Allan Blanchard

Introduction à la preuve de programmes C avec Frama-C et son greffon WP

2 novembre 2024

Reste de savoir

Table des matières

1. Introduction

Le code source de ce tutoriel est disponible sur GitHub, de même que les solutions aux différents exercices (incluant quelques preuves Coq de certaines propriétés).
Si vous trouvez des erreurs, n'hésitez pas à créer une *issue* ou une *pull request* sur :
https://github.com/AllanBlanchard/tutoriel_wp
ou à poster sur le sujet de la bêta sur Zeste de Savoir :
https://zestedesavoir.com/forums/sujet/7725/introduction-a-la-preuve-de-programmes-c-avec-frama-c-et-son-greffon-wp/

Le choix de certains exemples et d'une partie de l'organisation dans le présent tutoriel est le même que celui du tutoriel présenté à TAP 2013 par Nikolai Kosmatov, Virgile Prevosto et Julien Signoles du CEA List du fait de son cheminement didactique. Il contient également des exemples tirés de *ACSL By Example* de Jochen Burghardt, Jens Gerlach, Kerstin Hartig, Hans Pohl et Juan Soto du Fraunhofer. Pour les aspects formels, je me suis reposé sur le cours à propos de Why3 donné par Andrei Paskevich *à l'EJCP 2018* . Le reste vient de mon expérience personnelle avec Frama-C et WP.

Les versions des outils utilisés dans ce tutoriel sont les suivantes :

- Frama-C 30.0 Zinc
- Why3 1.7.2
- Alt-Ergo 2.6.0
- Coq 8.16.1 (pour les scripts proposés, Coq n'est pas utilisé dans le tutoriel)
- Z3 4.8.10 (utilisés dans un exemple, il n'est pas absolument nécessaire)

Selon les versions utilisées par le lecteur, quelques différences pourraient apparaître avec ce qui est prouvé et ce qui ne l'est pas. Quelques fonctionnalités ne sont disponibles que dans les versions récentes de Frama-C.

Le seul prérequis pour ce cours est d'avoir une connaissance basique du langage C, au moins jusqu'à la notion de pointeur.

Malgré son ancienneté, le C est un langage de programmation encore largement utilisé. Il faut dire qu'il n'existe, pour ainsi dire, aucun langage qui soit disponible sur une aussi large variété de plateformes (matérielles et logicielles) différentes, que son orientation bas-niveau et les années d'optimisations investies dans ses compilateurs permettent de générer à partir de programmes C des exécutables très performants (à condition bien sûr que le code le permette), et qu'il possède un nombre d'experts (et donc une base de connaissances) très conséquent.

De plus, de très nombreux systèmes reposent sur des quantités phénoménales de code historiquement écrit en C, qu'il faut maintenir et corriger, car ils coûteraient bien trop chers à re-développer.

Mais toute personne qui a déjà codé en C sait également que c'est un langage très difficile à maîtriser parfaitement. Les raisons sont multiples, mais les ambiguïtés présentes dans sa norme et la permissivité extrême qu'il offre au développeur, notamment en ce qui concerne les accès à la mémoire, font que créer un programme C robuste est très difficile même pour un programmeur chevronné.

Pourtant, C est souvent choisi comme langage de prédilection pour la réalisation de systèmes demandant un niveau critique de sûreté (aéronautique, ferroviaire, armement, ...) où il est apprécié pour ses performances, sa maturité technologique et la prévisibilité de sa compilation.

Dans ce genre de cas, les besoins de couverture par le test deviennent colossaux. Et, plus encore, la question « avons-nous suffisamment testé ? » devient une question à laquelle il est de plus en plus difficile de répondre. C'est là qu'intervient la preuve de programme. Plutôt que tester toutes les entrées possibles et (in)imaginables, nous allons prouver « mathématiquement » qu'aucun problème ne peut apparaître à l'exécution.

L'objet de ce tutoriel est d'utiliser Frama-C, un logiciel développé au CEA List, et WP, son greffon de preuve déductive, pour s'initier à la preuve de programmes C. Au-delà de l'usage de l'outil en lui-même, le but de ce tutoriel est de vous convaincre qu'il est possible d'écrire des programmes sans erreurs de programmation, mais également de sensibiliser à des notions simples permettant de mieux comprendre et de mieux écrire les programmes.

Merci aux différents bêta-testeurs pour leurs remarques constructives :
- Taurre
- barockobamo
- Vayel
- Aabu

Ainsi qu'aux validateurs qui ont encore permis d'améliorer la qualité de ce tutoriel :
- Taurre (oui, encore lui)
- Saroupille
- Aabu (oui, encore lui aussi)

Un grand merci à Jens Gerlach pour son aide lors de la traduction anglaise du tutoriel.
Merci finalement aux reviewers occasionnels sur GitHub :
- Alex Lyr
- Rafael Bachmann
- @charlesseizilles
- Myriam Clouet
- @Costava

— Daniel Rocha
— @GaoTamanrasset
— André Maroneze
— @MSoegtropIMC
— @rtharston
— @TrigDevelopment
— Quentin Santos
— Ricardo M. Correia
— Basile Desloges
pour leurs relectures et corrections.

2. La preuve de programmes et notre outil pour ce tutoriel : Frama-C

Le but de cette première partie est, dans une première section d'introduire rapidement en quoi consiste la preuve de programmes sans entrer dans les détails. Puis dans une seconde section de donner les quelques instructions nécessaires pour mettre en place Frama-C et les quelques prouveurs automatiques dont nous auront besoin pendant le tutoriel.

2.1. Preuve de programmes

2.1.1. Assurer la conformité des logiciels

Assurer qu'un programme a un comportement conforme à celui que nous attendons est souvent une tâche difficile. Plus en amont encore, il est déjà complexe d'établir sur quel critère nous pouvons estimer que le programme « fonctionne ».

— Les débutants « essayent » simplement leurs programmes et estiment qu'ils fonctionnent s'ils ne plantent pas.
— Les codeurs un peu plus habitués établissent quelques jeux de tests dont ils connaissent les résultats et comparent les sorties de leurs programmes.
— La majorité des entreprises établissent des bases de tests conséquentes, couvrant un maximum de code, tests exécutés de manière systématique sur les codes de leurs bases. Certaines font du développement dirigé par le test.
— Les entreprises de domaines critiques, comme l'aérospatial, le ferroviaire ou l'armement, passent par des certifications leur demandant de répondre à des critères très stricts de codage et de couverture de code par les tests.

Et bien sûr, il existe tous les « entre-deux » dans cette liste.

Dans toutes ces manières de s'assurer qu'un programme fait ce qui est attendu, il y a un mot qui revient souvent : *test*. Nous *essayons* des entrées de programme dans le but d'isoler des cas qui poseraient problème. Nous fournissons des entrées *estimées représentatives* de l'utilisation réelle du programme (laissant souvent de côté les usages non prévus, qui sont souvent les plus dangereux) et nous nous assurons que les résultats attendus sont conformes. Mais nous ne pouvons pas *tout* tester. Nous ne pouvons pas essayer *toutes* les combinaisons de *toutes* les entrées possibles du programme. Toute la difficulté réside donc dans le fait de choisir les bons tests.

Le but de la preuve de programmes est de s'assurer que, quelle que soit l'entrée fournie au programme, si elle est conforme à la spécification, alors le programme fera ce qui est attendu.

Cependant, comme nous ne pouvons pas tout essayer, nous allons établir formellement, mathématiquement, la preuve que le logiciel ne peut exhiber que les comportements qui sont spécifiés et que les erreurs d'exécution n'en font pas partie.

Une phrase très célèbre de Dijkstra exprime très clairement la différence entre test et preuve :

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

Le test de programme peut être utilisé pour montrer la présence de bugs, mais jamais pour montrer leur absence.

2.1.1.1. Le Graal du logiciel sans bug

Dans chaque nouvelle à propos d'attaque sur des systèmes informatiques, ou des virus, ou des bugs provoquant des crashs, il y a toujours la remarque séculaire « le programme inviolable/incassable/sans bugs n'existe pas ». Et il s'avère généralement que, bien qu'assez vraie, cette phrase est assez mal comprise.

Tout d'abord, nous ne précisons pas ce que nous entendons par « sans bug ». La création d'un logiciel fait toujours au moins intervenir deux étapes : la rédaction de ce qui est attendu sous la forme d'une spécification (souvent un cahier des charges) et la réalisation du logiciel répondant à cette spécification. À cela s'ajoute la spécification de notre langage de programmation qui nous définit la manière correcte de l'utiliser. Chacun de ces aspects peut donner lieu à l'introduction de bugs, que nous pouvons séparer en trois catégories :

- le programme n'est pas conforme, ou son comportement non défini, d'après la spécification du langage (par exemple, le programme accède en dehors d'un tableau pendant une recherche de l'indice de la valeur minimale) ;
- le programme n'est pas conforme à la spécification que nous en avons donné (par exemple, nous avons défini que le programme doit trouver l'indice de la valeur minimale d'un tableau, mais en fait il ne regarde pas sa dernière valeur à cause d'une erreur) ;
- la spécification ne reflète pas parfaitement « ce que nous voulons », et par conséquent, le programme non plus (par exemple, nous avons défini que le programme doit trouver l'indice de la valeur minimale d'un tableau, mais nous n'avons pas spécifié que s'il y en a plusieurs, il faut prendre la première, parce que cela me semblait trop évident, mais du coup ce n'est pas ce que fait le programme).

Chacune de ces catégories peut affecter la sûreté et la sécurité de nos programmes, qui ne sont pas des notions tout à fait équivalentes. Pour donner une idée de la différence qui existe entre ces deux notions, nous pouvons dire que dans le cas de la sécurité, on suppose qu'il existe une entité capable d'attaquer (volontairement ou pas) le système, tandis que dans la sûreté, nous voulons juste vérifier que lorsqu'il est utilisé de manière conforme, le système se comporte correctement. Par conséquent, sans sûreté, nous ne pouvons pas avoir la sécurité [1].

1. Selon votre domaine d'activité, le terme sûreté peut avoir un sens très différent. Plus précisément, un système sûr serait un système qui ne doit jamais mettre la vie d'un humain en danger. Et donc dans ce cas, la situation est inverse : sans sécurité, nous ne pouvons pas avoir la sûreté. Dans ce tutoriel, nous nous plaçons bien dans le cas « sûreté = le programme ne présente pas de problème lorsqu'on l'utilise de manière conforme ».

Tout au long de ce tutoriel, nous montrerons comment prouver que les implémentations de nos programmes ne contiennent pas de bugs correspondant aux deux premières catégories définies plus haut, à savoir qu'ils sont conformes :

— à la spécification de notre langage ;
— à la spécification de ce que nous attendons d'eux.

Mais quels sont les arguments de la preuve par rapport aux tests ? D'abord, la preuve est complète, elle n'oublie pas de cas s'ils sont présents dans la spécification (le test serait trop coûteux s'il était exhaustif). D'autre part, l'obligation de formaliser la spécification sous une forme logique demande de comprendre exactement le besoin auquel nous devons répondre.

Nous pourrions dire avec cynisme que la preuve nous montre finalement que l'implémentation « ne contient aucun bug de plus que la spécification », et donc que nous ne traitons pas la troisième catégorie de bugs que nous avons définie. Cependant, être sûr que le programme « ne contient aucun bug de plus que la spécification » est déjà un sacré pas en avant par rapport à savoir que le programme « ne contient pas beaucoup plus de bugs que la spécification », après tout cela représente deux catégories entières de bugs dont nous nous débarrassons, bugs qui peuvent déjà sévèrement compromettre la sûreté et la sécurité de nos programmes. Ensuite, il existe également des techniques pour traiter la troisième catégorie de bugs, en analysant les spécifications en quête d'erreurs ou d'insuffisance. Par exemple, les techniques de model checking - vérification de modèles - permettent de construire un modèle abstrait à partir d'une spécification et de produire un ensemble d'états accessibles du programme d'après le modèle. En caractérisant les états fautifs, nous sommes en mesure de déterminer si les états accessibles contiennent des états fautifs.

2.1.2. Un peu de contexte

En informatique, les méthodes dites *formelles* permettent de raisonner de manière rigoureuse, mathématique, à propos des programmes. Il existe un très large panel de méthodes formelles qui peuvent intervenir à tous les niveaux de la conception, l'implémentation, l'analyse et la validation des programmes ou de manière plus générale de tout système permettant le traitement de l'information.

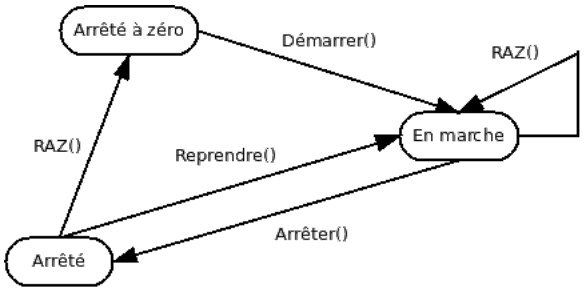
Ici, nous nous intéresserons à la vérification de la conformité de nos programmes au comportement attendu. Nous utiliserons des outils capables d'analyser le code et de nous dire si oui, ou non, notre code correspond à ce que nous voulons exprimer. La technique que nous allons étudier ici est une analyse statique, à opposer aux analyses dynamiques.

Le principe des analyses statiques est que nous n'exécuterons pas le programme pour nous assurer que son fonctionnement est correct, mais nous raisonnerons sur un modèle mathématique définissant l'ensemble des états qu'il peut atteindre. À l'inverse, les analyses dynamiques comme le test de programmes nécessitent d'exécuter le code analysé. Il existe également des analyses dynamiques et formelles, comme de la génération automatique de tests ou encore des techniques de *monitoring* de code qui pourront, par exemple, instrumenter un code source afin de vérifier à l'exécution que les allocations et désallocation de mémoire sont faites de manière sûre.

Dans le cas des analyses statiques, le modèle utilisé est plus ou moins abstrait selon la technique employée, c'est donc une approximation des états possibles de notre programme. Plus

l'approximation est précise, plus le modèle est concret, plus l'approximation est large, plus il est abstrait.

Pour illustrer la différence entre modèle concret et abstrait, prenons l'exemple d'un chronomètre simple. Une modélisation très abstraite du comportement de notre chronomètre est la suivante :



[Modélisation très abstraite d'un chronomètre]

Nous avons bien une modélisation du comportement de notre chronomètre avec différents états qu'il peut atteindre en fonction des actions qu'il subit. Cependant, nous n'avons pas modélisé comment ces états sont représentés dans le programme (Est-ce une énumération ? Une position précise atteinte au sein du code ?), ni comment est modélisé le calcul du temps (une seule variable en secondes ? Plusieurs variables heures, minutes, secondes ?). Nous aurions donc bien du mal à spécifier des propriétés à propos de notre programme. Nous pouvons ajouter des informations :

— état arrêté à zéro : temps = 0 s ;
— état en marche : temps > 0 s ;
— état arrêté : temps > 0 s.

Ce qui nous donne déjà un modèle plus concret, mais qui est toujours insuffisant pour poser des questions intéressantes à propos de notre système comme : « est-il possible que dans l'état arrêté, le temps continue de s'écouler ? ». Car nous n'avons pas modélisé l'écoulement du temps par le chronomètre.

À l'inverse, avec le code source du programme, nous avons un modèle concret du chronomètre. Le code source exprime bien le comportement du chronomètre puisque c'est lui qui sert à produire l'exécutable. Mais ce n'est pas pour autant le plus concret ! Par exemple, l'exécutable en code machine obtenu à la fin de la compilation est un modèle encore plus concret de notre programme.

Plus un modèle est concret, plus il décrit précisément le comportement de notre programme. Le code source exprime le comportement plus précisément que notre diagramme, mais il est moins précis que le code de l'exécutable. Cependant, plus un modèle est précis, plus il est difficile d'avoir une vision globale du comportement qu'il définit. Notre diagramme est compréhensible en un coup d'œil, le code demande un peu plus de temps, quant à l'exécutable ... Toute personne qui a déjà ouvert par erreur un exécutable avec un éditeur de texte sait que ce n'est pas très agréable à lire dans son ensemble [2].

2. Il existe des analyses formelles cherchant à comprendre le fonctionnement des exécutables en code machine, par exemple pour comprendre ce que font des logiciels malveillants ou pour détecter des failles de sécurité introduites lors de la compilation.

Lorsque nous créons une abstraction d'un système, nous l'approximons, pour limiter la quantité d'informations que nous avons à son sujet et faciliter notre raisonnement. Une des contraintes si nous voulons qu'une vérification soit correcte est bien sûr que nous ne devons jamais sous-approximer les comportements du programme : nous risquerions d'écarter un comportement qui contient une erreur. Inversement, si nous sur-approximons notre programme, nous ajoutons des exécutions qui ne peuvent pas arriver en réalité et si nous ajoutons trop d'exécutions inexistantes, nous pourrions ne plus être en mesure de prouver son bon fonctionnement dans le cas où certaines d'entre elles seraient fautives.

Dans le cas de l'outil que nous utiliserons, le modèle est plutôt concret. Chaque type d'instruction, chaque type de structure de contrôle d'un programme se voit attribuer une sémantique, une représentation de son comportement dans un monde purement logique, mathématique. Le cadre logique qui nous intéresse ici, c'est la logique de Hoare adaptée pour le langage C et toutes ses subtilités (qui rendent donc le modèle final très concret).

2.1.3. Les triplets de Hoare

La logique de Hoare est une méthode de formalisation des programmes proposée par Tony Hoare en 1969 dans un article intitulé *An Axiomatic Basis for Computer Programming* (une base axiomatique pour la programmation des ordinateurs). Cette méthode définit :

- des axiomes, qui sont des propriétés que nous admettons, comme « l'action "ne rien faire" ne change pas l'état du programme »,
- et des règles pour raisonner à propos des différentes possibilités de compositions d'actions, par exemple « l'action "ne rien faire" puis "faire l'action A" est équivalent à "faire l'action A" ».

Le comportement d'un programme est défini par ce que nous appelons les triplets de Hoare :

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

Où P et Q sont des prédicats (c'est-à-dire des formules logiques) qui nous disent dans quel état se trouve la mémoire traitée par le programme. C est un ensemble de commandes définissant un programme. Cette écriture nous dit « si nous sommes dans un état où P est vrai, après exécution de C et si C termine, alors Q sera vrai pour le nouvel état du programme ». Dit autrement, P est une précondition suffisante pour que C nous amène à la postcondition Q. Par exemple, le triplet correspondant à l'action « ne rien faire » (`skip`) est le suivant :

$$\{P\} \quad \texttt{skip} \quad \{P\}$$

Quand nous ne faisons rien, la postcondition est la même que la précondition.

Tout au long de ce tutoriel, nous verrons la sémantique de diverses constructions (blocs conditionnels, boucles, etc.) dans la logique de Hoare. Nous n'allons donc pas tout de suite rentrer dans ces détails puisque nous en aurons l'occasion plus tard. Il n'est pas nécessaire de mémoriser ces notions ni même de comprendre toute la théorie derrière, mais il est toujours utile d'avoir au moins une vague idée du fonctionnement de l'outil que nous utilisons.

Tout ceci nous donne les bases permettant de dire « voilà ce que fait cette action » mais ne nous donne pas encore de matériel pour mécaniser la preuve. L'outil que nous utiliserons repose sur la technique du calcul de plus faible précondition.

2.1.4. Calcul de plus faible précondition

Le calcul de plus faible précondition est une forme de sémantique de transformation de prédicats, proposée par Dijkstra en 1975 dans *Guarded commands, non-determinacy and formal derivation of programs*.

Cette phrase contient pas mal de mots méchants, mais le concept est en fait très simple. Comme nous l'avons vu précédemment, la logique de Hoare donne des règles expliquant comment se comportent les actions d'un programme. Mais elle ne nous dit pas comment appliquer ces règles pour établir une preuve complète du programme.

Dijkstra reformule la logique de Hoare en expliquant comment, dans le triplet $\{P\}C\{Q\}$, l'instruction ou le bloc d'instructions, C transforme le prédicat P, en Q. Cette forme est appelée « raisonnement vers l'avant » ou *forward-reasoning*. Nous calculons à partir d'une précondition et d'une ou plusieurs instructions, la plus forte postcondition que nous pouvons atteindre. Informellement, en considérant ce qui est reçu en entrée, nous calculons ce qui sera renvoyé au plus en sortie. Si la postcondition voulue est au plus aussi forte, alors nous avons prouvé qu'il n'y a pas de comportements non voulus.

Par exemple :

```
1 int a = 2;
2 a = 4;
3 //postcondition calculée : a == 4
4 //postcondition voulue   : 0 <= a <= 30
```

Pas de problème, 4 fait bien partie des valeurs acceptables pour a.

La forme qui nous intéresse, le calcul de plus faible précondition, fonctionne dans le sens inverse, nous parlons de « raisonnement vers l'arrière » ou *backward-reasoning*. À partir de la postcondition voulue et de l'instruction que nous traitons, nous déduisons la précondition minimale qui nous assure ce fonctionnement. Si notre précondition réelle est au moins aussi forte, c'est-à-dire, qu'elle implique la plus faible précondition, alors notre programme est valide.

Par exemple, si nous avons l'instruction (sous forme de triplet) :

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

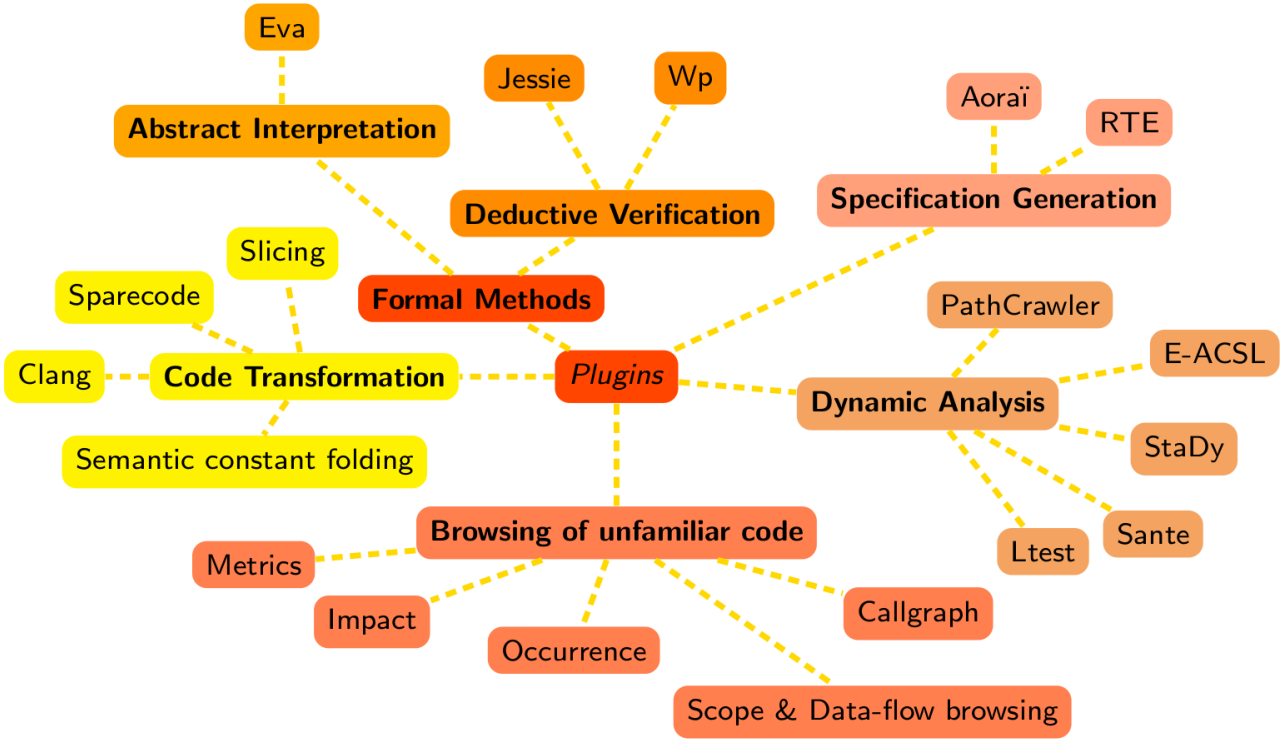
Quelle est la précondition minimale pour que la postcondition $\{x = 42\}$ soit vérifiée ? La règle nous dira que P est $\{\text{a}= 42\}$.

Nous n'allons pas nous étendre sur ces notions pour le moment, nous y reviendrons au cours du tutoriel pour comprendre ce que font les outils que nous utilisons. Et nos outils, parlons-en justement.

2.2. Frama-C

2.2.1. Frama-C ? WP ?

Frama-C (pour FRAmework for Modular Analysis of C code) est une plate-forme dédiée à l'analyse de programmes C créée par le CEA List et Inria. Elle est basée sur une architecture modulaire permettant l'utilisation de divers *plugins*. Les *plugins* fournis par défaut comprennent diverses analyses statiques (sans exécution du code analysé), dynamiques (avec exécution du code), ou combinant les deux. Ces *plugins* peuvent collaborer ou non, soit en communiquant directement entre eux, soit en utilisant le langage de spécification fourni par Frama-C.



Ce langage de spécification s'appelle ACSL (« Axel ») pour *ANSI C Specification Language* et permet d'exprimer les propriétés que nous souhaitons vérifier sur nos programmes. Ces propriétés seront écrites sous forme d'annotations dans les commentaires. Pour les personnes qui auraient déjà utilisé Doxygen, ça y ressemble beaucoup, sauf que tout sera écrit sous forme de formules logiques. Tout au long de ce tutoriel, nous parlerons beaucoup d'ACSL donc nous ne nous étendrons pas plus à son sujet ici.

L'analyse que nous allons utiliser ici est fournie par un plugin appelé WP pour *Weakest Precondition*, un plugin de vérification déduction. Il implémente la technique dont nous avons parlé

plus tôt : à partir des annotations ACSL et du code source, le plugin génère ce que nous appelons des conditions de vérification (ou obligations de preuves), qui sont des formules logiques dont nous devons vérifier si elles sont vraies ou fausses (on parle de « satisfiabilité »). Cette vérification peut être faite de manière manuelle ou automatique, ici nous n'utiliserons que des outils automatiques.

Nous allons en l'occurrence utiliser un solveur de formules SMT (satisfiabilité modulo théorie , et nous n'entrerons pas dans les détails). Ce solveur se nomme Alt-Ergo , initialement développé par le Laboratoire de Recherche en Informatique d'Orsay, aujourd'hui maintenu par OCamlPro.

2.2.2. Installation

Frama-C est un logiciel développé sous Linux et macOS. Son support est donc bien meilleur sous ces derniers. Il existe quand même de quoi faire une installation sous Windows et en théorie l'utilisation sera sensiblement la même que sous Linux, mais :

— le tutoriel présentera le fonctionnement sous Linux et l'auteur n'a pas expérimenté les différences d'utilisation qui pourraient exister avec Windows ;
— les versions récentes de Windows 10 permettent d'utiliser Windows Subsystem for Linux, en combinaison avec un Xserver installé sous Windows pour avoir la GUI ;
— La section « Bonus » un peu plus loin dans cette partie pourrait ne pas être accessible.

2.2.2.1. Linux

2.2.2.1.1. Via les gestionnaires de paquets Sous Debian, Ubuntu et Fedora, il existe des paquets pour Frama-C. Dans ce cas, il suffit de taper cette ligne de commande :

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

Par contre, ces dépôts ne sont pas systématiquement à jour. En soi, ce n'est pas très gênant, car il n'y a pas de nouvelle version de Frama-C tous les deux mois, mais il est tout de même bon de le savoir.

Les informations pour vérifier l'installation sont données dans la sous-section « Vérifier l'installation ».

2.2.2.1.2. Via Opam La deuxième solution consiste à passer par Opam, un gestionnaire de paquets pour les bibliothèques et applications OCaml.

D'abord, Opam doit être installé et configuré sur votre distribution (voir leur documentation). Ensuite, il faut également que quelques paquets de votre distribution soient présents préalablement à l'installation de Frama-C. Sur la plupart des distributions nous pouvons demander à

Opam d'aller chercher les bonnes dépendances pour le paquet que nous voulons installer. Pour cela, nous utilisons l'outil depext d'Opam, qu'il faut d'abord installer :

```
opam install depext
```

Puis nous lui demandons de prendre les dépendances de Frama-C :

```
opam depext frama-c
```

Si depext ne trouve pas les dépendances pour votre distribution, les paquets suivants doivent être présents sur votre système :

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

Sur les versions récentes de certaines distributions, GTK2 peut ne pas être disponible. Dans ce cas, ou si vous voulez avoir GTK3 et pas GTK2, les paquets GTK2, GTKSourceview2 et GnomeCanvas2 doivent être remplacés par GTK3 et GTKSourceview3.

Enfin, du côté d'Opam, il reste à installer Frama-C et Alt-Ergo.

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

Les informations pour vérifier l'installation sont données dans la sous-section « Vérifier l'installation ».

2.2.2.1.3. Via une compilation « manuelle » Pour installer Frama-C via une compilation manuelle, les paquets indiqués dans la section Opam sont nécessaires (mis à part Opam lui-même bien sûr). Il faut également une version récente d'OCaml et de son compilateur (y compris vers code natif). Il est aussi nécessaire d'installer Why3, en version 1.2.0, qui est disponible sur Opam ou sur leur site web (Why3).

Après décompression de l'archive disponible ici : https://frama-c.com/html/get-frama-c.html (Source distribution). Il faut se rendre dans le dossier et exécuter la commande :

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

Les informations pour vérifier l'installation sont données dans la sous-section « Vérifier l'installation ».

2.2.2.2. macOS

L'installation sur macOS passe par Homebrew et Opam. L'auteur n'ayant personnellement pas de macOS, voici une honteuse paraphrase du guide d'installation de Frama-C pour macOS.

Pour les utilitaires d'installation et de configuration :

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

Pour l'interface graphique :

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

Dépendances pour Alt-Ergo :

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

Frama-C et prouveur Alt-Ergo :

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

Les informations pour vérifier l'installation sont données dans la sous-section « Vérifier l'installation ».

2.2.2.3. Windows

Actuellement, la meilleure manière d'utiliser Frama-C sous Windows est de passer par Windows Subsystem for Linux. Une fois que le sous-système Linux est installé dans Windows, il suffit d'installer Opam et de suivre les instructions d'installation fournies dans la section Linux. Notons que pour profiter de l'interface graphique, il faudra installer un serveur X sous Windows.

Les informations pour vérifier l'installation sont données dans la sous-section « Vérifier l'installation ».

2.2.3. Vérifier l'installation

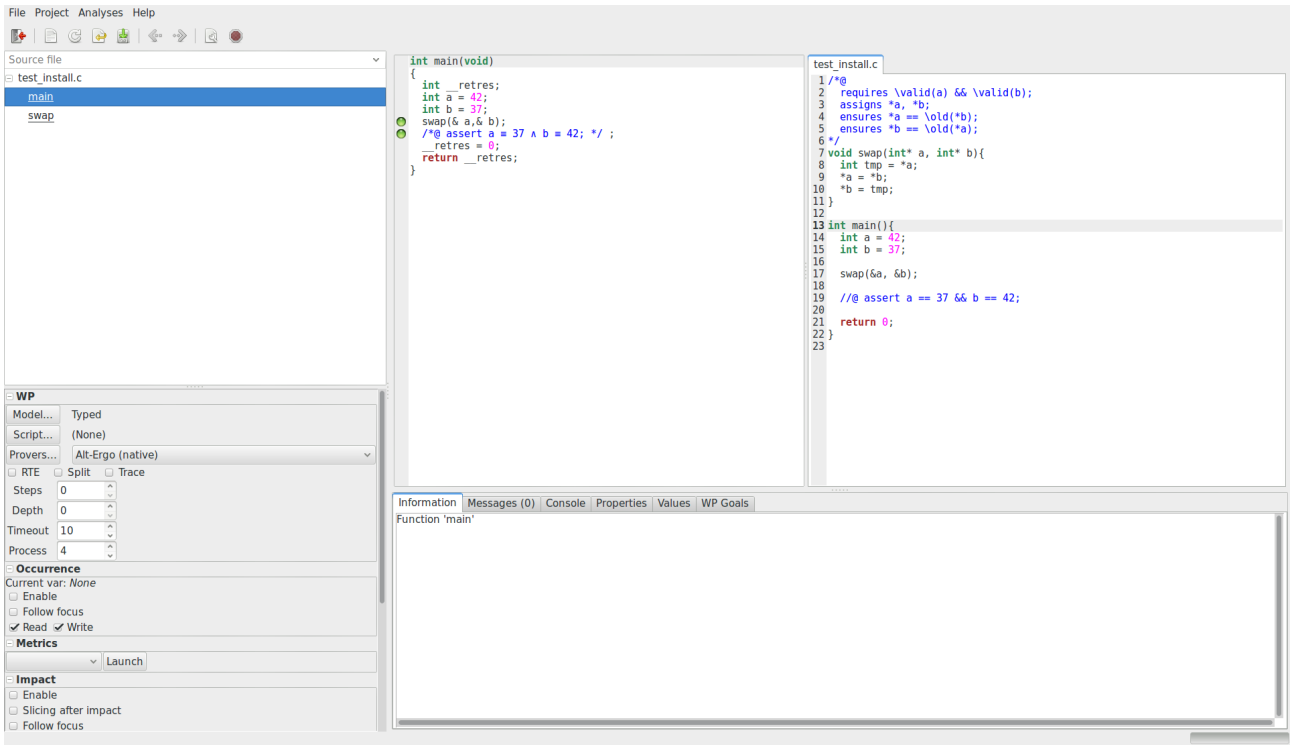
Pour vérifier votre installation, commencez par mettre ce code très simple dans un fichier « main.c » :

```
/*@
  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;
}
```

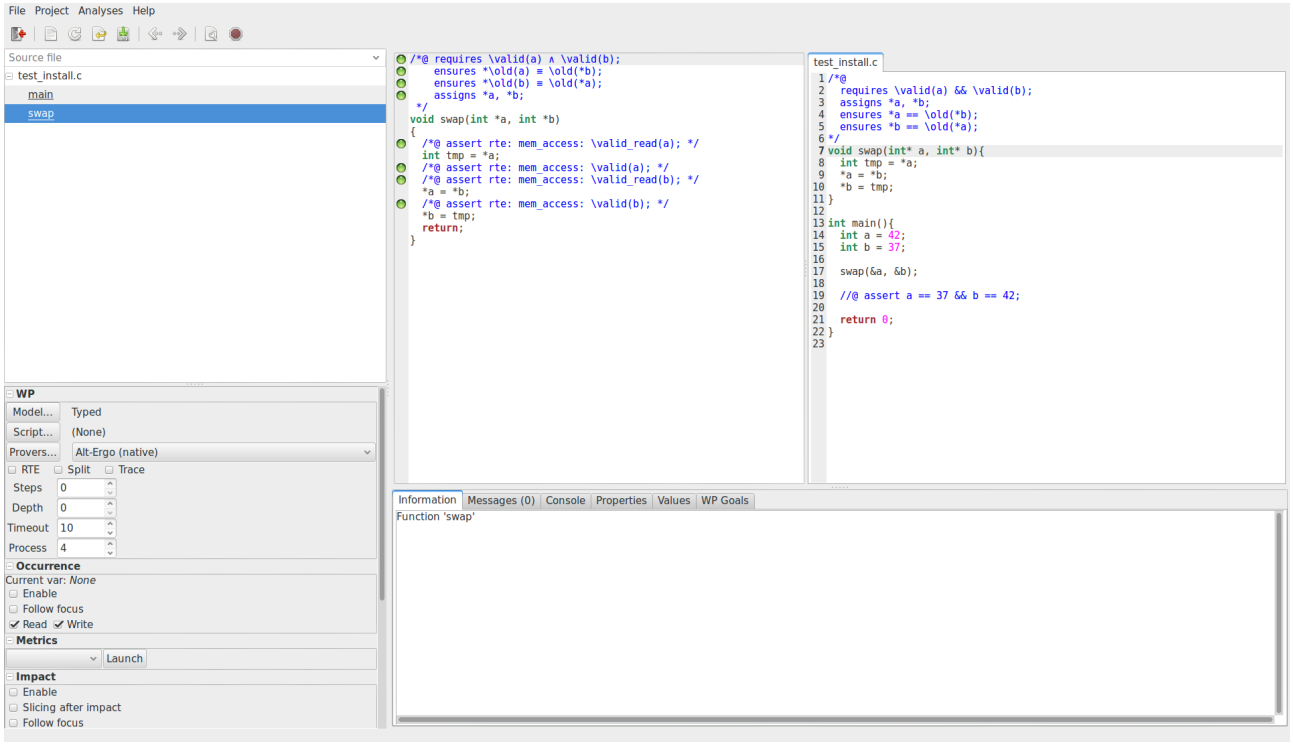
Ensuite, depuis un terminal, dans le dossier où ce fichier a été créé, nous pouvons lancer Frama-C avec la commande suivante :

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

Cette fenêtre devrait s'ouvrir.



En cliquant sur `main.c` dans le volet latéral gauche pour le sélectionner, nous pouvons voir le contenu du fichier `main.c` modifié et des pastilles vertes sur différentes lignes comme ceci :



Si c'est le cas, tant mieux, sinon il faudra d'abord vérifier que rien n'a été oublié au cours de l'installation (par exemple : l'oubli de bibliothèques graphiques ou encore l'oubli de l'installation d'Alt-Ergo). Si tout semble correct, divers forums pourront vous fournir de l'aide (Forum de Zeste de Savoir StackOverflow - Frama-C).

L'interface graphique de Frama-C ne permet pas l'édition du code source.

Pour les daltoniens, il est possible de lancer Frama-C avec un mode où les pastilles de couleurs sont remplacées par des idéogrammes noirs et blancs :

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

2.2.4. (Bonus) Installation de prouveurs supplémentaires

Cette partie est purement optionnelle, rien de ce qui est ici ne sera indispensable pendant le tutoriel. Cependant, lorsque l'on commence à s'intéresser vraiment à la preuve, il est possible de toucher assez rapidement aux limites du prouveur Alt-Ergo et d'avoir besoin d'autres prouveurs. Pour des propriétés simples, tous les prouveurs jouent à armes égales, pour des propriétés complexes, chaque prouveur à ses domaines de prédilection.

2.2.4.1. Why3

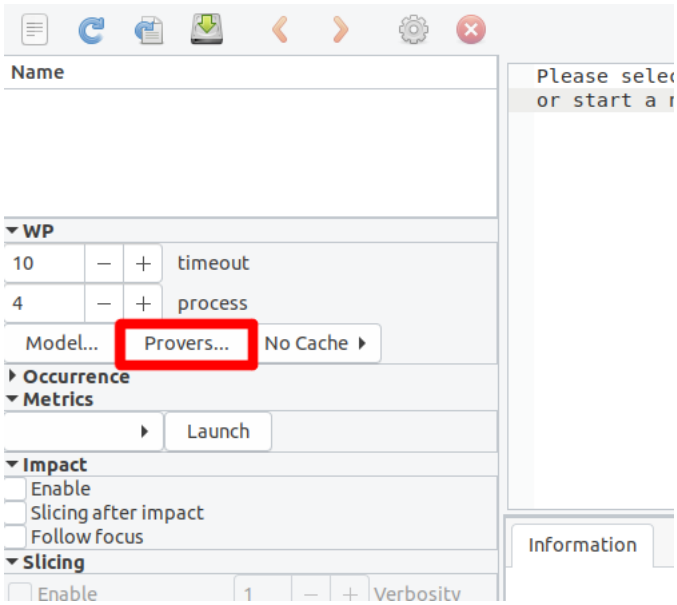
Why3 est une plateforme pour la preuve déductive développée par le LRI à Orsay. Elle embarque en outre un langage de programmation et de spécification ainsi qu'un module permettant le dialogue avec une large variété de prouveurs automatiques et interactifs. C'est en cela qu'il est utile dans le cas de Frama-C et WP. WP utilise Why3 comme *backend* pour dialoguer avec les prouveurs externes.

Nous pouvons retrouver sur ce même site la liste des prouveurs qu'il supporte. Il est vivement conseillé d'avoir Z3 , développé par Microsoft Research, et CVC4 , développé par des personnes de divers organismes de recherche (New York University, University of Iowa, Google, CEA List). Ces deux prouveurs sont très efficaces et relativement complémentaires.

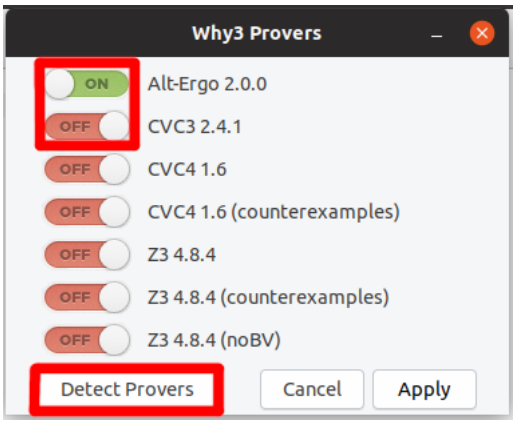
De nouveaux prouveurs peuvent être installés n'importe quand après l'installation de Frama-C. Cependant, la liste des prouveurs vus par Why3 doit être mise à jour :

```
1 why3 config detect
```

puis activée dans Frama-C. Dans le panneau latéral, dans la partie WP, cliquons sur « Provers... » :



Puis « Detect » dans la fenêtre qui apparaît. Une fois que c'est fait, les prouveurs peuvent être activés grâce au bouton qui se trouve à côté de leur nom.



2.2.4.2. Coq

Coq, développé par l'organisme de recherche Inria, est un assistant de preuve. C'est-à-dire que nous écrivons nous-mêmes les preuves dans un langage dédié et la plateforme se charge de vérifier (par typage) que cette preuve est valide.

Pourquoi aurait-on besoin d'un tel outil ? Il se peut parfois que les propriétés que nous voulons prouver soient trop complexes pour un prouveur automatique, typiquement lorsqu'elles nécessitent des raisonnements par induction avec des choix minutieux à chaque étape. Auquel cas, WP pourra générer des conditions de vérification traduites en Coq et nous laisser écrire la preuve en question.

Pour apprendre à utiliser Coq, ce tutoriel est très bon.

> *i* Si Frama-C est installé par l'intermédiaire du gestionnaire de paquets, il peut arriver que celui-ci ait directement intégré Coq.

Pour plus d'informations à propos de Coq et de son installation, voir The Coq Proof Assistant .

Voilà. Nos outils sont installés et prêts à fonctionner.

Le but de cette partie, en plus de l'installation de nos outils de travail pour la suite, est de faire ressortir deux informations claires :

— la preuve est un moyen d'assurer que nos programmes n'ont que des comportements conformes à notre spécification sans les exécuter ;
— il est toujours de notre devoir d'assurer que cette spécification est correcte.

3. Contrats de fonctions

Il est plus que temps d'entamer les hostilités. Contrairement aux tutoriels sur divers langages, nous commencerons par les fonctions. D'abord parce qu'il faut savoir en écrire avant d'entamer un tel tutoriel et surtout parce que cela permettra rapidement de produire des exemples simples que nous pouvons vérifier à l'aide de nos outils.

Au contraire, après le travail sur les fonctions, nous entamerons les notions les plus simples comme les affectations ou les structures conditionnelles pour comprendre comment fonctionne l'outil sous le capot.

Pour prouver qu'un code est valide, il faut d'abord pouvoir spécifier ce que nous attendons de lui. La preuve de programme consiste ensuite à s'assurer que le code que nous avons écrit effectue bien une action conforme à la spécification. Comme mentionné plus tôt dans le tutoriel, la spécification de code pour Frama-C est faite avec le langage ACSL, celui-ci nous permet (mais pas seulement, comme nous le verrons dans la suite) de poser un contrat pour chaque fonction.

3.1. Définition d'un contrat

Le principe d'un contrat de fonction est de poser les conditions selon lesquelles la fonction s'exécutera. On distinguera deux parties :

- **la précondition**, c'est-à-dire ce que doit assurer le code appelant à propos des variables passées en paramètres et de l'état de la mémoire globale pour que la fonction s'exécute correctement ;
- **la postcondition**, c'est-à-dire ce que s'engage à fournir la fonction en retour à propos de l'état de la mémoire et de la valeur de retour.

Ces propriétés sont exprimées en langage ACSL dont la syntaxe est relativement simple pour qui a déjà fait du C, puisqu'elle reprend la syntaxe des expressions booléennes du C. Cependant, elle ajoute également :

- certaines constructions et connecteurs logiques qui ne sont pas présents originellement en C pour faciliter l'écriture ;
- des prédicats pré-implémentés pour exprimer des propriétés souvent utiles en C (par exemple, la validité d'un pointeur) ;
- ainsi que des types plus généraux que les types primitifs du C, typiquement les types entiers ou réels.

Nous introduirons au fil du tutoriel les notations présentes dans le langage ACSL.

Les spécifications ACSL sont introduites dans nos codes source par l'intermédiaire d'annotations placées dans des commentaires. Syntaxiquement, un contrat de fonction est intégré dans les sources de la manière suivante :

```c
/*@
  //contrat
*/
void foo(int bar){

}
```

Notons bien le `@` à la suite du début du bloc de commentaire, c'est lui qui fait que ce bloc devient un bloc d'annotations pour Frama-C et pas un simple bloc de commentaires à ignorer.

Maintenant, regardons comment sont exprimés les contrats, à commencer par la postcondition, puisque c'est ce que nous attendons en priorité de notre programme (nous nous intéresserons ensuite aux préconditions).

3.1.1. Postcondition

La postcondition d'une fonction est précisée avec la clause `ensures`. Nous travaillerons avec la fonction suivante qui donne la valeur absolue d'un entier reçu en entrée. Une de ses postconditions est que le résultat (que nous notons avec le mot-clé `\result`) est supérieur ou égal à 0.

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

(Notons le `;` à la fin de la ligne de spécification comme en C).

Mais ce n'est pas tout, il faut également spécifier le comportement général attendu d'une fonction renvoyant la valeur absolue. À savoir : si la valeur est positive ou nulle, la fonction renvoie la même valeur, sinon elle renvoie l'opposé de la valeur.

Nous pouvons spécifier plusieurs postconditions, soit en les composants avec un `&&` comme en C, soit en introduisant une nouvelle clause `ensures`, comme illustré ci-dessous.

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

Cette spécification est l'opportunité de présenter un connecteur logique très utile que propose ACSL mais qui n'est pas présent en C : l'implication $A \Rightarrow B$, que l'on écrit en ACSL `A ==> B`. La table de vérité de l'implication est la suivante :

A	B	$A \Rightarrow B$
F	F	V
F	V	V
V	F	F
V	V	V

Ce qui veut dire qu'une implication $A \Rightarrow B$ est vraie dans deux cas : soit A est fausse (et dans ce cas, il ne faut pas se préoccuper de B), soit A est vraie et alors B doit être vraie aussi. Notons que cela signifie que $A \Rightarrow B$ est équivalente à $\neg A \vee B$. L'idée étant finalement « je veux savoir si dans le cas où A est vrai, B l'est aussi. Si A est faux, je considère que l'ensemble est vrai ». Par exemple, « s'il pleut, je veux vérifier que j'ai un parapluie, s'il ne pleut pas, ce n'est pas un problème de savoir si j'en ai un ou pas, tout va bien ».

Sa cousine l'équivalence $A \Leftrightarrow B$ (écrite `A <==> B` en ACSL) est plus forte. C'est la conjonction de l'implication dans les deux sens : $(A \Rightarrow B) \wedge (B \Rightarrow A)$. Cette formule n'est vraie que dans deux cas : A et B sont vraies toutes les deux, ou fausses toutes les deux (c'est donc la négation du ou-exclusif). Pour continuer avec notre petit exemple, « je ne veux plus seulement savoir si j'ai un parapluie quand il pleut, je veux être sûr de n'en avoir que dans le cas où il pleut ».

i

Profitons en pour rappeler l'ensemble des tables de vérités des opérateurs usuels en logique du premier ordre ($\neg$ = `!`, $\wedge$ = `&&`, $\vee$ = `||`) :

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

Revenons à notre spécification. Quand nos fichiers commencent à être longs et contenir beaucoup de spécifications, il peut être commode de nommer les propriétés que nous souhaitons vérifier. Pour cela, nous indiquons un nom (les espaces ne sont pas autorisées) suivi de `:` avant de mettre effectivement la propriété, il est possible de mettre plusieurs « étages » de noms pour catégoriser nos propriétés. Par exemple, nous pouvons écrire ceci :

```
1 /*@
2   ensures positive_value: function_result: \result >= 0;
3   ensures (val >= 0 ==> \result == val) &&
4           (val < 0 ==> \result == -val);
5 */
6 int abs(int val){
```

```
7    if(val < 0) return -val;
8    return val;
9  }
```

Dans une large part du tutoriel, nous ne nommerons pas les éléments que nous tenterons de prouver, les propriétés seront généralement relativement simples et peu nombreuses, les noms n'apporteraient pas beaucoup d'information.

Nous pouvons copier/coller la fonction `abs` et sa spécification dans un fichier `abs.c` et regarder avec Frama-C si l'implémentation est conforme à la spécification.

Pour cela, il faut lancer l'interface graphique de Frama-C (il est également possible de se passer de l'interface graphique, cela ne sera pas présenté dans ce tutoriel) soit par cette commande :

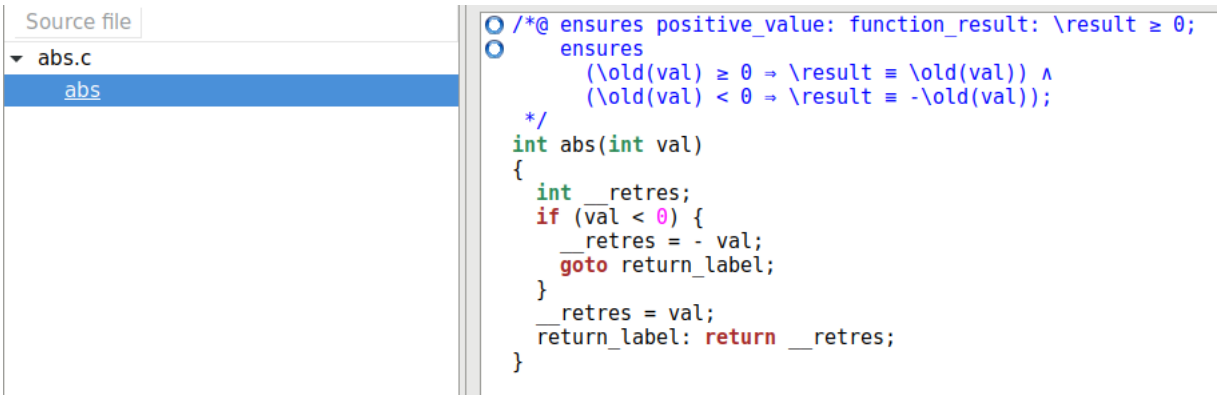
```
1  frama-c-gui
```

Soit en l'ouvrant depuis l'environnement graphique.

Il est ensuite possible de cliquer sur le bouton « *Create a new session from existing C files* », les fichiers à analyser peuvent être sélectionnés par double-clic, OK terminant la sélection. Par la suite, l'ajout d'autres fichiers à la session s'effectue en cliquant sur Files > Source Files.

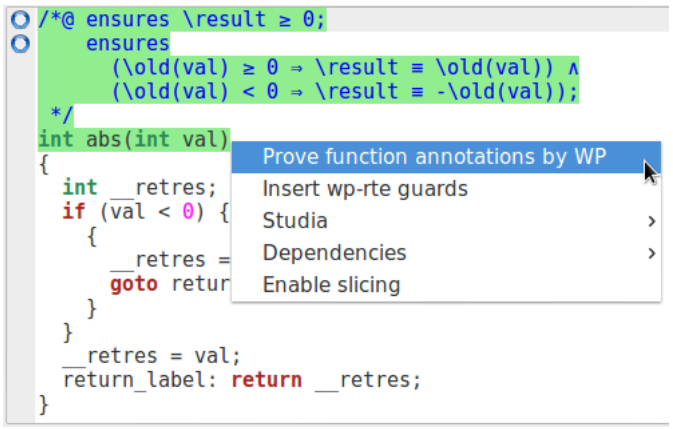
À noter également qu'il est possible d'ouvrir directement le(s) fichier(s) depuis la ligne de commande en le(s) passant en argument(s) de `frama-c-gui`.

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



La fenêtre de Frama-C s'ouvre, dans le volet correspondant aux fichiers et aux fonctions, nous pouvons sélectionner la fonction `abs`. Pour chaque ligne `ensures`, il y a un cercle bleu dans la marge. Ces cercles indiquent qu'aucune vérification n'a été tentée pour ces lignes.

Nous demandons de vérifier que le code répond à la spécification en faisant un clic droit sur le nom de la fonction et « *Prove function annotations by WP* » :



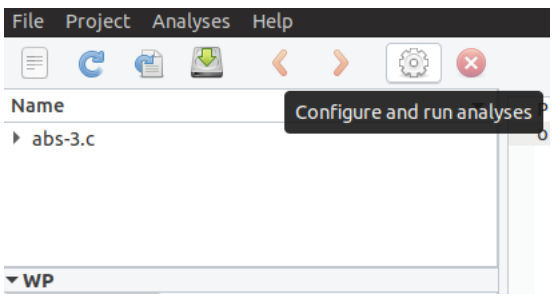
Nous pouvons voir que les cercles bleus deviennent des pastilles vertes, indiquant que la spécification est bien assurée par le programme. Il est possible de prouver les propriétés une à une en cliquant-droit sur celles-ci et pas sur le nom de la fonction. Nous pouvons également voir que deux annotations supplémentaires ont été générées par Frama-C pour notre fonction :

— `exits \false`
— `terminates \true`

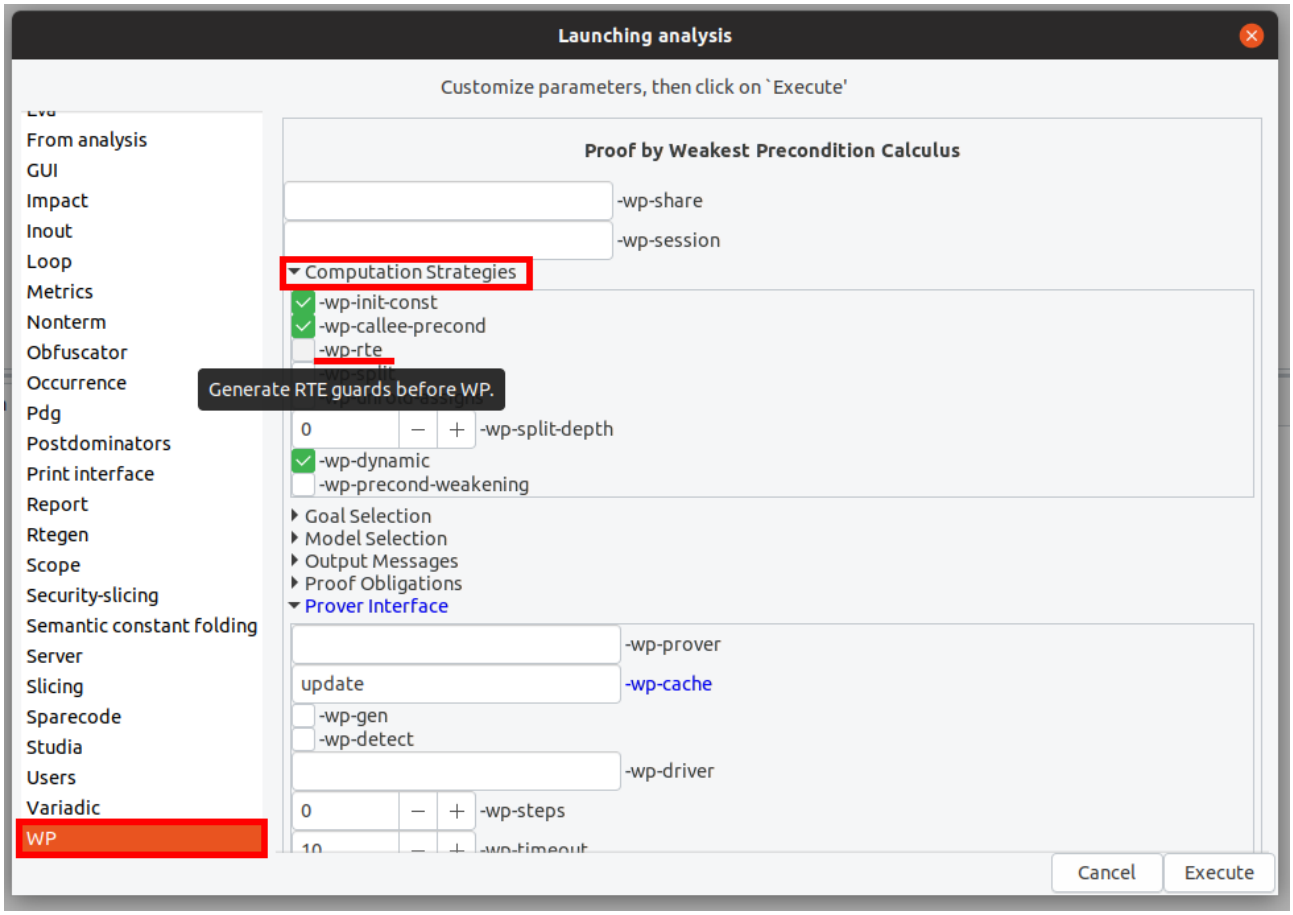
Le premier dit que la fonction ne doit pas appeler la fonction C `exit` (ou une fonction qui appellerait elle-même transitivement la fonction `exit` à un certain point), et qu'elle finira normalement son exécution en atteignant l'instruction `return`. La seconde dit que la fonction ne doit pas s'exécuter à l'infini. Ces annotations sont générées par défaut, cependant il est également possible d'ajouter sa propre clause `exits` ou `terminates` pour changer son comportement. Nous expliquerons cela plus tard, pour l'instant, ignorons simplement ces annotations.

