INTRODUCTION À LA PREUVE DE PROGRAMMES C AVEC FRAMA-C ET SON GREFFON WP

Allan Blanchard

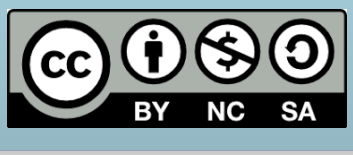
22 juin 2017

Reste de savoir

Table des matières

1 Introduction

License



Information

Le choix de certain 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. Le reste vient de mon expérience personnelle avec Frama-C et WP.
Le seul pré-requis 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 programme 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 redévelopper.

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 convaincre que nous pouvons de plus en plus souvent toucher du doigt l'idée 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.

Information

Merci aux différents bêta-testeurs pour leurs remarques constructives :
- Taurre (l'exemple en section III - 3 - 4 a honteusement été copié d'un de ses posts).
- barockobamo
- Vayel

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

Finalement, un grand merci à Jens Gerlach pour son aide lors de la traduction anglaise du tutoriel.

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 respecte 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 soit assez mal comprise.

Outre la différence entre sûreté et sécurité (qui peut **vaguement** être définie par la présence d'un élément malveillant dans l'histoire), 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. Et ce sont également les deux moments où les erreurs peuvent être introduites.

Tout au long de ce tutoriel, nous allons nous attacher à montrer comment nous pouvons prouver que l'implémentation est conforme à la spécification. 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 bugs de plus que la spécification ». D'une part, c'est un sacré pas en avant par rapport à « le test nous montre que l'implémentation ne contient pas beaucoup plus de bugs que la spécification ». Et d'autre part, il existe également des techniques permettant d'analyser les spécifications en quête d'erreurs ou de manquements. 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 du programme accessible 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

Les méthodes formelles, comme elles sont appelées, permettent dans le domaine de l'informatique 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 allons nous intéresser à la vérification que nos programmes sont conformes au comportement que nous attendons de leur part. Nous allons utiliser 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. A l'inverse, les analyses dynamiques comme le test de programme nécessite 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 pourrons, 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 peut être 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, nous pouvons prendre l'exemple d'un chronomètre simple. Une modélisation très abstraite du comportement de notre chronomètre est la suivante est présentée dans la figure 2.1.

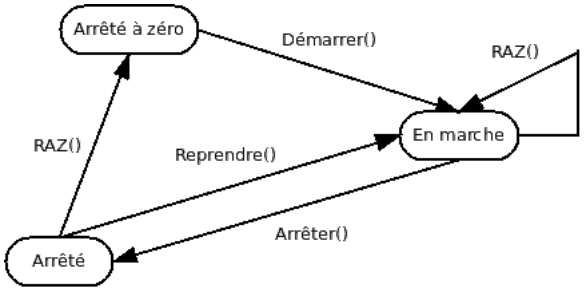


Figure 2.1 – Modélisation très abstraite d'un chronomètre

Nous avons bien une modélisation du comportement de notre chronomètre avec différents états qu'il peut atteindre en fonction des actions qui sont réalisées à son sujet. Cependant, nous n'avons pas modélisé comment ces états sont réprésentés 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 = 0s ;
— État en marche : temps > 0s ;
— État arrêté : temps > 0s.

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 va nous servir à produire l'exécutable. Mais ce n'est pour autant pas 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'oeil, 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 [1].

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 en réalité pas arriver 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 allons utiliser, 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 :

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

Où P et Q sont des prédicats, 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 ». Dis autrement, P est une pré-condition suffisante pour que C nous amène à la post-condition Q. Par exemple, le triplet correspondant à l'action « ne rien faire » (**skip**) est le suivant :

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

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

Quand nous ne faisons rien, la post-condition est la même que la pré-condition.

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 allons utiliser repose sur la technique de calcul de plus faible pré-condition.

2.1.4 Calcul de plus faible pré-condition

Le calcul de plus faible pré-condition 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 nous donne des règles nous 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 pré-condition et d'une ou plusieurs instructions, la plus forte post-condition 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 post-condition voulue est au plus aussi forte, alors nous avons prouvé qu'il n'y a pas de comportements non-voulus.