Le code est-il vraiment sans erreurs ? WP nous permet de nous assurer que le code répond à la spécification, mais il ne fait pas de contrôle d'erreur à l'exécution (*runtime error*, abrégé RTE) si nous ne le demandons pas. Un autre *plugin* de Frama-C, appelé sobrement RTE, peut être utilisé pour générer des annotations ACSL qui peuvent ensuite être vérifiées par d'autres *plugins*. Son but est d'ajouter des contrôles dans le programme pour les erreurs d'exécutions possibles (débordements d'entiers, déréférencements de pointeurs invalides, division par 0, etc).

Pour activer ce contrôle, nous devons activer l'option dans la configuration de WP. Pour cela, il faut d'abord cliquer sur le bouton de configuration des *plugins* :



Et ensuite cocher l'option `-wp-rte` dans les options liées à WP :



Il est également possible de demander à WP d'ajouter ces contrôles par un clic droit sur le nom de la fonction puis « Insert wp-rte guards ».

À partir de ce point du tutoriel, `-wp-rte` devra toujours être activé pour traiter les exemples, sauf indication contraire.

Enfin, nous relançons la vérification (nous pouvons également cliquer sur le bouton « *Reparse* » de la barre d'outils, cela aura pour effet de supprimer les preuves déjà effectuées).

Nous voyons alors que WP échoue à prouver l'impossibilité de débordement arithmétique sur le calcul de -val. Et c'est bien normal parce que -`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;
}
```

i

Il est bon de noter que le risque de dépassement est pour nous réel, car nos machines (dont Frama-C détecte la configuration) fonctionne en complément à deux pour lequel le dépassement n'est pas défini par la norme C.

Ici, nous pouvons voir un autre type d'annotation ACSL. La ligne `//@ assert propriete ;` nous permet de demander la vérification d'une propriété à un point particulier du programme. Ici, l'outil l'a insérée pour nous, car il faut vérifier que le `-val` ne provoque pas de débordement, mais il est également possible d'en ajouter manuellement dans un code.

Comme le montre cette capture d'écran, nous avons deux nouveaux codes couleur pour les pastilles : vert + marron et orange.

La couleur verte et marron nous indique que la preuve a été effectuée, mais qu'elle dépend potentiellement de propriétés pour lesquelles ce n'est pas le cas.

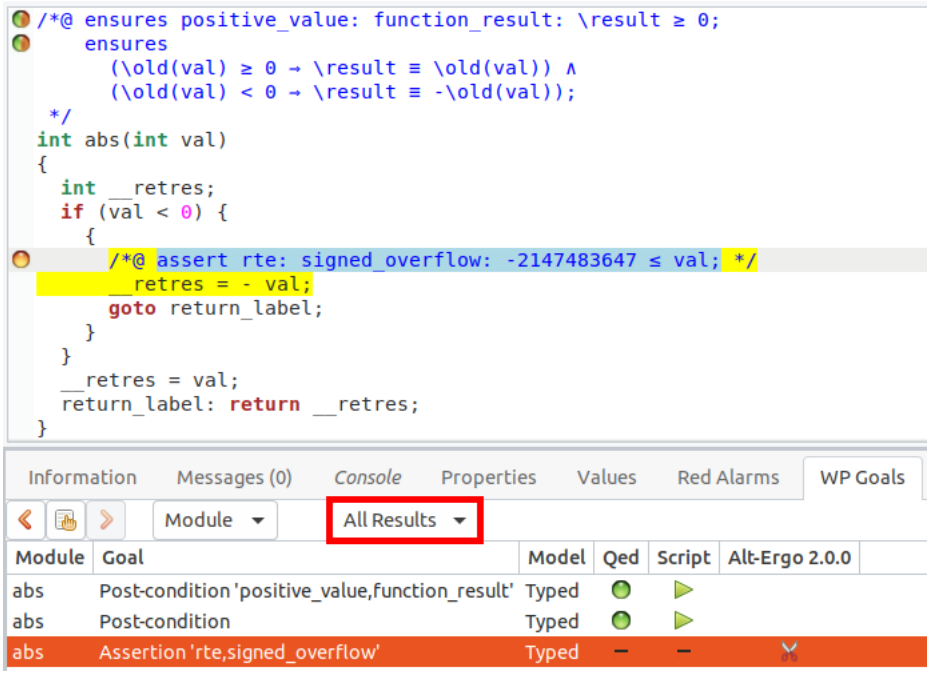
Si la preuve n'a pas été recommencée intégralement par rapport à la preuve précédente, ces pastilles ont dû rester vertes, car les preuves associées ont été réalisées avant l'introduction de la propriété nous assurant l'absence d'erreur d'exécution, et ne se sont donc pas reposées sur la connaissance de cette propriété puisqu'elle n'existait pas.

En effet, lorsque WP transmet une condition de vérification à un prouveur automatique, il transmet deux types de propriétés : G, le but, la propriété que l'on cherche à prouver, et S_1 ... S_n les diverses suppositions que l'on peut faire à propos de l'état du programme au point où l'on cherche à vérifier G. Cependant, il ne reçoit pas, en retour, quelles propriétés ont été utilisées par le prouveur pour valider G. Donc si S_3 fait partie des suppositions, et si WP n'a pas réussi à obtenir une preuve de S_3, il indique que G est vraie, mais en supposant que S_3 est vraie, pour laquelle nous n'avons actuellement pas établi de preuve.

La couleur orange nous signale qu'aucun prouveur n'a pu déterminer si la propriété est vérifiable. Les deux raisons peuvent être :

- qu'il n'a pas assez d'information pour le déterminer ;
- que malgré toutes ses recherches, il n'a pas pu trouver un résultat à temps. Auquel cas, il rencontre un *timeout* dont la durée est configurable dans le volet de WP.

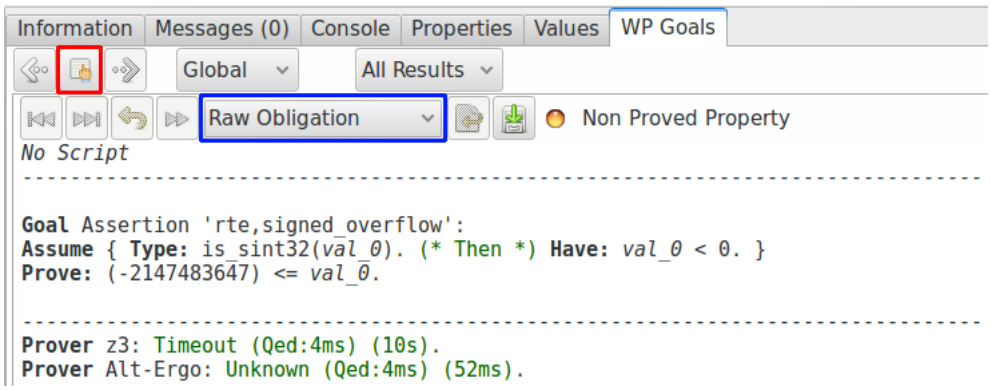
Dans le volet inférieur, nous pouvons sélectionner l'onglet « *WP Goals* », celui-ci nous affiche la liste des conditions de vérification et pour chaque prouveur indique un petit logo si la preuve a été tentée et si elle a été réussie, échouée ou a rencontré un *timeout* (logo avec les ciseaux). Pour voir la totalité des conditions de vérification, il faut s'assurer que « *All Results* » est bien sélectionné dans le champ encadré dans la capture.



Le tableau est découpé comme suit, en première colonne, nous avons le nom de la fonction où se trouve le but à prouver. En seconde colonne, nous trouvons le nom du but. Ici par exemple notre postcondition nommée est estampillée `postcondition 'positive_value,function_result'`, nous pouvons d'ailleurs noter que lorsqu'une propriété est sélectionnée dans le tableau, elle est également surlignée dans le code source. Les propriétés anonymes se voient assignées comme nom le type de propriété voulu. En troisième colonne, nous trouvons le modèle mémoire utilisé pour la preuve, (nous n'en parlerons pas dans ce tutoriel). Finalement, les dernières colonnes représentent les différents prouveurs accessibles à WP.

Dans ces prouveurs, le premier élément de la colonne est Qed. Ce n'est pas à proprement parler un prouveur. C'est un outil utilisé par WP pour simplifier les propriétés avant de les envoyer aux prouveurs externes. Ensuite, nous voyons la colonne Script, les scripts fournissent une manière de terminer les preuves à la main lorsque les prouveurs automatiques n'y arrivent pas. Finalement, nous trouvons la colonne Alt-Ergo, qui est un prouveur automatique. Notons que sur la propriété en question des ciseaux sont indiqués, cela veut dire que le prouveur a été stoppé à cause d'un *timeout*.

En fait, si nous double-cliquons sur la propriété « ne pas déborder » (surlignée en bleu dans la capture précédente), nous pouvons voir ceci (si ce n'est pas le cas, il faut s'assurer que « *Raw obligation* » est bien sélectionné dans le champ encadré en bleu) :



C'est la condition de vérification que génère WP par rapport à notre propriété et notre programme, il n'est pas nécessaire de comprendre tout ce qu'il s'y passe, juste d'avoir une idée globale. Elle contient (dans la partie « *Assume* ») les suppositions que nous avons pu donner et celles que WP a pu déduire des instructions du programme. Elle contient également (dans la partie « *Prove* ») la propriété que nous souhaitons vérifier.

Que fait WP avec ces éléments ? En fait, il les transforme en une formule logique puis demande aux différents prouveurs s'il est possible de la satisfaire (de trouver pour chaque variable, une valeur qui rend la formule vraie), cela détermine si la propriété est prouvable. Mais avant d'envoyer cette formule aux prouveurs, WP utilise un module qui s'appelle Qed et qui est capable de faire différentes simplifications à son sujet. Parfois, comme dans le cas des autres propriétés de `abs`, ces simplifications suffisent à déterminer que la propriété est forcément vraie, auquel cas, nous ne faisons pas appel aux prouveurs.

Lorsque les prouveurs automatiques ne parviennent pas à assurer que nos propriétés sont bien vérifiées, il est parfois difficile de comprendre pourquoi. En effet, les prouveurs sont généralement incapables de nous répondre autre chose que « oui », « non » ou « inconnu », ils ne sont pas capables d'extraire le « pourquoi » d'un « non » ou d'un « inconnu ». Il existe des outils qui sont capables d'explorer les arbres de preuve pour en extraire ce type d'information, Frama-C n'en possède pas à l'heure actuelle. La lecture des conditions de vérification peut parfois nous aider, mais cela demande un peu d'habitude pour pouvoir les déchiffrer facilement. Finalement, le meilleur moyen de comprendre la raison d'un échec est d'effectuer la preuve de manière interactive avec Coq. En revanche, il faut déjà avoir une certaine habitude de ce langage pour ne pas être perdu devant les conditions de vérification générées par WP, étant donné que celles-ci encodent les éléments de la sémantique de C, ce qui rend le code souvent indigeste.

Si nous retournons dans notre tableau des conditions de vérification (bouton encadré en rouge dans la capture d'écran précédente), nous pouvons donc voir que les hypothèses n'ont pas suffi aux prouveurs pour déterminer que la propriété « absence de débordement » est vraie (et nous l'avons dit : c'est normal), il nous faut donc ajouter une hypothèse supplémentaire pour garantir le bon fonctionnement de la fonction : une précondition d'appel.

3.1.2. Précondition

Les préconditions de fonctions sont introduites par la clause `requires`. De la même manière qu'avec `ensures`, nous pouvons composer nos expressions logiques et mettre plusieurs préconditions :

```
1 /*@
2   requires 0 <= a < 100;
3   requires b < a;
4 */
5 void foo(int a, int b){
6
7 }
```

Les préconditions sont des propriétés sur les entrées (et potentiellement sur des variables globales) qui seront supposées préalablement vraies lors de l'analyse de la fonction. La preuve que celles-ci sont effectivement validées n'interviendra qu'aux points où la fonction est appelée.

Dans ce petit exemple, nous pouvons également noter une petite différence avec le C dans l'écriture des expressions booléennes. Si nous voulons spécifier que `a` se trouve entre 0 et 100, il n'y a pas besoin d'écrire `0 <= a && a < 100` (c'est-à-dire en composant les deux comparaisons avec un `&&`). Nous pouvons simplement écrire `0 <= a < 100` et l'outil se chargera de faire la traduction nécessaire.

Si nous revenons à notre exemple de la valeur absolue, pour éviter le débordement arithmétique, il suffit que la valeur de `val` soit strictement supérieure à `INT_MIN` pour garantir que le débordement n'arrive pas. Nous l'ajoutons donc comme précondition (à noter : il faut également inclure l'en-tête où `INT_MIN` est défini) :

```
#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;
}
```

Rappel : la fenêtre de Frama-C ne permet pas l'édition du code source.

Une fois le code source modifié de cette manière, un clic sur « *Reparse* » et nous lançons à nouveau l'analyse. Cette fois, tout est validé pour WP ; notre implémentation est prouvée :

```
/*@ 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;
}
```

Nous pouvons également vérifier qu'une fonction qui appellerait `abs` satisfait bien la précondition qu'elle impose :

```
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
```

```
6 }
```

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

Notons qu'en cliquant sur la pastille à côté de l'appel de fonction, nous pouvons voir la liste des préconditions et voir quelles sont celles qui ne sont pas vérifiées. Ici, nous n'avons qu'une précondition, mais quand il y en a plusieurs, c'est très utile pour pouvoir voir quel est exactement le problème.

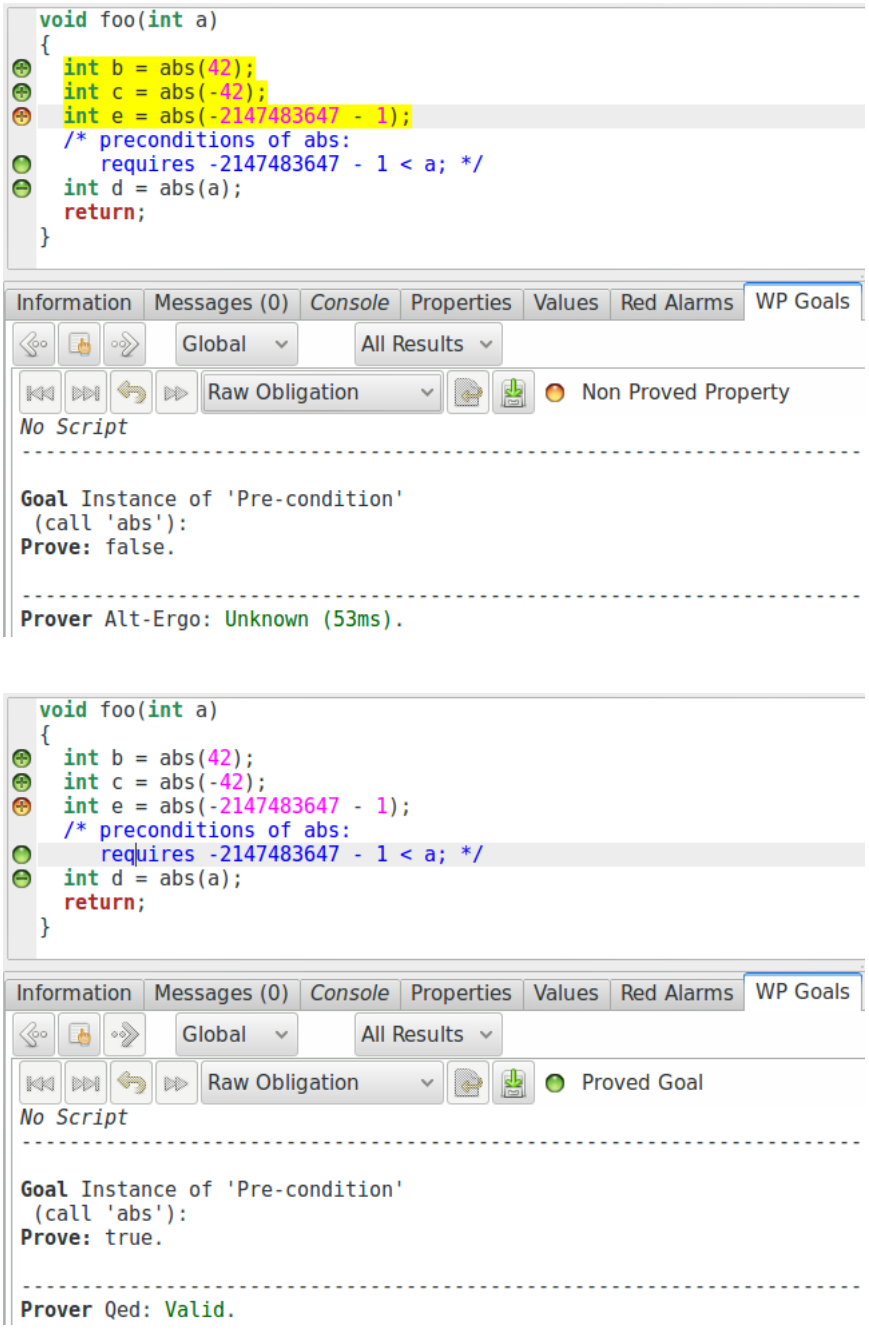
```
void foo(int a)
{
  int b = abs(42);
  int c = abs(-42);
  /* preconditions of abs:
     requires -2147483647 - 1 < a; */
  int d = abs(a);
  int e = abs(-2147483647 - 1);
  return;
}
```

Pour modifier un peu l'exemple, nous pouvons essayer d'inverser les deux dernières lignes. Auquel cas, nous pouvons voir que l'appel `abs(a)` est validé par WP s'il se trouve après l'appel `abs(INT_MIN)` ! Pourquoi ?

Il faut bien garder en tête que le principe de la preuve déductive est de nous assurer que si les préconditions sont vérifiées et que le calcul termine alors la postcondition est vérifiée.

Si nous donnons à notre fonction une valeur qui viole explicitement sa précondition, nous pouvons déduire que n'importe quoi peut arriver, incluant obtenir « faux » en postcondition. Plus précisément, ici, après l'appel, nous supposons que la précondition est vraie (puisque la fonction ne peut pas modifier la valeur reçue en paramètre), sinon la fonction n'aurait pas pu s'exécuter correctement. Par conséquent, nous supposons que `INT_MIN < INT_MIN` qui est trivialement faux. À partir de là, nous pouvons prouver tout ce que nous voulons, car ce « faux » devient une supposition pour tout appel qui viendrait ensuite. À partir de « faux », nous prouvons tout ce que nous voulons, car si nous avons la preuve de « faux » alors « faux » est vrai, de même que « vrai » est vrai. Donc tout est vrai.

En prenant le programme modifié, nous pouvons d'ailleurs regarder les conditions de vérification générées par WP pour l'appel fautif et l'appel prouvé par conséquent :



Nous pouvons remarquer que pour les appels de fonctions, l'interface graphique surligne le chemin d'exécution suivi avant l'appel dont nous cherchons à vérifier la précondition. Ensuite, si nous regardons l'appel `abs(INT_MIN)`, nous remarquons qu'à force de simplifications, Qed a déduit que nous cherchons à prouver « False ». Conséquence logique, l'appel suivant `abs(a)` reçoit dans ses suppositions « False ». C'est pourquoi Qed est capable de déduire immédiatement « True ».

La deuxième partie de la question est alors : pourquoi lorsque nous mettons les appels dans l'autre sens (`abs(a)` puis `abs(INT_MIN)`), nous obtenons quand même une violation de la précondition sur le deuxième ? La réponse est simplement que pour `abs(a)` nous ajoutons dans nos suppositions la connaissance `a < INT_MIN`, et tandis que nous n'avons pas de preuve que c'est vrai, nous n'en avons pas non plus que c'est faux. Donc si nous obtenons

nécessairement une preuve de « faux » avec un appel `abs(INT_MIN)`, ce n'est pas le cas de l'appel `abs(a)`, car `a` peut être autre chose que `INT_MIN`.

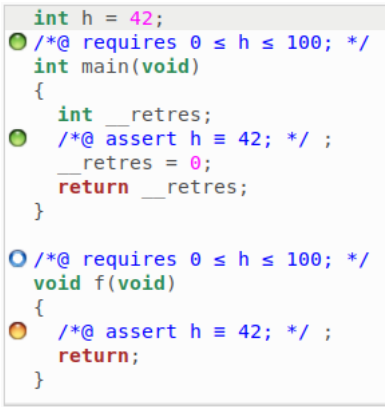
3.1.3. Le cas particulier de la fonction `main`

La fonction `main` n'est généralement pas appelée directement par le programme. Cependant, elle peut avoir des préconditions. Dans un tel cas, WP génère des conditions de vérification pour elles, mais avec une différence importante : le contexte fournit pour la preuve n'est pas le même que pour les autres fonctions. En effet, comme la fonction principale est appelée avant toute autre fonction du programme, à ce point de l'exécution, les variables globales ont nécessairement la valeur qui leur a été transmise à l'initialisation.

Illustrons cela avec l'extrait de code suivant :

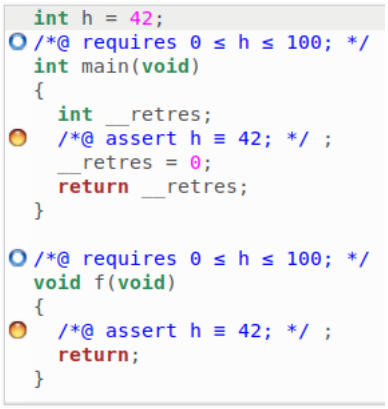
```
1  int h = 42 ;
2
3  //@ requires 0 <= h <= 100 ;
4  int main(void){
5    //@ assert h == 42 ;
6  }
7
8  //@ requires 0 <= h <= 100 ;
9  void f(void){
10   //@ assert h == 42 ;
11 }
```

Si nous exécutons WP sur cet exemple, la clause `requires` de la fonction `main` est prouvée, tandis qu'aucune tentative de preuve n'a été réalisée pour la précondition de la fonction `f` puisqu'elle n'est pas appelée dans le programme. La précondition de la fonction `main` est vérifiée, car après l'initialisation `h` a la valeur 42 qui est bien entendu entre 0 et 100. Remarquons que l'assertion dans la fonction `main` est, elle aussi, prouvée puisque la valeur de `h` n'est pas changée par la fonction, alors que dans la fonction `f` ce n'est pas le cas puisque WP ne fait aucune supposition à propos des valeurs des variables globales.

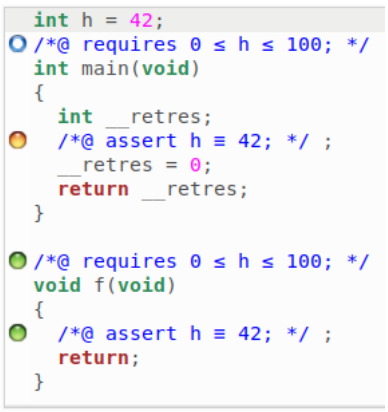


Notons que WP fait cette hypothèse à propos de la fonction `main` parce que c'est le comportement par défaut de Frama-C. Ce comportement peut être configuré à l'aide de l'option `-lib-entry` qui dit à Frama-C de considérer que toutes les fonctions (incluant donc `main`)

peuvent être appelées dans le programme. Donc, quand nous activons cette option sur la ligne de commande de Frama-C, pour notre exemple, aucune condition de vérification n'est générée pour les clauses `requires` de `main` puisqu'elle n'est pas appelée et aucune des assertions n'est prouvée :



Finalement, Frama-C permet de configurer quelle fonction du programme doit être considérée comme la fonction principale. Par exemple, si nous appelons Frama-C avec l'option `-main=f` sur la ligne de commande, la clause `requires` de la fonction `f` est prouvée, de même que l'assertion qu'elle contient. À l'inverse, aucune vérification n'est tentée pour la clause `requires` de la fonction `main` et l'assertion dans la fonction `main` n'est pas prouvée.



3.1.4. Exercices

Ces exercices ne sont pas absolument nécessaires pour lire les chapitres à venir dans ce tutoriel, nous conseillons quand même de les réaliser. Nous suggérons aussi fortement d'au moins lire le quatrième exercice qui introduit une notation qui peut parfois d'avérer utile.

3.1.4.1. Addition

Écrire la postcondition de la fonction d'addition suivante :

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

Lancer la commande :

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

Lorsque la preuve que la fonction est conforme à son contrat est établie, lancer la commande :

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

qui devrait échouer. Adapter le contrat en ajoutant la bonne précondition.

3.1.4.2. Distance

Écrire la postcondition de la fonction distance suivante, en exprimant la valeur de `b` en fonction de `a` et `\result` :

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

Lancer la commande :

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

Lorsque la preuve que la fonction est conforme à son contrat est établie, lancer la commande :

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

qui devrait échouer. Adapter le contrat en ajoutant la bonne précondition.

3.1.4.3. Lettres de l'alphabet

Écrire la postcondition de la fonction suivante, qui retourne vrai si le caractère reçu en entrée est une lettre de l'alphabet. Utiliser la relation d'équivalence `<==>`.

```c
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 ;
}
```

Lancer la commande :

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

Toutes les conditions de vérification devraient être prouvées, y compris les assertions dans la fonction main.

3.1.4.4. Jours du mois

Écrire la postcondition de la fonction suivante qui retourne le nombre de jours en fonction du mois reçu en entrée (NB : nous considérons que le mois reçu est entre 1 et 12), pour février, nous considérons uniquement le cas où il a 28 jours, nous verrons plus tard comment régler ce problème :

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

Lancer la commande :

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

Lorsque la preuve que la fonction est conforme à contrat est établie, lancer la commande :

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

Si cela échoue, adapter le contrat en ajoutant la bonne précondition.

Le lecteur aura peut-être constaté qu'écrire la postcondition est un peu laborieux. Il est possible de simplifier cela. ACSL fournit la notion d'ensemble mathématique et l'opérateur `\in` qui peut être utilisé pour vérifier si une valeur est dans un ensemble ou non.

Par exemple :

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

Modifier la postcondition en utilisant cette notation.

3.1.4.5. Le dernier angle d'un triangle

Cette fonction reçoit deux valeurs d'angle en entrée et retourne la valeur du dernier angle composant le triangle correspondant en se reposant sur la propriété que la somme des angles d'un triangle vaut 180 degrés. Écrire la postcondition qui exprime que la somme des trois angles vaut 180.

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

Lancer la commande :

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

Lorsque la preuve que la fonction est conforme à son contrat est établie, lancer la commande :

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

Si cela échoue, adapter le contrat en ajoutant la bonne précondition. Notons que la valeur de chaque angle ne peut pas être supérieure à 180 et que cela inclut l'angle résultant.

3.2. De l'importance d'une bonne spécification

3.2.1. Bien traduire ce qui est attendu

C'est certainement notre tâche la plus difficile. En soi, la programmation est déjà un effort consistant à écrire des algorithmes qui répondent à notre besoin. La spécification nous demande également de faire ce travail, la différence est que nous ne nous occupons plus de préciser la manière de répondre au besoin, mais le besoin lui-même. Pour prouver que la réalisation implémente bien ce que nous attendons, il faut donc être capable de décrire précisément le besoin.

Changeons d'exemple et spécifions la fonction suivante :

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

Le lecteur pourra écrire et prouver sa spécification. Pour la suite, nous travaillerons avec celle-ci :

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

Si nous donnons ce code à WP, il accepte sans problème de prouver la fonction. Pour autant cette spécification est-elle suffisante ? Nous pouvons par exemple essayer de voir si ce code est validé :

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

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

La réponse est non. En fait, nous pouvons aller plus loin en modifiant le corps de la fonction `max` et remarquer que le code suivant est également valide quant à la spécification :

```
#include <limits.h>

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

Même si elle est correcte, notre spécification est trop permissive. Il faut que nous soyons plus précis. Nous attendons du résultat non seulement qu'il soit supérieur ou égal à nos deux paramètres, mais également qu'il soit exactement l'un des deux :

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

Nous pouvons également prouver que cette spécification est vérifiée par notre fonction. Mais

nous pouvons maintenant prouver en plus l'assertion présente dans notre fonction `foo`, et nous ne pouvons plus prouver que l'implémentation qui retourne `INT_MAX` vérifie la spécification.

3.2.2. Préconditions incohérentes

Bien spécifier son programme est d'une importance cruciale. Typiquement, préciser une précondition fausse peut nous donner la possibilité de prouver FAUX :

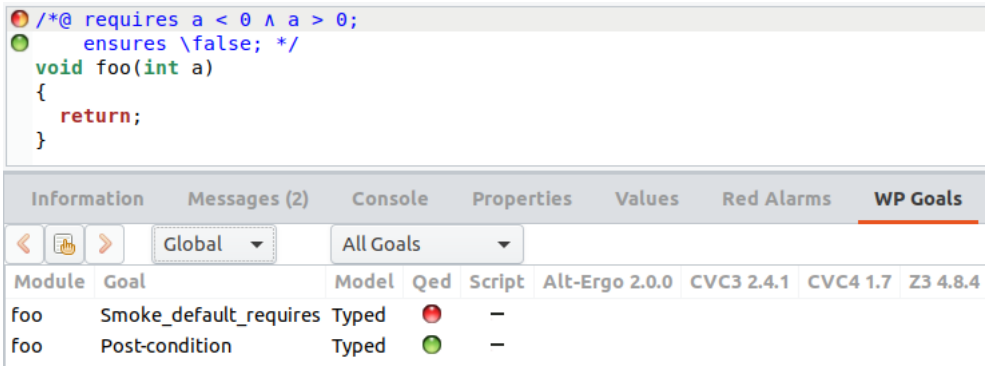
```
1 /*@
2   requires a < 0 && a > 0;
3   ensures \false;
4 */
5 void foo(int a){
6
7 }
```

Si nous demandons à WP de prouver cette fonction. Il l'acceptera sans rechigner, car la propriété que nous lui donnons comme précondition est nécessairement fausse. Par contre, nous aurons bien du mal à lui donner une valeur en entrée qui satisfait la précondition.

Pour cette catégorie particulière d'incohérences, une fonctionnalité utile de WP est l'option « *smoke tests* » du greffon. Ces tests préliminaires, effectués sur notre spécification, sont utilisés pour détecter que des préconditions ne peuvent pas être satisfaites. Par exemple, ici, nous pouvons lancer cette ligne de commande :

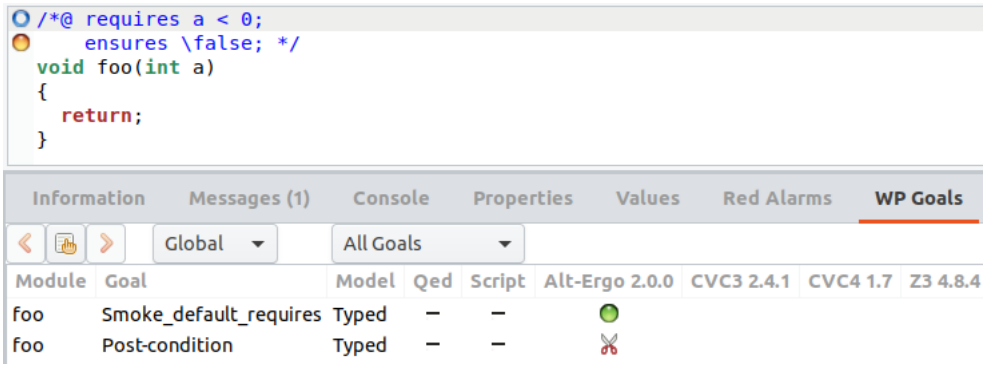
```
1   frama-c-gui -wp -wp-smoke-tests file.c
```

et nous obtenons le résultat suivant dans l'interface graphique :



Nous pouvons voir une pastille orange et rouge à côté de la précondition de la fonction, qui signifie que s'il existe un appel atteignable à la fonction dans le programme, la précondition sera nécessairement violée lors de cet appel ; et une pastille rouge dans la liste des conditions de vérification, indiquant qu'un prouveur a réussi à montrer que la précondition est incohérente.

Notons que lorsque ces tests préliminaires réussissent, par exemple si nous corrigeons la précondition de cette façon :



Cela ne signifie pas que la précondition est nécessairement cohérente, juste qu'aucun prouveur n'a été capable de montrer qu'elle est incohérente.

Certaines notions que nous verrons plus loin dans le tutoriel apporterons un risque encore plus grand de créer ce genre d'incohérence. Il faut donc toujours avoir une attention particulière pour ce que nous spécifions.

3.2.3. Pointeurs

S'il y a une notion à laquelle nous sommes confrontés en permanence en langage C, c'est bien la notion de pointeur. C'est une notion complexe et l'une des principales cause de bugs critiques dans les programmes, ils ont donc droit à un traitement de faveur dans ACSL. Pour avoir une spécification correcte des programmes utilisant des pointeurs, il est impératif de détailler la configuration de la mémoire que l'on considère.

Prenons par exemple une fonction `swap` pour les entiers (pour le moment seulement vérifiable sans la vérification des RTEs) :

```
1 /*@
2   ensures *a == \old(*b) && *b == \old(*a);
3 */
4 void swap(int* a, int* b){
5   int tmp = *a;
6   *a = *b;
7   *b = tmp;
8 }
```

3.2.3.1. Historique des valeurs

Ici, nous introduisons une première fonction logique fournie de base par ACSL : `\old`, qui permet de parler de l'ancienne valeur d'un élément. Ce que nous dit donc la spécification est « la fonction doit assurer que `*a` soit égal à l'ancienne valeur (au sens : la valeur avant l'appel) de `*b` et inversement ».

La fonction `\old` ne peut être utilisée que dans la postcondition d'une fonction. Si nous avons besoin de ce type d'information ailleurs, nous utilisons `\at` qui nous permet d'exprimer des propriétés à propos de la valeur d'une variable à un point donné. Elle reçoit deux paramètres. Le

premier est la variable (ou position mémoire) dont nous voulons obtenir la valeur et le second la position (sous la forme d'un label C) à laquelle nous voulons contrôler la valeur en question.

Par exemple, nous pourrions écrire :

```
1  int a = 42;
2 Label_a:
3  a = 45;
4
5  //@assert a == 45 && \at(a, Label_a) == 42;
```

En plus des labels que nous pouvons nous-mêmes créer, il existe 6 labels qu'ACSL nous propose par défaut :

- `Pre` / `Old` : valeur avant l'appel de la fonction,
- `Post` : valeur après l'appel de la fonction,
- `LoopEntry` : valeur en début de boucle,
- `LoopCurrent` : valeur en début du pas actuel de la boucle,
- `Here` : valeur au point d'appel.

Le comportement de `Here` est en fait le comportement par défaut lorsque nous parlons de la valeur d'une variable. Son utilisation avec `\at` nous servira généralement à s'assurer de l'absence d'ambiguïté lorsque nous parlons de divers points de programme dans la même expression.

À la différence de `\old`, qui ne peut être utilisée que dans les postconditions de contrats de fonction, `\at` peut être utilisée partout. En revanche, tous les points de programme ne sont pas accessibles selon le type d'annotation que nous sommes en train d'écrire. `Old` et `Post` ne sont disponibles que dans les postconditions d'un contrat, `Pre` et `Here` sont disponibles partout. `LoopEntry` et `LoopCurrent` ne sont disponibles que dans le contexte de boucles (dont nous parlerons plus loin dans le tutoriel).

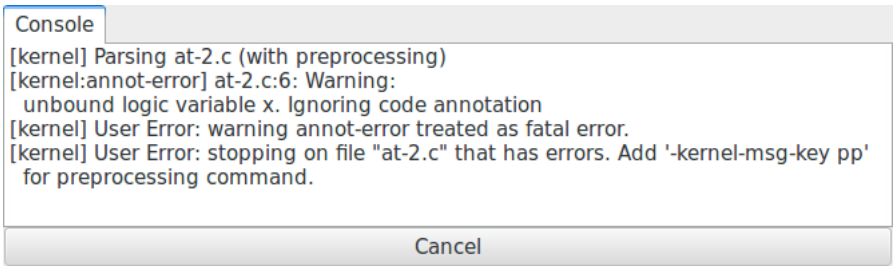
Notons qu'il est important de s'assurer que l'on utilise `\old` et `\at` pour des valeurs qui ont du sens. C'est pourquoi par exemple dans un contrat, toutes les valeurs reçues en entrée sont placées dans un appel à `\old` par Frama-C lorsqu'elles sont utilisées dans les postconditions, la nouvelle valeur d'une variable fournie en entrée d'une fonction n'a aucun sens pour l'appelant puisque cette valeur est inaccessible par lui : elles sont locales à la fonction appelée. Par exemple, si nous regardons le contrat de la fonction `swap` dans Frama-C, nous pouvons voir que dans la postcondition, chaque pointeur se trouve dans un appel à `\old` :

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

Pour la fonction `\at`, nous devons plus explicitement faire attention à cela. En particulier, le label transmis en entrée doit avoir un sens par rapport à la portée de la variable que l'on lui

transmet. Par exemple, dans le programme suivant, Frama-C détecte que nous demandons la valeur de la variable `x` à un point du programme où elle n'existe pas :

```
1  void example_1(void){
2   L: ;
3    int x = 1 ;
4    //@ assert \at(x, L) == 1 ;
5  }
```



Cependant, dans certains cas, tout ce que nous pouvons obtenir est un échec de la preuve, parce que déterminer si la valeur existe ou non à un label particulier ne peut être fait par une analyse purement syntaxique. Par exemple, si la variable est déclarée, mais pas définie, ou si nous demandons la valeur d'une zone mémoire pointée :

```
1  void example_2(void){
2    int x ;
3   L:
4    x = 1 ;
5    //@ assert \at(x, L) == 1 ;
6  }
7
8  void example_3(void){
9   L: ;
10   int x = 1 ;
11   int *ptr = &x ;
12   //@ assert \at(*\at(ptr, Here), L) == 1 ;
13 }
```

Ici, il est facile de remarquer le problème. Cependant, le label que nous transmettons à la fonction `\at` est propagé également aux sous-expressions. Dans certains cas, des termes qui paraissent tout à fait innocents peuvent en réalité nous donner des comportements surprenant si nous ne gardons pas cette idée en tête. Par exemple, dans le programme suivant :

```
1  /*@ requires x + 2 != p ; */
2  void example_4(int* x, int* p){
3    *p = 2 ;
4    //@ assert x[2]  == \at(x[2], Pre) ;
5    //@ assert x[*p] == \at(x[*p], Pre) ;
6  }
```

La première assertion est prouvée, et tandis que la seconde assertion a l'air d'exprimer la même propriété, elle ne peut pas être prouvée. La raison est simplement qu'elle n'exprime pas la même propriété. L'expression `\at(x[*p], Pre)` doit être lue comme `\at(x[\at(*p)], Pre)`

parce que le label est propagé à la sous-expression `*p`, pour laquelle nous ne connaissons pas la valeur au label `Pre` (qui n'est pas spécifié).

Pour le moment, nous n'utiliserons pas `\at`, mais elle peut rapidement se montrer indispensable pour écrire des spécifications précises.

3.2.3.2. Validité de pointeurs

Si nous essayons de prouver le fonctionnement de `swap` (en activant la vérification des RTE), notre postcondition est bien vérifiée, mais WP nous indique qu'il y a un certain nombre de possibilités de *runtime-error*. Ce qui est normal, car nous n'avons pas précisé à WP que les pointeurs que nous recevons en entrée de fonction sont valides.

Pour ajouter cette précision, nous allons utiliser le prédicat `\valid` qui reçoit un pointeur en entrée :

```
1  /*@
2    requires \valid(a) && \valid(b);
3    ensures  *a == \old(*b) && *b == \old(*a);
4  */
5  void swap(int* a, int* b){
6    int tmp = *a;
7    *a = *b;
8    *b = tmp;
9  }
```

À partir de là, les déréférencements qui sont effectués par la suite sont acceptés, car la fonction demande à ce que les pointeurs d'entrée soient valides.

Comme nous le verrons plus tard, `\valid` peut recevoir plus qu'un pointeur en entrée. Par exemple, il est possible de lui transmettre une expression de cette forme : `\valid(p + (s .. e))` qui voudra dire « pour tout `i` entre `s` et `e` (inclus), `p+i` est un pointeur valide », ce sera important notamment pour la gestion des tableaux dans les spécifications.

Si nous nous intéressons aux assertions ajoutées par WP dans la fonction `swap` avec la validation des RTEs, nous pouvons constater qu'il existe une variante de `\valid` sous le nom `\valid_read`. Contrairement au premier, celui-ci assure qu'il est uniquement nécessaire que le pointeur puisse être déréférencé en lecture et pas forcément en écriture, pour pouvoir réaliser l'opération de lecture. Cette subtilité est due au fait qu'en C, le *downcast* de pointeur vers un élément `const` est très facile à faire, mais n'est pas forcément légal.

Typiquement, dans le code suivant :

```
1  /*@ requires \valid(p); */
2  int unref(int* p){
3    return *p;
4  }
5
6  int const value = 42;
7
8  int main(){
9    int i = unref(&value);
```

```
10 }
```

Le déréférencement de `p` est valide, pourtant la précondition de `unref` ne sera pas validée par WP, car le déréférencement de l'adresse de `value` n'est légal qu'en lecture. Un accès en écriture sera un comportement indéterminé. Dans un tel cas, nous pouvons préciser que dans `unref`, le pointeur `p` doit être nécessairement `\valid_read` et pas `\valid`.

3.2.3.3. Effets de bord

Notre fonction `swap` est bien prouvable au regard de sa spécification et de ses potentielles erreurs à l'exécution, mais est-elle pour autant suffisamment spécifiée ? Pour voir cela, nous pouvons modifier légèrement le code de cette façon (nous utilisons `assert` pour analyser des propriétés ponctuelles) :

```
1  int h = 42;
2
3  /*@
4    requires \valid(a) && \valid(b);
5    ensures  *a == \old(*b) && *b == \old(*a);
6  */
7  void swap(int* a, int* b){
8    int tmp = *a;
9    *a = *b;
10   *b = tmp;
11 }
12
13 int main(){
14   int a = 37;
15   int b = 91;
16
17   //@ assert h == 42;
18   swap(&a, &b);
19   //@ assert h == 42;
20 }
```

Le résultat n'est pas vraiment celui escompté :

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

En effet, nous n'avons pas spécifié les effets de bord autorisés pour notre fonction. Pour cela, nous utilisons la clause `assigns` qui fait partie des postconditions de la fonction. Elle nous permet de spécifier quels éléments **non locaux** (on vérifie bien des effets de bord), sont susceptibles d'être modifiés par la fonction.

Par défaut, WP considère qu'une fonction a le droit de modifier n'importe quel élément en mémoire. Nous devons donc préciser ce qu'une fonction est en droit de modifier. Par exemple

pour notre fonction `swap`, nous pouvons spécifier que seules les valeurs pointées par les pointeurs reçus peuvent être modifiées :

```
/*@
  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;
}
```

Si nous rejouons la preuve avec cette spécification, la fonction et les assertions que nous avions demandées dans le `main` seront validées par WP.

Finalement, il peut arriver que nous voulions spécifier qu'une fonction ne provoque pas d'effets de bord. Ce cas est précisé en donnant `\nothing` à `assigns` :

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

  assigns \nothing ;

  ensures \result == *a + 5 ;
*/
int plus_5(int* a){
  return *a + 5 ;
}
```

Le lecteur pourra maintenant reprendre les exemples précédents pour y intégrer la bonne clause `assigns`.

3.2.3.4. Séparation des zones de la mémoire

Les pointeurs apportent le risque d'*aliasing* (plusieurs pointeurs ayant accès à la même zone de mémoire). Si dans certaines fonctions, cela ne pose pas de problème (par exemple si nous passons deux pointeurs égaux à notre fonction `swap`, la spécification est toujours vérifiée par le code source), dans d'autre cas, ce n'est pas si 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;
}
```

Si nous demandons à WP de prouver cette fonction (ignorons la vérification de l'absence de RTEs pour cet exemple), nous obtenons le résultat suivant :

```
/*@ 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;
}
```

La raison est simplement que rien ne garantit que le pointeur `a` est bien différent du pointeur `b`. Or, si les pointeurs sont égaux,

— la propriété `*a == \old(*a) + *b` signifie en fait `*a == \old(*a) + *a`, ce qui ne peut être vrai que si l'ancienne valeur pointée par `a` était 0, ce qu'on ne sait pas,
— la propriété `*b == \old(*b)` n'est pas validée, car potentiellement, nous la modifions.

? Pourquoi la clause `assigns` est-elle validée ?
C'est simplement dû au fait, qu'il n'y a bien que la zone mémoire pointée par `a` qui est modifiée étant donné que si `a != b` nous ne modifions bien que cette zone et que si `a == b`, il n'y a toujours que cette zone, et pas une autre.

Pour assurer que les pointeurs sont bien sur des zones séparées de mémoire, ACSL nous offre le prédicat `\separated(p1, ..., pn)` qui reçoit en entrée un certain nombre de pointeurs et qui nous assurera qu'ils sont deux à deux disjoints. Ici, nous spécifierions :

```
1  #include <limits.h>
2
3  /*@
4    requires \valid(a) && \valid_read(b);
5    requires \separated(a, b);
6    assigns  *a;
7    ensures  *a == \old(*a)+ *b;
8    ensures  *b == \old(*b);
9  */
10 void incr_a_by_b(int* a, int const* b){
11   *a += *b;
12 }
```

Et cette fois, la preuve est effectuée :

```
/*@ 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;
}
```

Nous pouvons noter que nous ne nous intéressons pas ici à la preuve de l'absence d'erreur à l'exécution, car ce n'est pas l'objet de cette section. Cependant, si cette fonction faisait partie d'un programme complet à vérifier, il faudrait définir le contexte dans lequel on souhaite l'utiliser et définir les préconditions qui nous garantissent l'absence de débordement en conséquence.

3.2.4. Écrire le bon contrat

Trouver les bonnes préconditions à une fonction est parfois difficile. Il est intéressant de noter qu'une bonne manière de vérifier qu'une spécification est suffisamment précise est d'écrire des tests pour voir si le contrat nous permet, depuis un code appelant, de déduire des propriétés intéressantes. En fait, c'est exactement ce que nous avons fait pour nos exemples `max` et `swap`. Nous avons écrit une première version de notre spécification et du code appelant qui nous a servi à déterminer si nous pouvions prouver des propriétés que nous estimions devoir être capables de prouver à l'aide du contrat.

Le plus important est avant tout de déterminer le contrat sans prendre en compte le contenu de la fonction (au moins dans un premier temps). En effet, nous essayons de prouver une fonction, mais elle pourrait contenir un bug, donc si nous suivons de trop près le code de la fonction, nous risquons d'introduire dans la spécification le même bug présent dans le code, par exemple en prenant en compte une condition erronée. C'est pour cela que l'on souhaitera généralement que la personne qui développe le programme et la personne qui le spécifie formellement soient différentes (même si elles ont pu préalablement s'accorder sur une spécification textuelle par exemple).

Une fois que le contrat est posé, alors seulement, nous nous intéressons aux spécifications dues au fait que nous sommes soumis aux contraintes de notre langage et notre matériel. Cela concerne principalement nos préconditions. Par exemple, la fonction valeur absolue n'a, au fond, pas vraiment de précondition à satisfaire : c'est la machine cible qui détermine qu'une condition supplémentaire doit être vérifiée en raison du complément à deux. Comme nous le verrons dans le chapitre 7, vérifier l'absence de runtime-errors peut aussi impacter nos postconditions, pour l'instant laissons cela de côté.

3.2.5. Exercices

3.2.5.1. Division et reste

Spécifier la postcondition de la fonction suivante, qui calcule le résultat de la division de `a` par `b` et le reste de cette division et écrit ces deux valeurs à deux positions mémoire `p` et `q` :

```
1 void div_rem(unsigned x, unsigned y, unsigned* q, unsigned* r){
2   *q = x / y ;
3   *r = x % y ;
4 }
```

Lancer la commande :

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

Une fois que la fonction est prouvée, lancer :

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

Si cela échoue, compléter le contrat en ajoutant la bonne précondition.

3.2.5.2. Remettre à zéro selon une condition

Donner un contrat à la fonction suivante qui remet à zéro la valeur pointée par le premier paramètre si et seulement si celle pointée par le second est vraie. Ne pas oublier d'exprimer que la valeur pointée par le second paramètre doit rester la même :

```
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 ;
}
```

Lancer la commande :

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

3.2.5.3. Addition de valeurs pointées

La fonction suivante reçoit deux pointeurs en entrée et retourne la somme des valeurs pointées. Écrire le contrat de cette fonction :

```
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 ;
}
```

Lancer la commande :

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

Une fois que la fonction et son code appelant sont prouvées, modifier la signature de la fonction comme suit :

```
void add(int* a, int* b, int* r);
```

Le résultat doit maintenant être stocké à la position mémoire `r`. Modifier l'appel dans la fonction `main` et le code de la fonction de façon à implémenter ce comportement. Modifier le contrat de la fonction `add` et recommencer la preuve. `*a` et `*b` devraient rester inchangés.

3.2.5.4. Maximum de valeurs pointées

Le code suivant calcule le maximum des valeurs pointées par `a` et `b`. Écrire le contrat de cette fonction :

```
int max_ptr(int* a, int* b){
  return (*a < *b) ? *b : *a ;
}

extern int h ;

int main(){
  h = 42 ;

  int a = 24 ;
  int b = 42 ;

  int x = max_ptr(&a, &b) ;

  //@ assert x == 42 ;
  //@ assert h == 42 ;
}
```

Lancer la commande :

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

Une fois que la fonction est prouvée, modifier la signature de la fonction comme suit :

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

La fonction doit maintenant s'assurer qu'après l'exécution, `*a` contient le maximum des valeurs pointées et `*b` contient l'autre valeur. Modifier le code de façon à assurer cela ainsi que le contrat. Notons que la variable `x` n'est plus nécessaire dans la fonction `main` et que nous pouvons changer l'assertion en ligne 15 pour mettre en lumière le nouveau comportement de la fonction.

3.2.5.5. Ordonner trois valeurs

La fonction suivante doit ordonner trois valeurs reçues en entrée dans l'ordre croissant. Écrire le code correspondant et la spécification de la fonction :

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

Et lancer la commande :

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

Il faut bien garder en tête qu'ordonner des valeurs ne consiste pas seulement à s'assurer qu'elles sont dans l'ordre croissant et que chaque valeur doit être l'une de celles d'origine. Toutes les valeurs d'origine doivent être présente et en même quantité. Pour exprimer cette idée, nous pouvons nous reposer à nouveau sur les ensembles. La propriété suivante est vraie par exemple :

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

Nous pouvons l'utiliser pour exprimer que l'ensemble des valeurs d'entrée et de sortie est le même. Cependant, ce n'est pas la seule chose à prendre en compte, car un ensemble ne contient qu'une occurrence de chaque valeur. Donc, si `*a == *b == 1`, alors `{ *a, *b } == { 1 }`.
Par conséquent, nous devons considérer trois autres cas particuliers :

- toutes les valeurs d'origine sont les mêmes ;
- deux valeurs d'origine sont les mêmes, la dernière est plus grande ;
- deux valeurs d'origine sont les mêmes, la dernière est plus petite.

Qui nous permet d'ajouter la bonne contrainte aux valeurs de sortie.

Pour la réalisation de la spécification, le programme de test suivant peut nous aider :

```
  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 ;
}

```

Si la spécification est suffisamment précise, chaque assertion devrait être prouvée. Cependant, cela ne signifie pas que tous les cas ont été considérés, il ne faut pas hésiter à ajouter d'autres tests.

3.3. Comportements

Il peut arriver qu'une fonction ait divers comportements potentiellement très différents en fonction de l'entrée. Un cas typique est la réception d'un pointeur vers une ressource optionnelle : si le pointeur est `NULL`, nous aurons un certain comportement et un comportement complètement différent s'il ne l'est pas.

Nous avons déjà vu une fonction qui avait des comportements différents, la fonction `abs`. Nous reprendrons comme exemple. Les deux comportements que nous pouvons isoler sont le cas où la valeur est positive et le cas où la valeur est négative.

Les comportements nous servent à spécifier les différents cas pour les postconditions. Nous les introduisons avec le mot-clé `behavior`. Chaque comportement a un nom. Pour un comportement donné, nous trouvons différentes hypothèses à propos de l'entrée de la fonction, elles sont introduites à l'aide du mot-clé `assumes` (notons que, puisqu'elles caractérisent les entrées, le mot-clé `\old` ne peut pas être utilisé ici). Cependant, chaque propriété exprimée par ces clauses n'a pas **besoin** d'être vérifiée avant à l'appel, elle **peut** être vérifiée et dans ce cas, les postconditions associées à ce comportement s'appliquent. Ces postconditions sont à nouveau introduites à l'aide du mot clé `ensures`. Finalement, nous pouvons également demander à WP de vérifier le fait que les comportements sont disjoints (pour garantir le déterminisme) et complets (pour garantir que nous couvrons toutes les entrées possibles).

Les comportements sont disjoints si pour toute entrée de la fonction, elle ne correspond aux hypothèses (*assumes*) que d'un seul comportement. Les comportements sont complets si les hypothèses recouvrent bien tout le domaine des entrées.

Par exemple pour `abs` :

```
#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;
}
```

Notons qu'introduire des comportements ne nous interdit pas de spécifier une postcondition globale. Par exemple ici, nous avons spécifié que quel que soit le comportement, la fonction doit retourner une valeur positive.

Pour comprendre ce que font précisément `complete` et `disjoint`, il est utile d'expérimenter deux possibilités :

— remplacer l'hypothèse de « pos » par `val > 0` auquel cas les comportements seront disjoints, mais incomplets (il nous manquera le cas `val == 0`) ;
— remplacer l'hypothèse de « neg » par `val <= 0` auquel cas les comportements seront complets, mais non disjoints (le cas `val == 0`) sera présent dans les deux comportements.

Même si `assigns` est une postcondition, à ma connaissance, il n'est pas possible de mettre des `assigns` pour chaque *behavior*. Si nous avons besoin d'un tel cas, nous spécifions :

— `assigns` avant les *behavior* (comme dans notre exemple) avec tout élément non-local susceptible d'être modifié,
— en postcondition de chaque *behavior* les éléments qui ne sont finalement pas modifiés en les indiquant égaux à leur ancienne (`\old`) valeur.

Les comportements sont très utiles pour simplifier l'écriture de spécifications quand les fonctions ont des effets très différents en fonction de leurs entrées. Sans eux, les spécifications passent systématiquement par des implications traduisant la même idée, mais dont l'écriture et la lecture sont plus difficiles (nous sommes susceptibles d'introduire des erreurs). D'autre part, la traduction de la complétude et de la disjonction devraient être écrites manuellement, ce qui serait fastidieux et une nouvelle fois source d'erreurs.

3.3.1. Exercices

3.3.1.1. Exercices précédents

Dans les sections précédentes, reprendre les exemples :

— à propos du calcul de la distance entre deux entiers ;
— « Remettre à zéro selon une condition » ;
— « Jours du mois » ;
— « Maximum des valeurs pointées ».

En considérant que les contrats étaient :

```
#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 ;
  }
}
```

Les réécrire en utilisant des comportements.

3.3.1.2. Deux autres exercices simples

Produire le code et la spécification des deux fonctions suivantes puis les prouver. La spécification devrait faire usage des comportements.

Tout d'abord une fonction qui retourne si un caractère est une voyelle ou une consonne, supposer (et exprimer) que la fonction reçoit une lettre minuscule.

```
enum Kind { VOWEL, CONSONANT };

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

Puis une fonction qui renvoie à quel quadrant d'un repère appartient une coordonnée. Lorsque la coordonnée se trouve sur un axe, choisir arbitrairement l'un des quadrants qu'elle touche.

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

3.3.1.3. Triangle

Compléter les fonctions suivantes qui reçoivent la longueur des différents côtés et retournent respectivement :

— si le triangle est scalène, isocèle, ou équilatéral ;
— si le triangle est rectangle, acutangle ou obtusangle.

```
#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){
  //
}
```

En supposant (et exprimant) que :

— les valeurs reçues forment bien un triangle,
— `a` est l'hypoténuse du triangle,

spécifier et prouver que les fonctions font la tâche prévue.

3.3.1.4. Maximum des valeurs pointées

Reprendre l'exemple « Maximum des valeurs pointées » de la section précédente et plus précisément la version qui retourne la plus grande valeur. En considérant que le contrat était :

```
/*@
  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. Le réécrire en utilisant des comportements
2. Modifier le contrat de 1 de sorte que les comportements ne soient pas disjoints. Excepté cette propriété, tout le reste devrait être correctement prouvé
3. Modifier le contrat de 1 de sorte que les comportements ne soient pas complets, puis ajouter un nouveau comportement pour le rendre de nouveau complet
4. Modifier la fonction de 1 de façon à accepter la valeur `NULL` pour les pointeurs d'entrées, si les deux pointeurs sont nuls, retourner `INT_MIN`, si l'un seulement est nul, retourner l'autre valeur, sinon retourner le maximum des deux valeurs. Modifier le contrat de façon à prendre en compte tout cela par de nouveaux comportements. Prendre soin d'assurer que les comportements sont complets et disjoints.

3.3.1.5. Ordonner trois valeurs

Reprendre l'exemple « Ordonner trois valeurs » de la section précédente, en considérant que le contrat était :

```
/*@
  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 ; }
}
```

Le réécrire en utilisant des comportements. Notons que le contrat devrait être composé d'un comportement général et de deux comportements spécifiques. Est-ce que ces comportements sont complets ? Sont-ils disjoints ?

3.4. Modularité du WP

Pour terminer cette partie nous allons parler de la composition des appels de fonctions et commencer à entrer dans les détails de fonctionnement de WP. Nous en profiterons pour regarder comment se traduit le découpage de nos programmes en fichiers lorsque nous voulons les spécifier et les prouver avec WP.

Notre but sera de prouver la fonction `max_abs` qui renvoie les maximums entre les valeurs absolues de deux valeurs :

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

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

Commençons par (sur-)découper les déclarations et définitions des différentes fonctions dont nous avons besoin (et que nous avons déjà prouvé) en couples *headers*/source, à savoir `abs` et `max`. Cela donne pour `abs` :

Fichier 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
```

Fichier abs.c :

```
#include "abs.h"

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

Nous découpons en mettant le contrat de la fonction dans le *header*. Le but est de pouvoir importer la spécification en même temps que la déclaration celle-ci lorsque nous aurons besoin de la fonction dans un autre fichier. En effet, WP en aura besoin pour montrer que les appels à cette fonction sont valides. D'abord pour prouver que la précondition est satisfaite (et donc que l'appel est légal) et ensuite pour savoir ce qu'il peut apprendre en retour (à savoir la postcondition) afin de pouvoir l'utiliser pour prouver la fonction appelante.

Nous pouvons créer un fichier sous le même formatage pour la fonction `max`. Dans les deux cas, nous pouvons rouvrir le fichier source (pas besoin de spécifier les fichiers *headers* dans la ligne de commande) avec Frama-C et remarquer que la spécification est bien associée à la fonction et que nous pouvons la prouver.

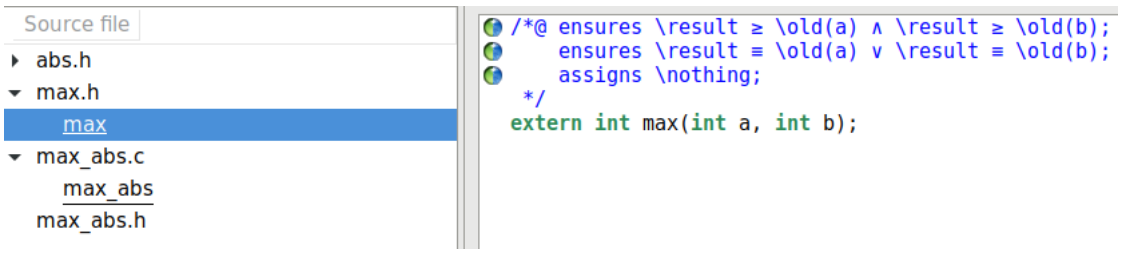
Maintenant, nous pouvons préparer le terrain pour la fonction `max_abs` dans notre *header* :

```
#ifndef _MAX_ABS
#define _MAX_ABS

int max_abs(int a, int b);

#endif
```

et dans le fichier source :

```
#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);
}
```

Et ouvrir ce dernier fichier dans Frama-C. Si nous regardons le panneau latéral, nous pouvons voir que les fichiers *header* que nous avons inclus dans le fichier `abs_max.c` y apparaissent et que les contrats de fonction sont décorés avec des pastilles particulières (vertes et bleues) :



Ces pastilles nous disent qu'en l'absence d'implémentation, les propriétés sont supposées vraies. Et c'est une des forces de la preuve déductive de programmes par rapport à certaines autres méthodes formelles : les fonctions sont vérifiées en isolation les unes des autres.

En dehors de la fonction, sa spécification est considérée comme étant vérifiée : nous ne cherchons pas à reprouver que la fonction fait bien son travail à chaque appel, nous nous contenterons de

vérifier que les préconditions sont réunies au moment de l'appel. Cela donne donc des preuves très modulaires et donc des spécifications plus facilement réutilisables. Évidemment, si notre preuve repose sur la spécification d'une autre fonction, cette fonction doit-elle même être vérifiable pour que la preuve soit formellement complète. Mais nous pouvons également vouloir simplement faire confiance à une bibliothèque externe sans la prouver.

Finalement, le lecteur pourra essayer de spécifier la fonction `max_abs`.

La spécification peut ressembler à ceci :

```
/*@
  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. Exercices

3.4.1.1. Jours du mois

Spécifier la fonction année bissextile qui retourne vrai si l'année reçue en entrée est bissextile. Utiliser cette fonction pour compéter la fonction jours du mois de façon à retourner le nombre de jours du mois reçu en entrée, incluant le bon comportement lorsque le mois en question est février et que l'année est bissextile.

```
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. Caractères alpha-numériques

Écrire et spécifier les différentes fonctions utilisées par `is_alpha_num`. Fournir un contrat pour chacune d'elles et fournir le contrat de `is_alpha_num`.

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

```
    is_digit(c) ;
}
```

Déclarer une énumération avec les valeurs `LOWER`, `UPPER`, `DIGIT` et `OTHER`, et une fonction `character_kind` qui retourne, en utilisant les différentes fonctions `is_lower`, `is_upper` et `is_digit`, la sorte de caractère reçue en entrée. Utiliser les comportements pour spécifier le contrat de cette fonction en s'assurant qu'ils sont complets et disjoints.

3.4.1.3. Ordonner trois valeurs

Reprendre la fonction `max_ptr` dans sa version qui « ordonne » les deux valeurs. Écrire une fonction `min_ptr` qui utilise la fonction précédente pour effectuer l'opération inverse. Utiliser ces fonctions pour compléter les quatre fonctions qui ordonnent trois valeurs. Pour chaque variante (ordre croissant et décroissant), l'écrire une première fois en utilisant uniquement `max_ptr` et une seconde en utilisant `min_ptr`. Écrire un contrat précis pour chacune de ces fonctions et les prouver.

```
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. Rendre la monnaie

Le but de cet exercice est d'écrire une fonction de rendu de monnaie. La fonction `make_change` reçoit la valeur due, la quantité d'argent reçue et un *buffer* pour indiquer quelle quantité de chaque billet/pièce doit être retournée au client.

Par exemple, pour une valeur due de 410 et une valeur reçue de 500, le tableau devrait contenir 1 dans la cellule `change[N50]` et 2 dans la cellule `change[N20]` après l'appel à la fonction.

Si le montant reçu est inférieur au prix, la fonction devrait retourner -1 (et 0 si ce n'est pas le cas).

```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;
}
```

La fonction `remove_max_notes` reçoit une valeur de pièce ou billet et ce qu'il reste à convertir (via un pointeur), supposé être supérieur à 0. Elle calcule le nombre maximal de billets ou pièces de cette valeur pouvant tenir dans le restant, diminue la valeur du restant conformément et retourne ce nombre. La fonction `make_change` doit ensuite faire usage de cette fonction pour calculer le rendu de monnaie.

Écrire le code de ces fonctions et leur spécification puis prouver la correction. Notons que le code ne devrait pas faire usage de boucles puisque nous ne savons pas encore les traiter.

3.4.1.5. Triangle

Dans cet exercice, nous voulons rassembler les résultats des fonctions que nous avons écrites dans la section précédente pour obtenir les propriétés de triangles dans une structure. La fonction `classify` reçoit trois longueurs `a`, `b`, et `c`, en supposant que `a` est l'hypoténuse. Si ces valeurs ne correspondent pas à un triangle, la fonction retourne -1, et 0 si tout est OK. Les propriétés sont collectées dans une structure `info` reçue via un pointeur.

```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){
```

```
20    // ...
21 }
```

Écrire, spécifier et prouver toutes les fonctions.

Notons qu'il y a beaucoup de comportements à lister et spécifier. Écrire une version qui ne requiert pas que `a` soit l'hypoténuse est possible. Par contre, il pourrait être difficile de terminer la preuve automatiquement avec Alt-Ergo parce qu'il y a vraiment beaucoup de combinaisons à considérer.

Pendant cette partie, nous avons vu comment spécifier les fonctions par l'intermédiaire de leurs contrats, à savoir leurs pré et postconditions, ainsi que quelques fonctionnalités offertes par ACSL pour exprimer ces propriétés. Nous avons également vu pourquoi il est important d'être précis dans la spécification et comment l'introduction des comportements nous permet d'avoir des spécifications plus compréhensibles et moins sujettes aux erreurs.

En revanche, nous n'avons pas encore vu un point important : la spécification des boucles. Avant d'entamer cette partie, nous devrions regarder plus précisément comment fonctionne l'outil WP.

4. Instructions basiques et structures de contrôle

Cette partie est plus formelle que ce que nous avons vu jusqu'à maintenant. Si le lecteur souhaite se concentrer sur l'utilisation de l'outil, l'introduction de ce chapitre et les deux premières sections (sur les instructions de base et « le bonus stage ») sont dispensables. Si ce que nous avons présenté jusqu'à maintenant a semblé ardu au lecteur sur un plan formel, il est également possible de réserver l'introduction et ces deux sections pour une deuxième lecture.
Les sections sur les boucles sont en revanches indispensables. Les éléments plus formels de ces sections seront signalés.

Pour chaque notion en programmation C, nous associerons la règle d'inférence qui lui correspond, la règle utilisée de calcul de plus faible précondition qui la régit et des exemples d'utilisation. Pas forcément dans cet ordre et avec plus ou moins de liaison avec l'outil. Les premiers points seront plus focalisés sur la théorie que sur l'utilisation, car ce sont les plus simples, au fur et à mesure, nous nous concentrerons de plus en plus sur l'outil, en particulier quand nous attaquerons le point concernant les boucles.

4.0.1. Règle d'inférence

Une règle d'inférence est de la forme :

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

et signifie que pour assurer que la conclusion C est vraie, il faut d'abord savoir que les prémisses P_1, ..., et P_n sont vraies. Quand il n'y a pas de prémisses :

$$\frac{}{C}$$

Alors, il n'y a rien à assurer pour conclure que C est vraie.

Inversement, pour prouver qu'une certaine prémisse est vraie, il peut être nécessaire d'utiliser une autre règle d'inférence, ce qui nous donnerait quelque chose comme :

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

Ce qui nous construit progressivement l'arbre de déduction de notre raisonnement. Dans notre raisonnement, les prémisses et conclusions manipulées seront généralement des triplets de Hoare.

4.0.2. Triplet de Hoare

Revenons sur la notion de triplet de Hoare :

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

Nous l'avons vu en début de tutoriel, ce triplet nous exprime que si avant l'exécution de C, la propriété P est vraie, et si C termine, alors la propriété Q est vraie. Par exemple, si nous reprenons notre programme de calcul de la valeur absolue (légèrement modifié) :

```
/*@
  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;
}
```

Ce que nous dit Hoare, est que pour prouver notre programme, les propriétés entre accolades dans ce programme doivent être vérifiées (j'ai omis une des deux postconditions pour alléger la lecture) :

```
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;
}
```

Cependant, Hoare ne nous dit pas comment nous pouvons obtenir automatiquement la propriété `P` de ce programme. Ce que nous propose Dijkstra, c'est donc un moyen de calculer, à partir d'une postcondition Q et d'une commande ou d'une liste de commandes C, la précondition minimale assurant Q après C. Nous pourrions donc, dans le programme précédent, calculer la propriété `P` qui nous donne les garanties voulues.

Nous présenterons tout au long de cette partie les différents cas de la fonction wp qui, à une postcondition voulue et un programme ou une instruction, nous associe la plus faible précondition qui permet de l'assurer. Nous utiliserons cette notation pour définir le calcul correspondant à une ou plusieurs instructions :

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

De plus la fonction wp est telle qu'elle nous garantit que le triplet de Hoare :

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

est effectivement un triplet valide.

Nous utiliserons souvent des assertions ACSL pour présenter les notions à venir :

```
1 //@ assert ma_propriete ;
```

Ces assertions correspondent en fait à des étapes intermédiaires possibles pour les propriétés indiquées dans nos triplets de Hoare. Nous pouvons par exemple reprendre le programme précédent et remplacer nos commentaires par les assertions ACSL correspondantes (nous omettons `P` car sa valeur est en fait simplement « vrai ») :

```
1  int abs(int val){
2    int res;
3    if(val < 0){
4      //@ assert val < 0 ;
5      res = - val;
6      //@ assert \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val ;
7    } else {
8      //@ assert !(val < 0) ;
9      res = val;
10     //@ assert \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val ;
11   }
12   //@ assert \at(val, Pre) >= 0 ==> res == val && \at(val, Pre) < 0 ==> res == -val ;
13
14   return res;
15 }
```

4.1. Concepts de base

4.1.1. Affectation

L'affectation est l'opération la plus basique que l'on puisse avoir dans un langage (mise à part l'opération « ne rien faire » qui manque singulièrement d'intérêt). Le calcul de plus faible précondition associée est le suivant :

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

où la notation $P[x \leftarrow E]$ signifie « la propriété P où x est remplacé par E ». Ce qui correspond ici à « la postcondition $Post$ où x a été remplacé par E ». Dans l'idée, pour que la formule en

postcondition d'une affectation de x à E soit vraie, il faut qu'en remplaçant chaque occurrence de x dans la formule par E, on obtienne une propriété qui est vraie. Par exemple :

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

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

La fonction wp nous permet donc de calculer la plus faible précondition de l'opération ($\{43*c = 258\}$), ce que l'on peut réécrire sous la forme d'un triplet de Hoare :

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

Pour calculer la précondition de l'affectation, nous avons remplacé chaque occurrence de x dans la postcondition, par la valeur $E = 43 * c$ affectée. Si notre programme était de la forme :

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

nous pourrions alors fournir la formule « $43*6 = 258$ » à notre prouveur automatique afin qu'il détermine si cette formule peut effectivement être satisfaite. Ce à quoi il répondrait évidemment « oui », puisque cette propriété est très simple à vérifier. En revanche, si nous avions donné la valeur 7 pour `c`, le prouveur nous répondrait que non, une telle formule n'est pas vraie.

Nous pouvons donc écrire la règle d'inférence pour le triplet de Hoare de l'affectation, où l'on prend en compte le calcul de plus faible précondition :

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

Nous noterons qu'il n'y a pas de prémisse à vérifier. Cela veut-il dire que le triplet est nécessairement vrai ? Oui. Mais cela ne dit pas si la précondition est satisfaite par le programme où se trouve l'instruction, ni que cette précondition est possible. C'est ce travail qu'effectuent ensuite les prouveurs automatiques.

Par exemple, nous pouvons demander la vérification de la ligne suivante avec Frama-C :

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

Ce qui est, bien entendu, prouvé directement par Qed car c'est une simple application de la règle de l'affectation.

Notons que d'après la norme C, l'opération d'affectation est une expression et non une instruction. C'est ce qui nous permet par exemple d'écrire `if( (a = foo()) == 42)`. Dans Frama-C, une affectation sera toujours une instruction. En effet, si une affectation est présente au sein d'une expression plus complexe, le module de création de l'arbre de syntaxe abstraite du programme analysé effectue une étape de normalisation qui crée systématiquement une instruction séparée.

4.1.1.1. Affectation de valeurs pointées

En C, grâce aux (à cause des ?) pointeurs, nous pouvons avoir des programmes avec des alias, à savoir que deux pointeurs peuvent pointer vers la même position en mémoire. Notre calcul de plus faible précondition doit donc considérer ce genre de cas. Par exemple, nous pouvons regarder ce triplet de Hoare :

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

Ce triplet de Hoare est correct, puisque `p` et `q` sont en alias, modifier la valeur `*p` modifie aussi la valeur `*q`, par conséquent, ces deux expressions s'évaluent à 1 et la postcondition est vraie. Cependant, regardons ce que nous obtenons en appliquant le calcul de plus faible précondition précédemment défini pour l'affectation sur cet exemple :

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

Nous obtenons la plus faible précondition : `1 + *q == 2`, et donc nous pourrions déduire que la plus faible précondition est `*q == 1` (ce qui est vrai), mais nous ne sommes pas en mesure de conclure que le programme est correct, car rien dans notre formule ne nous indique quelque chose comme : `p == q ==> *q == 1`. En fait, ici, nous voudrions être capable de calculer une plus faible précondition de la forme :

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

Pour cela, nous devons faire attention à la notion d'aliasing. Une manière commune de le faire est de considérer que la mémoire est une variable particulière du programme (nommons-la M) sur laquelle nous pouvons effectuer deux opérations : obtenir la valeur d'un emplacement particulier m en mémoire (qui nous retourne une expression) et changer la valeur à une position mémoire l pour y placer une nouvelle valeur v (qui nous retourne la nouvelle mémoire obtenue).

Notons :

— $get(M, l)$ par la notation $M[l]$
— $set(M, l, v)$ par la notation $M[l \mapsto v]$

Ces deux opérations peuvent être définies comme suit :

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

Si aucune valeur n'est associée à la position mémoire envoyée à *get*, la valeur est indéfinie (la mémoire est une fonction partielle). Bien sûr, au début d'une fonction, cette mémoire peut être remplie avec un ensemble de positions mémoire pour lesquelles nous savons que la valeur a été précédemment définie (par exemple parce que la spécification de la fonction nous l'indique).

Maintenant, nous pouvons changer légèrement notre calcul de plus faible précondition pour le cas particulier des affectations à travers un pointeur. Pour cela, nous considérons que nous avons dans le programme une variable implicite M qui modélise la mémoire, et nous définissons l'affectation d'une position en mémoire comme une mise à jour de cette variable, de telle manière à ce que cette position contienne maintenant l'expression fournie lors de l'affectation :

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

Évaluer une valeur pointée $*x$ dans une formule consiste maintenant à utiliser l'opération *get* pour demander la valeur à la mémoire. Nous pouvons donc appliquer notre calcul de plus faible précondition à notre programme précédent :

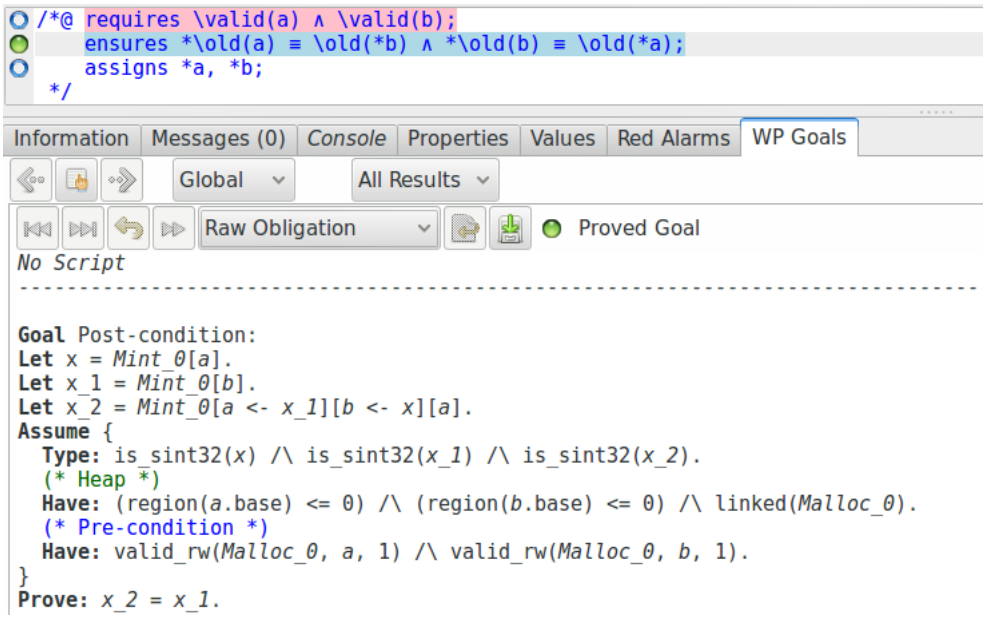
$$\begin{aligned} wp(*p = 1, *p + *q = 2) &= (*p + *q = 2)[M \leftarrow M[p \mapsto 1]] && (1) \\ &= (M[p] + M[q] = 2)[M \leftarrow M[p \mapsto 1]] && (2) \\ &= (M[p \mapsto 1][p] + M[p \mapsto 1][q] = 2) && (3) \\ &= (1 + M[p \mapsto 1][q] = 2) && (4) \\ &= (1 + (\texttt{if } q = p \texttt{ then } 1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

1. nous devons appliquer la règle pour l'affectation de valeur pointée, mais pour cela, nous devons d'abord introduire M,
2. nous remplaçons nos accès de pointeurs par un appel à *get* sur M,
3. nous appliquons le remplacement demandé par la règle d'affectation,
4. nous utilisons la définition de *get* pour p ($M[p \mapsto 1][p] = 1$),
5. nous utilisons la définition de *get* pour q ($M[p \mapsto 1][q] = \texttt{if } q = p \texttt{ then } 1 \texttt{ else } M[q]$)
6. nous appliquons quelques simplifications sur la formule …
7. …et nous pouvons finalement conclure que $M[q] = 1$ ou $p = q$,

et comme dans notre programme, nous savons que $p = q$, nous pouvons conclure que le programme est correct.

Le plugin WP ne fonctionne par exactement comme cela. En particulier, cela dépend du modèle mémoire sélectionné pour réaliser la preuve, qui fait différentes hypothèses à propos de la manière dont la mémoire est organisée. Pour le modèle mémoire que nous utilisons, le modèle

mémoire typé, WP crée différentes variables pour la mémoire. Cela dit, regardons tout de même les conditions de vérification générées par WP pour la postcondition de la fonction `swap` :



Nous pouvons voir, au début de la définition de cette condition de vérification qu'une variable `Mint_0` représentant une mémoire pour les valeurs de type entier a été créée, et que cette mémoire est mise à jour et accédée à l'aide des opérateurs que nous avons définis précédemment (voir par exemple la définition de la variable `x_2`).

4.1.2. Séquence d'instructions

Pour qu'une instruction soit valide, il faut que sa précondition nous permette, par cette instruction, de passer à la postcondition voulue. Maintenant, nous avons besoin d'enchaîner ce processus d'une instruction à une autre. L'idée est alors que la postcondition assurée par la première instruction soit compatible avec la précondition demandée par la deuxième et que ce processus puisse se répéter pour la troisième instruction, etc.

La règle d'inférence correspondant à cette idée, utilisant les triplets de Hoare est la suivante :

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

Pour vérifier que la séquence d'instructions $S1;\ S2$ (NB : où $S1$ et $S2$ peuvent elles-mêmes être des séquences d'instructions), nous passons par une propriété intermédiaire qui est à la fois la précondition de $S2$ et la postcondition de $S1$. Cependant, rien ne nous indique pour l'instant comment obtenir les propriétés P et R.

Le calcul de plus faible précondition wp nous dit simplement que la propriété intermédiaire R est trouvée par calcul de plus faible précondition de la deuxième instruction. Et que la propriété P est trouvée en calculant la plus faible précondition de la première instruction. La plus faible précondition de notre liste d'instruction est donc déterminée comme ceci :

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

Le plugin WP de Frama-C fait ce calcul pour nous, c'est pour cela que nous n'avons pas besoin d'écrire les assertions entre chaque ligne de code.

```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. Arbre de preuve

Notons que lorsque nous avons plus de deux instructions, nous pouvons simplement considérer que la dernière instruction est la seconde instruction de notre règle et que toutes les instructions qui la précèdent forment la première « instruction ». De cette manière, nous remontons bien les instructions une à une dans notre raisonnement, par exemple avec le programme précédent :

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

Nous pouvons par calcul de plus faibles préconditions construire la propriété Q_{-1} à partir de Q et i_3, ce qui nous permet de déduire Q_{-2}, à partir de Q_{-1} et i_2 et finalement P avec Q_{-2} et i_1.

Nous pouvons maintenant vérifier des programmes comprenant plusieurs instructions ; il est temps d'y ajouter un peu de structure.

4.1.3. Règle de la conditionnelle

Pour qu'un branchement conditionnel soit valide, il faut que la postcondition soit atteignable par les deux banches, depuis la même précondition, à ceci près que chacune des branches aura une information supplémentaire : le fait que la condition était vraie dans un cas et fausse dans l'autre.

Comme avec la séquence d'instructions, nous aurons donc deux points à vérifier (pour éviter de confondre les accolades, j'utilise la syntaxe $if\ B\ then\ S1\ else\ S2$) :

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

Nos deux prémisses sont donc la vérification que lorsque nous avons la précondition et que nous passons dans la branche `if`, nous atteignons bien la postcondition, et que lorsque nous avons

la précondition et que nous passons dans la branche `else`, nous obtenons bien également la postcondition.

Le calcul de précondition de wp pour la conditionnelle est le suivant :

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

À savoir que B doit impliquer la précondition la plus faible de $S1$, pour pouvoir l'exécuter sans erreur vers la postcondition, et que $\neg B$ doit impliquer la précondition la plus faible de $S2$ (pour la même raison).

4.1.3.1. Bloc `else` vide

En suivant cette définition, si le `else` ne fait rien, alors la règle d'inférence est de la forme suivante, en remplaçant $S2$ par une instruction « ne rien faire ».

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

Le triplet pour le `else` est :

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

Ce qui veut dire que nous devons avoir :

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

En résumé, si la condition du `if` est fausse, cela veut dire que la postcondition de l'instruction conditionnelle globale est déjà vérifiée avant de rentrer dans le `else` (puisqu'il ne fait rien).

Par exemple, nous pourrions vouloir remettre une configuration c à une valeur par défaut si elle a mal été configurée par un utilisateur du programme :

```
1  int c;
2
3  // ... du code ...
4
5  if(c < 0 || c > 15){
6    c = 0;
7  }
8  //@ assert 0 <= c <= 15;
```

Soit :

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

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

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

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

La formule est bien vérifiable : quelle que soit l'évaluation de $\neg(c \in [0;15])$ l'implication sera vraie.

4.1.4. Bonus Stage - Règle de conséquence

Parfois, il peut être utile pour la preuve de renforcer une postcondition ou d'affaiblir une précondition. Si la première sera souvent établie par nos soins pour faciliter le travail du prouveur, la seconde est plus souvent vérifiée par l'outil à l'issue du calcul de plus faible précondition.

La règle d'inférence en logique de Hoare est la suivante :

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

(Nous noterons que les prémisses, ici, ne sont pas seulement des triplets de Hoare mais également des formules à vérifier)

Par exemple, si notre postcondition est trop complexe, elle risque de générer une plus faible précondition trop compliquée et de rendre le calcul des prouveurs difficile. Nous pouvons alors créer une postcondition intermédiaire SQ, plus simple, mais plus restreinte et impliquant la vraie postcondition. C'est la partie $SQ \Rightarrow Q$.

Inversement, le calcul de précondition générera généralement une formule compliquée et souvent plus faible que la précondition que nous souhaitons accepter en entrée. Dans ce cas, c'est notre outil qui s'occupera de vérifier l'implication entre ce que nous voulons et ce qui est nécessaire pour que notre code soit valide. C'est la partie $P \Rightarrow WP$.

Nous pouvons par exemple illustrer cela avec le code qui suit. Notons bien qu'ici, le code pourrait tout à fait être prouvé par l'intermédiaire de WP sans ajouter des affaiblissements et renforcements de propriétés, car le code est très simple, il s'agit juste d'illustrer la règle de conséquence.

```
1  /*@
2    requires P: 2 <= a <= 8;
3    ensures  Q: 0 <= \result <= 100 ;
4    assigns  \nothing ;
5  */
6  int constrained_times_10(int a){
7    //@ assert P_imply_WP: 2 <= a <= 8 ==> 1 <= a <= 9 ;
8    //@ assert WP:         1 <= a <= 9 ;
9
10   int res = a * 10;
11
12   //@ assert SQ:         10 <= res <= 90 ;
13   //@ assert SQ_imply_Q: 10 <= res <= 90 ==> 0 <= res <= 100 ;
14
15   return res;
16 }
```

(À noter ici : nous avons omis les contrôles de débordement d'entiers).

Ici, nous voulons avoir un résultat compris entre 0 et 100. Mais nous savons que le code ne produira pas un résultat sortant des bornes 10 à 90. Donc nous renforçons la postcondition avec une assertion que `res`, le résultat, est compris entre 0 et 90 à la fin. Le calcul de plus faible précondition, sur cette propriété, et avec l'affectation `res = 10*a` nous produit une plus faible précondition `1 <= a <= 9` et nous savons finalement que `2 <= a <= 8` nous donne cette garantie.

Quand une preuve a du mal à être réalisée sur un code plus complexe, écrire des assertions produisant des postconditions plus fortes, mais qui forment des formules plus simples peut souvent nous aider. Notons que dans le code précédent, les lignes `P_imply_WP` et `SQ_imply_Q` ne sont jamais utiles, car c'est le raisonnement par défaut produit par WP, elles sont juste présentes pour l'illustration.

4.1.5. Bonus Stage - Règle de constance

Certaines séquences d'instructions peuvent concerner et faire intervenir des variables différentes. Ainsi, il peut arriver que nous initialisions et manipulions un certain nombre de variables, que nous commencions à utiliser certaines d'entre elles, puis les délaissions au profit d'autres pendant un temps. Quand un tel cas apparaît, nous avons envie que l'outil ne se préoccupe que des variables qui sont susceptibles d'être modifiées pour avoir des propriétés les plus légères possibles.

La règle d'inférence qui définit ce raisonnement est la suivante :

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

où c ne modifie aucune variable entrant en jeu dans R. Ce qui nous dit : « pour vérifier le triplet, débarrassons-nous des parties de la formule qui concernent des variables qui ne sont pas manipulées par c et prouvons le nouveau triplet ». Cependant, il faut prendre garde à ne pas supprimer trop d'information, au risque de ne plus pouvoir prouver nos propriétés.

Par exemple, nous pouvons imaginer le code suivant (une nouvelle fois, nous omettons les contrôles de débordements au niveau des entiers) :

```
1  /*@
2    requires a > -99 ;
3    requires b > 100 ;
4    ensures  \result > 0 ;
5    assigns  \nothing ;
6  */
7  int foo(int a, int b){
8    if(a >= 0){
9      a++ ;
10   } else {
11     a += b ;
12   }
13   return a ;
14 }
```

Si nous regardons le code du bloc `if`, il ne fait pas intervenir la variable `b`, donc nous pouvons omettre complètement les propriétés à propos de `b` pour réaliser la preuve que `a` sera bien supérieur à 0 après l'exécution du bloc :

```
1  /*@
2    requires a > -99 ;
3    requires b > 100 ;
4    ensures  \result > 0 ;
5    assigns  \nothing ;
6  */
```

```
7  int foo(int a, int b){
8    if(a >= 0){
9      //@ assert a >= 0; //et rien à propos de b
10     a++ ;
11   } else {
12     a += b ;
13   }
14   return a ;
15 }
```

En revanche, dans le bloc `else`, même si `b` n'est pas modifiée, établir des propriétés seulement à propos de `a` rendrait notre preuve impossible (en tant qu'humains). Le code serait :

```
1  /*@
2    requires a > -99 ;
3    requires b > 100 ;
4    ensures  \result > 0 ;
5    assigns  \nothing ;
6  */
7  int foo(int a, int b){
8    if(a >= 0){
9      //@ assert a >= 0; // et rien à propos de b
10     a++ ;
11   } else {
12     //@ assert a < 0 && a > -99 ; // et rien à propos de b
13     a += b ;
14   }
15   return a ;
16 }
```

Dans le bloc `else`, n'ayant que connaissance du fait que `a` est compris entre -99 et 0, et ne sachant rien à propos de `b`, nous pourrions difficilement savoir si le calcul `a += b` produit une valeur supérieure stricte à 0 pour `a`.

Naturellement ici, WP prouvera la fonction sans problème, puisqu'il transporte de lui-même les propriétés qui lui sont nécessaires pour la preuve. En fait, l'analyse des variables qui sont nécessaires ou non (et l'application, par conséquent de la règle de constance) est réalisée directement par WP.

Notons finalement que la règle de constance est une instance de la règle de conséquence :

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

Si les variables de R n'ont pas été modifiées par l'opération (qui par contre, modifie les variables de P pour former Q), alors effectivement $P \wedge R \Rightarrow P$ et $Q \Rightarrow Q \wedge R$.

4.1.6. Assertion

Les assertions ACSL doivent être prouvées (et utilisées). Nous avons donc besoin de règle dans le calcul de plus faible précondition. Il y a plusieurs types d'assertions en ACSL :

— `assert`
— `check`

— `admit`

Chacune d'elles à un comportement différent.

Commençons par celle que nous avons utilisée jusqu'à maintenant : `assert`. Une annotation `assert P` a la sémantique informelle suivante : la propriété `P` doit être vérifiée au point de programme, et ensuite le fait qu'elle soit vraie est ajouté comme hypothèse pour prouver les propriétés qui apparaissent plus tard dans le programme. Notons qu'elle pourrait ne pas être vraie, mais elle sera de toute façon ajoutée dans le contexte de preuve. En revanche, si elle n'est pas vraie, d'abord nous ne serons pas en mesure de la prouver et ensuite, les propriétés prouvées en l'admettant vraie seront marquée comme « *valid under hypothesis* ».

L'annotation `check P` signifie que la propriété `P` doit être vérifiée à ce point de programme, mais elle ne sera pas ajoutée au contexte de preuve pour les annotations qui la suivent.

Finalement, l'annotation `admit P` signifie qu'à partir de ce point de programme, la propriété `P` est considérée comme vraie et ajoutée au contexte de preuve pour annotations qui la suivent, même si nous ne la vérifions pas à ce point de programme.

Illustrons ces différents types d'annotations avec ce court exemple :

```
1  //@ ensures \false ;
2  void check_annot(void){
3    //@ check \false ;
4  }
5
6  //@ ensures \false ;
7  void admit_annot(void){
8    //@ admit \false ;
9  }
10
11 //@ ensures \false ;
12 void assert_annot(void){
13   //@ assert \false ;
14 }
```

```
/*@ 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;
}
```

Dans la fonction `check_annot`, ni le `check` ni la clause `ensures` ne sont prouvées. Les preuves ont été essayées séparément (et ont échoué).

Dans la fonction `admit_annot`, l'annotation `admit` apparaît bleue et verte (exactement comme dans les contrats de fonction sans corps), ce qui signifie qu'à partir de ce point, elle est considérée comme valide même si elle n'est pas prouvée. Par conséquent, la clause `ensures` est prouvée puisque `\false` est supposée vrai.

Finalement dans la fonction `assert_annot`, l'annotation `assert` n'est pas prouvée, mais à partir de ce point elle est supposée vraie. Par conséquent, la clause `ensures` est prouvée. En revanche, tandis que dans la fonction `admit_annot` elle apparaît entièrement prouvée (affichage vert), ici elle apparaît comme valide sous hypothèse : la validité dépend d'une propriété qui n'a pas encore été prouvée (et ne le sera pas).

Nous pouvons encoder ces différents comportements avec les règles de WP qui suivent.

La règle pour l'annotation `check` est la suivante :

$$wp(check\ A, Post) := A \wedge Post$$

qui signifie que nous ajoutons aux propriétés que nous devons vérifier une nouvelle propriété, qui est la propriété A.

La règle pour l'annotation `admit` est la suivante :

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

qui signifie que sous la condition A, nous devons vérifier la postcondition (de l'instruction, qui peut être une propriété calculée grâce à d'autres règles de WP).

Finalement, la règle pour l'annotation `assert` combine les deux précédentes. En effet :

```
1  assert P;
```

est équivalent à :

```
1  check P; // d'abord, vérifier que P est vraie
2  admit P; // puis le supposer
```

Donc :

$$wp(assert\ A, Post) := wp(check\ A, wp(admit\ A, Post)) \equiv A \wedge (A \Rightarrow Post)$$

Cela permet d'introduire une coupure dans la preuve. On prouve d'abord A et une fois que c'est fait, nous prouvons que lorsque A est vrai, $Post$ l'est aussi.

Il serait raisonnable de penser que l'annotation `admit` est dangereuse. Elle l'est. Mais elle peut être utile pour déboguer une preuve. En particulier, nous verrons dans le chapitre 7 que l'annotation `assert` peut être utilisée pour guider une preuve. Pendant le processus, l'annotation `admit` peut être utile pour essayer des annotations sans pour autant devoir les prouver immédiatement, ou pour considérer que certaines annotations sont valides pour accélérer la phase de design facile. Plus rarement, elle peut être utile pour expliciter des hypothèses à propos du matériel, ou de la plateforme logicielle, mais pour cela, nous utilisons plus souvent les options du noyau de Frama-C ou des axiomes (nous parlerons des axiomes dans la section 6.2).

4.1.7. WP plugin vs. WP calculus de Dijkstra

Depuis le début de cette section, nous avons vu que la fonction *wp* démarre depuis la postcondition et, instruction après instruction, change la formule jusqu'à atteindre le début de la fonction, où elle génère la condition de vérification (comme brièvement expliqué dans la section 4.1.4), que nous devons vérifier.

Ceci dit, le lecteur attentif aura peut-être également remarqué que depuis le début de ce livre, dans la plupart des exemples, nous n'avons pas *une* condition de vérification à vérifier, mais *plusieurs*. Sur un aspect théorique, cela ne fait pas grande différence, mais d'un point de vue pratique, cela permet de générer des conditions de vérification plus simples à vérifier pour les solveurs SMT.

En fait, le greffon WP génère plusieurs conditions de vérifications en parallèle pendant le calcul de WP. À chaque fois qu'il rencontre une nouvelle instruction, la fonction *wp* est appliqué pour cette instruction sur toutes les conditions de vérifications rencontrées sur le chemin de programme auquel l'instruction appartient (pour garantir cela, la règle de la conditionnelle est sensiblement différente de ce que nous avons présentés, mais nous n'entrerons pas dans ces détails). Cependant, quand cette étape inclut la preuve d'une propriété (par exemple, une assertion), au lieu d'appliquer la règle en l'ajoutant comme une conjonction, elle est simplement ajoutée à l'ensemble des conditions de vérification à prouver. Intuitivement : nous commençons un nouveau calcul de plus faible précondition en parallèle à partir de ce point de programme.

Illustrons cela sur cet exemple jouet :

```
1  /* run.config
2    DONTRUN:
3  */
4
5  int x ;
6  int y ;
7
8  // 8.
9  /*@ requires P(x) ;
10     ensures P(x) ;
11     ensures P(y) ;
12 */
13 void toy(void){
14   // 7.
15   if(x > 0){
16     // 6.
17     x ++ ;
18     // 5.
19     //@ assert Q(x);
20   } else {
21     // 4.
22     y ++ ;
23     // 3.
24     //@ check Q(y);
25   }
26   // 2.
27   x = x * y ;
28   // 1.
29 }
```

Nous commençons (en 1) avec deux conditions de vérification :

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

Nous appliquons la règle que l'affectation sur chacune d'elle, et nous atteignons 2 avec :

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

En 2, nous devons traiter la conditionnelle. Au lieu de directement appliquer la règle standard de WP, nous séparons l'analyse en deux ensembles de conditions de vérification, initialement équivalents. Considérons d'abord la branche `else` puis la branche `then`.

De 2 à 3, nous rencontrons une annotation `check`, donc nous ajoutons une nouvelle condition de vérification et nous obtenons l'ensemble :

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

Ensuite, nous appliquons la règle de l'affectation, et nous obtenons en 4 :

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

De 2 à 5, nous rencontrons une annotation `assert`, nous ajoutons donc cette connaissance à toutes les conditions de vérification, et nous en créons une nouvelle :

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

Ensuite, nous appliquons la règle de l'affectation et nous obtenons en 6 :

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

Maintenant, nous devons réconcilier les deux ensembles. Nous omettons certains détails ici (l'astuce se trouve dans les variables v' qui sont introduites et qui permettent de séparer les valeurs en fonction de la condition), et nous obtenons en 7, quelque chose comme :

```
1  VC1:
2    ((x <= 0 ==> y' == y+1 && x' == x) &&
3     (x >  0 ==> y' == y   && x' == x+1 && Q(x+1))) ==>
4       P(x' * y')
5
6  VC2:
7    ((x <= 0 ==> y' == y+1) &&
8     (x >  0 ==> y' == y    && Q(x+1))) ==>
9       P(y')
10 VC3:
11   (x <= 0) ==> Q(x+1)
12
13 VC4:
14   (x >  0) ==> Q(x+1)
```

Finalement, nous atteignons la précondition, et nous obtenons en 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)
```

Nous pouvons voir que pour chaque propriété que nous devons prouver, nous avons construit une condition de vérification spécifique qui rassemble la connaissance disponible le long du chemin de programme qui atteint le point où la vérification est demandée.

4.1.8. Exercices

4.1.8.1. Une série d'affectations

Calculer à la main la plus faible précondition du programme suivant :

```
/*@
  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 ;
}
```

En utilisant la bonne règle d'inférence, en déduire que le programme est conforme au contrat fixé pour cette fonction.

4.1.8.2. Branche « *then* » vide dans une conditionnelle

Nous avons précédemment montré que lorsqu'une structure conditionnelle a une branche « *else* » vide, cela signifie que la conjonction de la précondition et de la négation de la condition de notre structure conditionnelle est suffisante pour prouver la postcondition de la structure conditionnelle. Pour les deux questions qui suivent, nous avons uniquement besoin des règles d'inférence et pas du calcul de plus faible précondition.

Montrer que lorsque, au lieu de la branche « *else* », c'est la branche « *then* » qui est vide, la postcondition de structure conditionnelle est vérifiée par la conjonction de la précondition et

de la condition de notre structure conditionnelle (puisque la branche « *else* » est la seule à potentiellement modifier l'état de la mémoire).

Montrer que si les deux branches sont vides, la structure conditionnelle est juste une instruction *skip*.

4.1.8.3. Court-circuit (*Short circuit*)

Les compilateurs C implémentent le court-circuit pour les conditions (c'est d'ailleurs imposé par le standard C). Par exemple, cela signifie qu'un code comme (**sans bloc « *else* »**) :

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

peut être réécrit comme :

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

Montrer que sur ces deux morceaux de code, le calcul de plus faible précondition génère une plus faible précondition pour tout code qui se trouverait dans le bloc « *then* ». Notons que nous supposons que les conditions sont pures (ne modifient aucune position en mémoire).

4.1.8.4. Un plus gros programme

Calculer à la main la plus faible précondition du programme suivant :

```
/*@
  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 ;
}
```

En utilisant la bonne règle d'inférence, en déduire que le programme est conforme au contrat fixé pour cette fonction.

4.2. Les boucles

Les boucles ont besoin d'un traitement de faveur dans la vérification déductive de programmes. Ce sont les seules structures de contrôle qui vont nécessiter un travail conséquent de notre part. Nous ne pouvons pas y échapper, car sans les boucles, nous pouvons difficilement prouver des programmes intéressants.

Avant de s'intéresser à la spécification des boucles, il est légitime de se poser la question suivante : pourquoi les boucles sont-elles compliquées ?

4.2.1. Induction et invariance

La nature des boucles rend leur analyse difficile. Lorsque nous faisons nos raisonnements arrière, il nous faut une règle capable de dire à partir de la postcondition quelle est la précondition d'une certaine séquence d'instructions. Problème : nous ne pouvons *a priori* pas déduire combien de fois la boucle s'exécutera et donc, nous ne pouvons pas non plus savoir combien de fois les variables ont été modifiées.

Nous procéderons en raisonnant par induction. Nous devons trouver une propriété qui est vraie avant de commencer la boucle et qui, si elle est vraie au début d'un tour de boucle, sera vraie à la fin (et donc par extension, au début du tour suivant). Quand la boucle termine, nous gagnons la connaissance que la condition de boucle est fausse qui, en conjonction avec l'invariant, doit nous permettre de prouver que la postcondition de la boucle est vérifiée.

Ce type de propriété est appelé un invariant de boucle. Un invariant de boucle est une propriété qui doit être vraie avant et après chaque tour d'une boucle, et plus précisément chaque fois que l'on évalue la condition de la boucle. Par exemple, pour la boucle :

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

La propriété $0 \leq \texttt{i} \leq 10$ est un invariant de la boucle. La propriété $-42 \leq \texttt{i} \leq 42$ est également un invariant de la boucle (qui est nettement plus imprécis néanmoins). La propriété $0 < \texttt{i} \leq 10$ n'est pas un invariant, elle n'est pas vraie à l'entrée de la boucle. La propriété $0 \leq \texttt{i} < 10$ **n'est pas un invariant de la boucle**, elle n'est pas vraie à la fin du dernier tour de la boucle qui met la valeur `i` à 10.

Le raisonnement produit par l'outil pour vérifier un invariant I sera donc :
- vérifions que I est vrai au début de la boucle (établissement),
- vérifions que si I est vrai avant de commencer un tour, alors I est vrai après (préservation).

4.2.1.1. Formel - Règle d'inférence

En notant l'invariant I, la règle d'inférence correspondant à la boucle est définie comme suit :

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

Et le calcul de plus faible précondition est le suivant :

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

Détaillons cette formule :

- (1) le premier I correspond à l'établissement de l'invariant, c'est vulgairement la « précondition » de la boucle,
- la deuxième partie de la conjonction $((B \wedge I) \Rightarrow wp(c, I))$ correspond à la vérification du travail effectué par le corps de la boucle :
  - la précondition que nous connaissons du corps de la boucle (notons KWP, « *Known WP* ») est $(KWP = B \wedge I)$. Soit, le fait que nous sommes rentrés dedans (B est vrai), et que l'invariant est vérifié à ce moment (I, qui est vrai avant de commencer la boucle par (1), et dont veut vérifier qu'il sera vrai en fin de bloc de la boucle (2)),
  - (2) ce qu'il nous reste à vérifier c'est que KWP implique la précondition réelle* du bloc de code de la boucle $(KWP \Rightarrow wp(c, Post))$. Ce que nous voulons en fin de bloc, c'est avoir maintenu l'invariant I (B n'est peut-être plus vrai en revanche). Donc $KWP \Rightarrow wp(c, I)$, soit $(B \wedge I) \Rightarrow wp(c, I)$,
  - * cela correspond à une application de la règle de conséquence expliquée précédemment.
- Finalement, la dernière partie $((\neg B \wedge I) \Rightarrow Post)$ nous dit que le fait d'être sorti de la boucle ($\neg B$), tout en ayant maintenu l'invariant I, doit impliquer la postcondition voulue pour la boucle.

Dans ce calcul, nous pouvons noter que la fonction wp ne nous donne aucune indication sur le moyen d'obtenir l'invariant I. Nous allons donc devoir spécifier manuellement de telles propriétés à propos de nos boucles.

4.2.1.2. Retour à l'outil

Il existe des outils capables d'inférer des invariants (pour peu qu'ils soient simples, les outils automatiques restent limités). Ce n'est pas le cas de WP. Il nous faut donc écrire nos invariants à la main. Trouver et écrire les invariants des boucles de nos programmes sera toujours la partie la plus difficile de notre travail lorsque nous chercherons à prouver des programmes.

En effet, si en l'absence de boucle, la fonction de calcul de plus faible précondition peut nous fournir automatiquement les propriétés vérifiables de nos programmes, ce n'est pas le cas pour les invariants de boucle pour lesquels nous n'avons pas accès à une procédure automatique de calcul. Nous devons donc trouver et formuler correctement ces invariants, et selon l'algorithme, celui-ci peut parfois être très subtil.

Pour indiquer un invariant à une boucle, nous ajoutons les annotations suivantes en début de boucle :

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

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

RAPPEL : L'invariant est bien : $i \leq 30$!

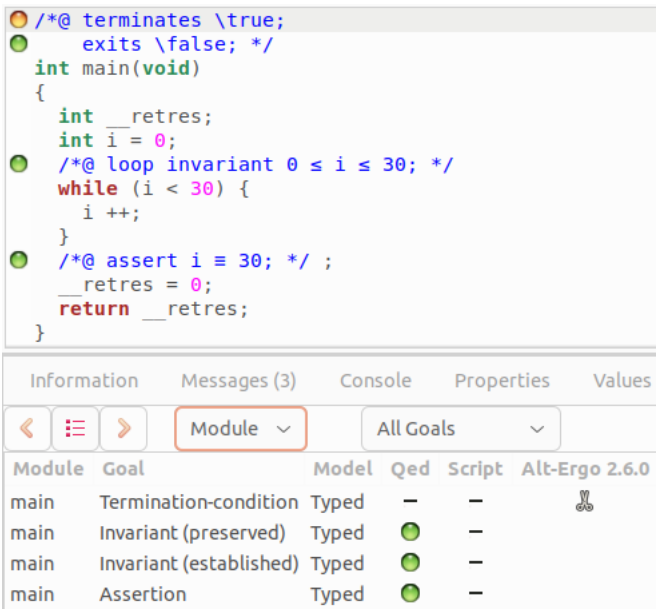
Pourquoi ? Parce que tout au long de la boucle `i` sera bien compris entre 0 et 30 **inclus**. 30 est même la valeur qui nous permettra de sortir de la boucle. Plus encore, une des propriétés demandées par le calcul de plus faible précondition sur les boucles est que lorsque l'on invalide la condition de la boucle, par la connaissance de l'invariant, on peut prouver la postcondition (Formellement : $(\neg B \wedge I) \Rightarrow Post$).

La postcondition de notre boucle est $\texttt{i} = 30$ et doit être impliquée par $\neg \texttt{i} < 30 \wedge 0 \leq \texttt{i} \leq 30$. Ici, cela fonctionne bien :

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

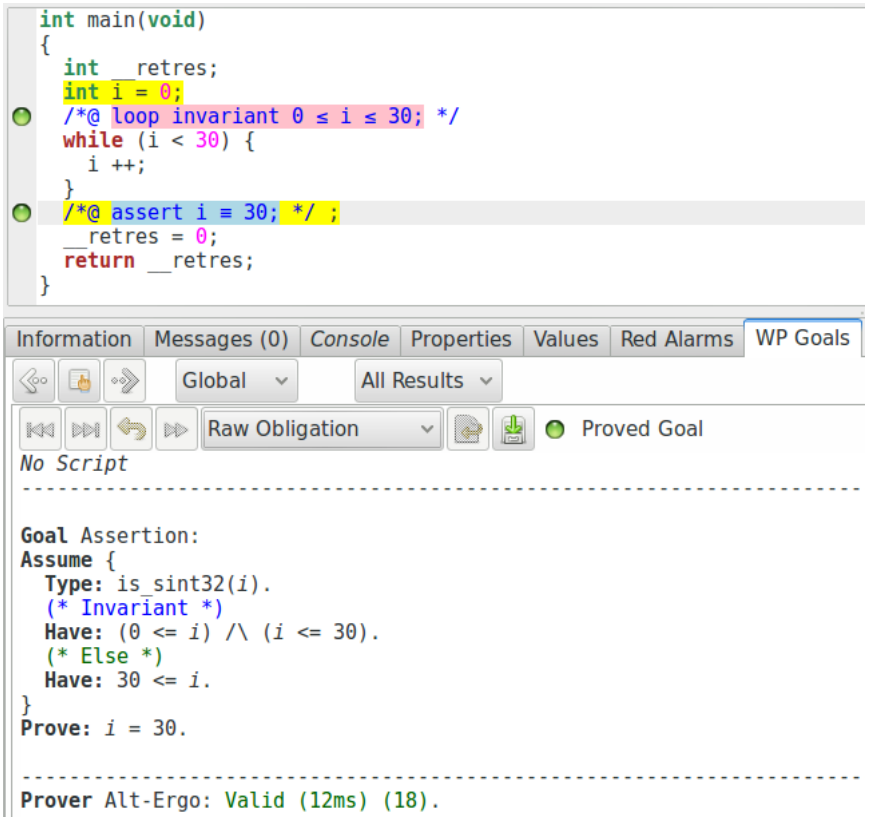
Si l'invariant excluait l'égalité à 30, la postcondition ne serait pas atteignable.

Une nouvelle fois, nous pouvons jeter un œil à la liste des buts dans « *WP Goals* » :



Nous remarquons que WP ne peut pas prouver la terminaison de la fonction, nous reviendrons sur ce sujet un peu plus loin. Plus important pour l'instant, nous voyons que WP décompose la preuve de l'invariant en deux parties : l'établissement de l'invariant et sa préservation. WP produit exactement le raisonnement décrit plus haut pour la preuve de l'assertion. Dans les versions récentes de Frama-C, Qed est devenu particulièrement puissant et la condition de

vérification générée ne le montre pas (affichant simplement « *True* »). En utilisant l'option `-wp-no-simpl` au lancement, nous pouvons quand même voir la condition de vérification correspondante :



Mais notre spécification est-elle suffisante ?

```
1  int main(){
2    int i = 0;
3    int h = 42;
4
5    /*@
6      loop invariant 0 <= i <= 30;
7    */
8    while(i < 30){
9      ++i;
10   }
11   //@assert i == 30;
12   //@assert h == 42;
13 }
```

Voyons le résultat :

```
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;
}
```

Il semble que non.

4.2.2. La clause `assigns` ... pour les boucles

En fait, à propos des boucles, WP ne considère vraiment *que* ce que lui fournit l'utilisateur pour faire ses déductions. Et ici l'invariant ne nous dit rien à propos de l'évolution de la valeur de `h`. Nous pourrions signaler l'invariance de toute variable du programme, mais ce serait beaucoup d'efforts. ACSL nous propose plus simplement d'ajouter des annotations `assigns` pour les boucles. Toute autre variable est considérée invariante. Par exemple :

```
1  int main(){
2    int i = 0;
3    int h = 42;
4
5    /*@
6      loop invariant 0 <= i <= 30;
7      loop assigns i;
8    */
9    while(i < 30){
10     ++i;
11   }
12   //@assert i == 30;
13   //@assert h == 42;
14 }
```

Cette fois, nous pouvons établir la preuve de correction de la boucle. Par contre, rien ne nous prouve sa terminaison. L'invariant de boucle n'est pas suffisant pour effectuer une telle preuve. Par exemple, dans notre programme, si nous réécrivons la boucle comme ceci :

```
1  /*@
2    loop invariant 0 <= i <= 30;
3    loop assigns i;
4  */
5  while(i < 30){
6
7  }
```

L'invariant est bien vérifié également, mais nous ne pourrons jamais prouver que la boucle se termine : elle est infinie.

4.2.3. Correction partielle et correction totale - Variant de boucle

En vérification déductive, il y a deux types de correction, la correction partielle et la correction totale. Dans le premier cas, la formulation est « si la précondition est validée et **si** le calcul termine, alors la postcondition est validée ». Dans le second cas, « si la précondition est validée, alors le calcul termine et la postcondition est validée ». WP s'intéresse par défaut à de la preuve de correction partielle, *mais* demande au noyau de Frama-C de générer une clause `terminates` pour nous forcer à vérifier ensuite la terminaison (sauf si nous changeons volontairement cette clause comme nous le présenterons dans la section 4.4.3). Par conséquent, si nous essayons de prouver le programme suivant :

```
1 void foo(){
2   while(1){}
3   //@ assert \false;
4 }
```

nous obtenons ceci :

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

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

L'assertion « FAUX » est prouvée ! La raison est simple : la non-terminaison de la boucle est triviale : la condition de la boucle est « VRAI » et aucune instruction du bloc de la boucle ne permet d'en sortir puisque le bloc ne contient pas de code du tout. Comme nous sommes en correction partielle, et que le calcul ne termine pas, nous pouvons prouver n'importe quoi au sujet du code qui suit la partie non terminante. Si, en revanche, la non-terminaison est non triviale, il y a peu de chances que l'assertion soit prouvée. Par contre, comme nous n'avons pas prouvé la terminaison de la boucle, la clause `terminates` n'est pas prouvée et nous pouvons voir qu'il pourrait y avoir un problème.

À noter qu'une assertion inatteignable est toujours prouvée comme vraie de cette manière :

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

Information | Messages (0) | Console | Properties | Values | WP Goals

```
[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
```

C'est également le cas lorsque l'on sait trivialement qu'une instruction produit nécessaire-

ment une erreur d'exécution (par exemple en déréférençant la valeur `NULL`), comme nous avions déjà pu le constater avec l'exemple de l'appel à `abs` avec la valeur `INT_MIN`.

4.2.3.1. Preuve de terminaison - Fournir une mesure

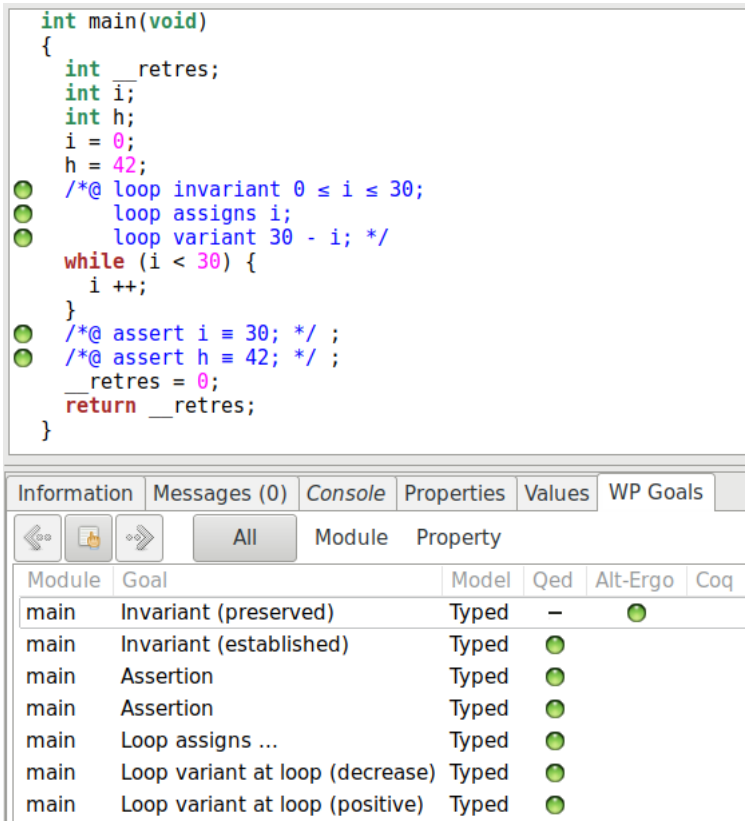
En preuve de programme, quand nous avons besoin de prouver la terminaison d'un algorithme, nous introduisons une notion de *mesure*. Une mesure est une expression qui doit strictement décroître d'après une relation bien-fondée R. D'une « étape » du calcul à une autre, nous voulons que la mesure décroisse d'après R. En ACSL, par défaut, la mesure est une expression entière décroissante et positive : $R(x, y) \Leftrightarrow x > y \wedge x \geq 0$, mais il est également possible de fournir une autre relation (voir Section 4.2.3.2).

Pour les boucles, nous utilisons la notion de variant de boucle et notre « étape » est l'itération de boucle. Le variant de boucle n'est pas une propriété, mais une expression faisant intervenir des éléments modifiés par la boucle et donnant une borne supérieure sur le nombre d'itérations restant à effectuer à un tour de la boucle. C'est donc une expression supérieure à 0 et strictement décroissante d'un tour de boucle à l'autre. Or, si pour tout tour de boucle, la valeur est décroissante, mais reste positive, c'est que la boucle s'arrête nécessairement après un nombre fini d'itérations.

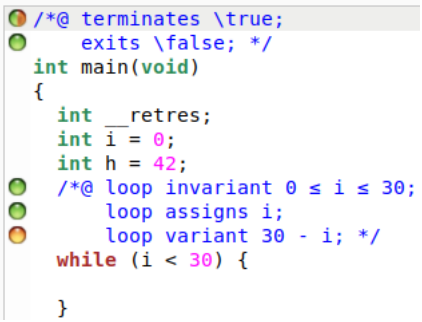
Si nous reprenons notre programme précédent, nous pouvons ajouter le variant de cette façon :

```
1  int main(){
2    int i = 0;
3    int h = 42;
4
5    /*@
6      loop invariant 0 <= i <= 30;
7      loop assigns i;
8      loop variant 30 - i;
9    */
10   while(i < 30){
11     ++i;
12   }
13   //@assert i == 30;
14   //@assert h == 42;
15 }
```

Une nouvelle fois nous pouvons regarder les buts générés :



WP génère deux conditions de vérification pour le variant de boucle : on doit assurer que la valeur de l'expression est positive et qu'elle décroît strictement pendant l'exécution de la boucle. Si nous supprimons la ligne de code qui incrémente `i`, WP ne peut plus prouver que la valeur `30 - i` décroît strictement et la clause `terminates` est maintenant « prouvée sous hypothèse ».



Nous donnerons plus de détail sur la preuve de la clause `terminates` plus tard. Pour l'instant, notons que, puisque le variant de boucle est une borne supérieure sur le nombre d'itérations restantes, être capable de donner un variant de boucle n'induit pas nécessairement d'être capable de donner le nombre exact d'itérations qui doivent encore être exécutées par la boucle, car nous n'avons pas toujours une connaissance aussi précise du comportement de notre programme. Nous pouvons par exemple avoir un code comme celui-ci :

```
#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);
  }
}
```

Ici, à chaque tour de boucle, nous diminuons la valeur de la variable `i` par une valeur dont nous savons qu'elle se trouve entre 1 et `i`. Nous pouvons donc bien assurer que la valeur de `i` est positive et décroît strictement, mais nous ne pouvons pas dire combien de tours de boucles vont être réalisés pendant une exécution.

Notons aussi que le variant de boucle n'a besoin d'être positif qu'au début de l'exécution du bloc de la boucle. Donc, dans le code suivant :

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

Même si `i` peut être négatif lorsque la boucle termine, cette valeur est bien un variant de la boucle puisque nous n'exécutons pas le corps de la boucle à nouveau.

4.2.3.2. Mesure généralisée

Dans cette courte section, nous présentons un usage particulier des variants de boucle. Il utilise une fonctionnalité d'ACSL qui ne sera présentée que dans la partie suivante du tutoriel. Ceci dit, cette section devrait rester compréhensible sans avoir besoin de lire la partie suivante. Notons que, puisque cette manière d'utiliser les variants de boucle est rarement utile, cela peut être ignoré dans une première lecture du tutoriel.

La plupart du temps une simple mesure entière est suffisante pour exprimer des variants de boucle et prouver la terminaison des boucles. Cependant, dans certaines situations, elle peut être difficile à utiliser. Pour ces situations, ACSL fournit la notion de variant généralisé où l'on peut fournir une expression de n'importe quel type, à condition de pouvoir fournir la relation bien-fondée qui correspond à ce type. Cependant, WP **ne vérifie pas** que la relation fournie est bien fondée, et cela doit donc être vérifié par ailleurs. Une mesure de ce type est fournie à l'aide de cette syntaxe :

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

Où le prédicat `Relation` doit être un prédicat binaire défini pour le type de `term`. Illustrons cela avec l'exemple suivant :

```
/*@ ensures \result >= 0;
    assigns \nothing ; */
int positive(void);

struct pair { int x, y ; };

/*@ predicate lexico(struct pair p1, struct pair p2) =
      p1.x > p2.x && p1.x >= 0 ||
      p1.x == p2.x && p1.y > p2.y && p1.y >= 0 ;
*/

//@ requires p.x >= 0 && p.y >= 0;
void f(struct pair p) {
  /*@ loop invariant p.x >= 0 && p.y >= 0;
      loop assigns p ;
      loop variant p for lexico;
  */
  while (p.x > 0 && p.y > 0) {
    if (positive()) {
      p.x--;
      p.y = positive();
    }
    else p.y--;
  }
}
```

Nous décrirons plus précisément la notion de prédicat utilisateur dans la Section 5.2, pour le moment considérons que le code ACSL aux lignes 12–15 définit une sorte de fonction qui reçoit deux structures en entrée et retourne une valeur booléenne. Ici, au lieu d'utiliser un entier pour notre mesure, nous utilisons une structure qui contient deux entiers. Le prédicat `lexico` définit un ordre lexicographique sur les structures de type `pair`. Si nous regardons la condition de vérification générée, dans la partie « Prove », nous voyons que nous devons prouver `P_lexico(p, p_1)`, qui utilise notre prédicat utilisateur.

Notons que, à la différence des clauses `loop variant` par défaut, cette clause génère une unique condition de vérification. En effet, dans WP, la relation par défaut $R(x, y) \Leftrightarrow x > y \wedge x \geq 0$ est séparée en deux parties $x > y$ et $x \geq 0$, mais pour les relations définies par l'utilisateur, ce n'est pas forcément possible, donc WP n'essaie jamais de telles transformations.

4.2.4. Lier la postcondition et l'invariant

Supposons le programme spécifié suivant. Notre but est de prouver que le retour de cette fonction est l'ancienne valeur de `a` à laquelle nous avons ajouté 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;
}
```

Le calcul de plus faibles préconditions ne permet pas de sortir de la boucle des informations qui ne font pas partie de l'invariant. Dans un programme comme :

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

en remontant les instructions depuis la postcondition, on conserve toujours les informations à propos de `a`. À l'inverse, comme mentionné plus tôt, en dehors de la boucle WP, ne considérera que les informations fournies par notre invariant. Par conséquent, notre fonction `add_ten` ne peut pas être prouvée en l'état : l'invariant ne mentionne rien à propos de `a`. Pour lier notre postcondition à l'invariant, il faut ajouter une telle information. Par exemple :

```
/*@
  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;
}
```

i Ce besoin peut apparaître comme une contrainte très forte. Il ne l'est en fait pas tant que cela. Il existe des analyses fortement automatiques capables de calculer les invariants de boucles. Par exemple, sans spécification, une interprétation abstraite calculera assez facilement `0 <= i <= 10` et `\old(a) <= a <= \old(a)+10`. En revanche, il est souvent bien plus difficile de calculer les relations qui existent entre des variables diffé-

rentes qui évoluent dans le même programme, par exemple l'égalité mentionnée par notre invariant ajouté.