Par exemple :

```
int a = 2;
a = 4;
//post-condition calculée : a == 4
//post-condition 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 pré-condition, fonctionne dans le sens inverse, nous parlons de « raisonnement vers l'arrière » ou *backward-reasoning*. À partir de la post-condition voulue et de l'instruction que nous traitons, nous allons trouver la pré-condition minimale qui nous assure ce fonctionnement. Si notre pré-condition réelle est au moins aussi forte, c'est-à-dire, qu'elle implique la plus faible pré-condition, alors notre programme est valide.

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

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

Quelle est la pré-condition minimale pour que la post-condition $\{x = 42\}$ soit respectée ? La règle nous dira que P est $\{\mathsf{a} = 42\}$.

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 avec ou sans collaborations. Les plugins fournis par défaut comprennent diverses analyses statiques (sans exécution du code analysé), dynamiques (avec exécution du code), ou combinant les deux.

Frama-C nous fournit également un langage de spécification appelé ACSL (« Axel ») pour *ANSI C Specification Language* et qui va nous permettre 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 allons beaucoup parler 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*, elle 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 obligations de preuves, qui sont des formules logiques dont nous devons vérifier la 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 (statisfiabilité 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 mis à jour et maintenu par OCamlPro.

2.2.2 Installation

Frama-C est un logiciel développé sous Linux et OSX. 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 :

Attention

- Le tutoriel présentera le fonctionnement sous Linux et l'auteur n'a pas expérimenté les différences d'utilisation qui pourraient exister avec Windows,
- 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 :

```
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.

Pour vérifier l'installation, c'est dans la sous-section « Vérifier l'installation » que les informations sont données.

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 soit présents préalablement à l'installation de Frama-C :

- lib gtk2 dev
- lib gtksourceview2 dev
- lib gnomecanvas2 dev
- (conseillé) lib zarith dev

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

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

Pour vérifier l'installation, c'est dans la sous-section « Vérifier l'installation » que les informations sont données.

2.2.2.1.3 Via compilation « manuelle » Pour installer Frama-C via 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).

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

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

Pour vérifier l'installation, c'est dans la sous-section « Vérifier l'installation » que les informations sont données.

2.2.2.2 OSX

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

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
```

Pour vérifier l'installation, c'est dans la sous-section « Vérifier l'installation » que les informations sont données.

2.2.2.3 Windows

L'installation de Frama-C sous Windows passe par l'usage de Cygwin et d'une version expérimentale d'Opam pour celui-ci. Il faut donc installer ces deux logiciels et le compilateur Ocaml basé sur MinGW.

Les instructions d'installation se trouvent ici :

Frama-C - Windows

Le lancement de Frama-C se fera par l'intermédiaire de cygwin.

Pour vérifier l'installation, c'est dans la sous-section « Vérifier l'installation » que les informations sont données.

2.2.3 Vérifier l'installation

Pour vérifier votre installation, nous allons utiliser le code très simple présenté en figure 2.2 dans un fichier « main.c » :

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
```

La fenêtre présentée en figure 2.3 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 illustré par la figure 2.4.

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 forum pourront vous fournir de l'aide.

```
/*@
  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;
}
```

Figure 2.2 – Code pour la vérification de l'installation

Attention

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

Information

Pour les daltoniens, il est possible de lancer Frama-C avec un mode où les pastilles de couleurs sont remplacées :

```
$ frama-c-gui -gui-theme colorblind fichier.c
```

2.2.4 (Bonus) Installation de prouveurs supplémentaires

Cette partie est purement optionnelle, rien de ce qui est ici ne sera complètement nécessaire 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 pré-intégré Alt-Ergo et d'avoir besoin d'outils plus puissants.

2.2.4.1 Coq

Coq, développé par l'organisme de recherche Inria, est un assistant de preuve. C'est-à-dire que nous écrivons nous-même 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

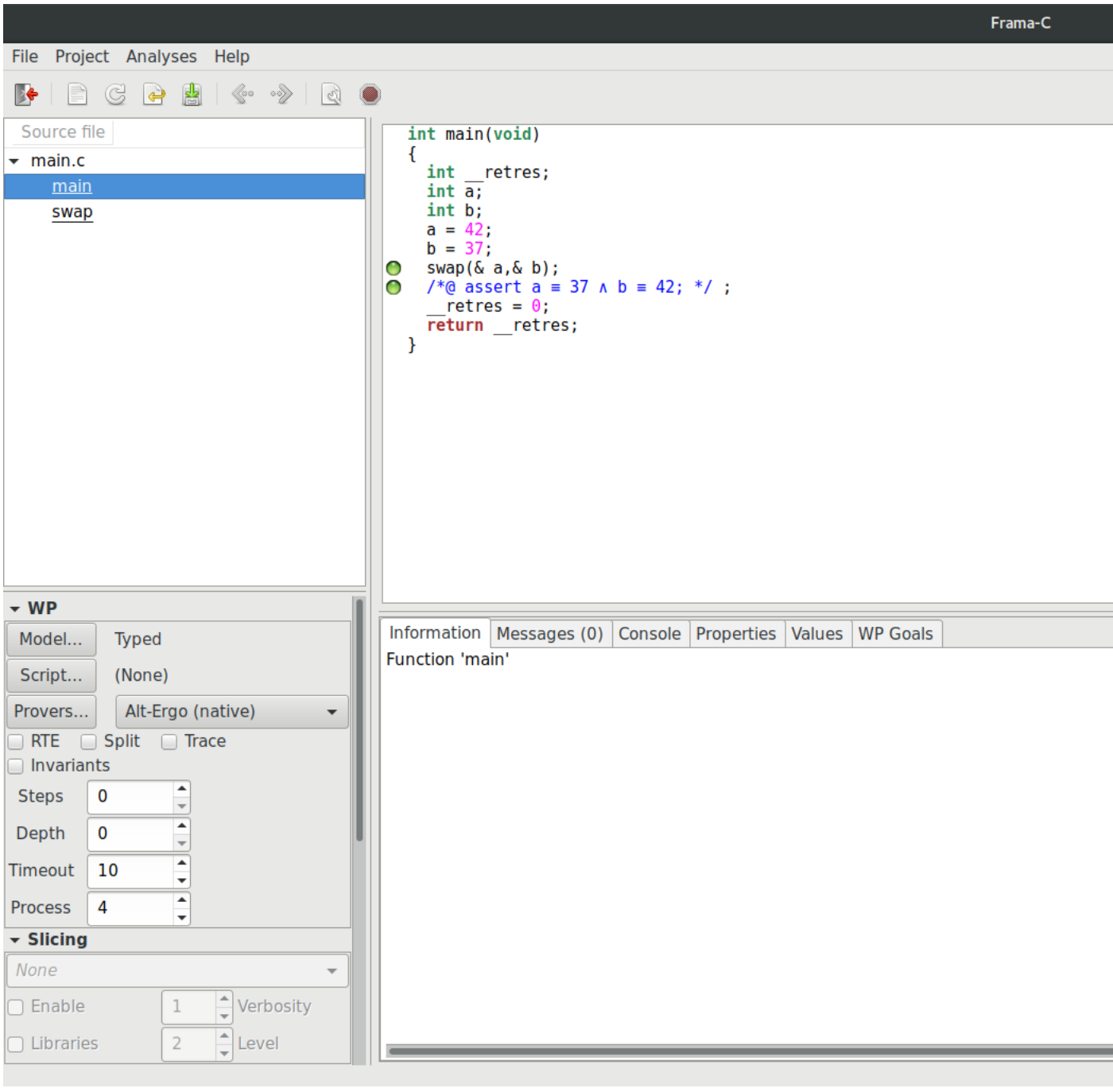


Figure 2.3 – Vérification installation 1

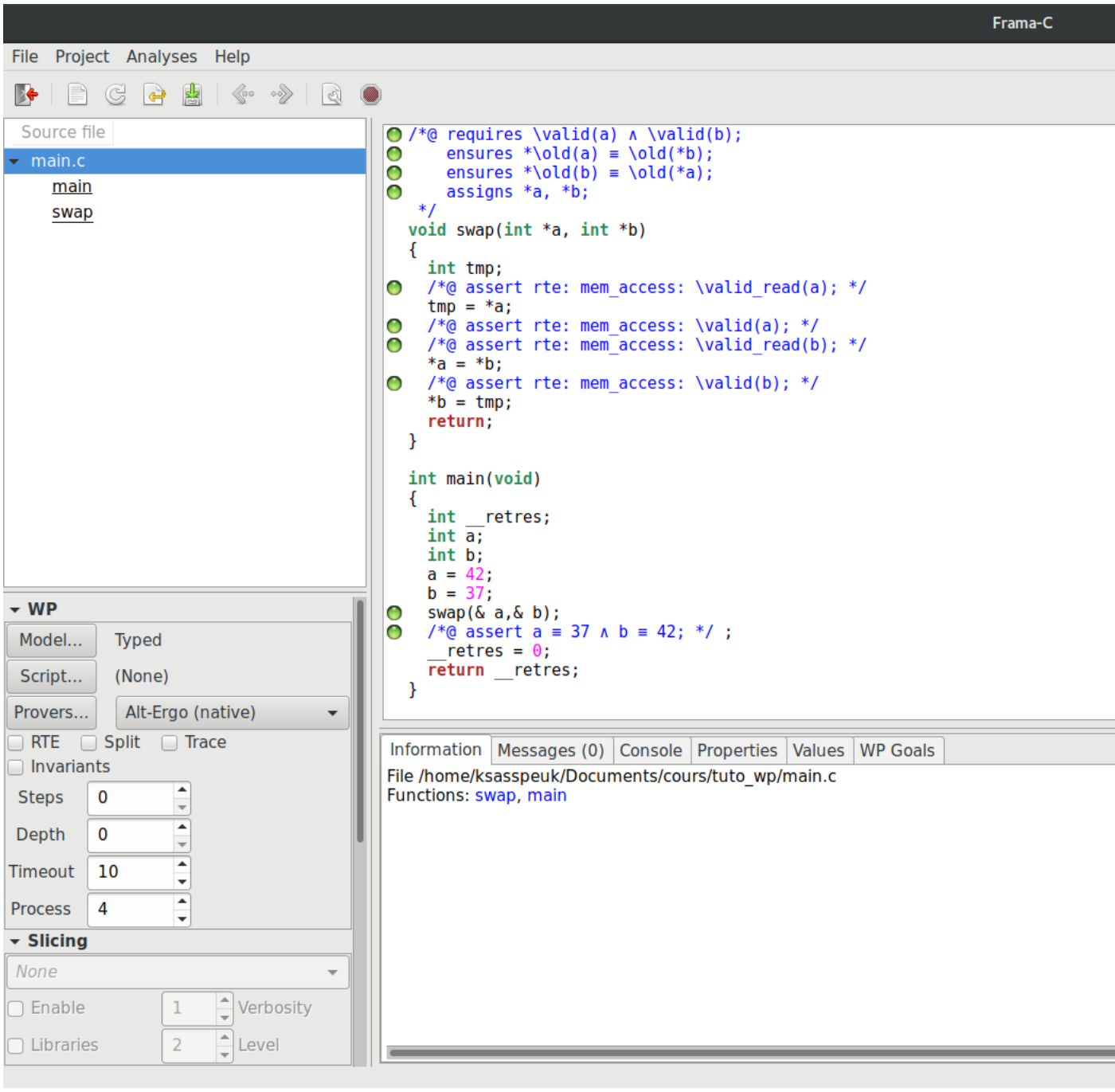


Figure 2.4 – Vérification installation 2

pourra générer des obligations de preuve traduites en Coq et nous laisser écrire la preuve en question.

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

Information

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 par ici : The Coq Proof Assistant.