4.2.5. Multiples invariants de boucles

Nous pouvons spécifier plusieurs clauses `loop invariant` sur une boucle. Il y a des similarités dans la manière dont les assertions et les invariants de boucles sont traités. En particulier, quand une assertion en suit une autre, elle peut être prouvée en utilisant cette dernière comme hypothèse, comme dans :

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

où `A2` peut être prouvée avec `A1` en hypothèse, nous pouvons faire la même chose avec les invariants de boucles. Cependant, il y a quelques subtilités. Considérons cet exemple :

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

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

Ici, nous avons 4 conditions de vérification à prouver :

- `I1` est établi ;
- `I2` est établi ;
- `I1` est préservé par chaque itération de boucle ;
- `I2` est préservé par chaque itération de boucle.

Pour l'établissement de `I1`, il y a peu de choses à dire : WP génère une condition de vérification avec toutes les hypothèses qui mènent à la boucle (ici, uniquement le fait que `x` vaut 0) et nous devons prouver que l'invariant est vrai (ici, `x <= 0`). Cela nous donne la condition de vérification suivante (nous avons utilisé l'option `-wp-no-let` pour éviter une preuve immédiate par Qed) :

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

Pour `I2`, c'est assez similaire. *Mais*, puisque nous avons prouvé que `I1` est établi, nous l'ajoutons dans les hypothèses pour prouver que `I2` est établi :

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

Si nous avions un troisième invariant de boucle `I3`, la preuve de son établissement recevrait la connaissance que `I1` et `I2` sont établis en hypothèse et ainsi de suite.

Ensuite, nous devons prouver que `I1` et `I2` sont préservés par chaque itération. Pour prouver que `I1` est préservé, nous supposons que tous les invariants de boucles sont vrais avant l'itération, et nous prouvons que `I1` est toujours vrai à la fin de l'itération. Cependant, nous pouvons ajouter des hypothèses supplémentaires. En particulier, nous pouvons ajouter que tous les invariants de boucles étaient vrais avant le début de la boucle (établissement). Ce qui nous donne :

— en supposant que `I1` et `I2` étaient vrai au début de la boucle ;
— en supposant que tous les invariants sont vérifiés quand l'itération commence ;
— prouvons que `I1` est toujours vraie à la fin de l'itération.

Notons que nous avons légèrement simplifié la condition de vérification pour nous concentrer sur les aspects les plus importants ici :

```
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.
```

Established invariant

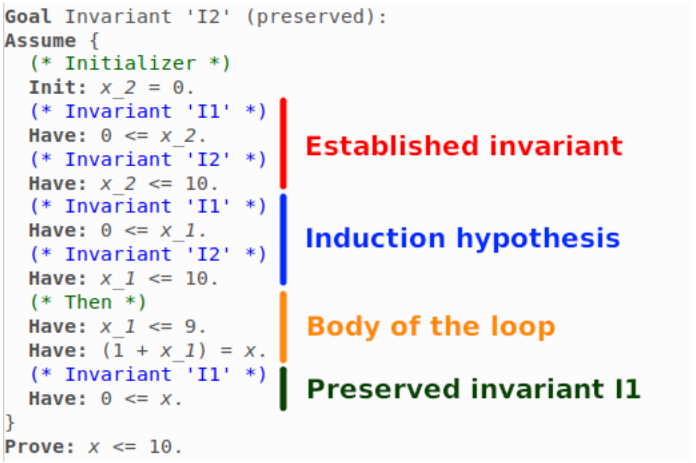
Induction hypothesis

Body of the loop

Finalement, pour prouver que `I2` est préservé, nous pouvons collecter encore plus d'hypothèses puisque nous savons maintenant que `I1` est préservé, donc :

— en supposant que `I1` et `I2` étaient vrai au début de la boucle ;
— en supposant que tous les invariants sont vérifiés quand l'itération commence ;
— en supposant aussi que `I1` est préservé,
— prouvons que `I2` est toujours vraie à la fin de l'itération.

Et à nouveau, nous ferions de même s'il y avait d'autres invariants à prouver.



4.2.6. Différentes sortes d'invariants de boucle

Tout comme les clauses `assert` qui ont deux variantes `check` et `admit`, nous pouvons spécifier des invariants de boucle qui ne sont que vérifiés ou admis. La syntaxe est la suivante :

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

À nouveau, cela change la génération des conditions de vérifications et les hypothèses reliées.

Le comportement associé aux clauses `admit loop invariant` n'a pas de subtilité particulière. Une telle close est ajoutée comme hypothèse exactement dans les mêmes situations que pour une clause `loop invariant` classique, de telle sorte que la seule différence entre ces deux clauses est le fait que WP ne génère pas de condition de vérification à vérifier pour s'assurer que la clause `admit loop invariant` est vraie.

La clause `check loop invariant` est un petit peu plus compliquée à traiter. Puisque la clause doit être vérifiée, WP génère des conditions de vérification comme il le fait pour la clause `loop invariant` classique. Ensuite, puisque cette clause est seulement vérifiée, nous ne devons pas la supposer vraie pour prouver d'autres propriétés. Cependant, pour prouver qu'une clause `check loop invariant` est préservée, nous devons quand même supposer qu'elle est vraie au début de la boucle, mais *seulement* pour cette preuve de préservation (pas pour les autres) invariants de boucle.

Illustrons ce comportement avec les différentes conditions de vérifications générées pour `C3` et `I4` de l'extrait de code suivant (à nouveau, nous utilisons l'option `-wp-no-let` et nous avons légèrement nettoyés les conditions de vérification) :

```
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 ;
```

```
8         loop assigns x ;
9     */
10    while (x < 10) {
11      x++ ;
12      y += 2 ;
13    }
14  }
```

La condition de vérification associée à l'établissement de `C3` est simple, puisqu'elle apparaît après les invariants de boucle `I1` et `A2` qui doivent tous deux être admis (même si `A2` n'a pas été prouvée), nous avons ces deux propriétés en hypothèse :

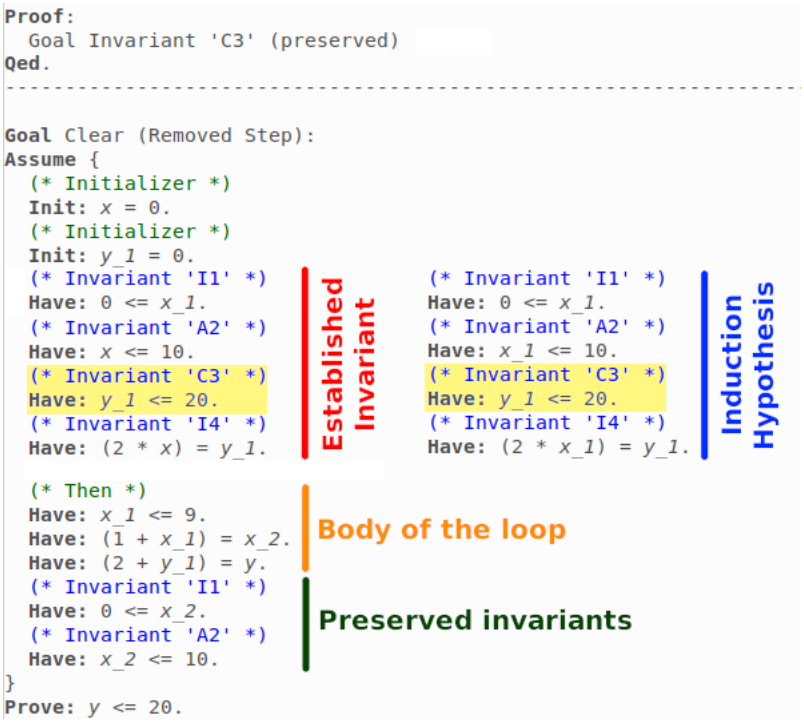
```
Goal Invariant 'C3' (established):
Assume {
  Type: is_sint32(x) /\ is_sint32(y).
  (* Initializer *)
  Init: x = 0.
  (* Initializer *)
  Init: y = 0.
  (* Invariant 'I1' *)
  Have: 0 <= x.
  (* Invariant 'A2' *)
  Have: x <= 10.
}
Prove: y <= 20.
```

La condition de vérification associée à l'établissement de `I4` est plus intéressante : elle reçoit bien `I1` et `A2` dans ses hypothèses, mais `C3` n'y est pas, car elle n'est que vérifiée et pas admise après cela.

```
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.
```

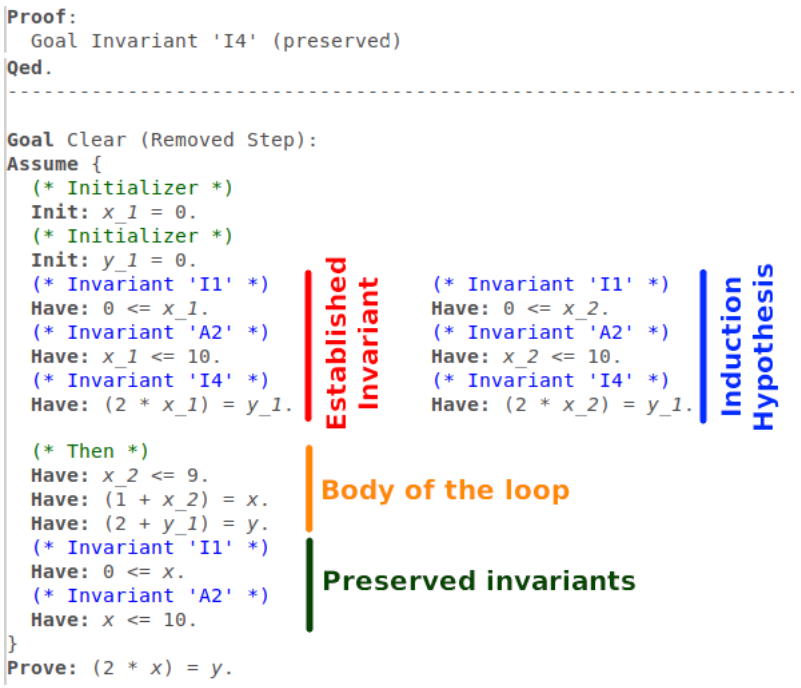
Ensuite, nous pouvons regarder la préservation de ces invariants. Pour `C3`, dans les hypothèses, nous trouvons :

— tous les invariants établis, incluant `C3` ;
— tous les invariants admis (hypothèse d'induction), incluant `C3` ;
— tous les invariants qui précèdent `C3`, dont la préservation est admise.



À l'inverse, si nous regardons la préservation de `I4`, dans les hypothèses, nous trouvons :

— tous les invariants établis, mais pas `C3` ;
— tous les invariants admis (hypothèse d'induction), mais pas `C3` ;
— tous les invariants qui précèdent `I3`, dont la préservation est admise, mais pas `C3`.



Comprendre comment les invariants sont utilisés par WP en fonction de l'ordre dans lequel ils apparaissent est utile pour prouver des invariants complexes. Plus nous transmettons d'hypothèses utiles pour la preuve, plus elle sera facile. En particulier, Qed peut faire beaucoup de simplification en faisant des réécritures dans les hypothèses à l'aide de règles de déduction. Par

conséquent, fournir d'abord les invariants les plus simples puis les plus complexes peut aider le processus de preuve en commençant par prouver les propriétés facile, puis en utilisant les connaissances obtenues pour prouver les plus complexes.

4.2.7. Terminaison prématurée de boucle

Un invariant de boucle doit être vrai chaque fois que la condition de la boucle est évaluée. En fait, cela signifie aussi qu'elle doit être vraie avant chaque itération, et après chaque itération **complète**. Illustrons cette idée importante avec un exemple.

```
1  int main(){
2    int i = 0;
3    int h = 42;
4
5    /*@
6      loop invariant 0 <= i <= 30;
7      loop assigns i;
8      loop variant 30 - i;
9    */
10   while(i < 30){
11     ++i;
12
13     if(i == 30) break ;
14   }
15   //@assert i == 30;
16   //@assert h == 42;
17 }
```

Dans cette fonction, quand la boucle atteint l'indice 30, elle effectue une opération `break` avant que la condition de la boucle soit à nouveau testée. L'invariant que nous avons écrit est bien sûr vérifié, mais nous pouvons en fait le restreindre encore.

```
1  int main(){
2    int i = 0;
3    int h = 42;
4
5    /*@
6      loop invariant 0 <= i <= 29;
7      loop assigns i;
8      loop variant 30 - i;
9    */
10   while(i < 30){
11     ++i;
12
13     if(i == 30) break ;
14   }
15   //@assert i == 30;
16   //@assert h == 42;
17 }
```

Ici, nous pouvons voir que nous avons exclu 30 de la plage des valeurs de `i` et la fonction est correctement vérifiée par WP. Cette propriété est particulièrement intéressante, car elle ne s'applique pas qu'à l'invariant. Aucune des propriétés de la boucle n'ont besoin d'être vérifiées pendant l'itération qui mène au `break`. Par exemple, nous pouvons écrire ce code qui est également vérifié :

```
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;
}
```

Nous voyons que nous pouvons écrire la variable `h` même si elle n'est pas listée dans la clause `loop assigns`, et que nous pouvons donner la valeur 42 à `i` alors que l'invariant l'interdirait, et aussi que nous pouvons rendre l'expression du variant négative. En fait, tout se passe exactement comme si nous avions écrit :

```
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;
}
```

C'est un schéma très pratique. Il correspond à tout algorithme qui cherche, à l'aide d'une boucle, une condition vérifiée par un élément particulier dans une structure de données et s'arrête quand cet élément est trouvé afin d'effectuer certaines opérations qui ne sont finalement pas vraiment des opérations de la boucle. D'un point de vue vérification, cela nous permet de simplifier le contrat associé à une boucle : nous savons que l'opération (potentiellement complexe) réalisée juste avant de sortir de la boucle ne nécessite pas d'être prise en compte dans l'invariant.

4.2.8. Exercice

4.2.8.1. Invariant de boucle

Écrire un invariant de boucle pour la boucle suivante et prouver qu'il est vérifié en utilisant la commande :

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

```
1  int x = 0 ;
2
3  while(x > -10){
4    -- x ;
5  }
```

Est-ce que la propriété $-100 \leq x \leq 100$ est un invariant correct ? Expliquer pourquoi.

4.2.8.2. Loop variant

Écrire un invariant et un variant corrects pour la boucle suivante et prouver l'ensemble à l'aide de la commande :

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

```
1  int x = -20 ;
2
3  while(x < 0){
4    x += 4 ;
5  }
```

Si le variant ne donne pas précisément le nombre d'itérations restantes, ajouter une variable qui enregistre exactement le nombre d'itérations restantes et l'utiliser comme variant. Il est possible qu'un invariant supplémentaire soit nécessaire.

4.2.8.3. Loop assigns

Écrire une clause `loop assigns` pour cette boucle, de manière que l'assertion ligne 8 soit prouvée ainsi que la clause `loop assigns`. Ignorons les erreurs à l'exécution dans cet exercice.

```
1  int h = 42 ;
2  int x = 0 ;
3  int e = 0 ;
```

```
4   while(e < 10){
5     ++ e ;
6     x += e * 2 ;
7   }
8   //@ assert h == 42 ;
```

Lorsque la preuve réussit, supprimer la clause `loop assigns` et trouver un autre moyen d'assurer que l'assertion soit vérifiée en utilisant des annotations différentes (notons que nous pouvons avoir besoin d'un label C dans le code). Que peut-on déduire à propos de la clause `loop assigns` ?

4.2.8.4. Terminaison prématurée

Écrire un contrat de boucle pour cette boucle qui permette de prouver les assertions aux lignes 9 et 10 ainsi que le contrat de boucle.

```
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. Plus d'exemples sur les boucles

4.3.1. Écrire des annotations de boucle

L'écriture des annotations de boucles demande un effort important pendant la vérification. Il n'y a pas de manière parfaite de faire ce travail. En particulier, trouver le bon invariant et la bonne manière de l'exprimer, est principalement une question d'expérience. Néanmoins, quelques idées simples peuvent aider, afin d'éviter certaines erreurs communes qui peuvent faire perdre beaucoup de temps pendant le processus.

Avant toute chose, nous devons exprimer une clause `loop assigns` correcte. Elle est généralement facile à écrire (il suffit de regarder les instructions d'affectation et les appels de fonctions), et si elle est incorrecte nous pourrions prouver des propriétés fausses, ce qui nous ferait perdre du temps. Il est plutôt déconseillé de chercher à fournir des clauses `loop assigns` trop précises. WP ne peut pas utiliser efficacement des clauses comme `array[x .. y]` lorsque `x` ou `y` sont elles-mêmes modifiées, on préférera donc des bornes qui les incluent tout en étant constante pendant l'exécution de la boucle. Si nous avons ensuite besoin d'informations plus précises pendant la preuve des invariants, il sera plus facile de fournir des invariants supplémentaires pour cela.

Ensuite, nous bornons les différentes variables qui sont assignées, en particulier les indices. Ces invariants sont généralement faciles à deviner, exprimer et vérifier. Nous mettons ces invariants au début de la liste des clauses `loop invariant`, puisque, comme nous l'avons expliqué dans les sections 4.2.5 et 4.2.6, l'ordre des invariants est important et ces propriétés très simples peuvent être propagées par WP dans les autres invariants pour simplifier la condition de vérification à prouver.

Pour la plupart des boucles, exceptées celles qui reposent sur une condition complexe, une fois que cette étape est réalisée, il est facile de fournir une clause `loop variant`. Les variables que nous venons de borner sont une bonne piste : nous avons juste à regarder la valeur qu'elles vont atteindre en fin d'exécution. Par exemple, si dans la boucle, on a une variable `i` qui va de 0 à `n`, `n - i` est un bon candidat. Pour des boucles plus complexes, nous pouvons utiliser du code fantôme (que l'on présente en section 6.3) pour rendre explicite une mesure utilisable pour un variant de boucle.

Ensuite, nous ajoutons nos invariants « principaux » à la boucle, c'est-à-dire ceux qui sont liés à la postcondition de la boucle (qui peut être aussi la postcondition de la fonction). Pour cela, nous pouvons utiliser la postcondition elle-même comme un guide. Si nous avons quelque chose comme `ensures P(n);` et une boucle qui itère `i` de 0 à `n`, il y a fort à parier que `loop invariant P(i);` soit un bon invariant de boucle. Notons que dans certaines situations, c'est la clause `assumes` d'un comportement qui se révèle être intéressante, typiquement quand le résultat de l'exécution est une simple valeur qui dépend de l'état en précondition, nous verrons cela dans un exemple plus tard. Ces invariants doivent généralement être positionnés en fin de liste des invariants de boucle.

Nous pouvons (optionnellement) avoir besoin de « glu » pour prouver les invariants « principaux ». Par exemple, si nous avons volontairement fourni une clause `loop assigns` imprécise, nous pourrions avoir besoin d'un invariant de boucle pour expliquer que certaines parties de la mémoire n'ont pas été modifiées, ou nous pourrions avoir besoin d'expliquer aux prouveurs que, parce que certaines propriétés sont vraies, nous pouvons déduire que l'invariant « principal » est vérifié, etc. Ces invariants doivent être placés avant les invariants « principaux », mais après les invariants simples qui bornent les variables. Ordonner ces invariants pourrait ne pas être si direct, généralement nous passerons par un processus d'essai et erreur.

Dans la suite de cette section, nous illustrons cette approche avec quelques exemples. Même si dans ce processus, il est fortement conseillé de lancer la preuve entre chaque étape, nous n'irons généralement pas jusqu'à ce niveau de détail dans les exemples futurs.

4.3.2. Exemple avec un tableau en lecture seule

S'il y a une structure de données que nous traitons avec les boucles, c'est bien le tableau. C'est une bonne base d'exemples pour les boucles, car ils permettent rapidement de présenter des invariants intéressants et surtout, ils nous permettront d'introduire des constructions très importantes d'ACSL.

Prenons par exemple la fonction qui cherche une valeur dans un tableau. Pour le moment, concentrons-nous sur le contrat de la fonction :

```c
#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;
*/
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;
}
```

Cet exemple est suffisamment fourni pour introduire des notations importantes.

D'abord, comme nous l'avons déjà mentionné, le prédicat `\valid_read` (de même que `\valid`) nous permet de spécifier non seulement la validité d'une adresse en lecture, mais également celle de tout un ensemble d'adresses contiguës. C'est la notation que nous avons utilisée dans cette expression :

```c
//@ requires \valid_read(a + (0 .. length-1));
```

Cette précondition nous atteste que les adresses `a+0`, `a+1`, …, `a+length-1` sont valides en lecture.

Nous avons également introduit deux notations qui vont nous être très utiles, à savoir `\forall` (∀) et `\exists` (∃), les quantificateurs de la logique. Le premier nous servant à annoncer que pour tout élément, la propriété suivante est vraie. Le second pour annoncer qu'il existe un élément tel que la propriété est vraie. Si nous commentons les deux lignes en questions, nous pouvons les lire de cette façon :

```c
/*@
//pour tout "off" de type "size_t", tel que SI "off" est compris entre 0 et "length"
//                                  ALORS la case "off" de "a" est différente de "element"
\forall size_t off ; 0 <= off < length ==> a[off] != element;

//il existe "off" de type "size_t", tel que "off" soit compris entre 0 et "length"
//                                  ET que la case "off" de "a" vaille "element"
\exists size_t off ; 0 <= off < length && a[off] == element;
*/
```

Si nous devions résumer leur utilisation, nous pourrions dire que sur un certain ensemble d'éléments, une propriété est vraie, soit à propos d'au moins l'un d'eux, soit à propos de la totalité d'entre eux. Un schéma qui reviendra typiquement dans ce cas est que nous restreindrons cet

ensemble à travers une première propriété (ici : `0 <= off < length`) puis nous voudrons prouver la propriété réelle qui nous intéresse à propos d'eux. **Mais il y a une différence fondamentale entre l'usage de `\exists` et celui de `\forall`**.

Avec `\forall type a ; p(a) ==> q(a)`, la restriction (`p`) est suivie par une implication. Pour tout élément, s'il satisfait une première propriété (`p`), alors il doit vérifier la seconde propriété `q`. Si nous mettions un ET comme pour le « il existe » (que nous expliquerons ensuite), cela voudrait dire que nous voulons que tout élément vérifie à la fois les deux propriétés. Parfois, cela peut être ce que nous voulons exprimer, mais cela ne correspond alors plus à l'idée de restreindre un ensemble dont nous voulons montrer une propriété particulière.

Avec `\exists type a ; p(a) && q(a)`, la restriction (`p`) est suivie par une conjonction, nous voulons qu'il existe un élément tel que cet élément est dans un certain état (défini par `p`), tout en satisfaisant l'autre propriété `q`. Si nous mettions une implication comme pour le « pour tout », alors une telle expression devient toujours vraie à moins que `p` soit une tautologie ! Pourquoi ? Existe-t-il « a » tel que p(a) implique q(a) ? Prenons n'importe quel « a » tel que p(a) est faux, l'implication devient vraie.

Notons que dans cet exemple, la clause `assume` du comportement `in` est exactement la négation de la clause `assumes` du comportement `notin`, c'est la raison pour laquelle les clauses `disjoint` et `complete` sont prouvées, en fait nous aurions pu l'exprimer comme suit :

```
/*@ ...
  behavior in:
    assumes !(\forall size_t off ; 0 <= off < length ==> array[off] != element);
  ...
*/
```

Parlons des annotations de la boucle. La première étape est d'ajouter la clause `loop assigns`. Ici, elle est simple : la boucle ne modifie que la variable `i`. La valeur de cette variable doit donc être bornée, elle va de `0` à `length`, c'est notre premier invariant : `0 <= i <= length`. Ceci nous donne également le variant de boucle : `length - i`. Maintenant, nous pouvons fournir notre invariant « principal ». Ici, il est relié à la clause `assumes`, et pas à la clause `ensures`. En particulier, la partie intéressante de la fonction est qu'à moins que l'on rencontre l'élément recherché, il ne se trouve pas dans le tableau, nous exploitons cela en partant de la clause `assumes` du comportement `notin` :

```
//@ \forall size_t off ; 0 <= off < length ==> array[off] != element;
```

La variable qui atteint `length` à la fin de la boucle est `i`, donc :

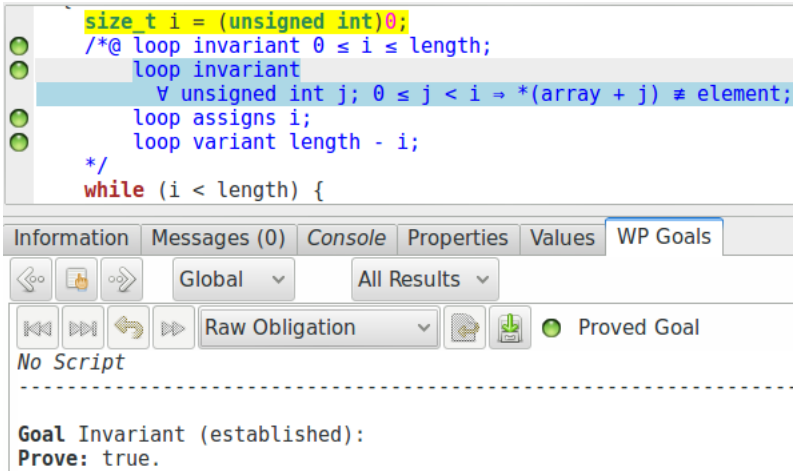
```
//@ loop invariant \forall size_t off ; 0 <= off < i ==> array[off] != element;
```

est certainement un bon candidat. Cela nous amène aux annotations de boucle suivantes :

```
/*@ 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];
```

Et effectivement, notre invariant de boucle final définit l'action de notre boucle, elle indique à WP ce que la boucle fera (ou apprendra dans le cas présent) tout au long de son exécution. Ici en l'occurrence, cette formule nous dit qu'à chaque tour, nous savons que pour toute case entre 0 et la prochaine que nous allons visiter (`i` exclue), elle stocke une valeur différente de l'élément recherché.

Le but de WP associé à la préservation de cet invariant est un peu compliqué, il n'est pour nous pas très intéressant de se pencher dessus. En revanche, la preuve de l'établissement de cet invariant est intéressante :



Nous constatons que cette propriété, pourtant complexe, est prouvée par Qed sans aucun problème. Si nous regardons sur quelles parties du programme la preuve se base, nous voyons l'instruction `i = 0` surlignée, et c'est bien la dernière instruction que nous effectuons sur `i` avant de commencer la boucle. Et donc effectivement, si nous faisons le remplacement dans la formule de l'invariant :

```
//@ loop invariant \forall size_t j; 0 <= j < 0 ==> array[j] != element;
```

« Pour tout j, supérieur ou égal à 0 et inférieur strict à 0 », cette partie est nécessairement fausse. Notre implication est donc nécessairement vraie.

4.3.3. Exemples avec tableaux mutables

Nous allons voir deux exemples avec la manipulation de tableaux en mutation. L'un avec une modification totale, l'autre en modification sélective.

4.3.3.1. Remise à zéro

Regardons la fonction effectuant la remise à zéro d'un tableau.

```
#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;
}
```

Cette fois, la boucle modifie le contenu du tableau, donc nous devons fournir cette information dans la clause `loop assigns`. Notons que nous pouvons utiliser la notation `n .. m` pour indiquer quelle partie du tableau a été modifiée. De plus, nous ne disons pas que la boucle assigne le contenu depuis `0` jusqu'à `i-1` (comme `i` est modifiée, WP ne peut pas exploiter cette écriture) mais depuis `0` jusqu'à `length-1`, c'est moins précis, mais cela peut être utilisé par WP en dehors de la boucle. Finalement, cette fois, nous relions directement l'invariant à la postcondition, le but de la fonction est de réinitialiser le tableau de 0 jusqu'à `length`, à une itération donnée, la boucle l'a déjà fait entre 0 et `i`.

4.3.3.2. Chercher et remplacer

Le dernier exemple qui nous intéresse est l'algorithme « chercher et remplacer ». C'est un algorithme qui modifie sélectivement des valeurs dans une certaine plage d'adresses. Il est toujours un peu difficile de guider l'outil dans ce genre de cas car, d'une part, nous devons garder « en mémoire » ce qui est modifié et ce qui ne l'est pas et, d'autre part, parce que l'induction repose sur ce fait.

À titre d'exemple, la première spécification et boucle annotée, que nous pouvons réaliser pour cette fonction ressemblerait à ceci (ce qui suit sensiblement le même processus que dans l'exemple précédent) :

```
#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]);
*/
```

```
12 void search_and_replace(int* array, size_t length, int old, int new){
13   /*@
14     loop invariant 0 <= i <= length;
15     loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
16                       ==> array[j] == new;
17     loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
18                       ==> array[j] == \at(array[j], Pre);
19     loop assigns i, array[0 .. length-1];
20     loop variant length-i;
21   */
22   for(size_t i = 0; i < length; ++i){
23     if(array[i] == old) array[i] = new;
24   }
25 }
```

Nous utilisons la fonction logique `\at(v, Label)` qui nous donne la valeur de la variable `v` au point de programme `Label`. Si nous regardons l'utilisation qui en est faite ici, nous voyons que dans l'invariant de boucle, nous cherchons à établir une relation entre les anciennes valeurs du tableau et leurs potentielles nouvelles valeurs :

```
1 /*@
2   loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
3                     ==> array[j] == new;
4   loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
5                     ==> array[j] == \at(array[j], Pre);
6 */
```

Pour tout élément que nous avons visité, s'il valait la valeur qui doit être remplacée, alors il vaut la nouvelle valeur, sinon il n'a pas changé. Alors que nous nous reposions sur la clause `assigns` pour ce genre de propriété dans les exemples précédents, ici nous ne pouvons pas le faire. Même si ACSL nous permettrait d'exprimer cette propriété de manière très précise, WP ne pourrait pas vraiment en tirer parti, dû à la manière dont cette clause est traitée. Nous devons donc utiliser un invariant et conserver une approximation des positions mémoire affectées.

En fait, si nous essayons de prouver l'invariant, WP n'y parvient pas, parce que la clause `assigns` n'est pas assez précise. Dans cette situation, nous fournissons un invariant supplémentaire pour détailler dans la plage modifiée quelles sont les positions en mémoire qui n'ont pas encore été modifiées par la boucle à une itération donnée :

```
1 for(size_t i = 0; i < length; ++i){
2     //@assert array[i] == \at(array[i], Pre); // échec de preuve
3     if(array[i] == old) array[i] = new;
4 }
```

Nous pouvons donc ajouter cette information comme invariant :

```
1   /*@
2     loop invariant 0 <= i <= length;
3     loop invariant \forall size_t j; i <= j < length
4                       ==> array[j] == \at(array[j], Pre);
5     loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
6                       ==> 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;
  }
```

Et cette fois, la preuve passera.

4.3.4. Exercices

Pour tous ces exercices, utiliser la commande suivante pour démarrer la vérification :

```
frama-c-gui -wp -wp-rte -warn-unsigned-overflow -warn-unsigned-downcast your-file.c
```

4.3.4.1. Fonctions sans modification : For all, Exists, ...

Actuellement, les pointeurs de fonction ne sont pas directement supportés par WP. Considérons que nous avons une fonction :

```
/*@
  assigns \nothing ;
  ensures \result <==> \true ; // some property about value instead
*/
int pred(int value){
  // your code
}
```

Écrire un corps (au choix) pour cette fonction et un contrat l'accompagnant. Ensuite, écrire les fonctions suivantes avec leurs contrats pour prouver leur correction. Notons qu'il n'est pas possible d'utiliser une fonction C dans un contrat, la propriété que choisie pour la fonction `pred` devra donc être inlinée dans la spécification des différentes fonctions.

- `forall_pred` retourne vrai si et seulement si `pred` est vraie pour tous les éléments ;
- `exists_pred` retourne vrai si et seulement si `pred` est vraie pour au moins un élément ;
- `none_pred` retourne vrai si et seulement si `pred` est fausse pour tous les éléments ;
- `some_not_pred` retourne vrai si et seulement si `pred` est fausse pour au moins un élément.

Les deux dernières fonctions devraient être écrites en appelant seulement les deux premières.

4.3.4.2. Fonction sans modification : Égalité de plages de valeurs

Écrire, spécifier et prouver la fonction `equal` qui retourne vrai si et seulement si deux plages de valeurs sont égales. Écrire, en utilisant la fonction `equal`, le code de `different` qui retourne vrai si et seulement si deux plages de valeurs sont différentes, votre postcondition devrait contenir un quantificateur existentiel.

```
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. Fonction sans modification : recherche dichotomique

La fonction suivante cherche la position d'une valeur fournie en entrée dans un tableau, en supposant que le tableau est trié. D'abord, considérons que la longueur du tableau est fournie en tant qu'`int` et utilisons des valeurs de ce même type pour gérer les indices. Nous pouvons noter qu'il y a deux comportements à cette fonction : soit la valeur existe dans le tableau (et le résultat est l'indice de cette valeur) ou pas (et le résultat est -1).

```
#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 ;
}
```

Cette fonction est un petit peu complexe à prouver, voici quelques conseils pour en venir à bout. D'abord, la longueur de la fonction est reçue en utilisant un type `int`, donc nous devons poser une restriction sur cette longueur en précondition pour qu'elle soit cohérente. Ensuite, l'un des invariants de la boucle devrait indiquer les bornes des valeurs `low` et `up`, mais nous pouvons noter que pour chacune d'elles, l'une des bornes n'est pas nécessaire. Finalement, la seconde propriété invariante devrait indiquer que si l'un des indices du tableau correspond à la valeur recherchée, alors cet indice devrait être correctement borné.

Plus dur : Modifier cette fonction de façon à recevoir `len` comme un `size_t`. Il faut modifier légèrement l'algorithme et la spécification/les invariants. Conseil : s'arranger pour que

`up` soit une borne exclue de la recherche.

4.3.4.4. Fonction avec modification : addition de vecteurs

Écrire, spécifier et prouver la fonction qui ajoute deux vecteurs dans un troisième. Fixer des contraintes arbitraires sur les valeurs d'entrée pour gérer le débordement des entiers. Considérer que le vecteur est résultant est spatialement séparé des vecteurs d'entrée. En revanche, le même vecteur devrait pouvoir être utilisé pour les deux vecteurs d'entrée.

```
1 void add_vectors(int* v_res, const int* v1, const int* v2, size_t len){
2
3 }
```

4.3.4.5. Fonction avec modification : inverse

Écrire, spécifier et prouver la fonction qui inverse un vecteur en place. Prendre garde à la partie non modifiée du vecteur à une itération donnée de la boucle. Utiliser la fonction `swap` précédemment prouvée.

```
1 void swap(int* a, int* b);
2
3 void reverse(int* array, size_t len){
4
5 }
```

4.3.4.6. Fonction avec modification : copie

Écrire, spécifier et prouver la fonction `copy` qui copie une plage de valeur dans un autre tableau, en commençant pas la première cellule du tableau. Considérer (et spécifier) d'abord que les deux plages sont entièrement séparées.

```
1 void copy(int const* src, int* dst, size_t len){
2
3 }
```

Plus dur : Les vraies fonctions `copy` et `copy_backward`.

En fait, une séparation aussi forte n'est pas nécessaire. Pour faire une copie en partant du début, la précondition réelle doit simplement garantir que si les deux plages se chevauchent en mémoire, le début de la destination ne doit pas être dans la plage source :

```
1 //@ requires \separated(&src[0 .. len-1], dst) ;
```

Essentiellement, en copiant des éléments dans cet ordre, nous pouvons les décaler depuis la fin d'une plage vers le début. En revanche, cela signifie que nous devons être plus précis dans notre contrat : nous ne garantissons plus une égalité avec le tableau source, mais avec les anciennes valeurs du tableau source. Nous devons également être plus précis dans notre invariant, d'abord en spécifiant aussi la relation avec l'état précédent de la mémoire, et ensuite en ajoutant un invariant qui nous dit que le tableau source n'est pas modifié entre le `i` ^{me} élément visité et le dernier.

Finalement, il est aussi possible d'écrire une fonction qui copie les éléments de la fin vers le début. Dans ce cas, à nouveau, les plages de valeurs peuvent se chevaucher, mais la condition n'est pas exactement la même. Écrire, spécifier et prouver la fonction `copy_backward` qui copie les éléments dans le sens inverse.

4.4. Appels de fonction

4.4.1. Appel de fonction

4.4.1.1. Formel - Calcul de plus faible précondition

Lorsqu'une fonction est appelée, le contrat de cette fonction est utilisé pour déterminer la précondition de l'appel, mais il est important de garder en tête deux aspects important pour exprimer le calcul de plus faible précondition.

Premièrement, la postcondition d'une fonction f qui serait appelée dans un programme n'est pas nécessairement directement la précondition calculée pour le code qui suit l'appel à f. Par exemple, si nous avons un programme : `x = f() ; c`, et si $wp(\texttt{c}, Q) = 0 \leq x \leq 10$ alors que la postcondition de la fonction `f` est $1 \leq x \leq 9$, nous avons besoin d'exprimer l'affaiblissement de la précondition réelle de `c` vers celle que l'on a calculé. Pour cela, nous renvoyons à la section 4.1.4, l'idée est simplement de vérifier que la postcondition de la fonction implique la précondition calculée.

Deuxièmement, en C, une fonction peut avoir des effets de bord. Par conséquent, les valeurs référencées en entrée de fonction ne restent pas nécessairement les mêmes après l'appel à la fonction, et le contrat devrait exprimer certaines propriétés à propos des valeurs avant et après l'appel. Donc, si nous avons des labels (C) dans la postcondition, nous devons faire les remplacements qui s'imposent par rapport au lieu d'appel.

Pour définir le calcul de plus faible précondition associée aux fonctions, introduisons d'abord quelques notations pour rendre les explications plus claires. Pour cela, considérons l'exemple suivant :

```
1  /*@ requires \valid(x) && *x >= 0 ;
2      assigns *x ;
3      ensures *x == \old(*x)+1 ; */
4  void inc(int* x);
5
6  void foo(int* a){
7    L1:
8    inc(a) ;
```

```
9   L2:
10  }
```

Le calcul de plus faible précondition de l'appel de fonction nous demande de considérer le contrat de la fonction appelée (ici, dans `foo`, quand nous appelons la fonction `inc`). Bien sûr, avant l'appel de la fonction, nous devons vérifier sa précondition, qui fait donc partie de la plus faible précondition. Mais nous devons aussi considérer la postcondition de la fonction, sinon cela voudrait dire que nous ne prenons pas en compte son effet sur l'état du programme.

Par conséquent, il est important de noter que dans la précondition, l'état mémoire considéré est bien celui pour lequel la plus faible précondition doit être vraie, tandis que pour la postcondition, ce n'est pas le cas : l'état considéré est celui qui suit l'appel, alors que dans la postcondition, lorsque nous parlons des valeurs avant l'appel nous devons explicitement ajouter le mot-clé `\old`. Par exemple, pour le contrat de `inc` lorsque nous l'appelons depuis `foo`, `*x` dans la précondition est `*a` au label `L1`, alors que `*x` dans la postcondition est `*a` au label `L2`. Par conséquent, la pré et la postcondition doivent être considérées de manières légèrement différentes lorsque nous devons parler des positions mémoire mutables. Notons que pour la valeur du paramètre `x` lui-même, ce n'est pas le cas : cette valeur ne peut pas être modifiée par l'appel (du point de vue de l'appelant).

Maintenant, définissons le calcul de plus faible précondition d'un appel de fonction. Pour cela, notons :

— $\vec{v}$ un vecteur de valeurs $v_1, ..., v_n$ et v_i la i^{me} valeur,
— $\vec{t}$ les arguments fournis à la fonction lors de l'appel,
— $\vec{x}$ les paramètres dans la définition de la fonction,
— $\vec{a}$ les valeurs modifiées (vues de l'extérieur, une fois instanciées),
— $here(x)$ une valeur en postcondition,
— $old(x)$ une valeur en précondition.

Nous nommons `f:Pre` la précondition de la fonction, et `f:Post`, la postcondition.

$$\begin{aligned} wp(f(\vec{t}), Q) := \quad & \texttt{f:Pre}[x_i \leftarrow t_i] \\ \wedge \quad & \forall \vec{v}, \quad (\texttt{f:Post}[x_i \leftarrow t_i, here(a_j) \leftarrow v_j, old(a_j) \leftarrow a_j] \Rightarrow Q[here(a_j) \leftarrow v_j]) \end{aligned}$$

Nous pouvons détailler les étapes du raisonnement dans les différentes parties de cette formule.

Premièrement, notons que dans les pré et postconditions, chaque paramètre x_i est remplacé par l'argument correspondant ($[x_i \leftarrow t_i]$), comme nous l'avons dit juste avant, nous n'avons pas de question d'état mémoire à considérer ici puisque ces valeurs ne peuvent pas être changées par l'appel de fonction. Par exemple dans le contrat de `inc`, chaque occurrence de `x` serait remplacée par l'argument `a`.

Ensuite, dans la partie de notre formule qui correspond à la postcondition, nous pouvons voir que nous introduisons un $\forall \vec{v}$. Le but ici est de modéliser la possibilité que la fonction change la valeur des positions mémoire spécifiées par la clause `assigns` du contrat. Donc, pour chaque position potentiellement modifiée a_j (qui est, pour notre exemple d'appel à `inc`, `*(&a)`),

nous générons une valeur v_j qui représente sa valeur après l'appel. Mais si nous voulons vérifier que la postcondition nous donne le bon résultat, nous ne pouvons pas accepter *toute valeur* pour les positions mémoire modifiées, nous voulons celles *qui permettent de satisfaire la postcondition.*

Nous utilisons donc ces valeurs pour transformer la postcondition de la fonction et pour vérifier qu'elle implique la postcondition reçue en entrée du calcul de plus faible précondition. Nous faisons cela en remplaçant, pour chaque position mémoire modifiée a_j, sa valeur *here* avec la valeur v_j qu'elle obtient après l'appel ($here(a_j) \leftarrow v_j$). Finalement, nous devons remplacer chaque valeur *old* par sa valeur avant l'appel, et pour chaque $old(a_j)$, cette valeur est simplement a_j ($old(a_j) \leftarrow a_j$).

4.4.1.2. Formel - Exemple

Illustrons tout cela sur un exemple en appliquant le calcul de plus faible précondition sur ce petit code, en supposant le contrat précédemment proposé pour la fonction `swap`.

```
1  int a = 4 ;
2  int b = 2 ;
3
4  swap(&a, &b) ;
5
6  //@ assert a == 2 && b == 4 ;
```

Nous pouvons appliquer le calcul de plus faible précondition :

$$
\begin{aligned}
&wp(a = 4; b = 2; swap(\&a, \&b), a = 2 \wedge b = 4) = \\
&\quad wp(a = 4, wp(b = 2; swap(\&a, \&b), a = 2 \wedge b = 4)) = \\
&\quad wp(a = 4, wp(b = 2, wp(swap(\&a, \&b), a = 2 \wedge b = 4)))
\end{aligned}
$$

Et considérons séparément :

$$wp(swap(\&a, \&b), a = 2 \wedge b = 4)$$

Par la clause `assigns`, nous savons que les valeurs modifiées par la fonction sont $*(\&a) = a$ et $*(\&b) = b$. (nous raccourcissons *here* avec H et *old* avec O).

$$
\begin{aligned}
&\texttt{swap:Pre}[x \leftarrow \&a,\ y \leftarrow \&b] \\
&\quad \wedge \forall v_a, v_b, (\texttt{swap:Post}\ \ [x \leftarrow \&a,\ y \leftarrow \&b, \\
&\qquad\qquad H(*(\&a)) \leftarrow v_a,\ H(*(\&b)) \leftarrow v_b, \\
&\qquad\qquad O(*(\&a)) \leftarrow *(\&a),\ O(*(\&b)) \leftarrow *(\&b)]) \\
&\qquad \Rightarrow (H(a) = 2 \wedge H(b) = 4)[H(a)) \leftarrow v_a, H(b)) \leftarrow v_b])
\end{aligned}
$$

Pour la précondition, nous obtenons :

$$valid(\&a) \wedge valid(\&b)$$

Pour la postcondition, commençons par écrire l'expression depuis laquelle nous travaillerons avant de faire les remplacements (et sans la syntaxe de remplacement pour rester concis) :

$$H(*x) = O(*y) \land H(*y) = O(*x) \Rightarrow H(a) = 2 \land H(b) = 4$$

Remplaçons d'abord les pointeurs ($x \leftarrow \&a,\ y \leftarrow \&b$) :

$$H(*(\&a)) = O(*(\&b)) \land H(*(\&b)) = O(*(\&a)) \Rightarrow H(a) = 2 \land H(b) = 4$$

Puis les valeurs *here*, avec les valeurs quantifiées v_i ($H(a)) \leftarrow v_a, H(b)) \leftarrow v_b$) :

$$v_a = O(*(\&b)) \land v_b = O(*(\&a)) \Rightarrow v_a = 2 \land v_b = 4$$

Et les valeurs *old*, avec les valeurs avant l'appel ($O(*(\&a)) \leftarrow *(\&a),\ O(*(\&b)) \leftarrow *(\&b)$) :

$$v_a = *(\&b) \land v_b = *(\&a) \Rightarrow v_a = 2 \land v_b = 4$$

Nous pouvons maintenant simplifier cette formule en :

$$v_a = b \land v_b = a \Rightarrow v_a = 2 \land v_b = 4$$

Donc, $wp(swap(\&a, \&b), a = 2 \land b = 4)$ est :

$$P : valid(\&a) \land valid(\&b) \land \forall v_a, v_b, \quad v_a = b \land v_b = a \Rightarrow v_a = 2 \land v_b = 4$$

nous pouvons immédiatement simplifier cette formule en constatant que les propriétés de validité sont trivialement vraies (puisque les variables sont allouées sur la pile juste avant) :

$$P : \forall v_a, v_b, \quad v_a = b \land v_b = a \Rightarrow v_a = 2 \land v_b = 4$$

Maintenant, calculons $wp(a = 4, wp(b = 2, P)))$, en remplaçant d'abord b par 2 par la règle d'affectation :

$$\forall v_a, v_b, \quad v_a = 2 \land v_b = a \Rightarrow v_a = 2 \land v_b = 4$$

et ensuite a par 4 par la même règle :

$$\forall v_a, v_b, \quad v_a = 2 \land v_b = 4 \Rightarrow v_a = 2 \land v_b = 4$$

Cette dernière propriété est trivialement vraie, le programme est vérifié.

4.4.1.3. Contrats de fonction : check et admit

Tout comme les assertions et les invariants de boucle, les clauses `requires` et `ensures` ont des variantes `check` et `admit` permettant de seulement vérifier ou admettre ces propriétés.

Commençons par la clause `ensures`. Quand une fonction `f` a une clause `ensures`, elle doit être prouvée correcte lorsque l'on vérifie `f`, en revanche lorsque l'on appelle `f`, cette propriété est admise. Maintenant, si `f` a une clause `check ensures`, nous devons la vérifier quand nous vérifions `f`, mais nous l'*ignorons* lorsque `f` est appelée. À l'inverse, quand `f` a une clause `admit ensures`, nous ne la vérifions pas quand nous vérifions `f`, mais elle est admise quand `f` est appelée. Dans l'exemple suivant :

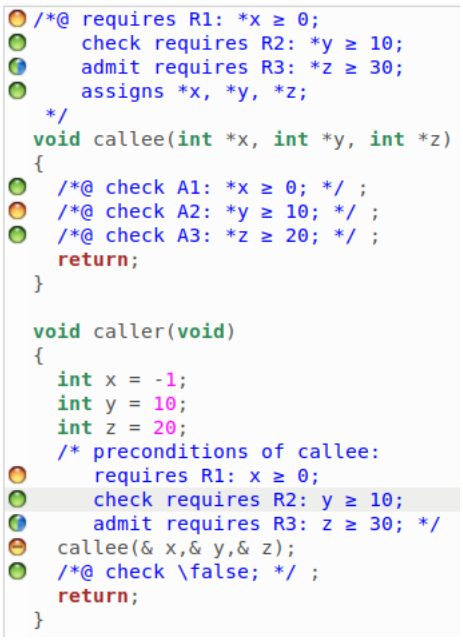
```
/*@ assigns *x, *y, *z ;
          ensures E1: *x >= 0 ;
    check ensures E2: *y >= 10 ;
    admit ensures E3: *z >= 30 ;
*/
void callee(int *x, int *y, int *z){
  *x = -1 ;
  *y = 10 ;
  *z = 20 ;
}

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;
}
```

Nous ne nous pouvons pas prouver la clause `E1` parce que `*x` est affectée à 1, mais nous pouvons prouver `A1`, car quand `f` est appelée, nous admettons que `E1` est vraie. Nous pouvons prouver la clause `E2` parce que `*y` est affectée à 10 qui est effectivement supérieur ou égal à 10. En revanche, nous ne pouvons pas prouver `A2` parce que la postcondition est ignorée à l'appel. Finalement, tandis que `*z` est affectée à 20, nous ne vérifions pas que `E3` est vraie (et elle ne l'est pas), pour autant, WP n'émet pas de *warning* à ce sujet : elle est admise, donc lors de l'appel à `f`, nous pouvons prouver que `z` est plus grand que 30, même si elle ne l'est pas.

Maintenant, présentons le comportement pour la clause `requires`. Quand une fonction `f` a une clause `requires`, elle est supposée vraie lorsque l'on vérifie `f`, et doit être vérifiée lorsque `f` est appelée. Quand c'est une clause `check requires`, la clause est vérifiée au point d'appel, mais elle n'est pas supposée vraie lorsque nous vérifions `f`. Finalement, quand `f` a une clause `admit requires`, elle est supposée vraie lors de la vérification de `f` et l'on n'essaie pas de la vérifier au point d'appel. Dans l'exemple suivant :

```
/*@       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 ;
}
```



Nous ne pouvons pas prouver que `R1` est vraie au point d'appel, mais nous admettons quand même qu'elle est vraie dans `callee`, donc nous pouvons prouver `A1`. Ensuite, même si `R2` est vérifiée au point d'appel, puisque nous l'ignorons pendant la vérification de `callee`, nous ne pouvons pas prouver que `A2` est vraie. Finalement, nous n'essayons pas de vérifier `R3` au point d'appel (et WP n'émet pas de *warning* à ce sujet même si elle est fausse), mais nous pouvons prouver que `A3` est vérifiée, car nous supposons que `R3` est vraie.

Il y a deuxième aspect important avec le comportement des clauses `requires`. Le lecteur attentif aura surement remarqué que dans l'exemple précédent, le `check \false` est vérifié. La raison est que les clauses `requires` normales et `admit` sont aussi supposée vraie localement après leur vérification. Par conséquent, même si dans cet exemple ces clauses sont fausses, nous introduisons localement « faux » dans le contexte de preuve.

4.4.1.4. Que devrions-nous garder en tête à propos des contrats de fonction ?

Les fonctions sont absolument nécessaires pour programmer modulairement, et le calcul de plus faible précondition est pleinement compatible avec cette idée, permettant de raisonner localement à propos de chaque fonction et donc de composer les preuves juste de la même manière que nous composons les appels de fonction.

Comme pense-bête, nous devrions toujours garder en tête le schéma suivant :

```
/*@       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...){
  // Ici, nous supposons que foo_R et foo_AR sont vraies


  // Ici, nous devons prouver que bar_R et bar_CR sont vraies
  bar(some parameters ...) ;
  // Ici, nous supposons que bar_E et bar_AE sont vraies


  // Ici, nous devons prouver que foo_E et foo_CE sont vraies
  return ... ;
}

```

Notons qu'à propos du dernier commentaire, en calcul de plus faible précondition, l'idée est plutôt de montrer que notre précondition est assez forte pour assurer que le code nous amène à la postcondition. Cependant, premièrement, cette vision est plus facile à comprendre et deuxièmement, un greffon comme WP (et comme n'importe quel outil réaliste pour la preuve de programme) ne suit pas strictement un calcul de plus faible précondition, mais une manière fortement optimisée de calculer les conditions de vérification qui ne suit pas exactement les mêmes règles.