Pour utiliser Coq lors d'une preuve dans Frama-C, il faudra le sélectionner par l'intermédiaire du panneau latéral à gauche, dans la partie qui concerne WP (voir figure 2.5).

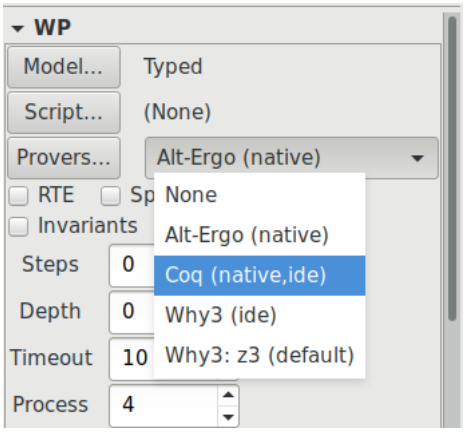


Figure 2.5 – Sélectionner l'assistant de preuve Coq

Information

Nous n'avons pas expérimenté cette procédure sous Windows.

2.2.4.2 Why3

Attention

À la connaissance de l'auteur, il n'est pas possible (ou vraiment pas facile) d'installer Why3 sous Windows. L'auteur ne saurait être tenu responsable de blessures subies pendant une telle opération.

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 peut nous intéresser. WP sera capable de traduire ses obligations de preuve vers le langage de Why3 et d'utiliser ce dernier pour dialoguer avec un certain nombre de prouveurs automatiques.

Pour plus d'informations sur Why3 c'est sur son site que ça se passe. Si Opam est installé, Why3 est disponible par son intermédiaire. Sinon, il y a une procédure d'installation proposée.

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.

Pour utiliser les prouveurs en question, la procédure est expliquée dans la partie sur Coq pour la sélection d'un prouveur différent d'Alt-Ergo. À noter qu'il faudra peut-être demander la détection des prouveurs fraîchement installé avec le bouton « Provers » puis « Detect Provers » dans la fenêtre qui s'ouvre.

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 ;
— 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 allons commencer 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 d'écrire quelques preuves jouables.

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 pouvoir prouver qu'un code est valide, il faut d'abord pouvoir spécifier ce que nous attendons de lui. La preuve de programme consistera donc à s'assurer que le code que nous avons écrit correspond bien à la spécification qui décrit le rôle qui lui a été attribué. 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 va s'exécuter. C'est-à-dire : ce que doit respecter 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 pré-condition** ; et ce que s'engage à respecter la fonction en retour à propos de l'état de la mémoire et de la valeur de retour : **la post-condition**.

Ces propriétés sont exprimées en langage ACSL, 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 : pointeur valide) ;
- 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. Syntaxiquement, un contrat de fonction est intégré dans les sources de la manière suivante :

```
/*@
  //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 post-condition, puisque c'est ce que nous attendons en priorité de notre programme (nous nous intéresserons ensuite aux pré-conditions).

3.1.1 Post-condition

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

```
/*@
  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'opposée de la valeur.

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

```
/*@
  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. 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 ».

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).

Information

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	T

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 :

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

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 :

```
$ 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 de directement ouvrir le(s) fichier(s) depuis la ligne de commande en le(s) passant en argument(s) de `frama-c-gui`.