4.4.1.5. Le cas particulier de la clause `exits`

En C, il est possible d'appeler la fonction `exit` pour arrêter l'exécution du programme avec un code d'erreur particulier. Dans un tel cas, le code qui suit l'appel ne sera pas exécuté et l'instruction `return` ne sera jamais atteinte. Par conséquent, lorsqu'une fonction *quitte*, la

postcondition est toujours vérifiée, car elle inatteignable (de la même manière que lorsqu'une fonction n'appelle pas `exit`, la clause `exits` peut être n'importe quoi) :

```
#include <stdlib.h>

/*@ exits \true ;
    ensures \false ;
*/
void this_function_exits(void){
  exit(1);
}
```

Ici, nous avons indiqué que `\true` doit être vrai quand la fonction quitte, mais en fait, nous pouvons indiquer n'importe quelle propriété d'intérêt. Par exemple, la valeur qui a été passée à `exit` :

```
#include <stdlib.h>

/*@ exits \exit_status == 1 ;
    ensures \false ;
*/
void this_function_exits(void){
  exit(1);
}
```

Bien sûr, quand on appelle une fonction qui peut quitter, ce risque est propagé :

```
#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();
}
```

[Formel] Calcul de WP Lorsque nous voulons prouver la postcondition d'une fonction, nous commençons le calcul depuis la postcondition. Donc, nous commençons depuis l'unique instruction `return` créée par Frama-C (bien que ce soit exactement comme commencer le calcul par toutes les instructions `return` du programme d'origine), et nous raisonnons en arrière le long du programme. Lorsque nous rencontrons un appel de fonction, nous utilisons son contrat à l'aide de la règle précédemment expliquée et la postcondition que nous considérons à ce point de raisonnement est celle que l'on trouve dans les clauses `ensures`.

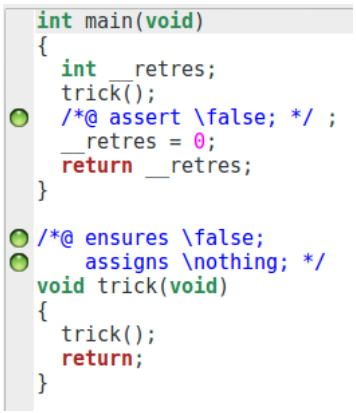
Pour prouver les clauses `exits`, c'est différent, mais pas tant que cela. Au lieu de commencer le calcul depuis les instructions `return`, nous commençons un nouveau calcul à partir de chaque appel de fonction (puisqu'elle pourrait appeler `exit`) et à cet appel, nous utilisons la clause `exits` comme postcondition. Ensuite, lorsque nous continuons le raisonnement arrière, nous utilisons les clauses `ensures` comme d'habitude puisque si nous avons atteint l'appel

de fonction depuis lequel nous avons démarré le calcul, cela veut dire que les appels qui ont précédé n'ont pas appelé `exit`.

4.4.2. Fonctions récursives

De la même manière que nous pouvons prouver n'importe quoi à propos de la postcondition d'une fonction qui contient une boucle infinie, il est très simple de prouver n'importe quoi à propos de la postcondition d'une fonction qui produit une récursion infinie :

```
/*@
  assigns \nothing ;
  ensures \false ;
*/
void trick(){
  trick() ;
}

int main(){
  trick();
  //@ assert \false ;
}
```



Nous pouvons voir que la fonction et l'assertion sont prouvées. Et, effectivement, la preuve est correcte : nous considérons une correction partielle (puisque nous ne pouvons pas prouver la terminaison), et cette fonction ne termine pas. Toute assertion suivant cette fonction est vraie : elle est inatteignable. Mais à nouveau, nous avons un chien de garde : la clause `terminates` n'est pas prouvée.

Une question que l'on peut alors se poser est : que peut-on faire dans un tel cas ? Nous pouvons à nouveau utiliser une mesure (comme dans la Section 4.2.3.1) pour borner la profondeur de récursion. En ACSL, c'est le rôle de la clause `decreases` :

```
/*@ requires n >= 0 ;
    decreases n ;
*/
void ends(int n){
  if(n > 0) ends(n-1);
}
```

Tout comme la clause `loop variant`, la clause `decreases` exprime une notion de mesure. Donc, une expression entière positive (ou une expression équipée d'une relation) qui décroît strictement. Tandis que la clause `loop variant` représente une borne supérieure sur le nombre d'itérations, la clause `decreases` représente une borne supérieure sur la profondeur de récursion (et pas le nombre d'appels de fonctions) :

```
1  /*@ requires n >= 0 ;
2      decreases n ;
3  */
4  void ends_2(int n){
5    if(n > 0) ends_2(n-1); // Ok: 0 <= n && n-1 < n
6    if(n > 0) ends_2(n-1); // Ok: 0 <= n && n-1 < n, no need to be less than call on l.5
7  }
```

Notons que, comme pour la clause `loop variant` où nous vérifions les propriétés de l'expression seulement lorsqu'une nouvelle itération peut apparaître, les propriétés de l'expression d'une clause `decreases` ne sont vérifiés que lorsque la fonction est à nouveau appelée. Ce qui signifie que lorsque l'on atteint la profondeur maximale d'appel, l'expression peut être négative :

```
1  /*@ requires n >= -10 ;
2      decreases n ; */
3  void go_negative(int n){
4    if(n >= 0) go_negative(n-10); // if n is 0, this call is rec(-10), it is fine
5  }
```

Nous pouvons voir la condition de vérification générée pour l'exemple précédent en désactivant la simplification du but (option `-wp-no-let`, ici, nous avons supprimé les informations redondantes de la capture d'écran) :

```
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)).
```

Ici, la condition `n - 10 < n` est reformulée en `n_1 (= n) <= 9 + n` à cause de la normalisation des formules.

Lorsque nous sommes en train de prouver la correction de la clause `decreases` d'une fonction (pour un appel de cette fonction), l'expression est évaluée pour deux entités : la fonction sous preuve et l'instruction d'appel. L'état d'évaluation de l'expression pour la fonction sous preuve est `Pre`, l'état d'évaluation de l'expression pour l'appel est `Here`. La valeur de l'expression lors de l'appel doit être plus petite que la valeur de l'expression associée à l'état `Pre`.

```
/*@ 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)
  }
}
```

Bien sûr, des fonctions récursives peuvent être mutuellement récursives. Par conséquent, la clause `decreases` peut être utilisée pour borner la profondeur d'appels récursifs dans cette situation. Cependant, nous ne voulons le faire *que* pour les fonctions qui sont effectivement dans l'ensemble de fonctions mises en jeu dans la récursion. Pour cela, WP calcule les composantes fortement connexes depuis l'ensemble complet des fonctions, dans la terminologie ACSL, on appelle ces composants des *clusters*.

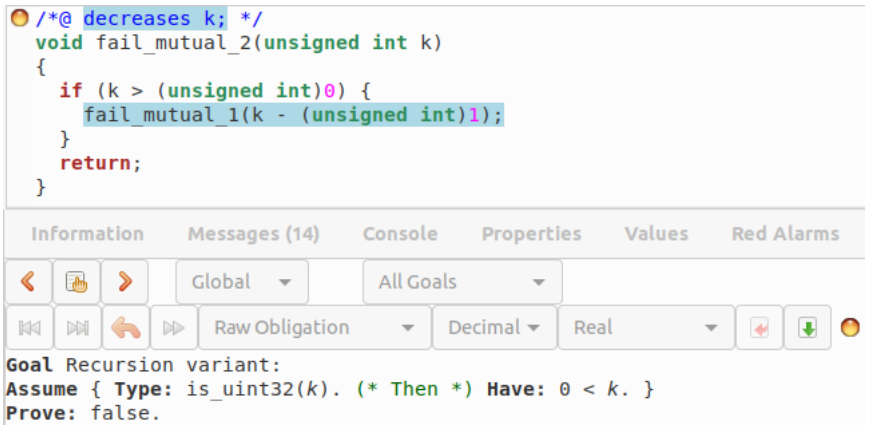
Nous pouvons donc préciser un peu comment est vérifiée la correction d'une clause `decreases`. Lorsqu'une fonction est (mutuellement) récursive, sa spécification doit être équipée avec une telle clause afin de prouver qu'elle termine. Vérifier que la clause `decreases` d'une fonction `f` donne une mesure de sa profondeur de récursion consiste à vérifier, pour chaque appel à une fonction *qui appartient au même cluster que* `f`, que l'expression fournie par la clause est effectivement positive et décroissante. Et par conséquent, aucune condition de vérification n'est générée lorsqu'une fonction récursive n'appartenant pas au même *cluster* est appelée :

```
//@ 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
}
```

Notons que le calcul des *clusters* est basé sur un critère syntaxique. Pour le moment, nous ne parlons par des pointeurs de fonctions dans ce tutoriel, le lecteur peut se référer à la documentation de l'option `-wp-dynamic` dans le manuel de WP.

Finalement, si une fonction d'un *cluster* ne fournit pas de clause `decreases`, une condition de vérification `\false` est générée et WP émet un avertissement ;

```
1  void fail_mutual_1(unsigned k);
2
3  //@ decreases k ;
4  void fail_mutual_2(unsigned k){
5    if(k > 0) fail_mutual_1(k-1);
6  }
7  /* Warning:
8    [wp] file.c:5: Warning:
9      Missing decreases clause on recursive function fail_mutual_1, call must be unreachable
10 */
11
12 void fail_mutual_1(unsigned k){
13   if(k > 0) fail_mutual_2(k-1);
14 }
```



i

Comme pour les clauses `loop variant`, il est possible de fournir une autre relation (voir Section 4.2.3.2) pour une clause `decreases`. La syntaxe est :

```
1  //@ decreases <term> for <Relation> ;
```

4.4.3. Spécifier et prouver la terminaison des fonctions

Une propriété désirable pour une fonction est souvent qu'elle doit terminer. En effet, dans la plupart des programmes, toutes les fonctions doivent terminer, et quand ce n'est pas le cas, il est très probable qu'en réalité, une seule fonction puisse boucler à l'infini (donc la quasi-totalité des fonctions dans le programme terminent).

4.4.3.1. Syntaxe et description

ACSL fournit une clause `terminates` pour spécifier qu'une fonction doit terminer quand une propriété particulière est vérifiée en précondition. La syntaxe est la suivante :

```
//@ terminates condition ;
void function(void){
  // ...
}
```

Ici, le contrat énonce que quand `condition` est vérifiée en précondition, alors la fonction doit terminer. Par exemple, la fonction `abs` doit toujours terminer :

```
/*@ requires x > INT_MIN ;
    terminates \true ;
*/
int abs(int x){
  return (x < 0) ? -x : x ;
}
```

alors que pour la fonction `main_loop` cela peut ne pas être le cas (notons qu'avec les options par défaut, le variant de la boucle n'est pas vérifié, nous expliquerons pourquoi plus tard) :

```
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
  }
}
```

Soulignons que la fonction *pourrait ne pas* terminer, elle n'est pas *forcée de boucler à l'infini*. Par exemple, dans la fonction suivante, la clause `terminates` est vérifiée, car *lorsque la condition de terminaison est vérifiée* (jamais), la fonction termine (toujours) :

```
//@ terminates \false ;
void may_not_terminate(void){

}
```

i

Si l'on veut vraiment vérifier qu'une fonction ne termine jamais, nous pouvons spécifier que la fonction ne retourne jamais, et ne quitte (`exits`) jamais. C'est-à-dire : les postconditions associées à ces types de terminaison sont inatteignables :

```
unsigned counter ;

/*@ terminates \false;
    exits \false;
    ensures \false;
*/
void does_not_terminate(void){
  //@ loop assigns counter ;
  while(1){
    counter++;
  }
}
```

En ACSL, il est spécifié que lorsqu'une fonction n'a pas de clause `terminates`, le comportement par défaut `terminates \true`. Ce comportement est automatiquement activé par WP au niveau du noyau de Frama-C. Donc, quand WP démarre la vérification d'une fonction, il demande au noyau de générer ces annotations. Ce comportement peut être désactivé **pour les fonctions définies par l'utilisateur** à l'aide de l'option :

— `-generated-spec-custom terminates:skip`

4.4.3.2. Vérification

Vérifier qu'une fonction termine demande de vérifier que toutes les instructions atteignables d'une fonction terminent. Les affectations terminent trivialement, donc nous n'avons rien de particulier à vérifier à leur sujet. Une instruction conditionnelle termine si toutes les instructions des différentes branches (atteignables) terminent, donc nous avons juste à vérifier que ces instructions terminent (ou sont inatteignables). Les instructions restantes sont les boucles et les appels de fonctions. Donc nous devons vérifier que :

— toutes les boucles ont une clause `loop variant` (vérifiée),
— toutes les fonctions appelées terminent avec leurs paramètres d'entrée,
— toutes les fonctions récursives ont une clause `decreases` (vérifiée),
— (ou qu'il n'y a pas de boucles ou d'appels atteignables dans la fonction).

Cependant, nous devons seulement faire ces vérifications lorsque la condition de terminaison de la fonction est vérifiée. Voyons donc maintenant quelles sont les conditions de vérification générées et quand WP les génère.

Lorsqu'une fonction a une clause `terminates`, WP visite toutes les instructions de la fonction et collecte les boucles pour lesquelles aucune clause `loop variant` n'est spécifiée et les appels de fonctions. Si aucune instruction de ce type n'est présente, la fonction termine trivialement.

```
/*@ requires x > -2147483647 - 1;
    terminates \true; */
int abs(int x)
{
```

Information Messages (3) Console

[wp] [CFG] Goal abs_terminates : Valid (Trivial)

S'il existe de telles instructions, leur terminaison doit être vérifiée *lorsque* la fonction doit terminer (disons, lorsque `T`). Donc, les conditions de vérification sont de la forme `\at(T, Pre)` ⇒ `<statement termination>`. Notons que la prémisse de l'implication doit être évaluée dans l'état `Pre`. Donc, dans ce code :

```
void call(int r);

//@ terminates r > 0 ;
void simple(int r){
  r -- ;
  call(r);
}
```

Même si `r` a été décrémentée, la vérification de la condition de la terminaison de `call(r)` est faite en prenant `\at(r, Pre) > 0` comme prémisse. Nous verrons dans la section suivante la condition de vérification complète générée pour un appel.

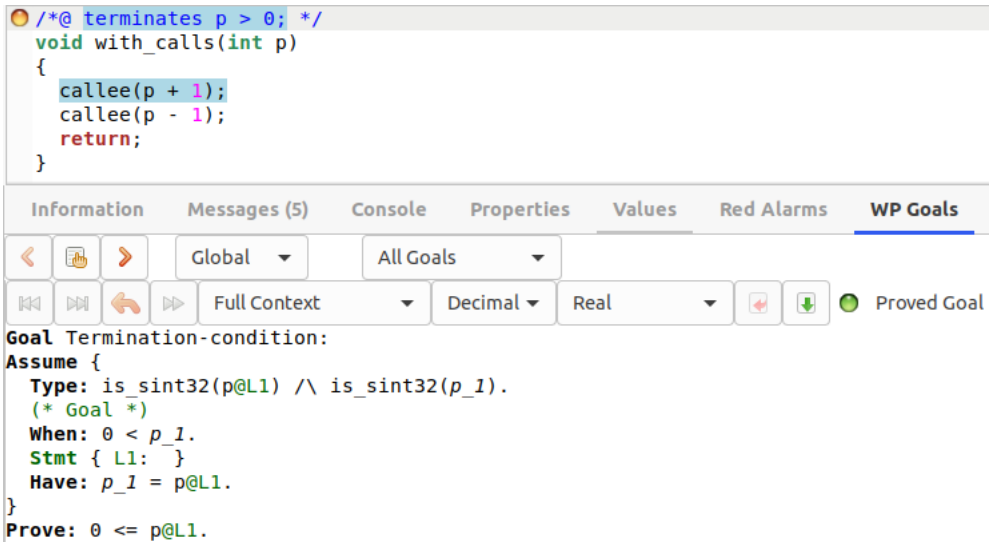
Notons aussi que cela signifie que lorsque nous atteignons un point de programme où `T` est fausse, la condition de vérification est toujours vérifiée :

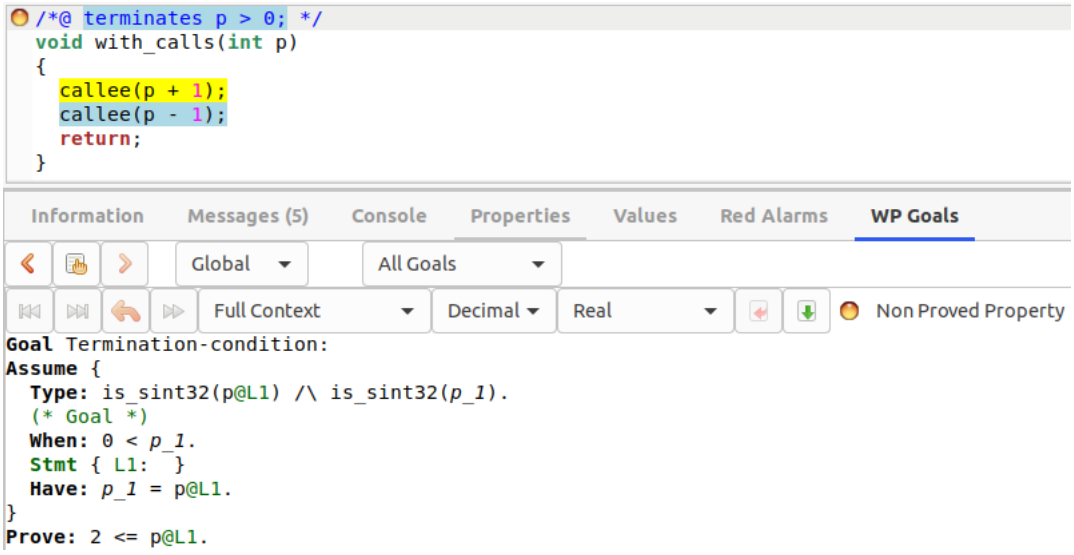
```
//@ terminates r > 0 ;
void cond_may_not_terminate(int r){
  if(r <= 0){ // if we enter here, r > 0 is false at Pre
    // here statements are not force to terminate: FALSE ==> P is always true
    while(1){}
  }
}
```

4.4.3.2.1. Appel de fonction Pour vérifier qu'un appel de fonction termine, nous vérifions lorsque la fonction est appelée, que la condition de terminaison est vraie. Par exemple, à partir du programme suivant :

```
//@ 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);
}
```

Nous obtenons les conditions de vérification suivante (en utilisant l'option `-wp-no-let` pour désactiver les simplifications du but) :





Où la terminaison du premier appel est en effet vérifiée, tandis que la terminaison du second ne l'est pas.

4.4.3.2.2. Variant de boucle Lorsqu'une fonction contient une boucle qui n'a pas de clause `loop variant`, sa terminaison ne peut pas être vérifiée, donc WP nous demande de vérifier que lorsque la condition de terminaison est vérifiée, la boucle est inatteignable.

```
1 //@ terminates value > 0 ;
2 void with_loop(int value){
3   if(value <= 0){
4     //@ loop assigns \nothing ;
5     while(1){
6       // code
7     }
8   }
9 }
```

Dans le code précédent, la boucle n'a pas de clause `loop variant`, donc nous devons vérifier `value > 0 ==> \false` à la position de la boucle, ce qui est bon : le code est inatteignable lorsque `value > 0`.

Finalement, lorsqu'une boucle a une clause `loop variant`, elle doit être vérifiée *seulement quand* la fonction doit terminer. Donc, si nous reprenons l'exemple présenté au début de cette section :

```
1  int debug_steps = -1 ;
2
3  /*@ requires debug_steps >= -1 ;
4      terminates debug_steps >= 0 ;
5   */
6  void main_loop(void){
7    /*@ loop invariant \at(debug_steps, Pre) == -1
8                     || 0 <= debug_steps <= \at(debug_steps, Pre) ;
9        loop assigns debug_steps ;
10       loop variant debug_steps ; */
11   while(1){
12     if(debug_steps == 0) return ;
13     else if(debug_steps > 0) debug_steps -- ;
14
15     // actual code
16   }
17 }
```

Nous devons vérifier que le variant de boucle est une valeur positive et décroissante seulement lorsque `debug_steps` n'est pas -1. Par contre, ce n'est pas le comportement par défaut de WP (qui cherche toujours à montrer que les variants de boucles sont corrects par défaut), cela peut être activé en utilisant l'option `-wp-variant-with-terminates` et dans ce cas, la fonction est entièrement vérifiée :

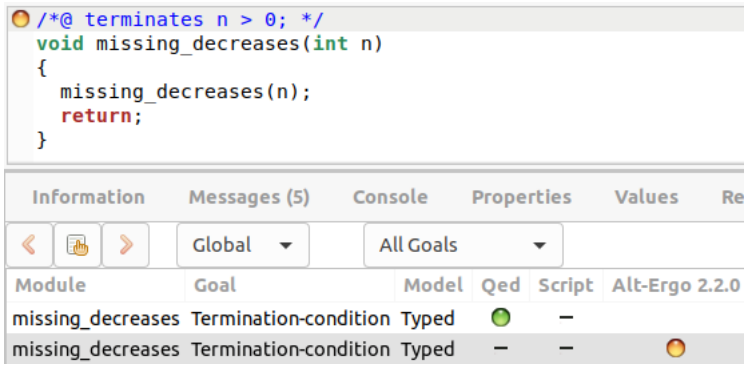
```
/*@ requires debug_steps ≥ -1;
    terminates debug_steps ≥ 0; */
void main_loop(void)
{
  /*@ loop invariant
        \at(debug_steps,Pre) ≡ -1 ∨
        (0 ≤ debug_steps ≤ \at(debug_steps,Pre));
      loop assigns debug_steps;
      loop variant debug_steps;
  */
  while (1) {
    {
      if (debug_steps == 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. Récursion Une fonction récursive doit avoir une clause `decreases` lorsque sa spécification indique qu'elle termine. Si une telle clause est manquante, une condition de véri-

fication `\false` est générée.

```
//@ terminates n > 0 ;
void missing_decreases(int n){
  missing_decreases(n);
}
```

Notons que ce code génère deux conditions de vérification :



La première correspond à la règle associée à la terminaison d'un appel de fonction précédemment présentée. La seconde correspond au fait que la clause `decreases` est manquante et donc non vérifiée.

À nouveau en ACSL, la vérification de la clause `decreases` n'est requise que lorsque la condition de terminaison est vérifiée dans l'état `Pre`. Le comportement de WP sur cet aspect est similaire au cas des variants de boucle. Par défaut, la vérification est toujours demandée, le comportement ACSL est activé via l'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. Exercices

4.4.4.1. Expliquer les échecs de preuve

Dans le programme suivant, quelques conditions de vérification ne sont pas vérifiées :

```
#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 ;
}
```

Expliquer pourquoi elles ne sont pas vérifiées et proposer une manière de corriger la spécification pour que tout soit vérifié.

4.4.4.2. Expliquer les résultats des preuves de terminaison

Dans le programme suivant :

```
#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){
```

```
21    return abs(b - a) ;
22  }
23
24  /*@ requires \valid_read(a + (0 .. len-1)) && \valid_read(b + (0 .. len-1));
25      requires \valid(result + (0 .. len-1));
26      requires \separated(a + (0..len-1), b + (0..len-1), result + (0..len-1));
27      requires \forall integer i ; 0 <= i < len ==> 0 <= b[i] - a[i] <= INT_MAX ;
28
29      terminates \true ;
30
31      assigns result[0 .. len-1] ;
32
33      ensures \forall integer i ; 0 <= i < len ==> a[i] + result[i] == b[i] ;
34  */
35  void distances(int const* a, int const* b, int* result, size_t len){
36    /*@ loop invariant 0 <= i <= len ;
37        loop invariant \forall integer k ; 0 <= k < i ==> a[k] + result[k] == b[k] ;
38        loop assigns i, result[0 .. len-1];
39    */
40    for(size_t i = 0 ; i < len ; ++i){
41      result[i] = distance(a[i], b[i]);
42    }
43  }
44
45  struct client ;
46
47  /*@ requires \valid(c) ;
48      terminates \true ;
49      assigns *c ;
50  */
51  void initialize(struct client *c);
52
53  /*@ requires \valid(c) ;
54      terminates \false ;
55      assigns *c ;
56      ensures \result \in { 0 , 1 } ;
57  */
58  int connect(struct client *c);
59
60  /*@ requires \valid(c);
61      terminates \true ;
62      assigns *c ;
63      ensures \result \in { 0 , 1 } ;
64  */
65  int prepare(struct client *c){
66    initialize(c);
67    return connect(c);
68  }
69
70  /*@ terminates x > 0 ; */
71  void terminates_f1(int x){
72    if(x <= 0) for(;;);
73  }
74
75  /*@ requires x > 0 ;
76      terminates \true ;
77  */
78  void terminates_f2(int x){
79    if(x <= 0) for(;;);
80  }
```

Expliquer pourquoi les clauses de terminaison sont vérifiées ou pas. Modifier les spécifications pour que tout soit vérifié.

4.4.4.3. Recherche

Spécifier et prouver la fonction de recherche récursive suivante :

```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);
}
```

La spécification doit inclure la condition de terminaison.

4.4.4.4. Somme des entiers

Le programme suivant calcule la somme des entiers entre `fst` et `lst` :

```c
int sum(int fst, int lst, int acc){
  if (fst == lst) return acc ;
  else return sum(fst+1, lst, fst+acc) ;
}
```

Prouver que la fonction termine. La preuve que la fonction calcule bien la somme des entiers et l'absence d'erreurs à l'exécution ne sont pas demandées.

4.4.4.5. Puissance

Le programme suivant calcule `x` à la puissance `n` :

```c
/*@
  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) ;
}
```

Prouver que la fonction termine. La preuve que la fonction calcule bien la puissance et l'absence d'erreurs à l'exécution ne sont pas demandées.

Dans cette partie nous avons pu voir comment se traduisent les affectations et les structures de contrôle d'un point de vue logique. Nous nous sommes beaucoup attardés sur les boucles parce que c'est là que se trouvent la majorité des difficultés lorsque nous voulons spécifier et prouver un programme par vérification déductive, les annotations ACSL qui leur sont spécifiques nous permettent d'exprimer le plus précisément possible leur comportement.

Pour la suite, nous nous attarderons plus précisément sur les constructions que nous offre le langage ACSL du côté de la logique. Elles sont très importantes parce que ce sont elles qui vont nous permettre de nous abstraire du code pour avoir des spécifications plus compréhensibles et plus aisément prouvables.

5. ACSL - Propriétés

Depuis le début de ce tutoriel, nous avons vu divers prédicats et fonctions logiques qui sont fournis par défaut en ACSL : `\valid`, `\valid_read`, `\separated`, `\old` et `\at`. Il en existe bien sûr d'autres, mais nous ne les présenterons pas un à un ; le lecteur pourra se référer à la documentation (ACSL implementation) pour cela (à noter : tout n'est pas nécessairement supporté par WP).

ACSL permet de faire plus que « simplement » spécifier notre code. Nous pouvons définir nos propres prédicats, fonctions, relations, etc. Le but est de pouvoir abstraire nos spécifications. Cela nous permet de les factoriser (par exemple en définissant ce qu'est un tableau valide), ce qui a deux effets positifs : d'abord nos spécifications deviennent plus lisibles donc plus faciles à comprendre, mais cela permet également de réutiliser des preuves déjà faites et donc de faciliter la preuve de nouveaux programmes.

5.1. Types primitifs supplémentaires

ACSL fournit différents types logiques qui permettent d'écrire des propriétés dans un monde plus abstrait, plus mathématique. Parmi les types qui peuvent être utiles, certains sont dédiés aux nombres et permettent d'exprimer des propriétés ou des fonctions sans avoir à nous soucier des contraintes dues à la taille en mémoire des types primitifs du C. Ces types sont `integer` et `real`, qui représentent respectivement les entiers mathématiques et les réels mathématiques (pour ces derniers, la modélisation est aussi proche que possible de la réalité, mais la notion de réel ne peut pas être parfaitement représentée).

Par la suite, nous utiliserons souvent des entiers à la place des classiques `int` du C. La raison est simplement que beaucoup de propriétés sont vraies quelle que soit la taille de l'entier (au sens C, cette fois) en entrée.

En revanche, nous ne parlerons pas de `real` VS `float/double`, parce que cela induirait que nous parlions de preuve de programmes avec du calcul en virgule flottante et que nous n'en parlerons pas ici. Par contre, ce tutoriel en parle : Introduction à l'arithmétique flottante
.

5.2. Prédicats

Un prédicat est une propriété portant sur des objets et pouvant être vraie ou fausse. En résumé, des prédicats, c'est ce que nous écrivons depuis le début de ce tutoriel dans les clauses de nos contrats et de nos invariants de boucle. ACSL permet de créer des versions nommées de ces prédicats, à la manière d'une fonction booléenne en C par exemple, à la différence près que

les prédicats (ainsi que les fonctions logiques que nous verrons par la suite) doivent être pures, c'est-à-dire qu'elles ne peuvent pas produire d'effets de bord en modifiant des valeurs pointées par exemple.

Ces prédicats peuvent prendre un certain nombre de paramètres. En plus de cela, ils peuvent également recevoir un certain nombre de *labels* (au sens C du terme) qui permettront d'établir des relations entre divers points du code.

5.2.1. Syntaxe

Les prédicats sont, comme les spécifications, introduits via des annotations. La syntaxe est la suivante :

```
1  /*@
2  predicate nom_du_predicat { Lbl0, Lbl1, ..., LblN }(type0 arg0, type1 arg1, ..., typeN
       argN) =
3  //une relation logique entre toutes ces choses.
4  */
```

Nous pouvons par exemple définir le prédicat nous disant qu'un entier en mémoire n'a pas changé entre deux points particuliers du programme :

```
1 /*@
2   predicate unchanged{L0, L1}(int* i) =
3     \at(*i, L0) == \at(*i, L1);
4 */
```

Il faut bien garder en mémoire que le passage se fait, comme en C, par valeur. Nous ne pouvons pas écrire ce prédicat en passant directement `i` :

```
1     /*@
2       predicate unchanged{L0, L1}(int i) =
3         \at(i, L0) == \at(i, L1);
4     */
5
```

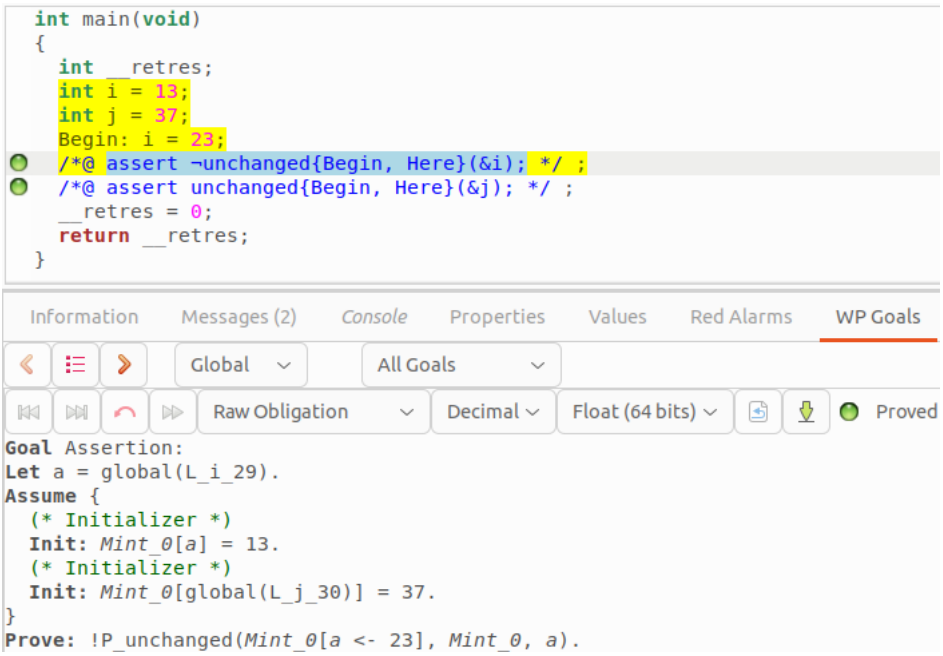
car `i` est juste une copie de la variable reçue en paramètre.

Nous pouvons par exemple vérifier ce petit code :

```
1 int main(void){
2   int i = 13;
3   int j = 37;
4
5  Begin:
6   i = 23;
7
```

```
8    //@assert ! unchanged{Begin, Here}(&i);
9    //@assert   unchanged{Begin, Here}(&j);
10 }
```

Nous pouvons également regarder les buts générés par WP et constater que, même s'il subit une petite transformation syntaxique, le prédicat n'est pas déroulé par WP. Ce sera au prouveur de déterminer s'il veut raisonner avec.



Comme nous l'avons dit plus tôt, une des utilités des prédicats et fonctions (que nous verrons un peu plus tard) est de rendre plus lisible nos spécifications et de les factoriser. Un exemple est l'écriture d'un prédicat pour la validité en lecture/écriture d'un tableau sur une plage particulière. Cela nous évite d'avoir à réécrire l'expression en question qui est moins compréhensible au premier coup d'œil.

```
1  /*@
2    predicate valid_range_rw(int* t, integer n) =
3      n >= 0 && \valid(t + (0 .. n-1));
4
5    predicate valid_range_r(int* t, integer n) =
6      n >= 0 && \valid_read(t + (0 .. n-1));
7  */
8
9  /*@
10   requires 0 < length;
11   requires valid_range_r(array, length);
12   //...
13 */
14 int* search(int* array, size_t length, int element);
```

Dans cette portion de spécification, le *label* pour les prédicats n'est pas précisé, ni pour leur création, ni pour leur utilisation. Pour la création, Frama-C en ajoutera automatiquement un dans la définition du prédicat. Pour l'appel, le *label* passé sera implicitement `Here`. La non-déclaration du *label* dans la définition n'interdit pour autant pas de passer explicitement un *label* lors de l'appel.

Bien entendu, les prédicats peuvent être déclarés dans des fichiers *headers* afin de produire une bibliothèque d'utilitaires de spécifications par exemple.

5.2.1.1. Surcharger des prédicats

Il est possible de surcharger les prédicats tant que les types des paramètres sont différents ou que le nombre de paramètres change. Par exemple, nous pouvons redéfinir `valid_range_r` comme un prédicat qui prend en paramètre à la fois le début et la fin de la plage considérée. Ensuite, nous pouvons écrire une surcharge qui utilise le prédicat précédent pour le cas particulier des plages qui commencent à 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

Une autre utilité importante des prédicats est de définir l'état logique de nos structures quand les programmes se complexifient. Nos structures doivent généralement avoir un invariant (encore) que chaque fonction de manipulation devra maintenir pour assurer que la structure sera toujours utilisable et qu'aucune fonction ne commettra de bavure.

Cela permet notamment de faciliter la lecture des spécifications. Par exemple, nous pourrions poser les spécifications nécessaires à la sûreté d'une pile de taille limitée. Et cela donnerait quelque chose comme :

```
#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;
```

```
18    ensures valid_stack_int(s) && empty_stack_int(s);
19  */
20  void initialize(struct stack_int* s);
21
22  /*@
23    requires valid_stack_int(s) && !full_stack_int(s);
24    assigns  *s;
25    ensures valid_stack_int(s);
26  */
27  void push(struct stack_int* s, int value);
28
29  /*@
30    requires valid_stack_int(s) && !empty_stack_int(s);
31    assigns \nothing;
32  */
33  int  top(struct stack_int* s);
34
35  /*@
36    requires valid_stack_int(s) && !empty_stack_int(s);
37    assigns *s;
38    ensures valid_stack_int(s);
39  */
40  void pop(struct stack_int* s);
41
42  /*@
43    requires valid_stack_int(s);
44    assigns \nothing;
45    ensures \result == 1 <==> empty_stack_int(s);
46  */
47  int  is_empty(struct stack_int* s);
48
49
50  /*@
51    requires valid_stack_int(s);
52    assigns \nothing;
53    ensures \result == 1 <==> full_stack_int(s);
54  */
55  int  is_full(struct stack_int* s);
```

(Notons qu'ici, nous ne fournissons pas la définition des prédicats, car ce n'est pas l'objet de cet exemple. Le lecteur pourra considérer ceci comme un exercice.)

Ici, la spécification n'exprime pas de propriétés fonctionnelles. Par exemple, rien ne nous spécifie que lorsque nous faisons un *push* d'une valeur puis demandons *top*, nous auront effectivement cette valeur. Mais elle nous donne déjà tout ce dont nous avons besoin pour produire un code où, à défaut d'avoir exactement les résultats que nous attendons (des comportements tels que « si j'empile une valeur v, l'appel à `top` renvoie la valeur v », par exemple), nous pouvons au moins garantir que nous n'avons pas d'erreur d'exécution (à condition de poser une spécification correcte pour nos prédicats et de prouver les fonctions d'utilisation de la structure).

5.2.3. Exercices

5.2.3.1. Les jours du mois

Reprendre la solution de l'exercice 3.4.1.1, écrire un prédicat pour exprimer qu'une année est bissextile et modifier le contrat de façon à l'utiliser.

5.2.3.2. Caractères alpha-numériques

Reprendre la solution de l'exercice 3.4.1.2 à propos des caractères alpha-numériques. Écrire des prédicats pour exprimer qu'un caractère est une majuscule, une autre pour les minuscules et un dernier pour les chiffres. Adapter les contrats des différentes fonctions en les utilisant.

5.2.3.3. Maximum de 3 valeurs

La fonction suivante retourne le maximum de 3 valeurs d'entrée :

```
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 ;
}
```

Écrire un prédicat qui exprime qu'une valeur est l'une de trois valeurs pointées à un état mémoire donné :

```
  /*@
    predicate one_of{L}(int value, int *a, int *b, int *c) =
      // ...
  */
```

Utiliser la notation ensembliste. Écrire un contrat pour la fonction et prouver qu'elle le vérifie.

5.2.3.4. Recherche dichotomique

Reprendre la solution de l'exercice 4.3.4.3 à propos de la recherche dichotomique utilisant des indices non signés. Écrire un prédicat qui exprime qu'une plage de valeur est triée entre `begin` et `end` (exclu). Écrire une surcharge de ce prédicat pour rendre `begin` optionnel avec une valeur par défaut à 0. Écrire un prédicat qui vérifie si un élément est dans un tableau pour des indices compris entre `begin` et `end` (exclu), à nouveau, écrire une surcharge qui rend la première borne optionnelle.

Utiliser ces prédicats pour simplifier le contrat de la fonction. Notons que les clauses `assumes` des deux comportements devraient être modifiées.

5.2.3.5. Chercher et remplacer

Reprendre l'exemple 4.3.3.2, à propos de la fonction « chercher et remplacer ». Écrire des prédicats qui exprime qu'une plage de valeurs dans un tableau pour des indices compris entre `begin` et `end` (exclu), les valeurs :

- restent inchangées entre deux états mémoire,
- sont remplacées par une valeur si elles sont égales à une valeur donnée, sinon sont laissées inchangées.

Surcharger les deux prédicats de manière à rendre la première borne optionnelle. Utiliser les prédicats obtenus pour simplifier le contrat et l'invariant de boucle de la fonction.

5.3. Fonctions logiques

Les fonctions logiques nous permettent de décrire des fonctions mathématiques qui contrairement aux prédicats nous permettent de renvoyer différents types. Elles ne seront utilisables que dans les spécifications. Cela nous permet d'une part, de les factoriser, et d'autre part de définir des opérations sur les types `integer` et `real` qui ne peuvent pas déborder contrairement aux types machines.

Comme les prédicats, elles peuvent recevoir divers *labels* et valeurs en paramètre.

5.3.1. Syntaxe

Pour déclarer une fonction logique, l'écriture est la suivante :

```
/*@
  logic type_retour ma_fonction{ Label0, ..., LabelN }( type0 arg0, ..., typeN argN ) =
    formule mettant en jeu les arguments ;
*/
```

Nous pouvons par exemple décrire une fonction affine générale du côté de la logique :

```
/*@
  logic integer ax_b(integer a, integer x, integer b) =
    a * x + b;
*/
```

Elle peut nous servir à prouver le code de la fonction suivante :

```
/*@
  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;
}
```

Le code est bien prouvé, mais les contrôles d'*overflow*, eux, ne le sont pas. Nous pouvons ajouter la contrainte en précondition que le calcul doit entrer dans les bornes d'un entier :

```
/*@
  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;
}
```

Certains contrôles de débordement ne sont pas encore prouvée. En effet, tandis que la borne fournie pour `x` par notre fonction logique est définie pour le calcul complet, elle ne nous dit rien à propos des calculs intermédiaires. Par exemple ici, le fait `3 * x + 4` ne soit pas inférieur à `INT_MIN` ne nous garantit pas que c'est aussi le cas pour `3 * x`. Nous pouvons imaginer deux manières différentes de résoudre le problème, ce choix doit être guidé par les besoins du projet.

Nous pouvons augmenter les restrictions sur l'entrée :

```
/*@
  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;
}
```

Ou nous pouvons modifier le code de manière à corriger le risque de débordement :

```
/*@
  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;
```

```
11 }
```

Notons que comme dans la spécification, les calculs sont effectués à l'aide d'entiers mathématiques, nous n'avons pas à nous préoccuper d'un quelconque risque de débordement lorsque nous utilisons la fonction logique `ax_b` :

```
1 void mathematical_example(void){
2   //@ assert ax_b(42, INT_MAX, 1) < ax_b(70, INT_MAX, 1) ;
3 }
```

est correctement déchargé par WP, qui ne génère aucune alarme liée aux débordements :

```
void mathematical_example(void)
{
  /*@ assert ax_b(42, 2147483647, 1) < ax_b(70, 2147483647, 1); */ ;
  return;
}
```

5.3.2. Récursivité et limites

Les fonctions logiques peuvent être définies récursivement. Cependant, une telle définition montrera très rapidement ses limites pour la preuve. En effet, pendant les manipulations des prouveurs automatiques sur les propriétés logiques, si l'usage d'une telle fonction est présente, elle devra être évaluée. Or, les prouveurs ne sont pas conçus pour faire ce genre d'évaluation, qui se révélera donc généralement très coûteuse et produisant alors des temps de preuve trop longs menant à des *timeouts*.

Exemple concret, nous pouvons définir la fonction factorielle, dans la logique et en C :

```
1  /*@
2    logic integer factorial(integer n) = (n <= 0) ? 1 : n * factorial(n-1);
3  */
4
5  /*@
6    assigns \nothing ;
7    ensures \result == factorial(n) ;
8  */
9  int facto(int n){
10   if(n < 2) return 1 ;
11
12   int res = 1 ;
13   /*@
14     loop invariant 2 <= i <= n+1 ;
15     loop invariant res == factorial(i-1) ;
16     loop assigns i, res ;
17     loop variant n - i ;
18   */
19   for(int i = 2 ; i <= n ; i++){
20     res = res * i ;
21   }
22   return res ;
23 }
```

Sans contrôle de borne, cette fonction se prouve rapidement. Si nous ajoutons le contrôle des RTE, nous voyons qu'il y a un risque de débordement arithmétique sur la multiplication.

Sur le type `int`, le maximum que nous pouvons calculer est la factorielle de 12. Au-delà, cela produit un dépassement. Nous pouvons donc ajouter cette précondition :

```
/*@
  requires n <= 12 ;
  assigns \nothing ;
  ensures \result == factorial(n) ;
*/
int facto(int n){
```

Si nous demandons la preuve avec cette entrée, Alt-Ergo échouera pratiquement à coup sûr. En revanche, le prouveur Z3 produit la preuve en moins d'une seconde. Parce que dans ce cas précis, les heuristiques de Z3 considèrent que c'est une bonne idée de passer un peu plus de temps sur l'évaluation de la fonction.

Les fonctions logiques peuvent donc être définies récursivement, mais sans astuces supplémentaires, nous venons vite nous heurter au fait que les prouveurs vont au choix devoir faire de l'évaluation, ou encore « raisonner » par induction, deux tâches pour lesquelles ils ne sont pas du tout faits, ce qui limite nos possibilités de preuve. Nous verrons plus tard dans ce tutoriel comment éviter cette limitation.

5.3.3. Exercices

5.3.3.1. Distance

Spécifier et prouver le programme suivant :

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

Pour cela, définir deux fonctions logiques `abs` and `distance`. Utiliser ces fonctions pour écrire la spécification de la fonction.

5.3.3.2. Carré

Écrire le corps de la fonction `square`. Spécifier et prouver le programme. Utiliser une fonction logique `square`.

```
int abs(int x){
  return (x < 0) ? -x : x ;
}

unsigned square(int x){
  return 0 ; // to complete
}
```

Attention aux types des différentes variables, de telle manière à ne pas sur-contraindre les entrées de la fonction. De plus, pour vérifier l'absence d'erreurs à l'exécution, utiliser les options `-warn-unsigned-overflow` et `-warn-unsigned-downcast`.

5.3.3.3. Iota

Voici une implémentation possible de la fonction `iota` :

```
#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 ;
    }
  }
}
```

Écrire une fonction logique qui retourne sa valeur d'entrée incrémentée de 1. Prouver qu'après l'exécution de `iota`, la première valeur du tableau est celle reçue en entrée et que chaque valeur du tableau correspond à la valeur précédente plus 1 (en utilisant la fonction logique définie).

5.3.3.4. Addition sur un vecteur

Dans le programme suivant, la fonction `vec_add` ajoute le second vecteur reçu en entrée dans le premier. Écrire un contrat pour la fonction `show_the_difference` qui exprime pour chaque valeur du vecteur `v1` la différence entre la pré et la postcondition. Pour cela, définir une fonction logique `diff` qui retourne la différence de valeur à une position mémoire entre un label `L1` et la valeur au label `L2`.

```
#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);
}
```

Ré-exprimer le prédicat `unchanged` en utilisant la fonction logique.

5.3.3.5. La somme des N premiers entiers

La fonction suivante calcule la somme des N premiers entiers. Écrire une fonction logique récursive qui retourne la somme des N premiers entiers et écrire la spécification de la fonction C effectuant ce calcul en spécifiant qu'elle retourne la même valeur que celle fournie par la fonction logique.

```
int sum_n(int n){
  if(n < 1) return 0 ;

  int res = 0 ;
  for(int i = 1 ; i <= n ; i++){
    res += i ;
  }
  return res ;
}
```

Essayer de vérifier l'absence d'erreurs à l'exécution. Le débordement entier n'est pas si simple à régler. Cependant, écrire une précondition qui devrait être suffisante pour cela (rappel : la somme des N premiers entiers peut être exprimée avec une formule très simple ...). Cela ne sera sûrement pas suffisant pour arriver au bout de la preuve, mais nous réglerons cela dans la prochaine section.

5.4. Lemmes

Les lemmes sont des propriétés générales à propos des prédicats ou encore des fonctions. Une fois ces propriétés exprimées, la preuve peut être réalisée en isolation du reste de la preuve du programme, en utilisant des prouveurs automatiques ou (plus souvent) des prouveurs interactifs. Une fois la preuve réalisée, la propriété énoncée peut être utilisée directement par les prouveurs automatiques sans que cela ne nécessite d'en réaliser la preuve à nouveau. Par exemple, si nous énonçons un lemme L qui dit que $P \Rightarrow Q$, et dans une autre preuve nous avons besoin de

prouver Q alors que nous savons déjà que P est vérifiée, nous pouvons utiliser directement le lemme L pour conclure sans avoir besoin de faire à nouveau le raisonnement complet qui amène de P à Q.

Dans la section précédente, nous avons dit que les fonctions récursives logiques peuvent rendre les preuves plus difficiles pour les solveurs SMT. Dans un tel cas, les lemmes peuvent nous aider. Nous pouvons écrire nous même les preuves qui nécessitent de raisonner par induction pour certaines propriétés que nous énonçons comme des lemmes. Ces lemmes peuvent ensuite être utilisés efficacement par les prouveurs pour effectuer les autres preuves à propos du programme.

5.4.1. Syntaxe

Une nouvelle fois, nous les introduisons à l'aide d'annotations ACSL. La syntaxe utilisée est la suivante :

```
/*@
  lemma name_of_the_lemma { Label0, ..., LabelN }:
    property ;
*/
```

Cette fois les propriétés que nous voulons exprimer ne dépendent pas de paramètres reçus (hors de nos *labels* bien sûr). Ces propriétés seront donc exprimées sur des variables quantifiées. Par exemple, nous pouvons poser ce lemme qui est vrai, même s'il est trivial :

```
/*@
  lemma lt_plus_lt:
    \forall integer i, j ; i < j ==> i+1 < j+1;
*/
```

Cette preuve peut être effectuée en utilisant WP. La propriété est bien sûr trivialement prouvée par Qed.

5.4.2. Exemple : propriété fonction affine

Nous pouvons par exemple reprendre nos fonctions affines et exprimer quelques propriétés intéressantes à leur sujet :

```
/*@
  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);
*/
```

Pour ces preuves, il est fort possible qu'Alt-Ergo ne parvienne pas à les décharger. Dans ce cas, le prouveur Z3 devrait, lui, y arriver. Nous pouvons ensuite construire cet exemple de 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;
}
```

Si nous ne renseignons pas les lemmes mentionnés plus tôt, il y a peu de chances qu'Alt-Ergo réussisse à produire la preuve que `fmin` est inférieur à `fmax`. Avec ces lemmes présents en revanche, il y parvient sans problème, car cette propriété est une simple instance du lemme `ax_b_monotonic_pos`, la preuve étant ainsi triviale, car notre lemme nous énonce cette propriété comme étant vraie. Notons que sur cette version généralisée, Z3 sera probablement plus efficace pour prouver l'absence d'erreurs à l'exécution.

5.4.3. Exemple : tableaux et labels

Plus tard dans ce tutoriel, nous verrons certains types de définitions à propos desquels il est parfois difficile de raisonner pour les solveurs SMT quand des modifications ont lieu en mémoire. Par conséquent, nous aurons souvent besoin de lemmes pour indiquer les relations qui existent à propos du contenu de la mémoire entre deux labels.

Pour le moment, illustrons cela avec un exemple simple. Considérons les deux prédicats suivants :

```
/*@
  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) ;
```

```
8 */
```

Nous pourrions par exemple vouloir énoncer que lorsqu'un tableau est trié, et que la mémoire est modifiée (créant donc un nouvel état mémoire), mais que le contenu du tableau reste inchangé, alors le tableau est toujours trié. Cela peut être réalisé avec le lemme suivant :

```
1 /*@
2   lemma unchanged_sorted{L1, L2}:
3     \forall int* array, integer b, integer e ;
4       sorted{L1}(array, b, e) ==>
5       unchanged{L1, L2}(array, b, e) ==>
6         sorted{L2}(array, b, e) ;
7 */
```

Nous énonçons ce lemme pour deux labels `L1` et `L2`, et exprimons que pour toute plage de valeurs dans un tableau, si elle est triée au label `L1` et reste inchangée depuis `L1` vers `L2`, alors elle reste triée au label `L2`.

Notons qu'ici, cette propriété est facilement prouvée par les prouveurs SMT. Nous verrons plus tard des exemples pour lesquels il n'est pas si simple d'obtenir une preuve.

5.4.4. Check lemma

Comme pour les assertions, il est possible d'utiliser une version `check` des lemmes :

```
1 /*@
2   check lemma name_of_the_lemma { Label0, ..., LabelN }:
3     property ;
4 */
```

Une annotation `check lemma` demande à WP de générer une condition de vérification. Mais, à l'inverse des lemmes standards, la connaissance que la propriété correspondante est vérifiée ne sera pas ajoutée dans le contexte des preuves futures. Par conséquent, les solveurs SMT ne pourront pas l'utiliser pour d'autres preuves. Ce type d'annotations peut être utile pour gagner en confiance à propos des annotations globales, en testant qu'elles peuvent être utilisées dans certaines situations, ou à l'inverse, que certaines propriétés non voulues ne peuvent pas être prouvées.

Il n'y a pas d'annotations `admit lemma`, l'annotation dédiée est l'annotation `axiom` qui sera décrite dans la section 6.2.

5.4.5. Exercices

5.4.5.1. Propriété de la multiplication

Écrire un lemme qui énonce que pour trois entiers x, y et z, si x est plus grand ou égal à 0, si z est plus grand ou égal à y, alors $x * z$ est plus grand ou égal à $x * y$.

Ce lemme ne sera probablement pas prouvé par les solveurs SMT. Une solution et la preuve Coq du lemme sont disponibles sur le GitHub de ce tutoriel.

5.4.5.2. Localement trié vers globalement trié

Le programme suivant contient une fonction qui demande à ce qu'un tableau soit trié au sens que chaque élément soit plus petit ou égal à l'élément qui le suit puis appelle la fonction de recherche dichotomique.

```
1  /*@
2    lemma element_level_sorted_implies_sorted:
3      \true ; // to complete
4  */
5
6  /*@
7    requires \valid_read(arr + (0 .. len-1));
8    requires element_level_sorted(arr, 0, len) ;
9    requires in_array(value, arr, len);
10
11   assigns \nothing ;
12
13   ensures 0 <= \result < len ;
14   ensures arr[\result] == value ;
15 */
16 size_t bsearch_callee(int* arr, size_t len, int value){
17   return bsearch(arr, len, value);
18 }
```

Pour cet exercice, reprendre la solution de l'exercice 5.2.3.4 à propos de la recherche dichotomique. La précondition de cette recherche peut sembler plus forte que celle reçue par la précondition de `bsearch_callee`. La première demande chaque paire d'éléments d'être ordonnée, la seconde simplement que chaque élément soit inférieur à celui qui le suit. Cependant, la seconde implique la première. Écrire un lemme qui énonce que si `element_level_sorted` est vraie pour un tableau, `sorted` est vraie aussi. Ce lemme ne sera probablement pas prouvé par un solveur SMT, toutes les autres propriétés devraient être prouvées automatiquement.

Une solution et la preuve Coq du lemme sont disponibles sur le GitHub de ce tutoriel.

5.4.5.3. Somme des N premiers entiers

Reprendre la solution de l'exercice 5.3.3.5 à propos de la somme des N premiers entiers. Écrire un lemme qui énonce que la valeur calculée par la fonction logique récursive qui permet la spécification de la somme des N premiers entiers est $n*(n+1)/2$. Ce lemme ne sera pas prouvé par un solveur SMT.

Une solution et la preuve Coq du lemme sont disponibles sur le GitHub de ce tutoriel.

5.4.5.4. Transitivité d'un glissement d'éléments

Le programme suivant est composé de deux fonctions. La première est `shift_array` et permet de faire glisser des éléments dans un tableau d'un certain nombre de cellules (nommé `shift`). La seconde effectue deux glissements successifs des éléments d'un même tableau.

```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) ;
}
```

Compléter les prédicats `shifted` et `unchanged`. Le prédicat `shifted` doit utiliser `shifted_cell`. Le prédicat `unchanged` doit utiliser `shifted`. Compléter le contrat de la fonction `shift_array` et prouver sa correction avec WP.

Exprimer deux lemmes à propos de la propriété `shifted`.

Le premier, nommé `shift_ptr`, doit énoncer que déplacer une plage de valeur `fst+s1` à `last+s1` dans un tableau `array` d'un décalage `s2` est équivalent à déplacer une plage de valeurs `fst` à `last` pour la position mémoire `array+s1` avec un décalage `s2`.

Le second doit énoncer que quand les éléments d'un tableau sont déplacés une première fois avec un décalage `s1` puis une seconde fois avec un décalage `s2`, alors le déplacement final correspond à un décalage avec un déplacement `s1+s2`.

Le lemme `shift_ptr` ne sera probablement pas prouvé par un solveur SMT. Nous fournissons une solution et la preuve Coq de ce lemme sur le GitHub de ce livre. Les propriétés restantes doivent être prouvées automatiquement.

5.4.5.5. Déplacement d'une plage triée

Le programme suivant est composé de deux fonctions. La fonction `shift_and_search` déplace les éléments d'un tableau puis effectue une recherche dichotomique.

```c
#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);
}
```

Reprendre la solution de la recherche dichotomique de l'exercice 5.2.3.4. Modifier cette recherche et sa spécification de façon à ce que la fonction permette de chercher dans toute plage triée de valeurs. La preuve doit toujours fonctionner.

Reprendre également la fonction prouvée `shift_array` de l'exercice précédent.

Compléter le contrat de la fonction `shift_and_search`. La précondition qui demande à ce que le tableau soit trié avant la recherche ne sera pas validée, ni la postcondition de l'appelant. Compléter le lemme `shifted_still_sorted` qui doit énoncer que si une plage de valeur est triée à un *label*, puis déplacée, alors elle reste triée. La précondition devrait maintenant être validée. Ensuite, compléter le lemme `in_array_shifted` qui doit énoncer que si une valeur est dans une plage de valeur, alors lorsque cette plage est déplacée, la valeur est toujours dans la nouvelle plage obtenue. La postcondition de l'appelant devrait maintenant être prouvée.

Ces lemmes ne seront probablement pas prouvés par un solveur SMT. Une solution et les preuves Coq sont disponibles sur le GitHub de ce livre.

Dans cette partie, nous avons vu les constructions de ACSL qui nous permettent de factoriser un peu nos spécifications et d'exprimer des propriétés générales pouvant être utilisées par les prouveurs pour faciliter leur travail.

Toutes les techniques expliquées dans cette partie sont sûres, au sens où elles ne permettent *a priori* pas de fausser la preuve avec des définitions fausses ou contradictoires. En tout cas, si la spécification n'utilise que ce type de constructions et que chaque lemme, chaque précondition (aux points d'appel), chaque postcondition, chaque assertion, chaque variant et chaque invariant est correctement prouvé, le code est juste.

Parfois ces constructions ne sont pas suffisantes pour exprimer toutes nos propriétés ou pour prouver nos programmes. Les prochaines constructions que nous verrons ajouteront de nouvelles possibilités à ce sujet, mais il faudra se montrer prudent dans leur usage, car des erreurs pourraient nous permettre de créer des hypothèses fausses ou d'altérer le programme que nous vérifions.

6. ACSL - Définitions logiques et code fantôme

Dans cette partie nous allons voir trois notions importantes d'ACSL :
- les définitions inductives,
- les définitions axiomatiques,
- le code fantôme.

Dans certaines configurations, ces trois notions sont absolument nécessaires pour faciliter le processus de spécification et de preuve. Soit en forçant l'abstraction de certaines propriétés, soit en explicitant des informations qui sont autrement implicites et plus difficiles à prouver.

Le risque de ces trois notions est qu'elles peuvent rendre notre preuve inutile si nous faisons une erreur dans leur usage. Les définitions inductives et axiomatiques introduisent le risque de faire entrer « faux » dans nos hypothèses, ou d'écrire des définitions imprécises. Le code fantôme, s'il ne respecte pas certaines propriétés, ouvre le risque de modifier le programme vérifié, nous faisant ainsi prouver un autre programme que celui que nous voulons prouver.

6.1. Définitions inductives

Les prédicats inductifs donnent une manière d'énoncer des propriétés dont la vérification nécessite de raisonner par induction, c'est-à-dire que pour une propriété $p(input)$, on peut assurer qu'elle est vraie, soit parce qu'elle correspond à un certain cas de base (par exemple, 0 est un entier naturel pair parce que l'on définit le cas 0 comme un cas de base), ou alors parce que sachant que p est vraie pour un certain *smallerinput* (qui est « plus proche » du cas de base) et sachant que *smallerinput* et *input* sont reliés par une propriété donnée (qui dépend de notre définition) nous pouvons conclure (par exemple nous pouvons définir que si un naturel n est pair, le naturel $n + 2$ est pair, donc pour vérifier qu'un naturel supérieur à 0 est pair, on peut regarder si ce naturel moins 2 est pair).

6.1.1. Syntaxe

Pour le moment, introduisons la syntaxe des prédicats inductifs :

```
/*@
  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 ;
  }
*/
```

où tous les `c_i` sont des noms et tous les `p_i` sont des formules logiques où `property` apparaît en conclusion. Pour résumer, `property` est vraie pour certains paramètres et labels mémoire, s'ils valident l'un des cas du prédicat inductif.

Jetons un œil à la petite propriété dont nous parlions plus tôt : comment définir qu'un entier naturel est pair par induction ? Nous pouvons traduire la phrase : « 0 est un naturel pair, et si n est un naturel pair, alors $n + 2$ est aussi un naturel pair ».

```
/*@
  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);
  }
*/
```

Ce prédicat définit bien les deux cas :

— 0 est un naturel pair (case de base),
— si un naturel n est pair, $n + 2$ est pair aussi (induction)

Cette définition peut être utilisée pour prouver les assertions suivantes :

```
void foo(){
  //@ assert even_natural(4);
  //@ assert even_natural(42);
}
```

Notons que, puisque le solveur doit récursivement dérouler la définition inductive, cela pourrait ne pas fonctionner pour n'importe quelle valeur. Cette capacité de preuve dépend des heuristiques des solveurs qui décident ou non de s'arrêter après N dépliages.

Même si cette définition fonctionne, elle n'est pas complètement satisfaisante. Par exemple, nous ne pouvons pas déduire qu'un naturel impair n'est pas pair. Si nous essayons de prouver que 1 est pair, nous devons vérifier que si -1 est pair, puis -3, -5, etc. Cela nous amène au fait que l'assertion suivante ne peut pas être prouvée :

```
void bar(){
  int a = 1 ;
  //@ assert !even_natural(a);
}
```

De plus, nous préférons généralement indiquer la condition selon laquelle une conclusion donnée est vraie en utilisant les variables quantifiées dans la conclusion. Par exemple, ici, pour montrer qu'un entier n est naturel, comment faire ? D'abord vérifier s'il est égal à 0, et si ce n'est pas le cas, vérifier qu'il est plus grand que 0, et dans ce cas, vérifier que $n - 2$ est pair :

```
/*@
  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) ;
  }
*/
```

Ici nous distinguons à nouveau deux cas :

— 0 est pair,
— un naturel n est pair s'il est plus grand que 0 et $n-2$ est un naturel pair.

Pour un entier naturel donné, s'il est plus grand que 0, nous diminuerons récursivement sa valeur jusqu'à atteindre 0 ou -1. Dans le cas 0, l'entier naturel est pair. Dans le cas -1, nous n'aurons aucun cas du prédicat inductif qui corresponde à cette valeur et nous pourrons conclure que la propriété est fausse.

```
void bar(){
  int a = 1 ;
  //@ assert !even_natural(a);
}
```

Bien sûr, définir la notion d'entier naturel pair par induction n'est pas une bonne idée, un modulo serait plus simple. Nous utilisons généralement les propriétés inductives pour définir des propriétés récursives complexes.

6.1.1.1. Consistance

Les définitions inductives apportent le risque d'introduire des inconsistances. En effet, les propriétés spécifiées dans les différents cas sont considérées comme étant toujours vraies, donc si nous introduisons des propriétés permettant de prouver `false`, nous sommes en mesure de prouver n'importe quoi. Même si nous parlerons plus longuement des axiomes dans la Section 6.2, nous pouvons donner quelques conseils pour ne pas construire une mauvaise définition.

D'abord, nous pouvons nous assurer que les prédicats inductifs sont bien fondés. Cela peut être fait en restreignant syntaxiquement ce que nous acceptons dans une définition inductive en nous assurant que chaque cas est de la forme :

```
/*@
  \forall y1,...,ym ; h1 ==> ··· ==> hl ==> P(t1,...,tn) ;
*/
```

où le prédicat `P` ne peut apparaître que positivement (donc sans la négation `!` - ¬) dans les différentes hypothèses `hx`. Intuitivement, cela assure que nous ne pouvons pas construire des occurrences à la fois positives et négatives de `P` pour un ensemble de paramètres donnés (ce qui serait incohérent).

Cette propriété est par exemple vérifiée pour notre définition précédente du prédicat `even_natural`. Tandis qu'une définition comme :

```
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 */
```

ne respecte pas cette contrainte, car la propriété `even_natural` apparaît négativement à la ligne 8.

Ensuite, nous pouvons simplement écrire une fonction dont le contrat nécessite `P`. Par exemple, nous pouvons écrire la fonction suivante :

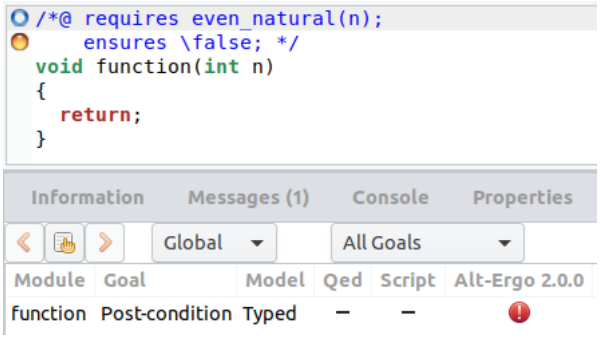
```
1  /*@
2    requires P( params ... ) ;
3    ensures  BAD: \false ;
4  */ void function(params){
5
6  }
```

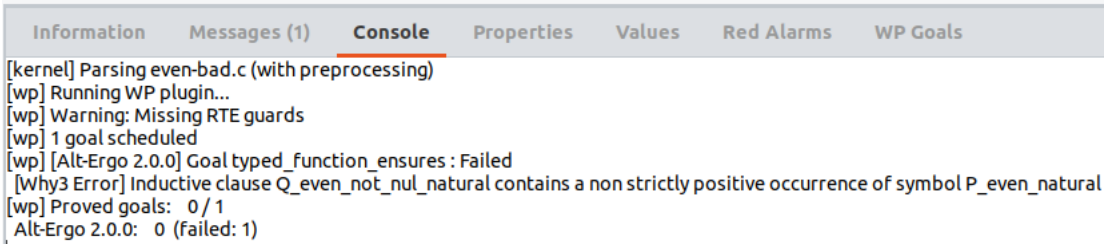
Pour notre définition de `even_natural`, cela donnerait :

```
1  /*@
2    requires even_natural(n) ;
3    ensures  \false ;
4  */ void function(int n){
5
6  }
```

Pendant la génération de la condition de vérification, WP demande à Why3 de créer une définition inductive pour celle que nous avons écrit en ACSL. Comme Why3 est plus strict que Frama-C et WP pour ce type de définition, si le prédicat inductif est trop étrange (s'il n'est pas bien fondé), il sera rejeté avec une erreur. Et en effet, pour la propriété `even_natural` que nous venons de définir, Why3 la refuse quand nous tentons de prouver `ensures \false`, parce qu'il existe une occurrence non positive de `P_even_natural` qui encode le `even_natural` que nous avons écrit en ACSL.

```
1  frama-c-gui -wp -wp-prop=BAD file.c
```





Cependant, cela ne signifie pas que nous ne pouvons pas écrire une définition inductive inconsistante. Par exemple, la définition inductive suivante est bien fondée, mais nous permet de prouver faux :

```
1  /*@ inductive P(int* p){
2        case c: \forall int* p ; \valid(p) && p == (void*)0 ==> P(p);
3      }
4  */
5
6  /*@ requires P(p);
7      ensures \false ; */
8  void foo(int *p){}
```

Ici nous pourrions détecter le problème avec `-wp-smoke-tests` qui trouverait que la précondition ne peut pas être satisfaite. Mais nous devons être prudents pendant la conception d'une définition inductive afin de ne pas introduire une définition qui nous permette de produire une preuve de faux.

Avant Frama-C 21 Scandium, les définitions inductives étaient traduites, en Why3, grâce à des axiomes. Cela signifie que ces vérifications n'étaient pas effectuées. En conséquence, pour avoir un comportement similaire avec une version plus ancienne de Frama-C, il faut utiliser Coq et pas un prouveur Why3.

6.1.2. Définitions de prédicats récursifs

Les prédicats inductifs sont souvent utiles pour exprimer des propriétés récursivement, car ils permettent souvent d'empêcher les solveurs SMT de dérouler la récursion quand c'est possible.

Par exemple, nous pouvons définir qu'un tableau ne contient que des zéros de cette façon :

```
/*@
  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);
  }
*/
```

et nous pouvons à nouveau prouver notre fonction de remise à zéro avec cette nouvelle définition :

```
/*@
  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;
}
```

Selon votre version de Frama-C et de vos prouveurs automatiques, la preuve de préservation de l'invariant peut échouer. Une raison à cela est que le prouveur ne parvient pas à garder l'information que ce qui précède la cellule en cours de traitement par la boucle est toujours à 0. Nous pouvons ajouter un lemme dans notre base de connaissance, expliquant que si l'ensemble des valeurs d'un tableau n'a pas changé, alors la propriété est toujours vérifiée :

```
/*@
  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);
*/
```

et d'énoncer une assertion pour spécifier ce qui n'a pas changé entre le début du bloc de la boucle (marqué par le *label* `L` dans le code) et la fin (qui se trouve être `Here` puisque nous posons notre assertion à la fin) :

```
  for(size_t i = 0; i < length; ++i){
  L:
    array[i] = 0;
    //@ assert same_elems{L,Here}(array,0,i);
  }
```

À noter que dans cette nouvelle version du code, la propriété énoncée par notre lemme n'est

pas prouvée par les solveurs automatiques, qui ne savent pas raisonner par induction. Pour les curieux, la (très simple) preuve en Coq est disponible ici : 6.4.

Dans le cas présent, utiliser une définition inductive est contre-productif : notre propriété est très facilement exprimable en logique du premier ordre comme nous l'avons déjà fait précédemment. Les axiomatiques sont faites pour écrire des définitions qui ne sont pas simples à exprimer dans le formalisme de base d'ACSL. Mais il est mieux de commencer avec un exemple facile à lire.

6.1.3. Exemple : le tri

Nous allons prouver un simple tri par sélection :

```
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]);
  }
}
```

Le lecteur pourra s'exercer en spécifiant et en prouvant les fonctions de recherche de minimum et d'échange de valeur. Nous cachons la correction (Réponses : 6.4) et nous nous concentrerons plutôt sur la spécification et la preuve de la fonction de tri qui sont une illustration intéressante de l'usage des définitions inductives.

En effet, une erreur commune que nous pourrions faire dans le cas de la preuve du tri est de poser cette spécification (qui est vraie !) :

```
/*@
  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]);
  }
}
```

Cette spécification est correcte. Mais si nous nous rappelons la partie concernant les spécifications, nous nous devons d'exprimer précisément ce que nous attendons. Avec la spécification actuelle, nous ne prouvons pas toutes les propriétés nécessaires d'un tri ! Par exemple, cette fonction remplit pleinement la spécification :

```
/*@
  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;
}
```

En fait, notre spécification oublie que tous les éléments qui étaient originellement présents dans le tableau à l'appel de la fonction doivent toujours être présents après l'exécution de notre fonction de tri. Dit autrement, notre fonction doit en fait produire la permutation triée des valeurs du tableau.

Une propriété comme la définition de ce qu'est une permutation s'exprime extrêmement bien par l'utilisation d'une définition inductive. En effet, pour déterminer qu'un tableau est la permutation d'un autre, les cas sont très limités. Premièrement, le tableau est une permutation de lui-même, puis l'échange de deux valeurs sans changer les autres est également une permutation, et finalement si nous créons la permutation p_2 d'une permutation p_1, puis la permutation p_3 de p_2, alors par transitivité p_3 est une permutation de p_1.

La définition inductive correspondante est la suivante :

```
/*@
  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);
  }
*/
```

Nous spécifions alors que notre tri nous crée la permutation triée du tableau d'origine et nous pouvons prouver l'ensemble en complétant l'invariant de la fonction :

```
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 }
```

Cette fois, notre propriété est précisément définie, la preuve reste assez simple à faire passer, ne nécessitant que l'ajout d'une assertion que le bloc de la fonction n'effectue qu'un échange des valeurs dans le tableau (et donnant ainsi la transition vers la permutation suivante du tableau). Pour définir cette notion d'échange, nous utilisons une annotation particulière (à la ligne 16), introduite par le mot-clé `ghost`. Ici, le but est d'introduire un *label* fictif dans le code qui est uniquement visible d'un point de vue spécification. C'est l'objet de la section finale de ce chapitre, parlons d'abord des définitions axiomatiques.

6.1.4. Exercices

6.1.4.1. La somme des N premiers entiers

Reprendre la solution de l'exercice 5.4.5.3 à propos de la somme des N premiers entiers. Réécrire la fonction logique en utilisant plutôt un prédicat inductif qui exprime l'égalité entre un entier et la somme des N premiers entiers.

```
1  #include <limits.h>
2
3  /*@
4    inductive is_sum_n(integer n, integer res) {
5    case C: \true; // to complete
6    }
7  */
8
9  /*@
10   requires n*(n+1) <= 2*INT_MAX ;
11   assigns \nothing ;
12   // ensures ... ;
13 */
14 int sum_n(int n){
15   if(n < 1) return 0 ;
16
```

```
17    int res = 0 ;
18    /*@
19      loop invariant 1 <= i <= n+1 ;
20      // loop invariant ... ;
21      loop assigns i, res ;
22      loop variant n - i ;
23    */
24    for(int i = 1 ; i <= n ; i++){
25      res += i ;
26    }
27    return res ;
28  }
```

Adapter le contrat de la fonction et le(s) lemme(s). Notons que les lemmes ne seront certainement pas prouvés par les solveurs SMT. Nous fournissons une solution et les preuves Coq sur le répertoire GitHub de ce livre.

6.1.4.2. Plus grand diviseur commun

Écrire un prédicat inductif qui exprime qu'un entier est le plus grand diviseur commun de deux autres. Le but de cet exercice est de prouver que la fonction `gcd` calcule le plus grand diviseur commun. Nous n'avons donc pas à spécifier tous les cas du prédicat. En effet, si nous regardons de près la boucle, nous pouvons voir qu'après la première itération, `a` est supérieur ou égal à `b`, et que cette propriété est maintenue par la boucle. Donc, considérons deux cas pour le prédicat inductif :

— `b` est 0,
— si une valeur `d` est le PGCD de `b` et `a % b`, alors c'est le PGCD de `a` et `b`.

```
1   #include <limits.h>
2
3   /*@ inductive is_gcd(integer a, integer b, integer div) {
4       case gcd_zero: \true ; // to complete
5       case gcd_succ: \true ; // to complete
6       }
7   */
8
9   /*@
10    requires a >= 0 && b >= 0 ;
11    assigns \nothing ;
12    // ensures ... ;
13  */
14  int gcd(int a, int b){
15    /*@
16      loop invariant \forall integer t ; \true ; // to complete
17    */
18    while (b != 0){
19      int t = b;
20      b = a % b;
21      a = t;
22    }
23    return a;
24  }
```

Exprimer la postcondition de la fonction et compléter l'invariant pour prouver qu'elle est vérifiée. Notons que l'invariant devrait utiliser le cas inductif `gcd_succ`.

6.1.4.3. Puissance

Dans cet exercice, nous ne considérerons pas les RTE.

Écrire un prédicat inductif qui exprime qu'un entier `r` est égal à x^n. Considérer deux cas : soit n est 0, soit il est plus grand et à ce moment-là, la valeur de `r` est reliée à la valeur x^{n-1}.

```
/*@
  inductive is_power(integer x, integer n, integer r) {
  case zero: \true; // to complete
  case N: \true; // to complete
  }
*/
```

Prouver d'abord la version naïve de la fonction puissance :

```
/*@
  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 ;
}
```

Maintenant tentons de prouver une version plus rapide de la fonction puissance :

```
/*@
  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 ;
}
```

Dans cette version, nous exploitons deux propriétés de l'opérateur puissance :

— $(x^2)^n = x^{2n}$

— $x \times (x^2)^n = x^{2n+1}$

qui permet de diviser n par 2 à chaque tour de boucle au lieu de le décrémenter de 1 (ce qui permet à l'algorithme d'être en $O(\log n)$ et pas $O(n)$). Exprimer les deux propriétés précédentes sous forme de lemmes :

```
/*@
  lemma power_even: \true; // to complete
  lemma power_odd: \true; // to complete
*/
```

Exprimer d'abord le lemme `power_even`, les solveurs SMT pourraient être capables de combiner ce lemme avec la définition inductive pour déduire `power_odd`. La preuve Coq de `power_even` est fournie sur le répertoire GitHub de ce livre, ainsi que la preuve de `power_odd` si les solveurs SMT échouent.

Finalement, compléter le contrat et l'invariant de la fonction `fast_power`. Pour cela, notons qu'au début de la boucle $x^{old(n)} = p^n$, et que chaque itération utilise les propriétés précédentes pour mettre à jour r, au sens que nous avons $x^{old(n)} = r \times p^n$ pendant toute la boucle, jusqu'à obtenir $n = 0$ et donc $p^n = 1$.

6.1.4.4. Permutation

Reprendre la définition des prédicats `shifted` et `unchanged` de l'exercice 5.4.5.4. Le prédicat `shited_cell` peut être *inliné* et supprimé. Utiliser le prédicat `shifted` pour exprimer le prédicat `rotate` qui exprime que les éléments d'un tableau sont « tournés » vers la gauche, dans le sens où tous les éléments sont glissés vers la gauche, sauf le dernier qui est mis à la première cellule de la plage de valeur. Utiliser ce prédicat pour prouver la fonction `rotate`.

```
/*@
  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 ;
}
```

Exprimer une nouvelle version de la notion de permutation avec un prédicat inductif qui considère les cas suivants :

— la permutation est réflexive ;

— si la partie gauche d'une plage de valeur (jusqu'à un certain indice) est « tournée » entre deux labels, nous avons toujours une permutation ;

— si la partie droite d'une plage de valeur (à partir d'un certain indice) est « tournée » entre deux labels, nous avons toujours une permutation ;
— la permutation est 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
  }
*/
```

Compléter le contrat de la fonction `two_rotates` qui fait des rotations successives, de la première et la seconde moitié de la plage considérée, et prouver qu'elle maintient la 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. Définitions axiomatiques

Les axiomes sont des propriétés dont nous énonçons qu'elles sont vraies quelle que soit la situation. C'est un moyen très pratique d'énoncer des notions complexes qui pourront rendre le processus très efficace en abstrayant justement cette complexité. Évidemment, comme toute propriété exprimée comme un axiome est supposée vraie, il faut également faire très attention à ce que nous définissons, car si nous introduisons une propriété fausse dans les notions que nous supposons vraies alors ... nous saurons tout prouver, même ce qui est faux.

6.2.1. Syntaxe

Pour introduire une définition axiomatique, nous utilisons la syntaxe suivante :

```
/*@
  axiomatic Name_of_the_axiomatic_definition {
    // ici nous pouvons définir ou déclarer des fonctions et prédicats

    axiom axiom_name { Label0, ..., LabelN }:
      // property ;

    axiom other_axiom_name { Label0, ..., LabelM }:
      // property ;

    // ... nous pouvons en mettre autant que nous voulons
  }
*/
```

Nous pouvons par exemple définir cette axiomatique :

```
/*@
  axiomatic lt_plus_lt{
    axiom always_true_lt_plus_lt:
      \forall integer i, j; i < j ==> i+1 < j+1 ;
  }
*/
```

et nous pouvons voir que dans Frama-C, la propriété est bien supposée vraie [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. Lien avec la notion de lemme

Les lemmes et les axiomes nous permettront d'exprimer les mêmes types de propriétés, à savoir des propriétés exprimées sur des variables quantifiées (et éventuellement des variables globales, mais cela reste assez rare puisqu'il est difficile de trouver une propriété qui soit globalement vraie à leur sujet tout en étant intéressante). Outre ce point commun, il faut également savoir que comme les axiomes, en dehors de leur définition, les lemmes sont considérés vrais par WP.

La seule différence entre lemme et axiome du point de vue de la preuve est donc que nous devrons fournir une preuve que le premier est valide alors que l'axiome est toujours supposé vrai.

6.2.2. Définition de fonctions ou prédicats récursifs

Les définitions axiomatiques de fonctions ou de prédicats récursifs sont particulièrement utiles, car elles permettent d'empêcher les prouveurs de dérouler la récursion quand c'est possible.

L'idée est alors de ne pas définir directement la fonction ou le prédicat, mais plutôt de la déclarer puis de définir des axiomes spécifiant son comportement. Si nous reprenons par exemple la factorielle, nous pouvons la définir axiomatiquement de cette manière :

```
/*@
  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, nous présentons un axiome extrêmement utile.

Dans cette définition axiomatique, notre fonction n'a pas de corps. Son comportement étant défini par les axiomes ensuite définis. Excepté ceci, rien ne change, en particulier, notre fonction peut être utilisée dans nos spécifications, exactement comme avant.

Une petite subtilité est qu'il faut prendre garde au fait que si les axiomes énoncent des propriétés à propos du contenu d'une ou plusieurs zones mémoires pointées, il faut spécifier ces zones mémoires en utilisant la notation `reads` au niveau de la déclaration. Si nous oublions une telle spécification, le prédicat, ou la fonction, sera considéré comme énoncé à propos du pointeur et non à propos de la zone mémoire pointée. Une modification de celle-ci n'entraînera donc pas l'invalidation d'une propriété connue axiomatiquement.

Par exemple, si nous prenons la définition inductive que nous avons rédigée pour `zeroed` dans le chapitre précédent, nous pouvons l'écrire à l'aide d'une définition axiomatique qui prendra cette forme :

```
/*@
  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);
  }
*/
```

Notons la présence de la clause `reads[b .. e-1]` qui spécifie la position mémoire dont le prédicat dépend. Tandis que nous n'avons pas besoin de spécifier les positions mémoires « lues » par une définition inductive, nous devons spécifier ces propriétés pour les propriétés définies axiomatiquement.

6.2.3. Consistance

En ajoutant des axiomes à notre base de connaissances, nous pouvons produire des preuves plus complexes, car certaines parties de cette preuve, mentionnées par les axiomes, ne nécessiteront plus de preuves qui allongeraient le processus complet. Seulement, en faisant cela **nous devons être extrêmement prudents**. En effet, la moindre hypothèse fausse introduite dans la base pourraient rendre tous nos raisonnements futiles. Notre raisonnement serait toujours correct, mais basé sur des connaissances fausses, il ne nous apprendrait donc plus rien de correct.

L'exemple le plus simple à produire est le suivant :

```
/*@
  axiomatic False{
    axiom false_is_true: \false;
  }
*/

int main(){
  // Examples of proved properties
```

```
9
10    //@ assert \false;
11    //@ assert \forall integer x; x > x;
12    //@ assert \forall integer x,y,z ; x == y == z == 42;
13    return *(int*) 0;
14  }
```

Tout est prouvé, y compris que le déréférencement de l'adresse 0 est OK :

```
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;
}
```

[Preuve de tout un tas de choses fausses]

Évidemment cet exemple est extrême, nous n'écririons pas un tel axiome. Le problème est qu'il est très facile d'écrire une axiomatique subtilement fausse lorsque nous exprimons des propriétés plus complexes, ou que nous commençons à poser des suppositions sur l'état global d'un système.

Quand nous commençons à créer de telles définitions, ajouter de temps en temps une preuve ponctuelle de « *false* » dont nous voulons qu'elle échoue permet de s'assurer que notre définition n'est pas inconsistante. Mais cela ne fait pas tout ! Si la subtilité qui crée le comportement faux est suffisamment cachée, les prouveurs peuvent avoir besoin de beaucoup d'informations autre que l'axiomatique elle-même pour être menés jusqu'à l'inconsistance, donc il faut toujours être vigilant !

Notamment parce que, par exemple, la mention des valeurs lues par une fonction ou un prédicat défini axiomatiquement est également importante pour la consistance de l'axiomatique. En effet, comme mentionné précédemment, si nous n'exprimons pas les valeurs lues dans le cas de l'usage d'un pointeur, la modification d'une valeur du tableau n'invalide pas une propriété que l'on connaîtrait à propos du contenu du tableau par exemple. Dans un tel cas, la preuve passe, mais l'axiome n'exprimant pas le contenu, nous ne prouvons rien.

Par exemple, si nous reprenons l'exemple de mise à zéro, nous pouvons modifier la définition de notre axiomatique en retirant la mention des valeurs dont dépendent le prédicat : `reads a[b .. e-1]`. La preuve passera toujours, mais n'exprimera plus rien à propos du contenu des tableaux considérés. Par exemple, la fonction suivante :

```
1  /*@
2    requires length > 10 ;
3    requires \valid(array + (0 .. length-1));
4    requires zeroed(array,0,length);
5    assigns  array[0 .. length-1];
6    ensures  zeroed(array,0,length);
7  */
8  void bad_function(int* array, size_t length){
9    array[5] = 42 ;
10 }
```

est prouvée correcte alors qu'une valeur a changé dans le tableau et donc elle n'est plus 0.

Notons qu'à la différence des définitions inductives, où Why3 nous permet de contrôler que ce que nous écrivons est relativement bien défini, nous n'avons pas de mécanisme de ce genre pour les définitions axiomatiques. Ces axiomes sont simplement traduits comme axiomes aussi du côté de Why3 et sont donc supposés vrais.

6.2.4. Cluster de blocs axiomatiques

La plupart des annotations globales (fonctions logiques et prédicats, lemmes, ...) peuvent être définies à deux niveaux différents : à *top-level*, le niveau des fonctions, des variables globales, etc. (sauf pour les axiomes et les fonctions et prédicats abstraits) ou dans des blocs axiomatiques. Les annotations à *top-level* sont toujours chargées dans le contexte des obligations de preuve, ce n'est pas le cas des annotations des blocs axiomatiques.

Dans l'exemple suivant :

```
1  /*@ axiomatic X {
2        predicate P(int* p) reads *p;
3        axiom x: \forall int *p ; *p == 0 ==> P(p) ;
4      }
5  */
6
7  /*@ axiomatic Y {
8        predicate Q(int *p) reads *p ;
9        axiom y: \forall int *p ; *p == 0 ==> P(p) && Q(p) ;
10     }
11 */
12
13 //@ ensures P(p) ;
14 void function(int* p){}
```

Puisque la clause `ensures` n'utilise que le prédicat `P` qui est défini dans le bloc axiomatique `X`, WP ne charge que l'axiome `x`. À l'inverse, si l'on remplace `P(p)` par `Q(p)` dans la clause `ensures`, WP charge l'axiomatique `Y` et donc l'axiome `y`, qui utilise également le prédicat `P`. Par conséquent, le bloc axiomatique `X` est également chargé. La clôture transitive des blocs axiomatiques ainsi chargés forme un *cluster* de définitions axiomatiques.

Ce comportement peut être utilisé pour éviter de fournir trop de lemmes aux solveurs SMT. Cela peut permettre d'améliorer les performances de la preuve dans certaines situations. Nous présenterons plus de détails à propos du guidage de la preuve avec des lemmes dans la section 7.2.

6.2.5. Exemple : comptage de valeurs

Dans cet exemple, nous cherchons à prouver qu'un algorithme compte bien les occurrences d'une valeur dans un tableau. Nous commençons par définir axiomatiquement la notion de comptage dans un tableau :

```
/*@
  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);
  }
*/
```

Nous avons trois cas à gérer :

— la plage de valeur concernée est vide : le nombre d'occurrences est 0 ;

— la plage de valeur n'est pas vide et le dernier élément est celui recherché : le nombre d'occurrences est le nombre d'occurrences dans la plage actuelle que l'on prive du dernier élément, plus 1 ;

— la plage de valeur n'est pas vide et le dernier élément n'est pas celui recherché : le nombre d'occurrences est le nombre d'occurrences dans la plage privée du dernier élément.

Par la suite, nous pouvons écrire la fonction C exprimant ce comportement et la prouver :

```
/*@
  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;
}
```

Une alternative au fait de spécifier que dans ce code `result` est au maximum `i` est d'exprimer un lemme plus général à propos de la valeur du nombre d'occurrences, dont nous savons qu'elle est comprise entre 0 et la taille maximale de la plage de valeurs considérée :

```
/*@
lemma l_occurrences_of_range{L}:
  \forall int v, int* array, integer from, to:
    from <= to ==> 0 <= l_occurrences_of(v, a, from, to) <= to-from;
```

```
5 */
```

La preuve de ce lemme ne pourra pas être déchargée par un solveur automatique. Il faudra faire cette preuve interactivement avec Coq par exemple. Exprimer des lemmes généraux prouvés manuellement est souvent une bonne manière d'ajouter des outils aux prouveurs pour manipuler plus efficacement les axiomatiques, sans ajouter formellement d'axiomes qui augmenteraient nos chances d'introduire des erreurs. Ici, nous devrons quand même réaliser les preuves des propriétés mentionnées.

6.2.6. Exemple : la fonction `strlen`

Dans cette section, prouvons la fonction C `strlen` :

```
1 #include <stddef.h>
2
3 size_t strlen(char const *s){
4   size_t i = 0 ;
5   while(s[i] != '\0'){
6     ++i;
7   }
8   return i ;
9 }
```

Premièrement, nous devons fournir un contrat adapté. Supposons que nous avons une fonction logique `strlen` qui retourne la longueur d'une chaîne de caractères, à savoir ce que nous attendons de notre fonction C.

```
1 /*@
2   logic integer strlen(char const* s) = // on verra plus tard
3 */
```

Nous voulons recevoir une chaîne C valide en entrée et nous voulons en calculer la longueur, une valeur qui correspond à celle fournie par la fonction logique `strlen` appliquée à cette chaîne. Bien sûr, cette fonction n'affecte rien. Définir ce qu'est une chaîne valide n'est pas si simple. En effet, précédemment dans ce tutoriel, nous avons uniquement travaillé avec des tableaux, en recevant en entrée à la fois un pointeur vers le tableau et la taille du tableau. Cependant, tel que c'est généralement fait en C, nous supposons que la chaîne termine avec un caractère `'\0'`. Cela signifie que nous avons en fait besoin de la fonction logique `strlen` pour définir ce qu'est une chaîne valide. Utilisons d'abord cette définition (notons que nous utilisons `\valid_read`, car nous ne modifions pas la chaîne) pour fournir un contrat pour `strlen` :

```
1 /*@
2   predicate valid_read_string(char * s) =
3     \valid_read(s + (0 .. strlen(s))) ;
4 */
5
6 /*@
```

```
  requires valid_read_string(s) ;
  assigns \nothing ;
  ensures \result == strlen(s) ;
*/
size_t strlen(char const *s)
```

Définir la fonction logique `strlen` n'est pas si simple. En effet, nous devons calculer la fonction d'une chaîne en trouvant le caractère `'\0'`, et nous espérons le trouver, mais en fait, nous pouvons facilement imaginer une chaîne qui n'en contiendrait pas. Dans ce cas, nous voudrions avoir une valeur d'erreur, mais il est impossible de garantir que le calcul termine : une fonction logique ne peut donc pas être utilisée pour exprimer cette propriété.

Définissons donc cette fonction axiomatiquement. D'abord définissons ce qui est lu par la fonction, à savoir : toute position mémoire depuis le pointeur jusqu'à une plage infinie d'adresses. Ensuite, considérons deux cas : la chaîne est finie, ou elle ne l'est pas, ce qui nous amène à deux axiomes : `strlen` retourne une valeur positive qui correspond à l'indice du premier caractère `'\0'` et retourne une valeur négative s'il n'y a pas de tel caractère.

```
/*@
  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 ;
```

Maintenant, nous pouvons être plus précis dans notre définition de `\valid_read_string`, une chaîne valide est une chaîne telle qu'est valide depuis le premier indice jusqu'à `strlen` de la chaîne et telle que cette valeur est plus grande que 0 (puisqu'une chaîne infinie n'est pas valide) :

```
/*@
  predicate valid_read_string(char * s) =
    strlen(s) >= 0 && \valid_read(s + (0 .. strlen(s))) ;
*/
```

Avec cette nouvelle définition, nous pouvons avancer et fournir un invariant utilisable pour la boucle de la fonction `strlen`. Il est plutôt simple : `i` est compris entre 0 et `strlen(s)`, pour toute valeur entre 0 et `i`, elles sont différentes de `'\0'`. Cette boucle n'affecte que `i` et le variant correspond à la distance entre `i` et `strlen(s)`. Cependant, si nous essayons de prouver cette fonction, la preuve échoue. Pour avoir plus d'information, nous pouvons relancer la preuve avec la vérification d'absence de RTE, avec la vérification de non-débordement des entiers non signés :

```
/*@ 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;
    }
  }
```

Nous pouvons voir que le prouveur échoue à montrer la propriété liée à la plage de valeur de `i`, et que `i` peut excéder la valeur maximale d'un entier non signé. Nous pouvons essayer de fournir une limite à la valeur de `strlen(s)` en précondition.

```
1  requires valid_read_string(s) && strlen(s) <= SIZE_MAX ;
```

Cependant, c'est insuffisant. La raison et que si nous avons défini la valeur de `strlen(s)` comme l'index du premier `'\0'` dans le tableau, l'inverse n'est pas vrai : savoir que la valeur de `strlen(s)` est positive n'est pas suffisant pour déduire que la valeur à l'indice correspondant est `'\0'`. Nous étendons donc notre définition axiomatique avec une autre proposition indiquant cette propriété (nous ajoutons également une autre proposition à propos des valeurs qui précèdent cet indice même si ici, ce n'est pas nécessaire) :

```
1   axiom index_of_strlen{L}:
2     \forall char* s ;
3       strlen(s) >= 0 ==> s[strlen(s)] == '\0' ;
4
5   axiom before_strlen{L}:
6     \forall char* s ;
7       strlen(s) >= 0 ==> (\forall integer i ; 0 <= i < strlen(s) ==> s[i] != '\0') ;
```

Cette fois la preuve réussit. Frama-C fournit ses propres headers pour la bibliothèque standard, et cela inclut une définition axiomatique pour la fonction logique `strlen`. Elle peut être trouvée dans le dossier de Frama-C, sous le dossier `libc`, le fichier est nommé `__fc_string_axiomatic.h` Notons que cette définition a beaucoup plus d'axiomes pour déduire plus de propriétés à propos de `strlen`.

6.2.7. Exercices

6.2.7.1. Comptage d'occurrences

Le programme suivant ne peut pas être prouvé avec la définition axiomatique que nous avons défini précédemment à propos du comptage d'occurrences :

```
/*@
  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;
}
```

Ré-exprimer la définition axiomatique dans une forme qui permet de prouver le programme.

6.2.7.2. Plus grand diviseur commun

Exprimer la fonction logique qui calcule le plus grand diviseur commun à l'aide d'une définition axiomatique et écrire le contrat de la fonction `gcd` puis la prouver ;

```
#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. Somme des N premiers entiers

Exprimer la fonction logique qui calcule la somme des N premiers entiers à l'aide d'une définition axiomatique. Écrire le contrat de la fonction `sum_n` et la prouver :

```c
#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

Reprendre l'exemple à propos du tri par sélection (section 6.1.3). Ré-exprimer le prédicat de permutation comme une définition axiomatique. Prendre garde à la clause `reads` (en particulier, noter que le prédicat relie deux labels mémoire).

```c
#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];
```

```
30    ensures sorted(a, beg, end);
31    ensures permutation{Pre, Post}(a,beg,end);
32  */
33  void sort(int* a, size_t beg, size_t end){
34    /*@
35      loop invariant beg <= i <= end;
36      loop invariant sorted(a, beg, i) && permutation{Pre, Here}(a, beg, end);
37      loop invariant \forall integer j,k; beg <= j < i ==> i <= k < end ==> a[j] <= a[k];
38      loop assigns i, a[beg .. end-1];
39      loop variant end-i;
40    */
41    for(size_t i = beg ; i < end ; ++i){
42      //@ ghost begin: ;
43      size_t imin = min_idx_in(a, i, end);
44      swap(&a[i], &a[imin]);
45      //@ assert swap_in_array{begin,Here}(a,beg,end,i,imin);
46    }
47  }
```

6.3. Code fantôme

Les techniques que nous avons vues précédemment dans ce chapitre ont pour but de rendre la spécification plus abstraite. Le rôle du code fantôme est inverse. Ici, nous enrichirons nos spécifications à l'aide d'information exprimées en langage C. L'idée est d'ajouter des variables et du code source qui n'intervient pas dans le programme réel, mais permettant de créer des états logiques qui ne seront visibles que depuis la preuve. Par cet intermédiaire, nous pouvons rendre explicites des propriétés logiques qui étaient auparavant implicites.

6.3.1. Syntaxe

Le code fantôme est ajouté par l'intermédiaire d'annotations qui contiennent du code C ainsi que la mention `ghost`.

```
1  /*@
2    ghost
3    // code en langage C
4  */
```

Dans un code fantôme, nous écrivons du C normal. Nous expliquerons certaines petites subtilités plus tard. Pour l'instant, intéressons-nous aux éléments basiques que nous pouvons écrire avec du code fantôme.

Nous pouvons déclarer des variables :

```
1  //@ ghost int ghost_glob_var = 0;
2
3  void foo(int a){
4    //@ ghost int ghost_loc_var = a;
5  }
```

Ces variables peuvent être modifiées via des opérations et structures conditionnelles :

```
//@ 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 ;
    }
  */
}
```

Nous pouvons déclarer des *labels* fantômes, que l'on peut utiliser dans nos annotations (ou même pour effectuer un `goto` depuis le code fantôme lui-même, sous certaines conditions que nous expliquerons plus tard) :

```
void foo(int a){
  //@ ghost Ghost_label: ;
  a = 28 ;
  //@ assert ghost_loc_var == \at(a, Ghost_label) == \at(a, Pre);
}
```

Une construction conditionnelle `if` peut être étendue avec un `else` fantôme s'il n'a pas de `else` à la base. Par exemple :

```
void foo(int a){
  //@ ghost int a_was_ok = 0 ;
  if(a < 5){
    a = 5 ;
  } /*@ ghost else {
    a_was_ok = 1 ;
  } */
}
```

Une fonction peut avoir des paramètres fantômes, cela permet de transmettre des informations supplémentaires pour la vérification de la fonction. Par exemple, si l'on imagine la vérification d'une fonction qui reçoit une liste chaînée, nous pourrions transmettre un paramètre fantôme qui représenterait la longueur de la liste :

```
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
}
```

Notons que si une fonction prend des paramètres fantômes, tous les appels doivent fournir les arguments fantômes correspondants.

Une fonction toute entière peut être fantôme. Par exemple, nous pourrions avoir une fonction fantôme qui calcule la longueur d'une liste que nous aurions utilisée au sein du code précédent :

```
/*@ 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 argument
}
```

Ici, nous pouvons voir une syntaxe spécifique pour le contrat de la fonction fantôme. En effet, il est souvent utile d'écrire des contrats ou des assertions dans du code fantôme. Comme nous devons écrire ces spécifications dans du code qui est déjà englobé dans des commentaires C, nous avons accès à une syntaxe spécifique pour fournir des contrats ou des assertions fantômes. Nous ouvrons une annotation fantôme avec la syntaxe `/@` et nous la fermons avec `@/`. Cela s'applique aussi aux boucles dans le code fantôme par exemple :

```
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. Validité du code fantôme, ce que Frama-C vérifie

Frama-C vérifie plusieurs propriétés à propos du code fantôme que nous écrivons :

- le code fantôme ne peut pas modifier le graphe de flot de contrôle du programme ;
- le code normal ne peut pas accéder à la mémoire fantôme ;
- le code fantôme ne peut modifier qu'une zone de mémoire fantôme.

Très simplement, ces propriétés cherchent à garantir que pour n'importe quel programme, son comportement observable pour toute entrée est le même avec ou sans le code fantôme.

Avant Frama-C 21 Scandium, la plupart de ces propriétés n'étaient pas vérifiées par le noyau de Frama-C. Par conséquent, si l'on travaille avec une version antérieure, il faut s'assurer soi-même que ces propriétés sont vérifiées.

Si certaines de ces propriétés ne sont pas vérifiées, cela voudrait dire que le code fantôme peut changer le comportement du programme vérifié. Analysons de plus prêt chacune de ces contraintes.

6.3.2.1. Maintien du flot de contrôle

Le flot de contrôle d'un programme est l'ordre dans lequel les instructions sont exécutées par le programme. Si le code fantôme change cet ordre, ou permet de ne plus exécuter certaines instructions du programme d'origine, alors le comportement du programme n'est plus le même, et nous ne vérifions donc plus le même programme.

Par exemple, la fonction suivante calcule la somme des n premiers entiers :

```
1  int sum(int n){
2    int x = 0 ;
3    for(int i = 0; i <= n; ++i){
4      //@ ghost break;
5      x += i ;
6    }
7    return x;
8  }
```

Par l'introduction, dans du code fantôme, de l'instruction `break` dans le corps de la boucle, le programme n'a plus le même comportement : au lieu de parcourir l'ensemble des i de 0 à $n + 1$, la boucle s'arrête dès la première itération. En conséquence, ce programme sera rejeté par Frama-C :

```
1  [kernel:ghost:bad-use] file.c:4: Warning:
2    Ghost code breaks CFG starting at:
3    /*@ ghost break; */
4    x += i;
```

Il est important de noter que lorsqu'un code fantôme altère le flot de contrôle, c'est le point de départ du code fantôme qui est pointé par l'erreur, par exemple si nous introduisons une conditionnelle autour de notre `break` :

```
1  int sum(int n){
2    int x = 0 ;
3    for(int i = 0; i <= n; ++i){
4      //@ ghost if(i < 3) break;
5      x += i ;
6    }
7    return x;
8  }
```

Le problème est indiqué pour le `if` englobant :

```
[kernel:ghost:bad-use] file.c:4: Warning:
  Ghost code breaks CFG starting at:
  /*@ ghost if (i < 3) break; */
  x += i;
```

Notons que la vérification que le flot de contrôle n'est pas altéré est purement syntaxique. Par exemple, si le `break` est inatteignable parce que la condition est toujours fausse, une erreur sera quand même levée :

```
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;
```

Finalement, remarquons qu'il existe deux manières générales d'altérer le flot de contrôle. La première est d'utiliser un saut (donc `break`, `continue`, ou `goto`), la seconde est d'introduire un code non terminant. Pour ce dernier, à moins que le code soit trivialement non terminant, le noyau ne peut pas vérifier la non-altération du flot de contrôle, et ne le fait donc jamais. Nous traiterons cette question dans la section 6.3.3.

6.3.2.2. Accès à la mémoire

Le code fantôme est un observateur du code normal. En conséquence, le code normal n'est pas autorisé à accéder au code fantôme, que ce soit sa mémoire ou ses fonctions. Le code fantôme lui, peut lire la mémoire du code normal, mais ne peut pas la modifier. Actuellement, le code fantôme ne peut pas non plus appeler de fonctions du code normal, nous parlerons de cette restriction à la fin de cette section.

Refuser que le code normal voie le code fantôme a une raison toute simple. Si le code normal tentait d'accéder à des variables fantômes, il ne pourrait même pas être compilé : le compilateur ne voit pas les variables déclarées dans les annotations. Par exemple :

```
int sum_42(int n){
  int x = 0 ;
  //@ ghost int r = 42 ;
  for(int i = 0; i < n; ++i){
    x += r;
  }
```

```
  return x;
}
```

ne peut pas être compilé :

```
# 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;
      |          ^
```

et n'est donc pas accepté par Frama-C non plus :

```
[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;
```

Dans le code fantôme, les variables normales ne doivent pas être modifiées. En effet, cela impliquerait de pouvoir par exemple modifier le résultat d'un programme en ajoutant du code fantôme. Par exemple dans le code suivant :

```
int sum(int n){
  int x = 0 ;
  for(int i = 0; i <= n; ++i){
    x += i ;
    //@ ghost x++;
  }
  return x;
}
```

Le résultat du programme ne serait pas le même avec ou sans le code fantôme. Frama-C interdit donc un tel code :

```
[kernel:ghost:bad-use] file.c:5: Warning:
  'x' is a non-ghost lvalue, it cannot be assigned in ghost code
```

Notons que cette vérification est faite grâce au type des différentes variables. Une variable déclarée dans du code normal a un statut de variable normale, tandis qu'une variable déclarée dans du code fantôme a un statut de variable fantôme. Par conséquent, une nouvelle fois, même si le code fantôme, dans les faits, n'altère pas le comportement du programme, toute écriture d'une variable normale dans le code fantôme est interdite :

```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
```

Cela s'applique également à la clause `assigns` lorsqu'elle est dans du code fantôme :

```c
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
```

En revanche, les contrats des fonctions et boucles normales peuvent (et doivent) permettre de spécifier les zones de mémoire fantôme modifiées. Par exemple, si nous corrigeons le petit programme précédent en rendant `x` fantôme, d'une part nos clauses `assigns` précédentes sont bien acceptées, mais en plus, nous pouvons spécifier que la fonction `foo` modifie la variable globale fantôme `x` :

```c
//@ 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. Typage des éléments fantômes

Il convient de donner quelques précisions au sujet des types des variables créées dans du code fantôme. Par exemple, parfois il est intéressant de pouvoir créer un tableau fantôme pour stocker des informations :

```
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;
  }
}
```

Ici, nous utilisons des indices pour accéder à nos tableaux, mais nous pourrions par exemple vouloir y accéder en utilisant un pointeur :

```
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++;
  }
}
```

Mais nous voyons immédiatement que Frama-C n'est pas d'accord avec notre manière de faire :

```
[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
```

En particulier, le premier message nous indique que nous essayons de transformer un pointeur sur `int \ghost` en pointeur sur `int`. En effet, lorsqu'une variable est déclarée dans du code fantôme, seule la variable est considérée fantôme. Donc dans le cas d'un pointeur, la mémoire pointée par ce pointeur, elle, n'est pas considérée comme fantôme (et donc ici, bien que `pe` soit fantôme, la mémoire pointée par `pe` ne l'est pas). Pour résoudre ce problème, Frama-C nous offre le qualificateur `\ghost`, qui nous permet d'ajouter un caractère fantôme à un type :

```
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++;
  }
}
```

Sur certains aspects, le qualificateur `\ghost` ressemble au mot-clé `const` du C. Cependant, son comportement n'est pas exactement le même pour deux raisons.

Tout d'abord, alors que la définition `const` suivante est autorisée, il n'est pas possible d'avoir une déclaration de forme similaire avec le qualificateur `\ghost` :

```
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
```

Déclarer un pointeur constant sur une zone que l'on peut modifier et qui contient des pointeurs vers de la mémoire constante ne pose pas de problème. En revanche, il est impossible de faire de même avec le qualificateur `\ghost`, cela signifierait que la mémoire normale contient des pointeurs vers la mémoire fantôme, ce qui n'est pas possible.

D'autre part, il est possible d'assigner un pointeur vers des données non-constantes à un pointeur vers des données constantes :

```
int a[10] ;
int const * p = a ;
```

Ce code ne pose pas de problème, car l'on ne fait que restreindre notre capacité à modifier les données lorsque l'on initialise (ou affecte) `p` à `&a[0]`. En revanche, les deux initialisations (ou affectations équivalentes) des pointeurs suivants sont refusées avec le qualificateur `\ghost` :

```
int non_ghost_int ;
//@ ghost int ghost_int ;

//@ ghost int \ghost * p = & non_ghost_int ;
//@ ghost int * q = & ghost_int ;
```

Si la raison du refus de la première initialisation est tout à fait directe : elle permettrait de modifier le contenu de la mémoire normale depuis du code fantôme, refuser la seconde peut être un peu moins intuitif. Et en effet, nous devons passer par des moyens détournés pour provoquer un problème avec cette 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 ;
}
```

Ici, nous faisons une conversion qui pourrait sembler autorisée. En effet, nous passons l'adresse d'un pointeur sur une zone de mémoire fantôme à une fonction qui attend un pointeur sur une zone de mémoire normale, cela ne fait que restreindre l'accès à la mémoire pointée. Cependant, par cet appel de fonction, la fonction `assign` assigne la valeur actuelle de `q` (qui est `&x`) à `p` et nous permet donc, par la dernière opération de modifier `x` dans du code fantôme. En conséquence, une telle conversion n'est jamais autorisée.

Finalement, le code fantôme ne peut actuellement pas appeler de fonction qui n'est pas fantôme, pour des raisons semblables à celle évoquée pour l'interdiction de toutes les conversions. Certains cas particuliers pourraient être traités de manière à accepter plus de code, mais ce n'est actuellement pas supporté par Frama-C.

6.3.3. Validité du code fantôme, ce qu'il reste à vérifier

Mis à part les restrictions que nous avons mentionnées dans la section précédente, le code fantôme est juste du code C normal. Cela veut dire que si nous voulons faire la vérification de notre programme d'origine, nous devons faire attention, nous-mêmes, à au moins deux aspects supplémentaires :

— l'absence d'erreurs à l'exécution,
— la terminaison du code fantôme.

Le premier cas ne nécessite pas plus d'attention que le reste de notre code. En effet, la vérification d'absence d'erreurs à l'exécution sera traitée par le plugin RTE comme pour le reste de notre programme.

Comme nous l'avons dit dans la section 4.2.3, il y a deux sortes de correction : la correction partielle et la correction totale, la seconde permettant de prouver qu'un programme termine. Dans le cas du code normal, montrer la terminaison n'est pas toujours souhaitable pour l'ensemble du programme. En revanche, si nous utilisons du code fantôme pour aider la vérification, montrer que la correction est totale est absolument nécessaire, car une boucle infinie dans le code fantôme peut nous permettre de prouver n'importe quoi à propos du programme.

```
/*@ ensures \false ; */
void foo(void){
  /*@ ghost
    while(1){}
```

```
5   */
6 }
```

6.3.4. Expliciter un état logique

Le but du code fantôme est de rendre explicite des informations généralement implicites. Par exemple, dans la vérification de l'algorithme de tri, nous nous en sommes servi pour ajouter un *label* dans le programme qui n'est pas visible par le compilateur, mais que nous avons pu utiliser pour la vérification. Le fait que les valeurs ont été échangées entre les deux points de programme était implicitement garanti par le contrat de la fonction d'échange, ajouter ce *label* fantôme nous a donné la possibilité de rendre cette propriété explicite par une assertion.

Prenons maintenant un exemple plus poussé. Nous voulons par exemple prouver que la fonction suivante nous retourne la valeur maximale des sommes de sous-tableaux possibles d'un tableau donné. Un sous-tableau d'un tableau `a` est un sous-ensemble contigu de valeur de `a`. Par exemple, pour un tableau `{ 0 , 3 , -1 , 4 }`, des exemples de sous tableaux peuvent être `{}`, `{ 0 }`, `{ 3 , -1 }`, `{ 0 , 3 , -1 , 4 }`, ... Notons que comme nous autorisons le tableau vide, la somme est toujours au moins égale à 0. Dans le tableau précédent, le sous-tableau de valeur maximale est `{ 3 , -1 , 4 }`, la fonction renverra donc 6.

```
1  int max_subarray(int *a, size_t len) {
2    int max = 0;
3    int cur = 0;
4    for(size_t i = 0; i < len; i++) {
5      cur += a[i];
6      if (cur < 0)   cur = 0;
7      if (cur > max) max = cur;
8    }
9    return max;
10 }
```

Pour spécifier la fonction précédente, nous aurons besoin d'exprimer axiomatiquement la somme. Ce n'est pas très complexe et le lecteur pourra s'exercer en exprimant les axiomes nécessaires au bon fonctionnement de cette axiomatique :

```
1  /*@
2    axiomatic Sum_array{
3      logic integer sum(int* array, integer begin, integer end) reads array[begin .. (end-1)];
```

La correction est disponible à la section 6.4.

La spécification de notre fonction est la suivante :

```
1  /*@
2    requires \valid(a+(0..len-1));
3    assigns \nothing;
4    ensures \forall integer l, h;  0 <= l <= h <= len ==> sum(a,l,h) <= \result;
5    ensures \exists integer l, h;  0 <= l <= h <= len &&  sum(a,l,h) == \result;
6  */
```

Pour toute paire de bornes, la valeur retournée par la fonction doit être supérieure ou égale à la somme des éléments entre les bornes, et il doit exister une paire de bornes telle que la somme des éléments entre ces bornes est exactement la valeur retournée par la fonction. Par rapport à cette spécification, si nous devons ajouter les invariants de boucles, nous nous apercevons rapidement qu'il nous manquera des informations. Nous avons besoin d'exprimer ce que sont les valeurs `max` et `cur`, et quelles relations existent entre elles, mais rien ne nous le permet !

En substance, notre postcondition a besoin de savoir qu'il existe des bornes `low` et `high` telles que la somme calculée correspond à ces bornes. Or, notre code n'exprime rien de tel d'un point de vue logique et rien ne nous permet *a priori* de faire cette liaison en utilisant des formulations logiques. Nous utiliserons du code *ghost* pour conserver ces bornes et exprimer l'invariant de notre boucle.

Nous aurons d'abord besoin de deux variables qui nous permettront de stocker les valeurs des bornes de la plage maximum, nous les appellerons `low` et `high`. Chaque fois que nous trouverons une plage où la somme est plus élevée nous les mettrons à jour. Ces bornes correspondront donc à la somme indiquée par `max`. Cela induit que nous avons encore besoin d'une autre paire de bornes : celle correspondant à la variable de somme `cur` à partir de laquelle nous pourrons construire les bornes de `max`. Pour celle-ci, nous n'avons besoin que d'ajouter une variable *ghost* : le minimum actuel `cur_low`, la borne supérieure de la somme actuelle étant indiquée par la variable `i` de la boucle.

```
1  /*@
2    requires \valid(a+(0..len-1));
3    assigns \nothing;
4    ensures \forall integer l, h;  0 <= l <= h <= len ==> sum(a,l,h) <= \result;
5    ensures \exists integer l, h;  0 <= l <= h <= len &&  sum(a,l,h) == \result;
6  */
7  int max_subarray(int *a, size_t len) {
8    int max = 0;
9    int cur = 0;
10   //@ ghost size_t cur_low = 0;
11   //@ ghost size_t low = 0;
12   //@ ghost size_t high = 0;
13
14   /*@
15     loop invariant BOUNDS: low <= high <= i <= len && cur_low <= i;
16
17     loop invariant REL :   cur == sum(a,cur_low,i) <= max == sum(a,low,high);
18     loop invariant POST:   \forall integer l;    0 <= l <= i      ==> sum(a,l,i) <= cur;
19     loop invariant POST:   \forall integer l, h; 0 <= l <= h <= i ==> sum(a,l,h) <= max;
20
21     loop assigns i, cur, max, cur_low, low, high;
22     loop variant len - i;
23   */
24   for(size_t i = 0; i < len; i++) {
25     cur += a[i];
26     if (cur < 0) {
27       cur = 0;
28       /*@ ghost cur_low = i+1; */
29     }
30     if (cur > max) {
31       max = cur;
32       /*@ ghost low = cur_low; */
33       /*@ ghost high = i+1; */
34     }
35   }
36   return max;
37 }
```

L'invariant `BOUNDS` exprime comment sont ordonnées les différentes bornes pendant le calcul. L'invariant `REL` exprime ce que signifient les valeurs `cur` et `max` par rapport à ces bornes. Finalement, l'invariant `POST` permet de faire le lien entre les invariants précédents et la postcondition de la fonction.

Le lecteur pourra vérifier que cette fonction est effectivement prouvée sans la vérification des RTE. Si nous ajoutons également le contrôle des RTE, nous pouvons voir que le calcul de la somme indique un dépassement possible sur les entiers.

Ici, nous ne chercherons pas à le corriger, car ce n'est pas l'objet de l'exemple. Le moyen de prouver cela dépend en fait fortement du contexte dans lequel on utilise la fonction. Une possibilité est de restreindre fortement le contrat en imposant des propriétés à propos des valeurs et de la taille du tableau. Par exemple, nous pourrions imposer une taille maximale et des bornes fortes pour chacune des cellules. Une autre possibilité est d'ajouter une valeur d'erreur en cas de dépassement (par exemple -1), et de spécifier qu'en cas de dépassement, c'est cette valeur qui est renvoyée.

6.3.5. Exercices

6.3.5.1. Validité du code ghost

Dans ces fonctions, sans exécuter Frama-C, expliquer en quoi le code fantôme pose problème. Lorsque Frama-C devrait rejeter le code, expliquer pourquoi. Notons qu'il est possible d'exécuter Frama-C sans contrôle du code fantôme en utilisant l'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. Multiplication par 2

Le programme suivant calcule `2 * x` en utilisant une boucle. Utiliser une variable fantôme `i` pour exprimer comme invariant que la valeur de `r` est `i * 2` et compléter la preuve.