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

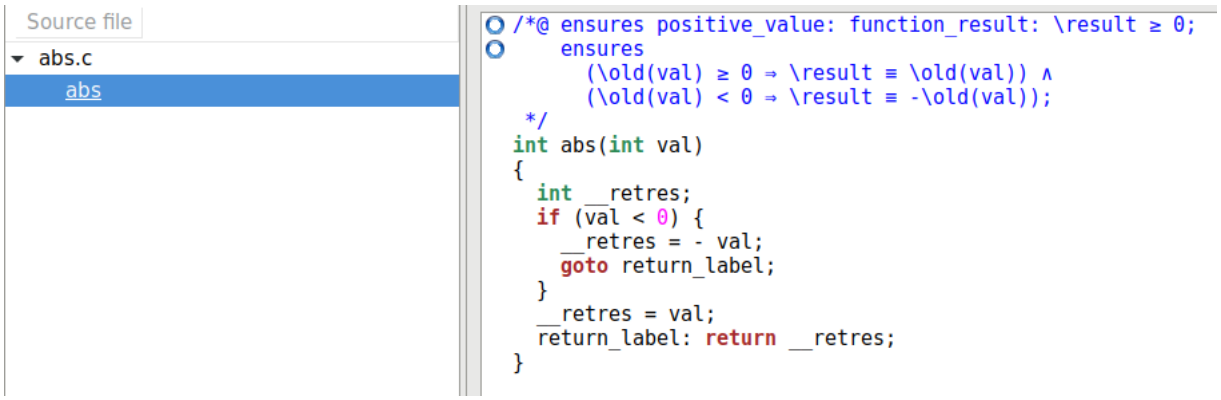


Figure 3.1 – Le volet latéral liste l'arbre des fichiers et des fonctions

La fenêtre de Frama-C s'ouvre, dans le volet correspondant aux fichiers et aux fonctions, nous pouvons sélectionner la fonction `abs`. Aux différentes lignes `ensures`, il y a un cercle bleu dans la marge, ils indiquent qu'aucune vérification n'a été tentée pour ces lignes (voir figure 3.1).

Nous demandons la vérification que le code répond à la spécification en faisant un clic droit sur le nom de la fonction et « *Prove function annotations by WP* » (voir figure 3.2).

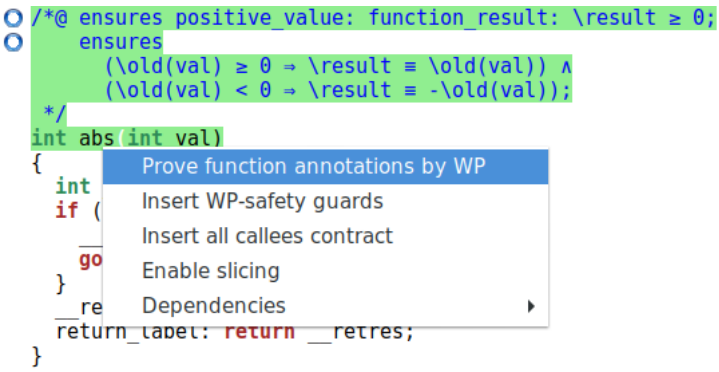


Figure 3.2 – Lancer la vérification de `abs` avec WP

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.

Mais le code est-il vraiment sans erreur pour autant ? 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 : RTE). C'est le rôle d'un autre petit plugin que nous allons utiliser ici et qui s'appelle sobrement RTE. 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 cochons la case montrée par cette capture (dans le volet de WP). Il est également possible de demander à Frama-C d'ajouter ces contrôles par un clic droit sur le nom de la fonction puis « Insert RTE guards » (voir figure 3.3).

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).

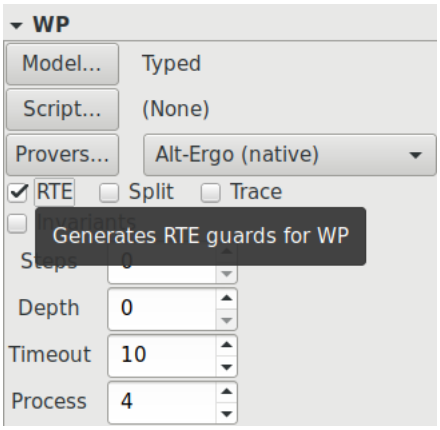


Figure 3.3 – Activer la vérification des erreurs d'exécution

Nous pouvons alors voir 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` ($2^{31} - 1$) (voir figure 3.4).

```
/*@ 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;
}
```

Figure 3.4 – Preuve incomplète de `abs`

Information

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 vert + marron nous indique que la preuve a été effectué mais qu'elle dépend potentiellement de propriétés qui, elle, ne l'ont pas été.

Si la preuve n'est pas 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 de runtime-error, et ne se sont donc pas reposées sur la connaissance de cette propriété puisqu'elle n'existait pas.