```
/*@
  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. Tableaux

Cette fonction reçoit un tableau et effectue une boucle dans laquelle nous ne faisons rien, sauf que nous avons indiqué que le contenu du tableau est modifié. Cependant, nous voudrions pouvoir prouver qu'en postcondition, le tableau n'a pas été modifié.

```
/*@
  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
  */

  /*@
```

```
13      loop invariant 0 <= i <= 10 ;
14      loop invariant \true ; // to complete
15      loop assigns i, a[0 .. 9] ;
16      loop variant 10 - i ;
17    */
18    for(int i = 0; i < 10; i++);
19  }
```

Sans modifier la clause `assigns` de la boucle et sans utiliser le mot clé `\at`, prouver que la fonction ne modifie pas le contenu du tableau. Pour cela, compléter le code fantôme et l'invariant de boucle en assurant que le contenu du tableau `g` représente l'ancien contenu de `a`.

Lorsque c'est fait, créer une fonction fantôme qui effectue cette même copie, et l'utiliser dans la fonction `foo` pour effectuer la même preuve.

6.3.5.4. Chercher et remplacer

Le programme suivant effectue une opération de recherche et remplacement :

```
1  #include <stddef.h>
2
3  void replace(int *a, size_t length, int old, int new) {
4    for (size_t i = 0; i < length; ++i) {
5      if (a[i] == old)
6        a[i] = new;
7    }
8  }
9
10 /*@
11   requires \valid(a + (0 .. length-1));
12   assigns a[0 .. length-1];
13   ensures \forall integer i ; 0 <= i < length ==> -100 <= a[i] <= 100 ;
14 */
15 void initialize(int *a, size_t length);
16
17 void caller(void) {
18   int a[40];
19
20   initialize(a, 40);
21
22   //@ ghost L: ;
23
24   replace(a, 40, 0, 42);
25
26   // here we want to obtain the updated locations via a ghost array
27 }
```

En supposant que la fonction `replace` demande à ce que `old` et `new` soient différents, écrire un contrat pour `replace` et prouver que la fonction le satisfait.

Maintenant, nous voudrions savoir quelles cellules du tableau ont été mises à jour par la fonction. Ajouter un paramètre fantôme à la fonction `replace` de manière à pouvoir recevoir un second tableau qui servira à enregistrer les cellules mises à jour (ou non) par la fonction. En ajoutant également le code suivant après l'appel à `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); @/
      }
    }
  */

```

Tout devrait être prouvé.

6.4. Contenu caché

6.4.1. Preuve Coq du lemme `no_changes`

```
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

```

```
40          holds on the first part of the array *)
41
42      apply IHP_zeroed ; auto.
43      intros i' ; intros.
44      apply Same ; lia.
45  Qed.
```

6.4.2. Fonctions utilisées pour le tri spécifiées

```
1  /*@
2    requires \valid_read(a + (beg .. end-1));
3    requires beg < end;
4
5    assigns  \nothing;
6
7    ensures  \forall integer i; beg <= i < end ==> a[\result] <= a[i];
8    ensures  beg <= \result < end;
9  */
10 size_t min_idx_in(int* a, size_t beg, size_t end){
11   size_t min_i = beg;
12
13   /*@
14     loop invariant beg <= min_i < i <= end;
15     loop invariant \forall integer j; beg <= j < i ==> a[min_i] <= a[j];
16     loop assigns min_i, i;
17     loop variant end-i;
18   */
19   for(size_t i = beg+1; i < end; ++i){
20     if(a[i] < a[min_i]) min_i = i;
21   }
22   return min_i;
23 }
24
25 /*@
26   requires \valid(p) && \valid(q);
27   assigns  *p, *q;
28   ensures  *p == \old(*q) && *q == \old(*p);
29 */
30 void swap(int* p, int* q){
31   int tmp = *p; *p = *q; *q = tmp;
32 }
```

6.4.3. Un important axiome

Actuellement, nos prouveurs automatiques n'ont pas la puissance nécessaire pour calculer *la réponse à la grande question sur la vie, l'univers et le reste*. Qu'à cela ne tienne nous pouvons l'énoncer comme axiome ! Reste à comprendre la question pour savoir où ce résultat peut être utile ...

```
1  /*@
2    axiomatic Ax_answer_to_the_ultimate_question_of_life_the_universe_and_everything {
3      logic integer the_ultimate_question_of_life_the_universe_and_everything{L} ;
4
5      axiom answer{L}:
6        the_ultimate_question_of_life_the_universe_and_everything{L} = 42;
7    }
```

```
*/
```

6.4.4. Axiomes pour la somme des éléments d'un tableau

```
/*@
  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];
  }
*/
```

Dans cette partie, nous avons vu des constructions plus avancées du langage ACSL qui nous permettent d'exprimer et de prouver des propriétés plus complexes à propos de nos programmes.

Mal utilisées, ces fonctionnalités peuvent fausser nos analyses, il faut donc se montrer attentif lorsque nous manipulons ces constructions et ne pas hésiter à les relire ou encore à exprimer des propriétés à vérifier à leur sujet afin de s'assurer que nous ne sommes pas en train d'introduire des incohérences dans notre programme ou nos hypothèses de travail.

7. Méthodologies de preuve

Maintenant que nous avons présenté les fonctionnalités les plus importantes d'ACSL pour la preuve de programme, intéressons nous plus spécifiquement à la manière de prouver un programme avec Frama-C et WP. Nous allons présenter différentes approches qui peuvent être utilisées, selon la cible de vérification, le type de propriétés que l'on cherche à montrer et les fonctionnalités d'ACSL que nous utilisons.

7.1. Absence d'erreurs à l'exécution : contrats minimaux

Nous avons vu que la preuve d'un programme permet de vérifier deux aspects principaux à propos de sa correction ; d'abord que le programme ne contient pas d'erreur d'exécution, et ensuite que le programme répond correctement à sa spécification. Cependant, il est parfois difficile d'obtenir le second aspect et le premier est déjà une étape intéressante pour la correction de notre programme.

En effet, les erreurs à l'exécution entraînent souvent la présence des fameux « *undefined behaviors* » dans les programmes C. Ces comportements peuvent être des vecteurs de failles de sécurité. Par conséquent, garantir leur absence nous protège déjà d'un grand nombre de ces vecteurs d'attaques. L'absence d'erreur à l'exécution peut être vérifiée avec WP à l'aide d'une approche appelée « contrats minimaux ».

7.1.1. Principe

L'approche par contrats minimaux est guidée par l'usage du greffon RTE de Frama-C. L'idée est simple : pour toutes les fonctions d'un module ou d'un projet, nous générons les assertions nécessaires à vérifier l'absence d'erreurs à l'exécution, et nous écrivons ensuite l'ensemble des spécifications (correctes) qui sont suffisantes pour prouver ces assertions et les contrats ainsi rédigés. La plupart du temps, cela permet d'avoir beaucoup moins de lignes de spécifications que ce qui est nécessaire pour prouver la correction fonctionnelle du programme.

Commençons par un exemple simple avec la fonction valeur absolue.

```
1 int abs(int x){
2   return (x < 0) ? -x : x ;
3 }
```

Ici, nous pouvons générer les assertions nécessaires à prouver pour montrer l'absence d'erreurs à l'exécution, ce qui génère ce programme :

```
/* 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;
}
```

Donc nous avons seulement besoin de spécifier la précondition qui nous dit que `x` doit être plus grand que `INT_MIN` :

```
/*@
  requires x > INT_MIN ;
*/
int abs(int x){
  return (x < 0) ? -x : x ;
}
```

Cette condition est suffisante pour montrer l'absence d'erreurs à l'exécution dans cette fonction.

Comme nous le verrons plus tard, généralement une fonction est cependant utilisée dans un contexte particulier. Il est donc probable que ce contrat ne soit en réalité pas suffisant pour assurer la correction dans son contexte d'appel. Par exemple, il est commun en C d'avoir des variables globales ou des pointeurs, il est donc probable que nous devions spécifier ce qui est assigné par la fonction. La plupart du temps, les clauses `assigns` ne peuvent pas être ignorées (ce qui est prévisible dans un langage où tout est mutable par défaut). De plus, si une personne demande la valeur absolue d'un entier, c'est probablement qu'elle a besoin d'une valeur positive. En réalité, le contrat ressemblera probablement à ceci :

```
/*@
  requires x > INT_MIN ;
  assigns  \nothing ;
  ensures  \result >= 0 ;
*/
int abs(int x){
  return (x < 0) ? -x : x ;
}
```

Mais cette addition ne devrait être guidée que par la vérification du ou des contextes dans lesquels la fonction est appelée, une fois que nous avons prouvée l'absence d'erreur d'exécution dans cette fonction.

7.1.2. Exemple : la fonction recherche

Maintenant que nous avons le principe en tête, travaillons avec un exemple un peu plus complexe, Celui-ci en particulier nécessite une boucle.

```c
#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;
}
```

Lorsque nous générons les assertions liées aux erreurs à l'exécution, nous obtenons le programme suivant :

```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;
}
```

Nous devons prouver que toute cellule visitée par le programme peut être lue, nous avons donc besoin d'exprimer comme précondition que ce tableau est `\valid_read` sur la plage de valeurs correspondante. Cependant, ce n'est pas suffisant pour terminer la preuve puisque nous avons une boucle dans ce programme. Nous devons donc aussi fournir un invariant, nous voulons aussi probablement prouver que la boucle termine.

Nous obtenons donc la fonction suivante, spécifiée minimalement :

```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;
}
```

Ce contrat peut être comparé avec le contrat fourni pour la fonction de recherche de la section 4.3.2, et nous pouvons voir qu'il est beaucoup plus simple.

Maintenant imaginons que cette fonction est utilisée dans le programme suivant :

```
void foo(int* array, size_t length){
  int* p = search(array, length, 0) ;
  if(p){
    *p += 1 ;
  }
}
```

Nous devons à nouveau fournir un invariant pour cette fonction, à nouveau en regardant l'assertion générée par le plugin RTE :

```
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;
}
```

Nous devons donc vérifier que :

— le pointeur que la fonction reçoit est valide,

— `*p+1` ne fait pas de débordement,

— la précondition de la fonction `search` est satisfaite.

En plus du contrat de `foo`, nous devons fournir plus d'informations dans le contrat de `search`. En effet, nous ne pourrons pas prouver que le pointeur est valide si la fonction ne nous garantit pas qu'il est dans la plage correspondant à notre tableau dans ce cas. De plus, nous ne pourrons pas prouver que `*p` a une valeur inférieure à `INT_MAX` si la fonction peut modifier le tableau.

Cela nous amène donc au programme complet annoté suivant :

```
#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)) ;
*/
```

```
25 void foo(int *array, size_t length){
26   int *p = search(array,length,0);
27   if (p){
28     *p += 1 ;
29   }
30 }
```

7.1.3. Avantages et limitations

L'avantage le plus évident de cette approche est le fait qu'elle permet de garantir qu'un programme ne contient pas d'erreurs à l'exécution dans toute fonction d'un module ou d'un programme en (relative) isolation des autres fonctions. De plus, cette absence d'erreurs à l'exécution est garantie pour tout usage de la fonction dont l'appel satisfait ses préconditions. Cela permet de gagner une certaine confiance dans un système avec une approche dont le coût est relativement raisonnable.

Cependant, comme nous avons pu le voir, lorsque nous utilisons une fonction, cela peut changer les connaissances que nous avons besoin d'avoir à son sujet, nécessitant d'enrichir son contrat progressivement. Nous pouvons par conséquent atteindre un point où nous avons prouvé la correction fonctionnelle de la fonction.

De plus, prouver l'absence d'erreur à l'exécution peut parfois ne pas être trivial comme nous avons pu le voir précédemment avec des fonctions comme la factorielle ou la somme des N premiers entiers, qui nécessitent de donner une quantité notable d'information aux solveurs SMT pour montrer qu'elle ne déborde pas.

Finalement, parfois les contrats minimaux d'une fonction ou d'un module sont simplement la spécification fonctionnelle complète. Et dans ce cas, effectuer la vérification d'absence d'erreur à l'exécution correspond à réaliser la vérification fonctionnelle complète du programme. C'est communément le cas lorsque nous devons travailler avec des structures de données complexes où les propriétés dont nous avons besoin pour montrer l'absence d'erreurs à l'exécution dépendent du comportement fonctionnel des fonctions, maintenant des invariants non triviaux à propos de la structure de donnée.

7.1.4. Exercices

7.1.4.1. Exemple simple

Prouver l'absence d'erreurs à l'exécution dans le programme suivant en utilisant une approche par contrats minimaux :

```
1 void max_ptr(int* a, int* b){
2   if(*a < *b){
3     int tmp = *b ;
4     *b = *a ;
5     *a = tmp ;
6   }
7 }
8
9 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. Inverse

Prouver l'absence d'erreurs à l'exécution dans la fonction `reverse` suivante et ses dépendances en utilisant une approche par contrats minimaux. Notons que la fonction `swap` doit également être spécifiée par contrats minimaux. Ne pas oublier d'ajouter les options `-warn-unsigned-overflow` et `-warn-unsigned-downcast`.

```
#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. Recherche dichotomique

Prouver l'absence d'erreurs à l'exécution dans la fonction `bsearch` suivante en utilisant une approche par contrats minimaux. Ne pas oublier d'ajouter les options `-warn-unsigned-overflow` et `-warn-unsigned-downcast`.

```
#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 ;
```

```
16 }
```

7.1.4.4. Tri

Prouver l'absence d'erreurs à l'exécution dans la fonction `sort` et ses dépendances en utilisant une approche par contrats minimaux. Notons que ces dépendances doivent également être spécifiées par contrats minimaux. Ne pas oublier d'ajouter les options `-warn-unsigned-overflow` et `-warn-unsigned-downcast`.

```
1  #include <stddef.h>
2
3  size_t min_idx_in(int* a, size_t beg, size_t end){
4    size_t min_i = beg;
5    for(size_t i = beg+1; i < end; ++i){
6      if(a[i] < a[min_i]) min_i = i;
7    }
8    return min_i;
9  }
10
11 void swap(int* p, int* q){
12   int tmp = *p; *p = *q; *q = tmp;
13 }
14
15 void sort(int* a, size_t beg, size_t end){
16   for(size_t i = beg ; i < end ; ++i){
17     size_t imin = min_idx_in(a, i, end);
18     swap(&a[i], &a[imin]);
19   }
20 }
```

7.2. Assertions de guidage et déclenchement de lemmes

Il y a différents niveaux d'automatisation dans la vérification de programmes, depuis les outils complètement automatiques, comme les interpréteurs abstraits qui ne nécessitent aucune aide de la part de l'utilisateur (ou en tout cas, très peu), jusqu'aux outils interactifs comme les assistants de preuve, où les preuves sont principalement écrites à la main et l'outil est juste là pour vérifier que nous le faisons correctement.

Les outils comme WP (et beaucoup d'autres comme Why3, Spark, ...) visent à maximiser l'automatisation. Cependant, plus les propriétés que nous voulons prouver sont complexes, plus il sera difficile d'obtenir automatiquement toute la preuve. Par conséquent, nous devons souvent aider les outils pour terminer la vérification. Nous faisons souvent cela en fournissant plus d'annotations pour aider le processus de génération des conditions de vérification. Ajouter un invariant de boucle est par exemple une manière de fournir les informations nécessaires pour produire le raisonnement par induction permettant son analyse, alors que les prouveurs automatiques sont plutôt mauvais à cet exercice.

Cette technique de vérification a été appelée « *auto-active* ». Ce mot est la contraction de « *automatic* » et « *interactive* ». Elle est automatique au sens où la majorité de la preuve est effectuée par des outils automatiques, mais elle est aussi en partie interactive puisqu'en tant qu'utilisateurs, nous fournissons manuellement de l'information aux outils.

Dans cette section, nous allons voir plus en détails comment nous pouvons utiliser des assertions pour guider la preuve. En ajoutant des assertions, nous créons une certaine base de connaissances (des propriétés que nous savons vraies), qui sont collectées par le générateur de conditions de vérification pendant le calcul de WP et qui sont données aux prouveurs automatiques qui ont par conséquent plus d'information et peuvent potentiellement prouver des propriétés plus complexes.

7.2.1. Contexte de preuve

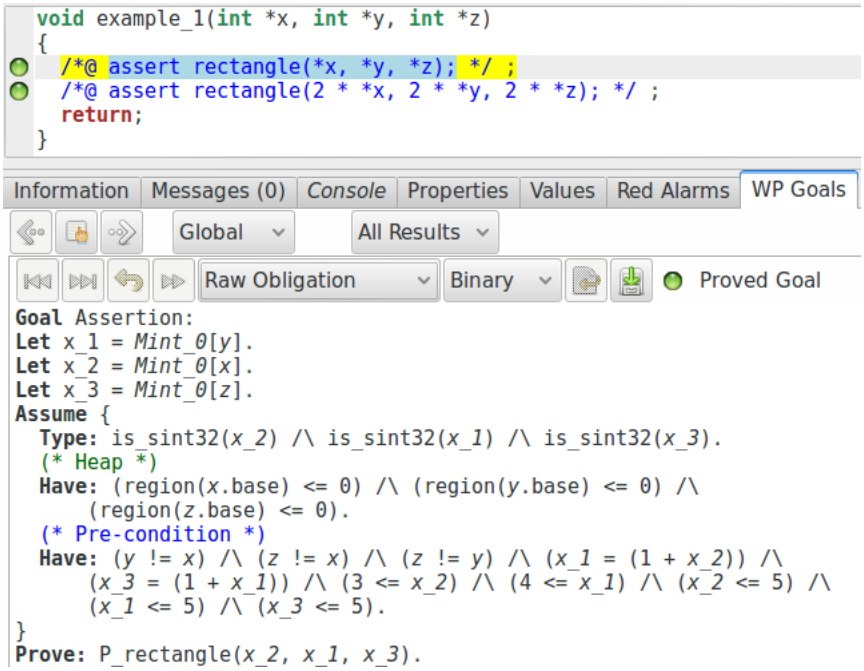
Pour comprendre exactement le bénéfice que représente l'ajout d'assertions dans les annotations d'un programme, commençons par regarder de plus près les conditions de vérification générées par WP à partir du code source annoté et comment les assertions sont prises en compte. Pour cela, nous allons utiliser le prédicat suivant (qui ressemble furieusement au théorème de Pythagore) :

```
/*@
  predicate rectangle{L}(integer c1, integer c2, integer h) =
    c1 * c1 + c2 * c2 == h * h ;
*/
```

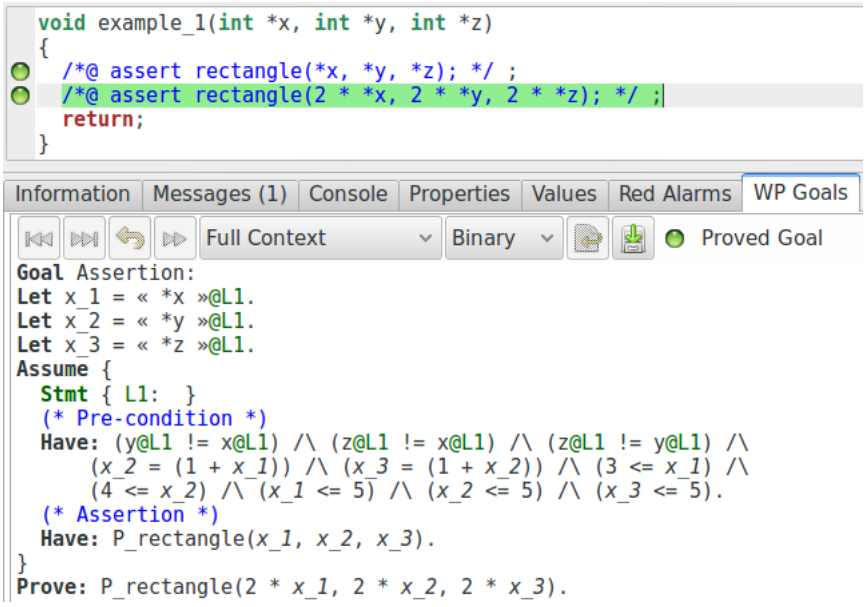
Regardons d'abord cet exemple :

```
/*@
  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));
}
```

Ici, nous avons spécifié une précondition suffisamment complexe pour que WP ne puisse par directement deviner les valeurs en entrée de fonction. En fait, ces valeurs sont exactement : `*x == 3`, `*y == 4` et `*z == 5`. Maintenant, si nous regardons la condition de vérification générée pour notre première assertion, nous pouvons voir ceci (il faut bien sélectionner la vue « *Full Context* » ou « *Raw obligation* » - elles ne sont pas exactement identiques, mais assez similaires, la première est juste légèrement plus jolie) :



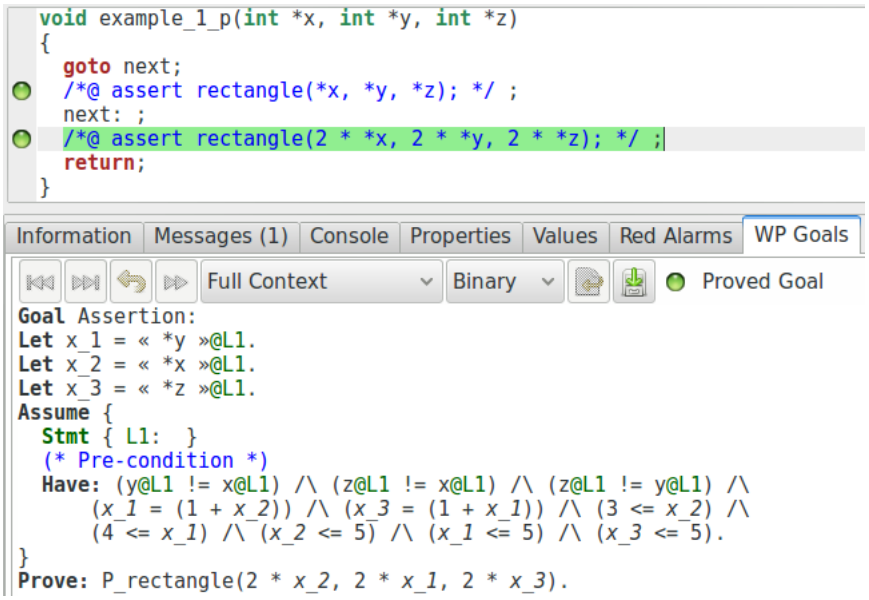
Nous y voyons les différentes contraintes que nous avons formulées comme préconditions de fonction (notons que les valeurs ne sont pas exactement les mêmes, et que quelques propriétés supplémentaires ont été générées). Maintenant, regardons plutôt la condition de vérification générée pour la seconde assertion (notons que nous avons édité les captures d'écran restantes de cette section pour nous concentrer sur ce qui est important, les autres propriétés pouvant être ignorées dans notre cas) :



Ici, nous pouvons voir que dans le contexte utilisable pour la preuve de la seconde assertion, WP a collecté et ajouté la première assertion et en a fait une supposition. WP considère que les solveurs SMT peuvent supposer que cette propriété est vraie. Cela signifie que les prouveurs peuvent l'utiliser, mais également qu'elle doit être prouvée pour que la condition de vérification actuelle soit complètement vérifiée.

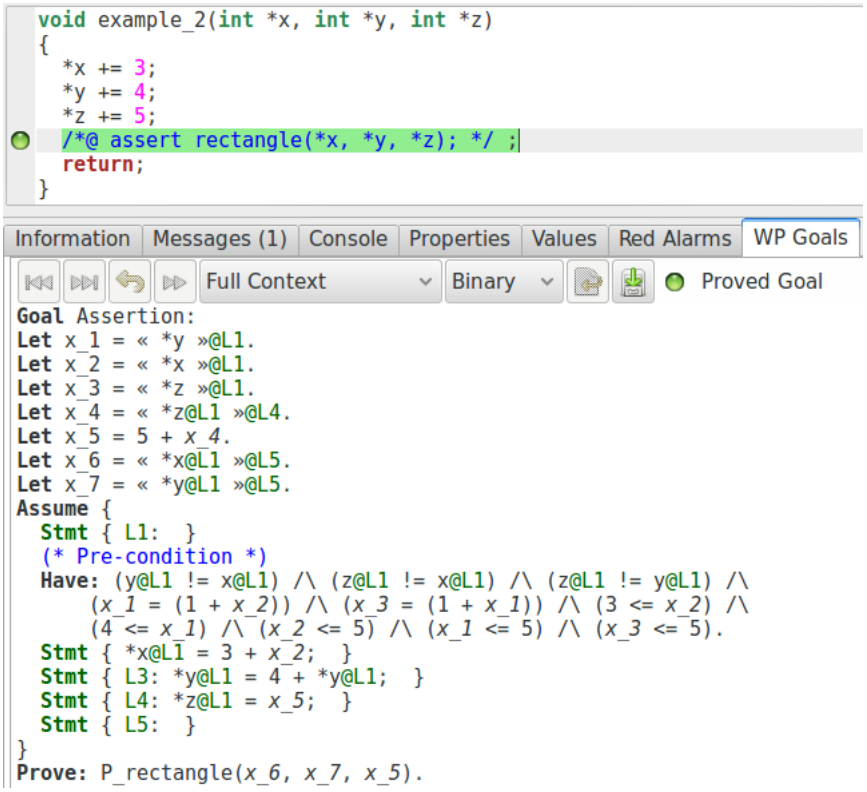
Notons que WP ne collecte que ce qu'il trouve sur les différents chemins d'exécution qui permettent d'atteindre l'assertion. Par exemple, si nous modifions le code de telle manière à ce que le chemin qui mène à la seconde assertion saute le chemin qui passe par la première, celle-ci n'apparaît pas dans le contexte de la seconde assertion.

```
1  void example_1_p(int* x, int* y, int* z){
2    goto next;
3    //@ assert rectangle(*x, *y, *z);
4   next: ;
5    //@ assert rectangle(2* (*x), 2* (*y), 2* (*z));
6  }
```



Maintenant, modifions un peu notre exemple de manière à illustrer comment les assertions peuvent changer la manière de prouver un programme. Par exemple, nous pouvons modifier les différentes positions mémoire (en doublant chaque valeur) et vérifier que le triangle résultant est rectangle.

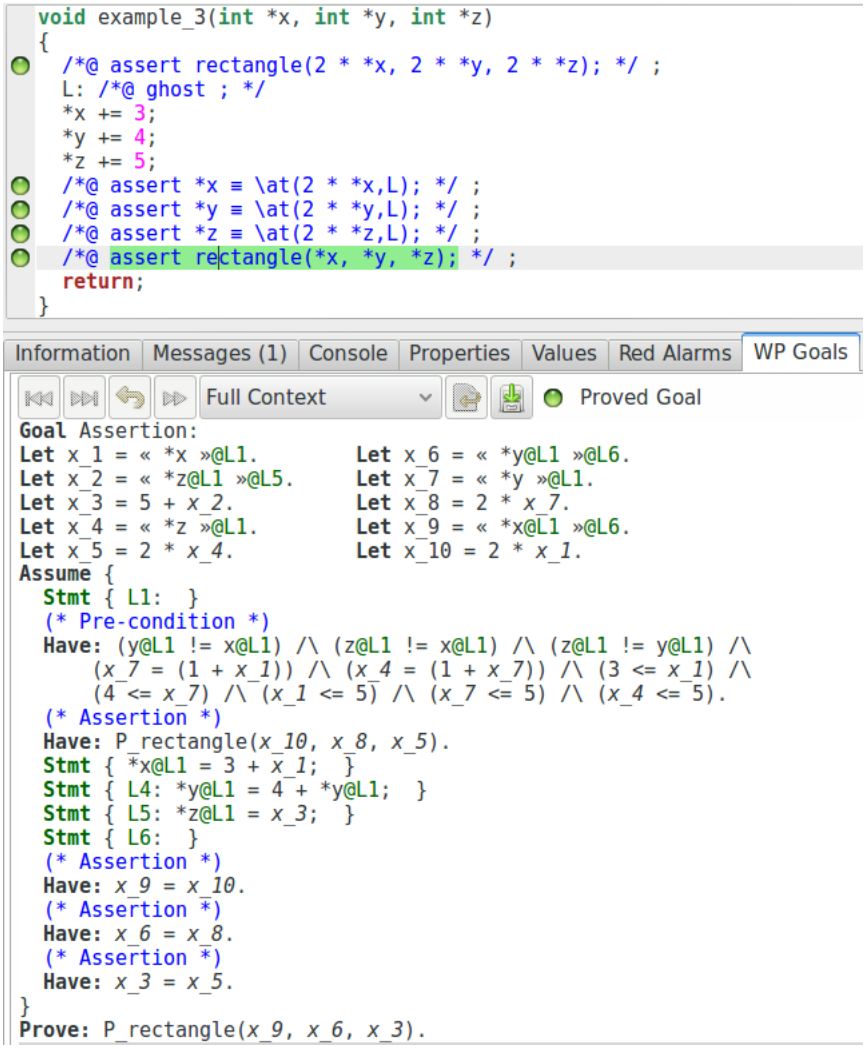
```
1  /*@
2    requires \separated(x, y , z);
3    requires 3 <= *x <= 5 ;
4    requires 4 <= *y <= 5 ;
5    requires *z <= 5 ;
6    requires *x+2 == *y+1 == *z ;
7  */
8  void example_2(int* x, int* y, int* z){
9    *x += 3 ;
10   *y += 4 ;
11   *z += 5 ;
12
13   //@ assert rectangle(*x, *y, *z);
14 }
```



Ici, le solveur déroulera probablement le prédicat et vérifiera directement que la propriété qui y est définie est vraie. En effet, depuis la condition de vérification, il n'y a pas vraiment d'autres informations qui pourrait nous amener à obtenir une preuve. Maintenant, ajoutons de l'information dans les annotations :

```
1  /*@
2    requires \separated(x, y , z);
3    requires 3 <= *x <= 5 ;
4    requires 4 <= *y <= 5 ;
5    requires *z <= 5 ;
6    requires *x+2 == *y+1 == *z ;
7  */
8  void example_3(int* x, int* y, int* z){
9    //@ assert rectangle(2* (*x), 2* (*y), 2* (*z));
10   //@ ghost L: ;
11
12   *x += 3 ;
13   *y += 4 ;
14   *z += 5 ;
15
16   //@ assert *x == \at(2* (*x), L) ;
17   //@ assert *y == \at(2* (*y), L) ;
18   //@ assert *z == \at(2* (*z), L);
19   //@ assert rectangle(*x, *y, *z);
20 }
```

Nous prouvons d'abord que si nous multiplions par 2 chacune des valeurs, le prédicat est vrai pour les nouvelles valeurs. Le solveur prouvera d'abord la même propriété, bien sûr, mais ce n'est pas ce que nous voulons montrer ici. Nous ajoutons ensuite que chaque valeur a été multipliée par 2. Maintenant, nous pouvons regarder la condition de vérification générée pour la dernière assertion :



Alors que nous devons prouver exactement la même propriété qu'avant (avec un peu de renommage), nous pouvons voir que nous avons une autre manière de la prouver. En effet, en combinant ces propriétés :

```
(* 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.
```

Il est facile de déduire :

```
Prove: P_rectangle(x_9, x_6, x_3).
```

En remplaçant simplement les valeurs `x_9`, `x_6` et `x_3`. Donc le solveur pourrait utiliser ceci pour faire la preuve sans avoir à déplier le prédicat. Cependant, il ne le fera pas forcément :

les solveurs SMT sont basés sur des méthodes heuristiques, nous pouvons juste leur fournir des propriétés et espérer qu'ils les utiliseront.

Ici la propriété est simple à prouver, donc il n'était pas vraiment nécessaire d'ajouter ces assertions (et donc de faire plus d'efforts pour faire la même chose). Cependant, dans d'autre cas, comme nous allons le voir maintenant, nous devons donner la bonne information au bon endroit de façon à ce que les prouveurs trouvent les informations dont ils ont besoin pour finir les preuves.

7.2.2. Déclencher les lemmes

Nous utilisons souvent des assertions pour exprimer des propriétés qui correspondent aux prémisses d'un lemme ou à ses conclusions. En faisant cela, nous maximisons les chances que les prouveurs automatiques « reconnaissent » que ce que nous avons écrit correspond à un lemme en particulier et qu'il devrait l'utiliser.

Illustrons cela avec l'exemple suivant. Nous utilisons des axiomes et non des lemmes parce qu'ils sont considérés de la même manière par WP lorsque nous nous intéressons à la preuve d'une propriété qui en dépend. Regardons d'abord notre définition axiomatique. Nous définissons deux prédicats `P` et `Q` à propos d'une position mémoire particulière `x`. Nous avons deux axiomes : `ax_1` qui énonce que si `P(x)` est vraie, alors `Q(x)` est vraie, et un second axiome `ax_2` qui énonce que si la position mémoire pointée ne change pas entre deux labels (ce que l'on représente par le prédicat `eq`) et que `P(x)` est vraie pour le premier label, alors elle est vraie pour le second.

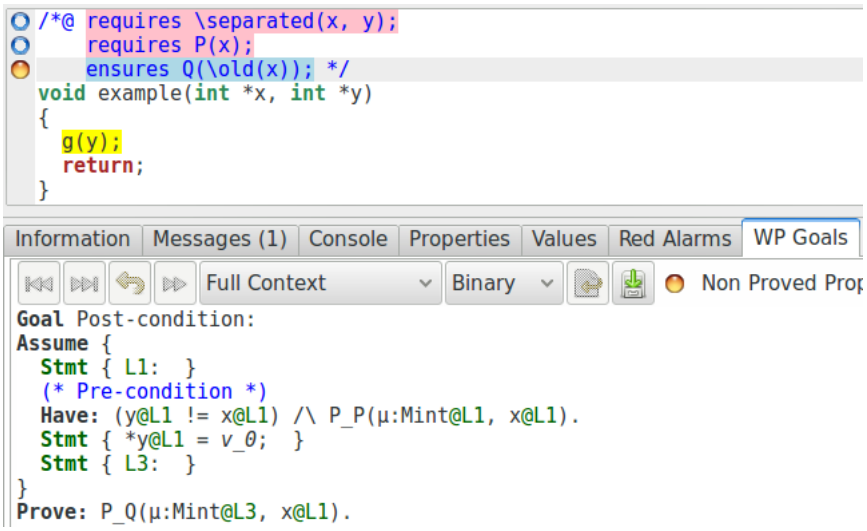
```
/*@
  predicate eq{L1, L2}(int* x) =
    \at(*x, L1) == \at(*x, L2) ;
*/

/*@
  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);
  }
*/
```

Et nous voulons prouver le programme suivant :

```
/*@ assigns *x ; */
void g(int* x);

/*@
  requires \separated(x, y);
  requires P(x) ;
  ensures  Q(x) ;
*/
void example(int* x, int* y){
  g(y);
```

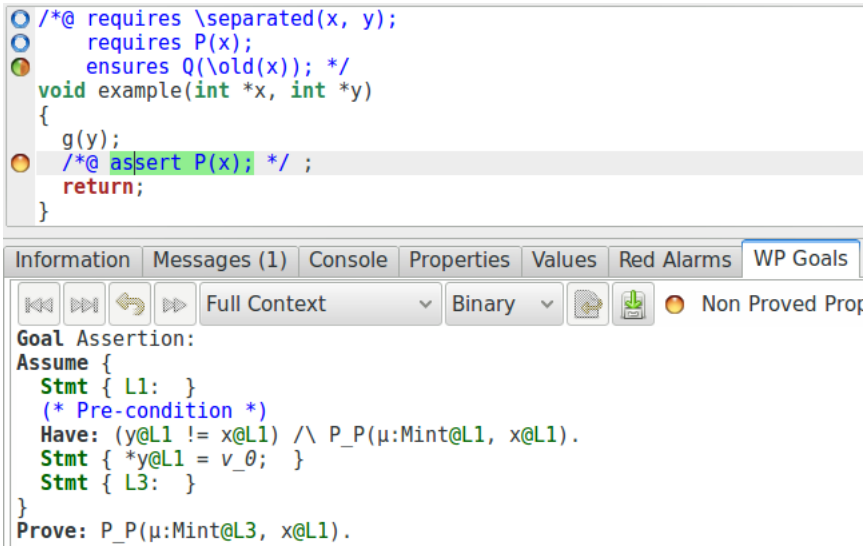
```
11 }
```

Cependant, nous pouvons voir que la preuve échoue sur la condition de vérification suivante (nous avons, à nouveau, retiré les éléments qui ne sont pas intéressants pour notre explication) :



D'après cela, notre prouveur automatique semble incapable d'utiliser l'un des axiomes de notre définition : soit il ne peut pas montrer qu'après l'appel `g(y)`, `P(x)` est toujours vraie, soit il le peut, et dans ce cas, cela veut dire qu'il n'arrive pas à montrer que cela implique `Q(x)`. Essayons donc d'ajouter une assertion pour vérifier qu'il arrive à montrer `P(x)` après l'appel :

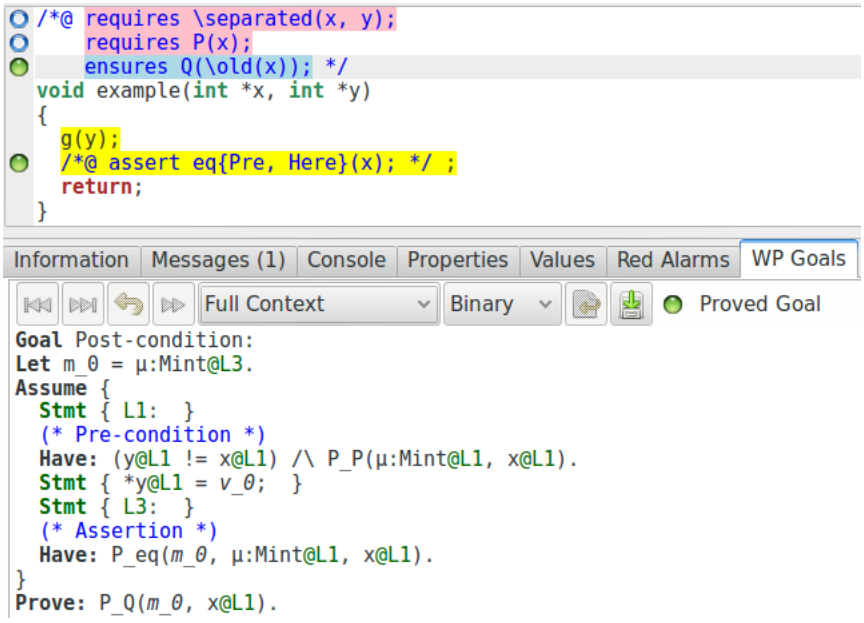
```
1 /*@
2   requires \separated(x, y);
3   requires P(x) ;
4   ensures  Q(x) ;
5 */
6 void example(int* x, int* y){
7   g(y);
8   //@ assert P(x);
9 }
```



Il semble que malgré le fait qu'il est clair que `*x` n'a pas changé pendant l'appel `g(y)`, et donc que `eq{Pre, Here}(x)` est vraie après l'appel, puisque la propriété n'est pas directement fournie dans notre condition de vérification, le prouveur automatique n'utilise pas l'axiome `ax_2` correspondant. Fournissons donc cette information au prouveur automatique :

```
1 /*@
2   requires \separated(x, y);
3   requires P(x) ;
4   ensures  Q(x) ;
5 */
6 void example(int* x, int* y){
7   g(y);
8   //@ assert eq{Pre, Here}(x);
9 }
```

Maintenant, tout est prouvé. Si nous regardons la condition de vérification générée, nous pouvons voir que l'information nécessaire est bien fournie, ce qui permet au prouveur automatique d'en faire usage :



7.2.3. Un exemple plus complexe : du tri à nouveau

Travaillons maintenant avec un exemple plus complexe utilisant une définition axiomatique réelle. Cette fois, nous nous intéresserons à montrer la correction d'un tri par insertion :

```
1  #include <stddef.h>
2  #include <limits.h>
3
4  void insert(int* a, size_t beg, size_t last){
5    size_t i = last ;
6    int value = a[i] ;
7
8    while(i > beg && a[i - 1] > value){
9      a[i] = a[i - 1] ;
10     --i ;
11   }
12   a[i] = value ;
13 }
14
15 void insertion_sort(int* a, size_t beg, size_t end){
16   for(size_t i = beg+1; i < end; ++i)
17     insert(a, beg, i);
18 }
```

La fonction `insertion_sort` visite chaque valeur, du début du tableau jusqu'à la fin. Pour chaque valeur v, elle est insérée (en utilisant la fonction `insert`) à la bonne place dans la plage des valeurs déjà triées (et qui se trouvent dans le début du tableau), en décalant les éléments jusqu'à rencontrer un élément qui est plus petit que v, ou le début du tableau.

Nous voulons prouver la même postcondition que ce que nous avons déjà prouvé pour le tri par sélection, c'est-à-dire : nous voulons créer une permutation triée des valeurs originales. À nouveau, chaque itération de la boucle doit assurer que la nouvelle configuration obtenue est une permutation des valeurs originales, et que la plage de valeurs allant du début à la cellule actuellement visitée est triée. Toutes ces propriétés sont garanties par la fonction `insert`. Si l'on regarde cette fonction de plus près, nous voyons qu'elle enregistre la valeur à insérer (qui se

trouve à la fin de la plage de valeurs) dans une variable `value`, et en commençant à la fin de la plage, décale itérativement les valeurs rencontrées jusqu'à rencontrer une valeur plus petite que la valeur à insérer ou la première cellule du tableau, et insère ensuite la valeur. Pour cette preuve, nous activons les options `-warn-unsigned-overflow` et `-warn-unsigned-downcast`.

Tout d'abord, fournissons un contrat et un invariant pour la fonction de tri par insertion. Le contrat est équivalent à celui que nous avions fourni pour le tri par sélection. Notons cependant que l'invariant est plus faible : nous n'avons pas besoin que les valeurs restant à trier soient plus grandes que les valeurs déjà visitées : nous insérons chaque valeur à la bonne position.

```
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 insertion_sort(int* a, size_t beg, size_t end){
8    /*@
9      loop invariant beg+1 <= i <= end ;
10     loop invariant sorted(a, beg, i) ;
11     loop invariant permutation{Pre, Here}(a,beg,end);
12     loop assigns a[beg .. end-1], i ;
13     loop variant end-i ;
14   */
15   for(size_t i = beg+1; i < end; ++i) {
16     insert(a, beg, i);
17   }
18 }
```

Maintenant, nous pouvons fournir un contrat à la fonction d'insertion. La fonction requière que la plage de valeurs considérée soit triée du début jusqu'à l'avant-dernière valeur. En échange elle doit garantir que la plage finale soit triée et soit une permutation des valeurs originales :

```
1  /*@
2    requires beg < last < SIZE_MAX && \valid(a + (beg .. last));
3    requires sorted(a, beg, last) ;
4
5    assigns a[ beg .. last ] ;
6
7    ensures permutation{Pre, Post}(a, beg, last+1);
8    ensures sorted(a, beg, last+1) ;
9  */
10 void insert(int* a, size_t beg, size_t last){
11   size_t i = last ;
12   int value = a[i] ;
13
14   while(i > beg && a[i - 1] > value){
15     a[i] = a[i - 1] ;
16     --i ;
17   }
18   a[i] = value ;
19 }
```

Ensuite, nous devons fournir un invariant utilisable pour expliquer le comportement de la boucle de la fonction `insert`. Cette fois, nous pouvons voir qu'avec notre précédente définition de la notion de permutation, nous sommes un peu embêtés. En effet, notre définition inductive de la permutation spécifie trois cas : une plage de valeur est une permutation d'elle-même, ou deux (et seulement deux) valeurs ont été changées, ou finalement la permutation d'une permutation est une permutation. Mais aucun de ces cas ne peut être appliqué à notre fonction d'insertion

puisque la plage obtenue ne l'est pas par une succession d'échanges de valeurs et les deux autres cas ne peuvent évidemment pas s'appliquer ici.

Nous avons donc besoin d'une meilleure définition pour notre permutation. Nous pouvons constater que ce dont nous avons vraiment besoin, c'est une manière de dire « chaque valeur qui était dans le tableau est toujours dans le tableau et si plusieurs valeurs étaient équivalentes, le nombre d'occurrences de ces valeurs ne change pas ». En fait, nous n'avons besoin que de la dernière partie de cette définition pour exprimer notre permutation. Une permutation est une plage de valeurs telles que pour toute valeur, le nombre d'occurrences de cette valeur dans cette plage ne change pas d'un point de programme à un autre :

```
/*@
  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) ;
*/
```

En partant de cette définition, nous sommes capables de fournir des lemmes qui nous permettront de raisonner efficacement à propos des permutations, à supposer que certaines propriétés sont vraies à propos du tableau entre deux points de programme. Par exemple, nous pourrions définir le cas `Swap` de notre définition inductive précédente en utilisant un lemme. C'est bien entendu aussi le cas pour notre plage de valeur « décalée ».

Déterminons quels sont les lemmes requis en considérant d'abord la fonction `insert_sort`. La seule propriété non prouvée est l'invariant qui exprime que le tableau est une permutation du tableau original. Comment pouvons-nous le déduire ? (Nous nous intéresserons aux preuves de ces lemmes plus tard).

Nous pouvons observer deux faits : la première plage du tableau (de `beg` à `i+1`) est une permutation de la même plage au début de l'itération (par le contrat de la fonction `insert`). La seconde partie (de `i+1` à `end`) est inchangée, donc c'est aussi une permutation. Essayons d'utiliser quelques assertions pour voir parmi ces propriétés ce qui peut être prouvé et ce qui ne peut pas l'être. Tandis que la première propriété est bien prouvée, nous pouvons voir que la seconde ne l'est pas :

```
  /*@
    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
  }
```

Nous avons donc besoin d'un premier lemme pour cette propriété. Définissons deux prédicats `shifted` et `unchanged`, le second étant utilisé pour définir le premier (nous verrons pourquoi un peu plus tard) et exprimer qu'une plage inchangée est une 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) ;
*/
```

Maintenant, nous pouvons vérifier que ces deux sous-tableaux sont des permutations, nous faisons cela en ajoutant une assertion qui montre que la plage allant de `i+1` à `end` est inchangée, afin de déclencher l'usage de notre lemme `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) ;
  }
```

Comme ces deux sous-parties du tableau sont des permutations, le tableau global est une permutation des valeurs initialement présentes au début de l'itération. Cependant, cela n'est pas prouvé directement, nous ajoutons donc aussi un lemme pour cela :

```
/*@ 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) ;
*/
```

Maintenant, nous pouvons déduire qu'une itération de la boucle produit une permutation en ajoutant cette conclusion comme une assertion :

```
    //@ ghost L: ;
    insert(a, beg, i);
    //@ assert permutation{L, Here}(a, beg, i+1);
    //@ assert unchanged{L, Here}(a, i+1, end);
```

```
5      //@ assert permutation{L, Here}(a, i+1, end);
6      //@ assert permutation{L, Here}(a, beg, end); // PROVED
```

Finalement, nous devons ajouter une information supplémentaire, la permutation d'une permutation est aussi une permutation. Cette fois, nous n'avons pas besoin d'une assertion supplémentaire. Le contexte contient :

— `permutation{Pre, L}(a, beg, end)` (invariant)
— `permutation{L, Here}(a, beg, end)` (assertion)

qui est suffisant pour conclure `permutation{Pre, Here}(a, beg, end)` à la fin du bloc de la boucle en utilisant le lemme suivant :

```
1  /*@ lemma transitive_permutation{L1, L2, L3}:
2    \forall int* a, integer beg, integer end ;
3      permutation{L1, L2}(a, beg, end) ==>
4      permutation{L2, L3}(a, beg, end) ==>
5        permutation{L1, L3}(a, beg, end) ;
6  */
```

Maintenant, nous pouvons regarder de plus près notre fonction d'insertion en nous intéressant d'abord à comment obtenir la connaissance que la fonction produit une permutation.

Elle décale les différents éléments vers la gauche jusqu'à rencontrer le début du tableau ou un élément plus petit que l'élément à insérer qui est initialement à la fin de la plage de valeur et inséré à la position ainsi atteinte. Les cellules du début du tableau jusqu'à la position d'insertion restent inchangées, c'est donc une permutation. Nous avons un lemme pour cela, mais nous devons ajouter cette connaissance que le début du tableau ne change pas comme un invariant de la boucle pour pouvoir déclencher le lemme après celle-ci. La seconde partie du tableau est une permutation parce que nous faisons « tourner » les éléments, nous avons besoin d'un lemme pour exprimer cela et d'indiquer dans l'invariant de boucle que les éléments sont décalés par la boucle. Finalement, l'union de deux permutations est une permutation et nous avons déjà un lemme pour cela.

Tout d'abord, donnons un invariant pour la permutation :

— nous fournissons les bornes de `i`,
— nous indiquons que la première partie du tableau est inchangée,
— nous indiquons que la seconde partie est décalée vers la gauche,

et nous ajoutons quelques assertions pour vérifier quelques propriétés d'intérêt :

— d'abord, pour déclencher `unchanged_permutation`, nous plaçons une première assertion pour énoncer que la première partie du tableau est inchangée, ce qui nous permet de prouver que ...
— la seconde assertion, qui nous dit que la première partie du tableau est une permutation de l'originale, et que l'on utilise en combinaison avec ...
— la troisième assertion qui nous dit que la seconde partie du tableau est une permutation de l'originale (qui nous permet de déclencher l'usage de `union_permutation` et de prouver la 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
```

Pour la dernière assertion, nous avons besoin d'un lemme à propos de la rotation des éléments :

```
/*@
  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) ;
*/
```

Nous devons également aider un peu les prouveurs automatiques pour montrer que l'ensemble des valeurs est trié après l'insertion. Pour cela, nous fournissons un nouvel invariant pour montrer que les valeurs « décalées » sont plus grandes que la valeur à insérer. Puis, nous ajoutons également des assertions pour montrer que le tableau est trié avant l'insertion, et que toutes les valeurs avant la cellule où nous insérons sont plus petites que la valeur insérée, et que la plage est en conséquence triée après l'insertion. Cela nous amène à la fonction `insert` complètement annotée suivante :

```
/*@
  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] ;

  /*@
```

```
15      loop invariant beg <= i <= last ;
16      loop invariant \forall integer k ; i <= k < last ==> a[k] > value ;
17      loop invariant \forall integer k ; beg <= k <= i    ==> a[k] == \at(a[k], Pre) ;
18      loop invariant \forall integer k ; i+1 <= k <= last ==> a[k] == \at(a[k-1], Pre) ;
19
20      loop assigns i, a[beg .. last] ;
21      loop variant i ;
22    */
23    while(i > beg && a[i - 1] > value){
24      a[i] = a[i - 1] ;
25      --i ;
26    }
27    //@ assert sorted(a, beg, last+1) ;
28    //@ assert \forall integer k ; beg <= k < i ==> a[k] <= value ;
29    a[i] = value ;
30    //@ assert sorted(a, beg, last+1) ;
31
32    //@ assert unchanged{Pre, Here}(a, beg, i) ;
33    //@ assert permutation{Pre, Here}(a, beg, i) ;
34
35    //@ assert rotate_left{Pre, Here}(a, i, last+1) ;
36    //@ assert permutation{Pre, Here}(a, i, last+1) ;
37  }
```

En tout, nous avons six lemmes à prouver :

- `l_occurrences_union`,
- `shifted_maintains_occ`,
- `unchanged_is_permutation`,
- `rotate_left_is_permutation`,
- `union_permutation`,
- `transitive_permutation`.

Tandis que les preuves Coq de ces lemmes sont en dehors des préoccupations de ce tutoriel (et nous verrons plus tard que dans le cas particulier de cette preuve nous pouvons nous en débarrasser), donnons quelques indications pour obtenir une preuve de ces lemmes (notons que les scripts Coq de ces preuves sont disponibles sur le répertoire GitHub de ce livre).

Pour prouver `l_occurrences_union`, nous raisonnons par induction sur la taille de la seconde partie du tableau. Le cas de base est trivial : si la taille est 0, nous avons immédiatement l'égalité puisque `split == to`. Maintenant, nous devons prouver que si l'égalité est vraie pour une plage de taille i, elle est vraie pour une plage de taille $i+1$. Puisque nous savons que c'est le cas jusqu'à i par hypothèse d'induction, nous analysons simplement les différents cas pour le dernier élément de la plage (au rang i) : soit cet élément est celui que nous comptons, soit il ne l'est pas. Quoi qu'il en soit, cela ajoute la même valeur des deux côtés de l'égalité.

Pour `shifted_maintains_occ`, nous raisonnons par induction sur la plage complète. Le premier cas est trivial (la plage est vide). Pour le cas d'induction, nous avons juste à montrer que la valeur ajoutée a été décalée, et qu'elle est donc la même.

La propriété `unchanged_is_permutation` peut être prouvée par les solveurs SMT grâce au fait que nous avons exprimé `unchanged` en utilisant `shifted`, le prouveur peut donc directement instancier le lemme précédent. Si ce n'est pas le cas, la preuve peut être réalisée en instanciant `shifted_maintains_occ` avec la valeur 0 pour la propriété de décalage.

Pour prouver `rotate_left_is_permutation`, nous séparons la plage pour `L1` en deux sous-plages `beg .. beg+1` et `beg+1 .. end` et la plage pour `L2` en deux sous-plages

`beg .. end-1` et `end-1 .. end` en utilisant la propriété `l_occurrences_union`. Nous montrons que le nombre d'occurrences dans `beg+1 .. end` pour `L1` et `beg .. end-1` pour `L2` n'a pas changé grâce à `shifted_maintains_occ` et que le nombre d'occurrences dans `beg .. beg+1` pour `L1` et `end-1 .. end` pour `L2` est le même en analysant par cas (et en utilisant le fait que la valeur correspondante est la même).

Pour prouver `union_permutation`, nous instancions le lemme `l_occurrences_union`. Finalement, le lemme `transitive_permutation` est prouvé automatiquement par transitivité de l'égalité.

7.2.4. Comment utiliser correctement les assertions ?

Il n'y a pas de guide précis à propos de quand utiliser des assertions ou non. La plupart du temps nous les utilisons d'abord pour comprendre pourquoi certaines preuves échouent en exprimant des propriétés dont nous pensons qu'elles sont vraies à un point particulier de programme. De plus, il n'est pas rare que les conditions de vérification soient longues ou un peu complexes à lire directement. Bien utiliser les assertions nécessite que l'on garde en tête les lemmes déjà exprimés pour savoir quelle assertion utiliser pour déclencher l'usage d'un lemme nous amenant à la propriété voulue. S'il n'y a pas de tel lemme, par exemple parce que la preuve de la propriété voulue nécessite un raisonnement par induction à propos d'une propriété ou d'une valeur, nous avons probablement besoin d'ajouter un nouveau lemme.

Avec un peu d'expérience, l'utilisation des assertions et des lemmes devient de plus en plus naturelle. Cependant, il est important de garder en tête qu'il est facile d'abuser de cela. Plus nous ajoutons de lemmes et d'assertions, plus notre contexte de preuve est riche et est susceptible de contenir les informations nécessaires à la preuve. Cependant, il y a aussi un risque d'ajouter trop d'information de telle manière à ce que le contexte de preuve finisse par contenir des informations inutiles qui polluent le contexte de preuve, rendant le travail des solveurs SMT plus difficile. Nous devons donc essayer de trouver le bon compromis.

7.2.5. Exercices

7.2.5.1. Comprendre le contexte de preuve

Dans la fonction suivante, la dernière assertion est prouvée automatiquement par le solveur SMT, probablement en dépliant le prédicat pour prouver directement la propriété. En utilisant des assertions, fournir une nouvelle manière de prouver la dernière propriété. Dans le contexte de preuve, trouver les propriétés générées qui peuvent amener à une autre preuve de l'assertion et expliquer comment :

```
1 /*@
2   predicate rectangle{L}(integer c1, integer c2, integer h) =
3     c1 * c1 + c2 * c2 == h * h ;
4 */
5
6 /*@
7   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. Déclencher les lemmes

Dans le programme suivant, WP échoue à prouver que la postcondition de la fonction `g` est vérifiée. Ajouter la bonne assertion, au bon endroit, de façon à ce que la preuve réussisse.

```
/*@
  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. Déclencher les lemmes sous condition

Dans le programme suivant, WP échoue à prouver que la postcondition de la fonction `example` est vérifiée. Cependant, nous pouvons noter que la fonction `g` assure indirectement que la valeur pointée est soit augmentée soit diminuée. Ajouter deux assertions qui montrent que le prédicat est vérifié en fonction de la valeur de `*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);
}
```

Les assertions devraient ressembler à :

```
//@ assert *x ... ==> ... ;
//@ assert *x ... ==> ... ;
```

Une autre manière d'ajouter de l'information au contexte est d'utiliser du code fantôme. Par exemple, la valeur de vérité d'une conditionnelle apparaît dans le contexte d'une condition de vérification. Modifier les annotations pour que le code ressemble à :

```
void example(int* x){
  g(x);
  /*@ ghost
   if ( ... ){
    /@ assert ... @/
   } else {
    /@ assert ... @/
   }
  */
}
```

Comparer la condition de vérification générée pour chaque assertion avec les précédentes.

Finalement, nous pouvons remarquer que « la valeur pointée a été augmentée ou diminuée » peut être exprimée en une seule assertion. Écrire l'annotation correspondante et recommencer

la preuve.

7.2.5.4. Un exemple avec la somme

La fonction suivante incrémenter de 1 la valeur d'une cellule du tableau, donc elle augmente aussi la valeur de la somme du contenu du tableau. Écrire un contrat pour la fonction qui exprime ce fait :

```
#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]++ ;
}
```

Pour prouver que cette fonction remplit son contrat, nous avons besoin de fournir des assertions qui guideront la preuve. Plus précisément, nous devons montrer que puisque toutes les valeurs avant la cellule modifiée n'ont pas été modifiées, la somme n'a pas été modifiée pour cette partie du tableau, et de même pour les cellules qui suivent la cellule modifiée.

Nous avons donc besoin de deux lemmes :

— `sum_separable` doit exprimer que nous pouvons séparer le tableau en deux sous parties, compter dans chaque partie et sommer les résultats pour obtenir la somme totale,

— `unchanged_sum` devrait exprimer que si une plage dans un tableau n'a pas changé entre deux labels, la somme du contenu est la même.

Compléter les code des lemmes et utiliser des assertions pour assurer qu'ils sont utilisés pour compléter la preuve. Nous ne demandons par la preuve des lemmes, les preuves Coq sont disponibles sur le répertoire GitHub de ce livre.

7.3. Plus de code fantôme, fonctions lemmes et macros lemmes

Les assertions nous permettent de donner des indices au générateur de conditions de vérification pour les solveurs SMT obtiennent assez d'information pour produire la preuve dont nous avons besoin. Cependant, il est parfois difficile d'écrire une assertion qui créera exactement la propriété dont le solveur SMT a besoin pour déclencher le bon lemme (par exemple, puisque le générateur effectue des optimisations sur les conditions de vérification, ils peuvent légèrement la modifier, ainsi que le contexte de preuve). De plus, nous reposons sur des lemmes qui ont souvent besoin d'être prouvés en Coq, et pour cela nous avons besoin d'apprendre Coq.

Dans cette section, nous verrons quelques techniques qui peuvent être utilisées pour rendre tout cela plus prédictible et nous éviter d'utiliser l'assistant de preuve Coq. Tandis que ces techniques ne peuvent pas toujours être utilisées (et nous expliquerons quand cela n'est pas applicable), elles sont généralement efficaces pour obtenir de la preuve presque complètement automatique. Cela repose sur l'usage de code fantôme.

7.3.1. Preuve par induction

Précédemment, nous avons mentionné que les solveurs SMT sont mauvais pour effectuer des preuves par induction (la plupart du temps), et c'est la raison pour laquelle nous avons souvent besoin d'exprimer des lemmes que nous prouvons avec l'assistant de preuve Coq qui nous permet de faire notre preuve par induction. Cependant, dans la section 4.2 à propos des boucles, nous trouvons une sous-section 4.2.1 nommée « Induction et invariant », où nous expliquons que pour prouver un invariant de boucle, nous procédons ... par induction. L'auteur de ce tutoriel aurait-il honteusement menti au lecteur pendant tout ce temps ?

En fait non. La raison est plutôt simple. Lorsque nous prouvons un invariant de boucle par induction en utilisant des solveurs SMT, ils n'ont pas besoin d'effectuer le raisonnement par induction eux-mêmes. Le travail qui consiste à séparer la preuve en deux sous-preuves, la première pour l'établissement de l'invariant (le cas de base de la preuve), et la seconde pour la préservation (le cas d'induction) est effectué par le générateur de conditions de vérification. Par conséquent, quand les conditions de vérifications sont transmises aux solveurs SMT, ce travail n'est plus nécessaire.

Comment pouvons-nous exploiter cette idée ? Nous avons expliqué précédemment que le code fantôme peut être utilisé pour fournir plus d'information que ce qui est explicitement fourni par le code source. Pour cela, nous ajoutons du code fantôme (et possiblement des annotations à propos de ce code) qui nous permet de déduire plus de propriétés. Illustrons cela avec un exemple simple. Dans un exercice précédent (5.4.5.2), nous voulions prouver l'appel de fonction suivant (nous avons exclus la postcondition pour raccourcir l'exemple) :

```
1 #include <stddef.h>
2 #include <limits.h>
3
4 /*@ predicate sorted(int* arr, integer end) =
5       \forall integer i, j ; 0 <= i <= j < end ==> arr[i] <= arr[j] ;
6     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);
}
```

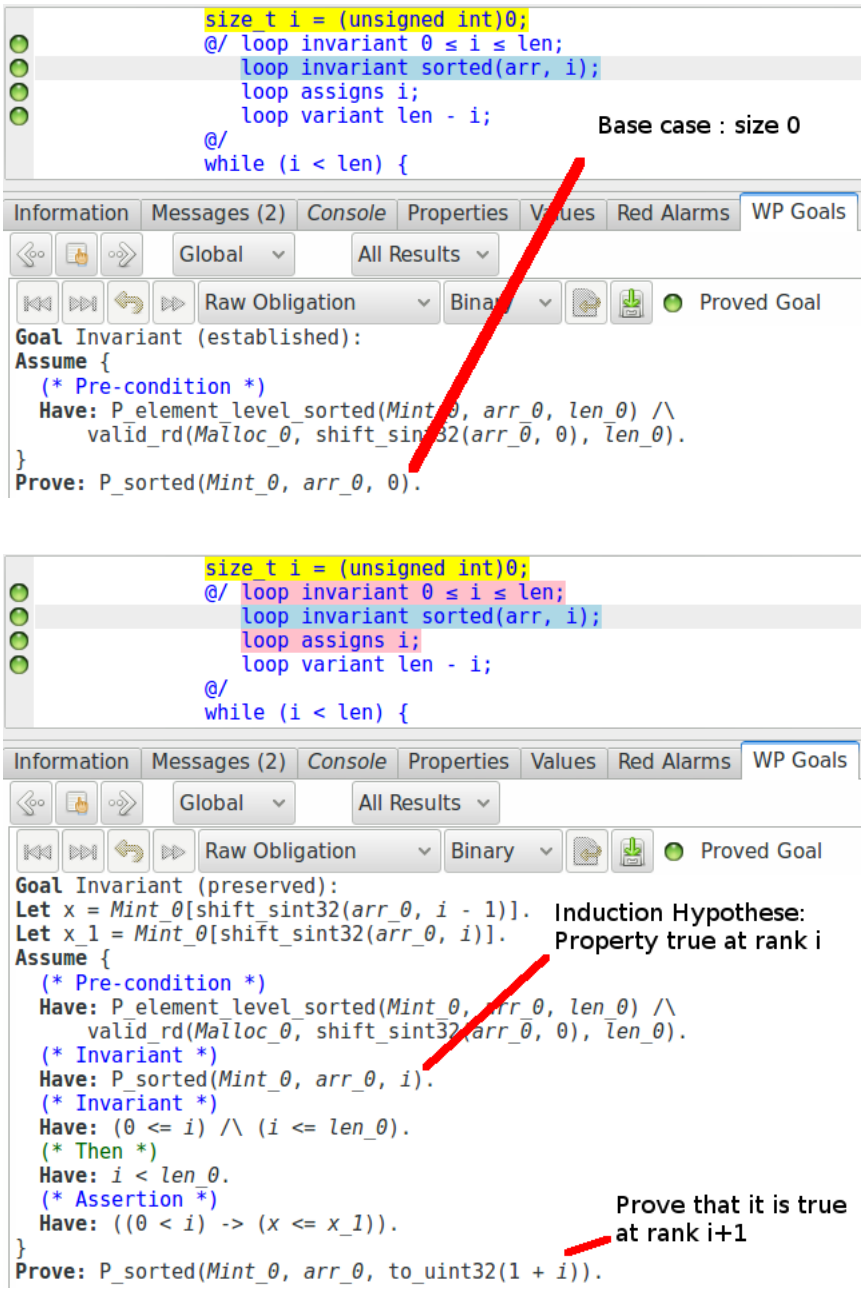
Pour cela, la solution que nous avions demandée dans l'exercice était de fournir un lemme qui énonce qui si la plage est « triée localement », au sens où chaque élément est supérieur ou égal à l'élément qui le précède, alors nous pouvons dire qu'elle est « globalement triée », c'est-à-dire que pour chaque paire d'indices i et j, si $i \leq j$ alors le j^{ime} élément du tableau est supérieur ou égal au i^{ime} élément. Donc, la précondition de la fonction pouvait être prouvée par les solveurs SMT, mais pas le lemme lui-même qui nécessite une preuve Coq. Est-ce que l'on ne pourrait pas faire quelque chose de mieux à ce sujet ?

La réponse est oui. Avant d'appeler la fonction, nous pouvons construire une preuve qui montre que puisque le tableau est trié localement, nous pouvons déduire qu'il est trié globalement (ce qui est simplement la preuve du lemme dont nous aurions besoin). Pour écrire cette preuve à la main, nous procéderions par induction sur la taille de la plage. Nous avons deux cas. D'abord si la plage est vide, la propriété est trivialement vraie. Ensuite, supposons qu'une plage donnée de taille i avec $i < length$ ($length$ étant la taille de la plage complète) est globalement triée et montrons que si c'est le cas, alors la plage de taille $i+1$ est triée. C'est facile parce que, d'après notre précondition, nous savons que le i^{ime} élément est supérieur ou égal au $(i-1)^{ime}$ élément, qui est lui-même plus grand que tous les éléments qui le précèdent.

Comment pouvons-nous traduire cela en code fantôme ? Nous écrivons une boucle qui va de 0 (notre cas de base), à la fin `len` et nous fournissons un invariant montrant que le tableau est globalement trié depuis 0 jusqu'à la cellule actuellement visitée. Nous ajoutons également une assertion pour aider le prouveur (qui nous dit que l'élément courant est plus grand que les éléments qui précèdent) :

```
/*@ 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);
}
```

Nous pouvons voir que toutes les conditions de vérification sont facilement vérifiées par les solveurs SMT, sans que cela ne nécessite d'écrire une preuve Coq ou un lemme. Les conditions de vérification respectivement créées pour l'établissement et la préservation de l'invariant correspondent aux deux cas que nous avions besoin dans notre preuve par induction :



Ce type de code est appelé un « *proof carrying code* » : nous avons écrit un code et les annotations qui amènent la preuve d'une propriété que nous voulons vérifier.

Notons qu'ici, puisque nous devons écrire beaucoup de code fantôme, cela augmente notre risque d'introduction d'une erreur qui changerait les propriétés du code vérifié. Nous devons donc impérativement nous assurer que le code fantôme que nous avons écrit termine et qu'il ne contient pas d'erreurs à l'exécution (grâce au plugin RTE) pour avoir confiance dans notre vérification.

Dans cet exemple, nous avons directement écrit le code fantôme comme annotation du programme, cela signifie que si nous avons un autre appel comme celui-ci quelque part dans le code avec une précondition similaire, nous aurions à le faire à nouveau. Rendons cela plus simple et modulaire avec des « fonctions lemmes » (*lemma functions*).

7.3.2. Fonction lemme

Le principe des « fonctions lemmes » est le même que celui des lemmes : à partir de certaines prémisses, nous voulons arriver à une conclusion particulière. Et une fois que c'est fait, nous voulons les utiliser à d'autres endroits pour directement déduire la conclusion depuis les prémisses sans avoir à faire la preuve à nouveau, en instanciant les valeurs nécessaires.

La manière de faire cela est d'utiliser une fonction, en utilisant les clauses `requires` pour exprimer les prémisses du lemme et les clauses `ensures` pour exprimer la conclusion du lemme. Les variables quantifiées universellement peuvent rester quantifiées ou correspondre à un paramètre de la fonction. Plus précisément, si une variable est uniquement liée aux prémisses ou uniquement aux conclusions, elle peut rester une variable quantifiée, à supposer qu'il ne soit pas nécessaire de la lier à une variable du code de preuve (puisque qu'une variable quantifiée n'est pas visible depuis le code C). Si elle est liée aux prémisses et conclusions, elle doit être un paramètre de la fonction (puisqu'ACSL ne nous permet pas de quantifier une variable pour un contrat de fonction entier).

Considérons l'exemple suivant où nous n'utilisons pas (directement) de variable universellement quantifiée dans le contrat, avec notre précédent exemple à propos des valeurs triées. Depuis la propriété `element_level_sorted(arr, len)`, nous voulons déduire `sorted(arr, len)`. Le lemme correspondant pourrait être :

```
/*@
  lemma element_level_sorted_is_sorted:
    \forall int* arr, integer len ;
       element_level_sorted(arr, len) ==> sorted(arr, len) ;
*/
```

Écrivons donc une fonction qui prend deux paramètres, `arr` et `len`, et requiert que le tableau soit trié localement et assure qu'il est trié globalement :

```
/*@ ghost
  /@
    requires element_level_sorted(arr, len) ;
    assigns  \nothing ;
    ensures  sorted(arr, len);
  @/
  void element_level_sorted_implies_sorted(int* arr, size_t len);
*/
```

Notons que cette fonction doit affecter `\nothing`. En effet, nous l'utilisons pour déduire des propriétés à propos du programme, dans du code fantôme, et donc elle ne devrait pas modifier le contenu du tableau, sinon le code fantôme modifierait le comportement du programme. Maintenant, produisons un corps pour cette fonction, le code qui nous amène la preuve que la conclusion est bien vérifiée en supposant que la précondition est vérifiée. Cela correspond au

code que nous avons écrit précédemment pour prouver la précondition de l'appel de la fonction `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] ; @/
    }
  }
*/
```

Avec cette boucle spécifiée, nous obtenons une preuve par induction que le lemme est vrai. Maintenant, nous pouvons utiliser cette fonction lemme en l'appelant tout simplement à l'endroit où nous avons besoin de faire cette déduction :

```
/*@ 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);
}
```

Ce qui nous demande d'établir la preuve que les prémisses sont établies grâce à la précondition de la fonction lemme (et qui est trivialement vraie, puisque nous l'obtenons de la précondition de `bsearch_callee`), et qui nous donne en retour la conclusion gratuitement puisque c'est la postcondition de la fonction lemme (et nous pouvons utiliser cette propriété comme précondition de l'appel à la fonction `bsearch`).

Comme nous l'avons expliqué, quand des variables universellement quantifiées sont liés à la fois aux prémisses et aux conclusions, elles doivent être des paramètres, c'est par exemple le cas ici pour les variables `arr` et `len`, tandis que pour les variables quantifiées dans les prédicats :

```
/*@ 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] ;
*/
```

puisqu'elles ne sont respectivement liées qu'aux prémisses et aux conclusions restent universellement quantifiées (même si elles sont cachées dans les prédicats). Nous pourrions avoir écrit le contrat comme ceci :

```
/*@ 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] ; @/
    }
  }
*/
```

où nous voyons parfaitement que les variables sont toujours universellement quantifiées. Cependant, nous ne sommes pas obligés de les maintenir quantifiées universellement, et nous pourrions parfaitement les transformer en paramètres (en supposant que la conclusion que nous obtenons des prémisses a toujours du sens). Faisons par exemple cela pour les variables `i` et `j` de la conclusion :

```
/*@ 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);
  }
*/
```

Ce qui est tout à fait bon et nous pourrions par exemple utiliser cette fonction pour déduire des propriétés à propos du contenu du tableau. Notons qu'ici, nous utilisons un appel à la précédente fonction lemme pour rendre la preuve plus facile. Nous pouvons même aller plus loin en transférant la « prémisse de notre conclusion » en tant que prémisse d'un nouveau lemme :

```
/*@ 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);
  }
*/
```

Tous ces lemmes énoncent la même relation globale, la différence est liée à la quantité d'informations requises pour les instancier (et par conséquent la précision de la propriété que nous obtenons en retour).

Finalement, présentons un dernier usage des fonctions lemmes. Dans tous les exemples précédents, nous avons considéré des variables universellement quantifiées. En fait, ce que nous avons dit précédemment est aussi applicable aux variables existentiellement quantifiées : si elles sont liées aux prémisses et aux conclusions, elles doivent être des paramètres, sinon elles peuvent donner lieu à des paramètres ou rester quantifiées. Cependant, à propos des variables existentiellement quantifiées, nous pouvons parfois aller plus loin en construisant une fonction qui nous fournit directement une valeur qui satisfait la propriété à propos de la variable existentiellement quantifiée.

Par exemple, considérons la définition axiomatique du comptage d'occurrences et imaginons qu'à un certain point de notre programme, nous voulons prouver l'assertion suivante à partir de la précondition :

```
/*@
  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
}
```

Bien sûr, il existe un indice `n` tel que `in[n]` est `v`, sinon le nombre d'occurrences de cette valeur serait 0. Mais au lieu de prouver que cet index existe, montrons que nous pouvons trouver un index qui respecte les contraintes à propos de `n` en utilisant une fonction lemme qui le retourne :

```
/*@ 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 ;
  }
*/
```

Si nous regardons seulement le corps de la fonction, il a deux comportements : soit il existe une cellule du tableau qui contient `v` et la fonction retourne son indice, ou ce n'est pas le cas, et dans ce cas la fonction retourne `SIZE_MAX`. Le premier comportement est facile à montrer, le retour correspondant à cette découverte n'est effectuée que dans une branche où l'on a trouvé une cellule qui correspond à la valeur recherchée.

Nous prouvons que le second comportement satisfait la postcondition en montrant qu'il mène à une contradiction. S'il n'y a pas de cellule dont la valeur est `v`, alors le nombre d'occurrences de `v` est 0. C'est exprimé grâce au second invariant qui nous dit qu'aucun `v` n'a été rencontré depuis le début de la boucle, et donc le nombre d'occurrences est 0. Cependant, la précondition de la fonction nous énonce que le nombre d'occurrences est supérieur à 0, ce qui mène à une contradiction que nous modélisons par une assertion de faux (notons qu'elle n'est pas nécessaire, nous l'écrivons explicitement pour notre explication) ce qui signifie que ce chemin est infaisable.

Finalement, nous pouvons appeler cette fonction pour montrer qu'il existe un indice qui nous permet de valider notre assertion :

```
/*@
  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
}
```

L'utilisation des fonctions lemmes nous permet de raisonner par induction à propos de lemmes sans avoir besoin de preuve interactive. De plus, le déclenchement des lemmes devient beaucoup plus prévisible puisque nous le faisons à la main. Cependant, les lemmes nous permettent de travailler sur plusieurs labels :

```
/*@
  lemma my_lemma{L1, L2}:  P{L1} ==> P{L2} ;
*/
```

Les fonctions lemmes ne nous fournissent pas de mécanisme équivalent, car elles sont simplement des fonctions C normales, qui ne peuvent pas prendre de labels comme entrée. Regardons ce que nous pouvons faire à ce sujet.

7.3.3. Macro lemme

Lorsque nous devons traiter de multiples labels, l'idée est « d'injecter » directement le code de preuve à l'endroit où c'est nécessaire comme nous l'avons fait au début de ce chapitre. En revanche, nous ne voulons pas écrire ce code à la main chaque fois que nous en avons besoin, utilisons donc des macros pour le faire.

Pour le moment, traduisons notre code précédent en une macro au lieu d'une fonction. Comme nous utilisons la macro dans du code fantôme (donc en annotation), nous devons faire attention à utiliser la syntaxe pour les annotations fantômes lorsque nous écrivons l'invariant de notre boucle et les 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);
}
```

Au lieu de fournir un pré et une postcondition, nous énonçons ces propriétés en utilisant des assertions avant et après le code de preuve. Ce code de preuve est simplement le même qu'avant, et est utilisé exactement comme il était utilisé dans le cas de la fonction. Cependant, nous pouvons voir que cela fait une différence importante une fois qu'il a été préprocessé par Frama-C, puisque le bloc de code et les annotations sont directement injectées dans la fonction `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;
}
```

En fait, nous utilisons une macro pour générer le code que nous écrivions précédemment. Dans le cas présent, ce n'est pas vraiment intéressant puisque l'appel de fonction nous permettait d'avoir une preuve plus modulaire. Étudions donc un exemple où nous n'avons pas d'autre choix qu'utiliser une macro.

Nous utiliserons le lemme suivant :

```
/*@
  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) ;
*/
```

Avec pour objectif de prouver le programme suivant :

```
/*@
  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) ;
}
```

où le lemme `shift_ptr` est nécessaire pour prouver que la postcondition de `callee` depuis la postcondition de `shift_array`. Notre but est bien sûr de ne pas avoir besoin du lemme en le remplaçant par une macro lemme.

Il n'y a pas de guide précis pour concevoir une macro utilisée pour injecter un code de preuve. Cependant, la plupart des lemmes énoncés à propos de labels multiples sont relativement similaires dans leur manière de lier les labels. Illustrons donc avec cet exemple ; la plupart du temps, concevoir une macro dans une telle situation suivra plus ou moins ce schéma.

Pour construire la macro, nous avons besoin d'un contexte dans lequel travailler. Nous construisons ce contexte en utilisant une fonction, nommons-la `context_to_prove_shift_ptr`. L'idée est d'utiliser cette fonction pour construire notre macro en isolation du reste du programme pour rendre la vérification de la propriété plus facile. Cependant, tandis que les fonctions lemmes sont ensuite appelées dans d'autres fonctions pour déduire des propriétés, cette fonction ne sera jamais appelée, son rôle est juste de nous permettre d'avoir un « endroit » où nous pouvons construire notre preuve. En particulier, comme nous avons besoin de plusieurs labels mémoire, notre fonction **a besoin** de modifier le contenu de la mémoire (sinon, nous aurions qu'un seul état mémoire pour toute la fonction).

Illustrons cela avec notre problème actuel pour rendre tout cela plus clair. Premièrement, nous créons la macro `shift_array` qui contiendra notre code de preuve, pour le moment indiquons juste que c'est une instruction vide. Dans les paramètres de ce lemme, nous prenons les labels considérés. Notons que les règles précédemment exprimées à propos des variables quantifiées s'appliquent aussi pour les macros.

```
1  #define shift_ptr(_L1, _L2, _arr, _fst, _last, _s1, _s2) ;
```

Ensuite nous créons notre fonction de contexte :

```
1  /*@
2    assigns arr[fst+s1+s2 .. last+s1+s2] ;
3    ensures shifted{Pre, Post}(arr, fst+s1, last+s1, s2) ;
4  */
5  void assign_array(int* arr, size_t fst, size_t last, size_t s1, size_t s2);
6
7  /*@
8    requires fst <= last ;
9    requires s1+s2+last <= SIZE_MAX ;
10 */
11 void context_to_prove_shift_ptr(int* arr, size_t fst, size_t last, size_t s1, size_t s2){
12  L1: ;
13   assign_array(arr, fst, last, s1, s2);
14  L2: ;
15   //@ assert shifted{L1, L2}(arr, fst+s1, last+s1, s2) ;
16
17   //@ ghost shift_ptr(L1, L2, arr, fst, last, s1, s2) ;
18
19   //@ assert shifted{L1, L2}(arr+s1, fst, last, s2) ;
20 }
```

Décomposons ce code, en commençant par la fonction de contexte. En entrée, nous recevons les variables du lemme. Nous énonçons également quelques propriétés à propos des bornes des entiers considérés, généralement cela devrait simplement être les préconditions qui ne sont pas liées aux états mémoire ou liées au premier état mémoire. Ensuite, nous introduisons le label `L1` et nous appelons la fonction `assign_array` qui nous amène au label `L2`. Le rôle de cet appel est de s'assurer que WP créera un nouvel état mémoire (et qu'il ne considérera donc pas que la mémoire est la même), et d'établir les prémisses. En effet, si nous regardons le contrat de `assign_array`, nous voyons qu'elle assigne le tableau (ce qui garantit la création d'un nouvel état mémoire) et en postcondition, elle assure que le contenu du tableau, entre la pré et la postcondition (donc, quand nous l'appelons, `L1` et `L2`) satisfont les prémisses de notre lemme (que l'on répète en ligne 15, en ajoutant une assertion). Ensuite nous utilisons notre macro `shift_ptr` (qui contiendra par la suite notre code de preuve), et nous voulons être capables de prouver la postcondition de notre lemme (ligne 19).

En faisant cela, nous assurons que nous avons construit un contexte qui ne contient que les informations nécessaires pour construire le code de preuve permettant de déduire la conclusion (ligne 19) depuis les prémisses (ligne 15). Maintenant écrivons la 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) ; @/
```

Nous ne détaillerons pas ce code, car il est très similaire à ce que nous avons écrit au début de cette section. La seule petite subtilité est l'assertion qui permet d'aider les solveurs SMT à relier les positions mémoire entre `L1` et `L2` aux lignes 8–9. Avec cette macro, nous pouvons voir que l'assertion à la fin de la fonction `context_to_prove_shift_ptr` est correctement validée. Par conséquent, nous pouvons espérer qu'elle sera capable d'aider les prouveurs à obtenir une conclusion similaire dans un contexte similaire (c'est-à-dire un contexte où nous savons que `shifted` est validée pour un certain tableau entre labels).

Finalement, nous pouvons compléter la preuve de notre fonction `callee` en utilisant notre macro lemme :

```
1  /*@
2    requires \valid(array+(0 .. len+s1+s2-1)) ;
3    requires s1+s2 + len <= SIZE_MAX ;
4    assigns array[s1 .. s1+s2+len-1];
5    ensures shifted{Pre, Post}(array+s1, 0, len, s2) ;
6  */
7  void callee(int* array, size_t len, size_t s1, size_t s2){
8    shift_array(array, s1, s1+len, s2) ;
9    //@ ghost shift_ptr(Pre, Here, array, 0, len, s1, s2) ;
10 }
```

Nous pouvons constater que même si cette technique permet d'injecter le code de preuve avec une seule ligne de code, elle peut injecter beaucoup de code à la position où nous l'utilisons. De plus, lorsque nous injectons ce code à un endroit où nous savons qu'il sera utile, le contexte correspondant peut déjà être plutôt complexe. Il n'est donc pas rare d'avoir à modifier légèrement la macro pour ajouter un peu d'information qui n'est pas nécessaires dans un contexte bien propre comme celui que nous utilisons pour produire la macro.

Notons qu'en cela, à la différence de la fonction lemme qui a un comportement très proche d'un lemme « classique », au sens où elle nous permet de faire une déduction immédiate d'une conclusion à partir de certaines prémisses à un point particulier de programme sans avoir à en refaire la preuve ; la macro lemme, elle, s'éloigne beaucoup du comportement d'un lemme « classique », car elle implique de refaire la preuve à chaque point où l'on a besoin de cette déduction.

Tout ceci peut rendre le contexte beaucoup plus gros, et plus difficile à utiliser pour les solveurs SMT. Il y a d'autres limitations à cette technique et le lecteur très attentif aura déjà pu les constater. Parlons-en.

7.3.4. Limitations

La principale limitation des fonctions lemmes et macros lemmes est le fait que nous sommes limités à des types C. Par exemple, si nous comparons la fonction lemme qui correspond à notre lemme `element_level_sorted_is_sorted`, le type original de la valeur `len` est un type entier mathématique, tandis que dans la fonction lemme, ce type est `size_t`. Cela signifie que là où notre lemme était vrai pour tout entier, et donc qu'il pouvait être utilisé peu importe que la variable représentant la taille soit un `int`, ou un `unsigned` (ou n'importe quel autre type entier), à l'opposé, notre fonction ne peut être utilisée que pour les types qui peuvent être convertis de manière sure vers `size_t`. Cependant, cette limitation n'est généralement pas

un problème : nous avons juste à exprimer notre spécification dans le type le plus gros que nous avons à considérer dans notre programme et la plupart du temps, cela sera suffisant. Et si ce n'est pas le cas, nous pouvons par exemple dupliquer le lemme pour les types qui nous intéressent. La plupart du temps cette limitation est largement gérable puisque nous travaillons avec les types utilisés dans le programme à prouver.

Dans certains cas, en revanche, cela contraint notre manière de modéliser des propriétés, ce qui est principalement lié aux types logiques que nous pouvons utiliser pour modéliser certaines structures de données concrètes. Par exemple, pour modéliser une liste chaînée, nous pouvons utiliser le type logique ACSL `\list<Type>` et exprimer une propriété inductive ou axiomatique pour définir comment une liste chaînée concrète peut être modélisée par une liste logique. Nous pourrions donc avoir des lemmes à propos des listes logiques, comme :

```
/*@
  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))) ;
*/
```

Nous ne pouvons pas écrire une fonction lemme avec du code de preuve pour cette propriété puisque nous n'avons pas de moyen d'utiliser ce type logique dans du code C, et donc, aucun moyen d'écrire une boucle et un invariant qui nous permettraient de prouver cette propriété.

L'autre limitation est liée aux macros lemmes et ce que nous avons déjà mentionné dans le chapitre précédent à propos des assertions. En ajoutant trop d'assertions, le contexte de preuve peut devenir trop gros et trop complexe, et donc difficile à manipuler pour les solveurs SMT. Utiliser des macros lemmes peut générer beaucoup de code et d'annotations et amener à de plus gros contextes preuve. Elles devraient donc être utilisées avec beaucoup de parcimonie.

Finalement, selon la propriété à prouver, il peut être difficile de trouver un code de preuve. En effet, les assistants de preuve comme Coq sont conçus pour être capables d'exprimer des preuves même pour des propriétés très complexes, en se reposant sur une vue très haut niveau de nos problèmes, tandis que C a été conçu pour écrire des programmes, et avec une vue très détaillée de la manière de résoudre le problème. Il peut donc parfois être difficile d'écrire un programme C permettant de prendre en compte certaines propriétés et plus encore de trouver un invariant utilisable lié à nos boucles.

7.3.5. Encore un peu de tri par insertion

Maintenant, revenons à notre preuve du tri par insertion et voyons comment nous pouvons nous débarrasser de nos preuves interactives pour cette fonction. Notons cependant que dans cette preuve, nous avons souvent besoin de macros puisqu'il n'a pas été particulièrement écrit avec comme objectif de le vérifier plus tard (pour cela, le lecteur peut se référer au livre ACSL by Example qui peut être adapté avec une technique similaire et est beaucoup plus facile à prouver). Donc dans cet exemple, nous poussons les solveurs SMT vers leurs limites à cause des gros contextes de preuve. En fonction de la puissance de la machine sur laquelle la preuve est lancée, la preuve pourrait approcher les 60 secondes (ce qui est déjà long pour un solveur SMT). Nous utilisons à nouveau les options `-warn-unsigned-overflow` et

`-warn-unsigned-downcast`. Dans cet exemple, nous illustrerons les trois cas d'usage que nous avons vu du code fantôme jusqu'ici :

— écrire directement un code pour construire une preuve,
— écrire (et utiliser) des fonctions lemmes,
— écrire (et utiliser) des macros lemmes.

Nous utilisons également des assertions pour rendre le contexte de preuve plus riche afin que les solveurs SMT puissent prouver les propriétés qui nous intéressent. Certaines parties des annotations que nous avons écrites précédemment seront équivalentes à ce que nous avons fait précédemment. Nous rappellerons leur but dans chaque fonction. Ensuite, nous utilisons la même définition axiomatique pour le comptage d'occurrences. De plus, nous conservons ces définitions de prédicats :

```
/*@
  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) ;
*/
```

puisque nous en avons besoin, ainsi que le lemme à propos de la transitivité du comptage d'occurrences, puisqu'il était prouvé automatiquement par les solveurs SMT (nous pouvons donc le garder puisqu'il ne nécessite pas de preuve interactive de notre part).

```
/*@ 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) ;
*/
```

Commençons par la fonction `insertion_sort`. Dans cette fonction, nous avons écrit trois 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) ;
```

```
11      loop invariant permutation{Pre, Here}(a,beg,end);
12      loop assigns a[beg .. end-1], i ;
13      loop variant end-i ;
14    */
15    for(size_t i = beg+1; i < end; ++i) {
16      //@ ghost L:;
17      insert(a, beg, i);
18      //@ assert permutation{L, Here}(a, beg, i+1);
19      //@ assert unchanged{L, Here}(a, i+1, end) ;
20      //@ assert permutation{L, Here}(a, beg, end) ;
21    }
22  }
```

La première nous assure que le début du tableau où nous avons inséré une valeur est une permutation de la même plage de valeur avant l'appel à `insert`. Comme c'est la postcondition de la fonction, elle n'est pas nécessaire, mais nous la conservons pour illustration. La dernière assertion est la propriété que nous voulons prouver pour obtenir suffisamment de connaissance pour que le lemme à propos de la transitivité de la permutation soit utilisé (et montre qu'à la fin du bloc de la boucle, puisque le tableau est une permutation du tableau au début, qui est lui-même une permutation du tableau original, alors le tableau à la fin du corps est une permutation du tableau original).

La seconde assertion dit que la seconde partie du tableau reste inchangée, et nous voulons utiliser cette connaissance pour montrer que le nombre d'occurrences des valeurs n'a pas changé. Ici, nous pourrions utiliser une combinaison des fonctions et macros lemmes pour prouver que la plage complète est une permutation (comme nous le ferons pour l'autre fonction). Cependant, écrire directement le code est un peu plus simple et requiert moins de preuves (comme nous le verrons plus tard), écrivons donc directement le code qui permet de prouver notre propriété.

Pour montrer que la plage complète est une permutation, nous devons montrer que le nombre d'occurrences de chaque valeur n'a pas changé. Nous savons que la première partie du tableau est une permutation de la même plage au début du corps de la boucle. Donc, nous savons déjà que le nombre d'occurrences de chaque v n'a pas changé pour une partie de notre tableau. En utilisant une boucle avec `j` allant de `i` à `end` et un invariant `permutationL,PI(a, beg, j)`, nous pouvons continuer le comptage des occurrences pour le reste de notre tableau, avec la connaissance que la fin n'a pas changé (quand `i+1` est plus petit que `end` sinon nous n'avons simplement plus rien à compter) :

```
1   for(size_t i = beg+1; i < end; ++i) {
2     //@ ghost L: ;
3     insert(a, beg, i);
4     //@ ghost PI: ;
5     //@ assert permutation{L, PI}(a, beg, i+1);
6     //@ assert unchanged{L, PI}(a, i+1, end) ;
7     /*@ ghost
8       if(i+1 < end){
9         /@ loop invariant i+1 <= j <= end ;
10           loop invariant permutation{L, PI}(a, beg, j) ;
11           loop assigns j ;
12           loop variant end - j ;
13        @/
14        for(size_t j = i+1 ; j < end ; ++j);
15      }
16    */
17  }
```

ce qui est suffisant pour assurer que la fonction `insertion_sort` est conforme à sa spécification à condition de finir la preuve de la fonction `insert`. Cette seconde fonction réalise des actions plus complexes, nous partirons de cette version annotée :

```
/*@
  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) ;
}
```

À nouveau, la preuve que cette fonction maintient la permutation du tableau est la partie la plus difficile de notre travail. Le fait que cette fonction garantisse que les valeurs sont bien triées est déjà établie. Utiliser la même technique que pour la fonction `insertion_sort` n'est pas si simple ici. En effet, la seconde partie du tableau a été « tournée » ce qui rend la propriété un peu plus complexe. Du coup, commençons par séparer notre tableau à la position d'insertion en deux parties, où nous montrons respectivement que :

— pour la première partie, puisqu'elle est inchangée, pour tout v le nombre d'occurrences n'a pas changé non plus ;
— pour la seconde partie, puisqu'elle a tourné, pour tout v, le nombre d'occurrences n'a pas changé.

D'abord, définissons une fonction lemme qui permet d'explicitement couper une plage de valeur en deux sous-parties dans lesquelles nous pouvons compter séparément :

```
/*@ 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);
  }
*/
```

Nous pouvons noter que cette propriété est prouvée d'une manière qui est très similaire à ce que nous avons écrit dans le corps de la boucle de la fonction `insertion_sort`, nous commençons au point à partir duquel nous voulons compter et nous montrons que la propriété reste vraie pour le reste du tableau.

Nous pouvons utiliser notre fonction pour couper le tableau à la bonne position après la boucle. Cependant, nous ne pouvons le faire que pour le nouveau contenu du tableau. En effet, pour établir cela pour le tableau original, nous devons appeler la fonction sur le tableau original pour lequel nous ne connaissons pas encore la valeur de i. Par conséquent, écrivons une autre version de la propriété « *split* » qui nous montrer que nous pouvons couper le tableau à toute position, donc rendons la variable `split` universellement quantifiée et utilisons la fonction précédente pour montrer que cette nouvelle propriété est vraie.

```
/*@ 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);
    }
  }
*/
```

Et nous pouvons souper notre tableau original et le nouveau :

```
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);
}
```

Maintenant, les seules parties restantes de la preuve sont de montrer qu'un tableau inchangé est une permutation et ensuite que la rotation maintient également une permutation. Ici, nous avons besoin d'une macro. Commençons avec la plus facile : le tableau inchangé, que l'on a prouvé quasiment à l'identique dans la fonction `insertion_sort`. Nous commençons par construire notre contexte de preuve :

```
/*@
  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) ;
}
```

La fonction `unchanged_permutation_premise` assure que nous avons modifié le tableau (et donc créé un nouvel état mémoire) et que le tableau est inchangé entre la précondition et la postcondition. Nous pouvons construire notre macro lemme :

```
#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); @/
```

qui correspond presque à ce que nous avons écrit pour `insert_sort`, et utiliser cette macro là où nous en avons besoin dans la fonction `insert`.

```
1   //@ assert unchanged{Pre, Here}(a, beg, i) ;
2   //@ ghost unchanged_permutation(Pre, Here, a, beg, i) ;
3   //@ assert permutation{Pre, Here}(a, beg, i) ;
```

La seule propriété restante est la plus complexe et concerne le prédicat `rotate_left`. Écrivons d'abord un contexte pour préparer notre 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 }
```

Comment pouvons-nous prouver cette propriété ? Nous devons d'abord constater que puisque tous les éléments depuis le début jusqu'à l'avant-dernier ont été décalées d'une cellule, le nombre d'occurrences dans cette partie décalée n'a pas changé. Ensuite, nous devons montrer que le nombre d'occurrences de v respectivement dans la dernière cellule du tableau original et la première cellule du nouveau tableau est le même (puisque l'élément correspondant est le même). À nouveau, nous nous reposons sur la fonction `split` pour compter séparément les éléments décalés et l'élément qui est déplacé de la fin vers le début. Cependant, l'appel correspondant dans le tableau original doit à nouveau être réalisée avant le code qui modifie le tableau original (voir ligne 12) dans le code précédent, et nous devons prendre cela en compte quand nous insérerons notre utilisation de la macro dans la fonction `insert`.

Présentons maintenant la macro qui est utilisée pour prouver que notre lemme est valide :

```
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); @/
```

L'invariant de boucle est très similaire à ce que nous avons écrit jusqu'à maintenant, la seule différence est que nous devons tenir compte du glissement des éléments. De plus, pour l'invariant, nous avons dû ajouter une assertion pour aider les prouveurs automatiques à remarquer que le dernier élément de chaque plage est le même (notons que selon les versions des prouveurs ou la puissance de la machine, cela peut parfois ne pas être nécessaire). Une différence plus importante comparativement à nos exemples précédents est le fait qu'ici, nous devons fournir plus d'information aux SMT solveurs en ajoutant d'autres appels de fonction fantôme (ligne 15, pour couper le premier élément du tableau), ainsi que des assertions pour guider les dernières étapes de preuve :

— 16–29 : nous rappelons que le tableau original peut être coupé au niveau du dernier élément,
— 21–25 : nous montrons que comme le premier élément du tableau est le dernier élément du tableau original (21), le nombre d'occurrences pour toute valeur dans ces plages est le même (22–24).

Nous pouvons utiliser cette macro dans notre programme :

```
1   //@ assert rotate_left{Pre, Here}(a, i, last+1) ;
2   //@ ghost rotate_left_permutation(Pre, Here, a, i, last+1) ;
3   //@ assert permutation{Pre, Here}(a, i, last+1) ;
```

Cependant, nous avons besoin de montrer que la plage au label `Pre` peut être coupée à `last`. Pour cela, nous utilisons une autre variante de la fonction `split`, qui montre que toute sous-plage peut être coupée avant le dernier élément (si elle n'est pas vide) :

```
1  /*@ ghost
2    /@
3      requires beg < end ;
4
5      assigns \nothing ;
6
7      ensures \forall int v, size_t any ; beg <= any < end ==>
8        l_occurrences_of(v, a, any, end) ==
9        l_occurrences_of(v, a, any, end-1) + l_occurrences_of(v, a, end-1, end) ;
10   @/
11   void l_occurrences_of_from_any_split_last(int* a, size_t beg, size_t end){
12     /@
13       loop invariant beg <= i <= end-1 ;
14       loop invariant \forall int v, size_t j ;
15         beg <= j < i ==>
16         l_occurrences_of(v, a, j, end) ==
17         l_occurrences_of(v, a, j, end-1) + l_occurrences_of(v, a, end-1, end) ;
18       loop assigns i ;
19       loop variant (end - 1) - i ;
20     @/
21     for(size_t i = beg ; i < end-1 ; ++i){
```

```
22        l_occurrences_of_explicit_split(a, i, end-1, end);
23      }
24    }
25  */
```

que nous pouvons ensuite appeler avant la boucle de la fonction `insert` :

```
1  void insert(int* a, size_t beg, size_t last){
2    size_t i = last ;
3    int value = a[i] ;
4
5    //@ ghost l_occurrences_of_split(a, beg, last+1);
6    //@ ghost l_occurrences_of_from_any_split_last(a, beg, last+1);
```

Notons que selon les versions des prouveurs automatiques, les assertions en lignes 21 à 25 de la macro, à propos de l'élément au début et à la fin du tableau, pourraient ne pas être prouvées à cause de la complexité du contexte de preuve. Aidons les solveurs une dernière fois en ajoutant un dernier lemme, automatiquement prouvé par les solveurs SMT, qui nous énonce la relation en question pour tout tableau et toute position du tableau :

```
1  /*@ lemma one_same_element_same_count{L1, L2}:
2    \forall int* a, int* b, int v, integer pos_a, pos_b ;
3      \at(a[pos_a], L1) == \at(b[pos_b], L2) ==>
4      l_occurrences_of{L1}(v, a, pos_a, pos_a+1) ==
5      l_occurrences_of{L2}(v, b, pos_b, pos_b+1) ;
```

qui nous garantit que la fonction annotée résultante est entièrement prouvée :

```
1  /*@
2    requires beg < last < SIZE_MAX && \valid(a + (beg .. last));
3    requires sorted(a, beg, last) ;
4
5    assigns a[ beg .. last ] ;
6
7    ensures permutation{Pre, Post}(a, beg, last+1);
8    ensures sorted(a, beg, last+1) ;
9  */
10 void insert(int* a, size_t beg, size_t last){
11   size_t i = last ;
12   int value = a[i] ;
13
14   //@ ghost l_occurrences_of_split(a, beg, last+1);
15   //@ ghost l_occurrences_of_from_any_split_last(a, beg, last+1);
16
17   /*@
18     loop invariant beg <= i <= last ;
19     loop invariant \forall integer k ; i <= k < last ==> a[k] > value ;
20     loop invariant \forall integer k ; beg <= k <= i    ==> a[k] == \at(a[k], Pre) ;
21     loop invariant \forall integer k ; i+1 <= k <= last ==> a[k] == \at(a[k-1], Pre) ;
22
23     loop assigns i, a[beg .. last] ;
24     loop variant i ;
25   */
26   while(i > beg && a[i - 1] > value){
27     a[i] = a[i - 1] ;
28     --i ;
29   }
30   a[i] = value ;
```

```
31    //@ assert sorted(a, beg, last+1) ;
32
33    /*@ assert
34        \forall int v ;
35        l_occurrences_of{Pre}(v, a, \at(i, Here), last+1) ==
36          l_occurrences_of{Pre}(v, a, \at(i, Here), last) +
37          l_occurrences_of{Pre}(v, a, last, last +1);
38    */
39
40    //@ assert rotate_left{Pre, Here}(a, i, last+1) ;
41    //@ ghost rotate_left_permutation(Pre, Here, a, i, last+1) ;
42    //@ assert permutation{Pre, Here}(a, i, last+1) ;
43
44    //@ assert unchanged{Pre, Here}(a, beg, i) ;
45    //@ ghost unchanged_permutation(Pre, Here, a, beg, i) ;
46    //@ assert permutation{Pre, Here}(a, beg, i) ;
47
48    //@ ghost l_occurrences_of_explicit_split(a, beg, i, last+1);
49  }
```

Nous mettons finalement en valeur le fait que le contexte de preuve peut rendre le travail vraiment difficile pour les solveurs SMT. Tout simplement, si nous inversons les preuves à propos du chaque partie du tableau, à savoir, en commençant par la partie `unchanged` puis la partie `rotate`, la preuve a de très bonnes chances d'échouer, car elle fait grossir le contexte de preuve pour la preuve la plus difficile.

7.3.6. Exercices

7.3.6.1. La somme des N premiers entiers

En utilisant des fonctions lemmes, nous pouvons prouver que le lemme à propos de la somme des N premiers entiers. Vous avez peut-être déjà fait cette preuve au lycée, maintenant, il est temps de faire cette preuve en C et ACSL. Écrire un contrat pour la fonction suivante qui exprimer en postcondition que la somme des N premiers entiers est `N(N+1)/2`. Compléter le corps de la fonction avec une boucle pour prouver la propriété. Nous conseillons de légèrement modifier l'invariant pour faire disparaître la division (qui sur les entiers a certaines propriétés qui rendent son utilisation difficile avec des solveurs SMT en fonction des contraintes qui existent sur les valeurs utilisées).

```
1  /*@
2    logic integer sum_of_n_integers(integer n) =
3      (n <= 0) ? 0 : sum_of_n_integers(n-1) + n ;
4  */
5
6  /*@ ghost
7    /@
8      assigns \nothing ;
9      ensures \true ; // to complete
10   @/
11   void lemma_value_of_sum_of_n_integers_2(unsigned n){
12     // ...
13   }
14 */
```

Maintenant, généralisons à toutes bornes avec la somme de tous les entiers entre deux bornes `fst` et `lst`. Nous fournissons la fonction logique et le contrat, à vous d'écrire le corps

de la fonction de manière à vérifier la postcondition. À nouveau, nous conseillons d'exprimer l'invariant sans 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){
    // ...
  }
*/
```

Finalement, prouver cette fonction :

```
/*@
  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 ;
}
```

Cela ne devrait pas être trop difficile, et ce que nous obtenons est une preuve que nous avons écrit une optimisation correcte pour la fonction qui calcule la somme des N premiers entiers.

7.3.6.2. Propriétés à propos du comptage d'occurrence

Dans cet exercice, nous voulons prouver un ensemble de propriétés intéressantes à propos de notre définition logique `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
  }
```

```
20
21   size_t occ_pos_find(int v, int* arr, size_t len){
22     // ...
23     // needs occ_monotonic
24   }
25
26   void occ_pos_exists(int v, int* arr, size_t len){
27     // ...
28     // should use occ_pos_find
29   }
30 */
```

La fonction `occ_bounds` doit énoncer que le nombre d'occurrences de `v` dans un tableau est compris entre 0 et `len`.

La fonction `not_in_occ_0` doit énoncer que si `v` n'est pas dans le tableau alors le nombre d'occurrences de `v` est 0.

La fonction `occ_monotonic` doit énoncer que le nombre d'occurrences de `v` dans un tableau entre 0 et `pos` est inférieur ou égal au nombre d'occurrences de `v` entre 0 et `more` si `more` est supérieur ou égal à `pos`

La fonction `occ_0_not_in` doit énoncer que si le nombre d'occurrences de `v` dans le tableau est 0 alors `v` n'est pas dans le tableau. Notons que `occ_monotonic` serait probablement utile.

La fonction `occ_pos_find` doit trouver un indice `i` tel que `arr[i]` est `v`, à supposer que le nombre d'occurrences de `v` est positif. `occ_monotonic` peut être utile.

Finalement, la fonction `occ_pos_exists` doit traduire le contrat de la fonction précédente en utilisant une variable quantifiée existentiellement et utiliser la fonction précédente pour obtenir gratuitement la preuve.

Pour toutes ces fonctions, WP doit être paramétré avec le contrôle d'absence d'erreurs d'exécution ainsi que les options `-warn-unsigned-overflow` et `-warn-unsigned-downcast`.

7.3.6.3. Un vrai exemple avec la somme

Reprendre la preuve effectuée dans le chapitre précédent pour l'exercice 7.2.5.4. Modifier les annotations pour assurer que plus aucun lemme classique n'est nécessaire. Voici un squelette pour le fichier :

```
1  #include <limits.h>
2  #include <stddef.h>
3
4  /*@
5    axiomatic Sum_array{
6      logic integer sum(int* array, integer begin, integer end) reads array[begin .. (end-1)];
7
8      axiom empty:
9        \forall int* a, integer b, e; b >= e ==> sum(a,b,e) == 0;
10     axiom range:
11       \forall int* a, integer b, e; b < e ==> sum(a,b,e) == sum(a,b,e-1)+a[e-1];
12   }
13 */
```

```
14
15 /*@
16   predicate unchanged{L1, L2}(int* array, integer begin, integer end) =
17     \forall integer i ; begin <= i < end ==> \at(array[i], L1) == \at(array[i], L2) ;
18 */
19
20 /*@ ghost
21   void sum_separable(int* array, size_t begin, size_t split, size_t end){
22     // ...
23   }
24 */
25
26 #define unchanged_sum(_L1, _L2, _arr, _beg, _end) ;
27
28
29 /*@
30   requires i < len ;
31   requires array[i] < INT_MAX ;
32   requires \valid(array + (0 .. len-1));
33   assigns array[i];
34   ensures sum(array, 0, len) == sum{Pre}(array, 0, len)+1;
35 */
36 void inc_cell(int* array, size_t len, size_t i){
37   // ...
38   array[i]++ ;
39   // ...
40 }
```

A mesure que nous essayons de prouver des propriétés plus complexes, particulièrement quand les programmes contiennent des boucles, et des structures de données complexes, il y a un part de « *trial and error* » pour comprendre ce qui manque aux prouveurs pour établir la preuve.

Il peut manquer des hypothèses. Dans ce cas, nous pouvons essayer d'ajouter des assertions pour guider les prouveurs, ou écrire du code ghost avec les bons invariants, ce qui permet d'effectuer une part du raisonnement nous même lorsque c'est trop difficile pour les solveurs SMT.

Avec un peu d'expérience, il est possible de lire le contenu des conditions de vérification ou essayer de faire la preuve soi-même avec l'assistant de preuve Coq pour voir si la preuve semble réalisable. Parfois, le prouveur a juste besoin de plus de temps, dans ce cas, nous pouvons augmenter (parfois beaucoup) le temps de *timeout*. Bien sûr la propriété peut parfois être trop difficile pour le prouveur et le code ghost ne pas être adapté, dans ce cas, il sera nécessaire de terminer la preuve nous mêmes.

Finalement, l'implémentation peut être incorrecte, et dans ce cas, nous devons la corriger. A ce moment là, nous utiliserons du test et non de la preuve, car un test nous permettra de mettre en évidence la présence du bug et de l'analyser.

8. Conclusion

> Voilà, c'est fini ...
>
> *Jean-Louis Aubert, Bleu Blanc Vert, 1989*

... pour cette introduction à la preuve de programmes avec Frama-C et WP.

Tout au long de ce tutoriel, nous avons vu comment utiliser ces outils pour spécifier ce que nous attendons de nos programmes et vérifier que le code produit correspond effectivement à la spécification que nous en avons faite. Cette spécification passe par l'annotation de nos fonctions avec le contrat auquel elle doit se conformer, c'est-à-dire les propriétés que doivent vérifier les entrées de la fonction pour garantir son bon fonctionnement et les propriétés que celle-ci s'engage à assurer sur les sorties en retour, associées aux contrôles que nous impose l'outil à propos des problèmes spécifiques au langage C (plus particulièrement, montrer l'absence de *runtime errors*).

À partir de nos programmes spécifiés, WP est capable de produire un raisonnement nous disant si, oui ou non, notre programme répond au besoin exprimé. La forme de raisonnement utilisée étant complètement modulaire, elle nous permet de prouver les fonctions une à une et de les composer.

WP ne comprend pas, à l'heure actuelle, l'allocation dynamique. Une fonction qui en utiliserait ne pourrait donc pas être prouvée. Mais même sans cela, une large variété de fonctions n'ont pas besoin d'effectuer d'allocation dynamique, travaillant donc sur des données déjà allouées. Et ces fonctions peuvent ensuite être appelées avec l'assurance qu'elles font correctement leur travail. Si nous ne voulons pas prouver le code appelant, nous pouvons toujours écrire quelque chose comme cela :

```
/*@
  requires some_properties_on(a);
  requires some_other_on(b);

  assigns ...
  ensures ...
*/
void ma_fonction(int* a, int b){
  //je parle bien du "assert" de "assert.h"
  assert(/*properties on a*/ && "must respect properties on a");
  assert(/*properties on b*/ && "must respect properties on b");
}
```

Ce qui nous permet ainsi de bénéficier de la robustesse de notre fonction tout en ayant la possibilité de déboguer un appel incorrect dans un code que nous ne voulons ou pouvons pas prouver.

Écrire les spécifications est parfois long, voire fastidieux. Les constructions de plus haut niveau d'ACSL (prédicats, fonctions logiques, axiomatisations) permettent d'alléger un peu ce travail, de la même manière que nos langages de programmation nous permettent de définir des types englobant d'autres types et des fonctions appelant des fonctions, nous menant vers le programme final. Mais malgré cela, l'écriture de spécification dans un langage formel quel qu'il soit représente une tâche ardue.

Cependant, cette étape de **formalisation** du besoin est **très importante**. Concrètement, une telle formalisation est, à bien y réfléchir, un travail que tout développeur devrait s'efforcer de faire. Et pas seulement quand il cherche à prouver son programme. Même la production de tests pour une fonction nécessite de bien comprendre son but si nous voulons tester ce qui est nécessaire et uniquement ce qui est nécessaire. Et écrire les spécifications dans un langage formel est une aide formidable (même si elle peut être frustrante, reconnaissons-le) pour avoir des spécifications claires.

Les langages formels, proches des mathématiques, sont précis. Les mathématiques ont cela pour elles. Qu'y a-t-il de plus terrible que lire une spécification écrite en langue naturelle pure beurre, avec toute sa panoplie de phrase à rallonge, de verbes conjugués à la forme conditionnelle, de termes imprécis, d'ambiguïtés, compilée dans des documents administratifs de centaines de pages, et dans laquelle nous devons chercher pour déterminer « bon alors cette fonction, elle doit faire quoi ? Et que faut-il valider à son sujet ? ».

Les méthodes formelles ne sont, à l'heure actuelle, probablement pas assez utilisées, parfois par méfiance, parfois par ignorance, parfois à cause de préjugés datant des balbutiements des outils, il y a 20 ans. Nos outils évoluent, nos pratiques dans le développement changent, probablement plus vite que dans de nombreux autres domaines techniques. Ce serait probablement faire un raccourci bien trop rapide que dire que ces outils ne pourront jamais être mis en œuvre quotidiennement. Après tout, nous voyons chaque jour à quel point le développement est différent aujourd'hui par rapport à il y a 10 ans, 20 ans, 40 ans. Et nous pouvons à peine imaginer à quel point il sera différent dans 10 ans, 20 ans, 40 ans.

Ces dernières années, les questions de sûreté et de sécurité sont devenues de plus en plus présentes et cruciales. Les méthodes formelles connaissent également de fortes évolutions et leurs apports pour ces questions sont très appréciés. Par exemple, ce lien mène vers un rapport d'une conférence sur la sécurité ayant rassemblé des acteurs du monde académique et industriel, dans lequel nous pouvons lire :

> 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*

Afficher le contenu masqué

8.1. Pour aller plus loin

8.1.1. Avec Frama-C

Frama-C propose divers moyens d'analyser les programmes. Dans les outils les plus courants qui sont intéressants à connaître d'un point de vue vérification statique et dynamique pour l'évaluation du bon fonctionnement d'un programme, on peut citer :

— l'analyse par interprétation abstraite avec EVA ,
— la transformation d'annotation en vérification à l'exécution avec E-ACSL .

Le but de la première est de calculer les domaines des différentes variables à tous les points de programme. Quand nous connaissons précisément ces domaines, nous sommes capables de déterminer si ces variables peuvent provoquer des erreurs. Par contre, cette analyse est effectuée sur la totalité du programme et pas modulairement, elle est par ailleurs fortement dépendante du type de domaine utilisé (nous n'entrerons pas plus dans les détails ici) et elle conserve moins bien les relations entre les variables. En compensation, elle est vraiment complètement automatique (modulo les entrées du programme), il n'y a même pas besoin de poser des invariants de boucle ! La partie plus « manuelle » sera de déterminer si oui ou non les alarmes lèvent des vraies erreurs ou si ce sont de fausses alarmes.

La seconde analyse permet de générer depuis un programme d'origine, un nouveau programme où les assertions sont transformées en vérification à l'exécution. Si les assertions échouent, le programme échoue. Si elles sont valides, le programme a le même comportement que s'il n'y avait pas d'assertions. Un exemple d'utilisation est d'utiliser l'option `-rte` de Frama-C pour générer les vérifications d'erreur d'exécution comme assertion et de générer le programme de sortie qui contiendra les vérifications en question.

Il existe divers autres greffons pour des tâches très différentes dans Frama-C.

Et finalement, la dernière possibilité qui motivera l'utilisation de Frama-C, c'est la possibilité de développer ses propres greffons d'analyse, à partir de l'API fournie au niveau du noyau. Beaucoup de tâches peuvent être réalisées par l'analyse du code source et Frama-C permet de forger différentes analyses.

8.1.2. Avec la preuve déductive

Tout au long du tutoriel, nous avons utilisé WP pour générer des conditions de vérification à partir de programmes et de leurs spécifications. Par la suite, nous avons utilisé des solveurs automatiques pour assurer que les propriétés en question sont bien vérifiées.

Lorsque nous passons par d'autres solveurs qu'Alt-Ergo, le dialogue avec ceux-ci est assuré par une traduction vers le langage de Why3 qui fait ensuite le pont avec les prouveurs automatiques. Mais ce n'est pas la seule manière d'utiliser Why3. Celui-ci peut tout à fait être utilisé seul pour écrire et prouver des algorithmes. Il embarque notamment un ensemble de théories déjà présentes pour un certain nombre de structures de données.

Il y a un certain nombre de preuves qui ne peuvent être déchargées par les prouveurs automatiques. Dans ce genre de cas, nous devons nous rabattre sur de la preuve interactive. Why3 peut par exemple extraire des conditions de vérification vers Coq, et il est très intéressant d'étudier ce langage, il peut par exemple servir à constituer des bibliothèques de lemmes génériques et

prouvés. À noter que Why3 peut également extraire ses conditions de vérification vers Isabelle ou PVS qui sont d'autres assistants de preuve.

Finalement, il existe d'autres logiques de programmes, comme la logique de séparation ou les logiques pour les programmes concurrents. Encore une fois ce sont des notions intéressantes à connaître, elles peuvent inspirer la manière dont nous spécifions nos programmes pour la preuve avec WP, elles pourraient également donner lieu à de nouveaux greffons pour Frama-C. Bref, tout un monde de méthodes à explorer.