En effet, lorsque WP transmet une obligation de preuve à un prouveur automatique, il transmet (basiquement) 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 à condition seulement que l'on arrive un jour à prouver S_3.

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* » (voir figure 3.5), celui-ci nous affiche la liste des obligations de preuve 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* (ici nous pouvons voir un essai avec Z3 sur le contrôle de la RTE pour montrer le logo des ciseaux associé au timeout).

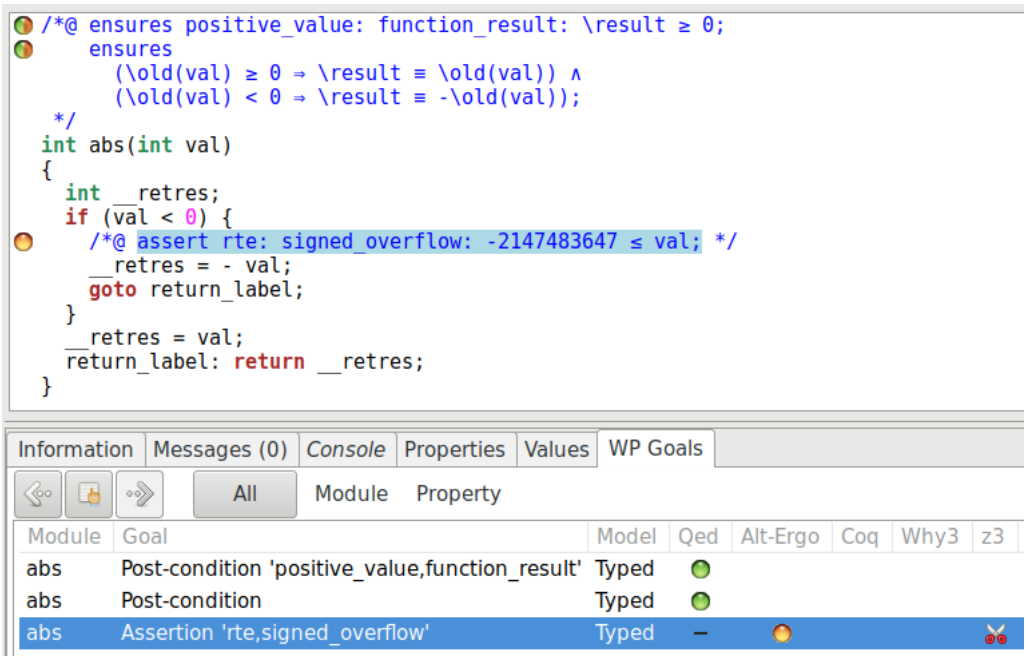


Figure 3.5 – Tableau des obligations de preuve de WP pour `abs`

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 post-condition nommée est estampillée "Post-condition 'positive_value,function_result' ", nous pouvons d'ailleurs noter que lorsqu'une propriété est sélectionnée dans le tableau, elle est également sur-lignée dans le code source. Les propriétés non-nommées 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. En fait, si nous double-cliquons sur la propriété "ne pas déborder" (surlignée en bleu dans la capture précédente), nous pouvons voir l'obligation correspondante (voir figure 3.6).

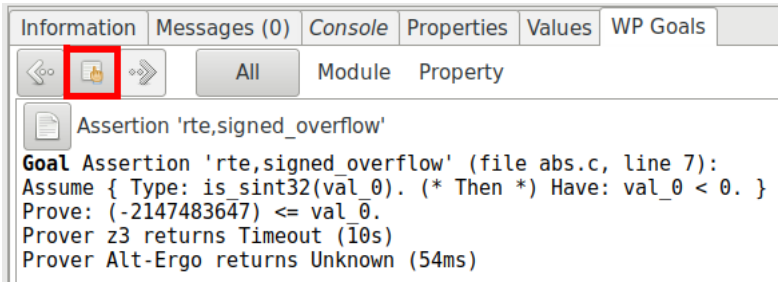


Figure 3.6 – Obligation de preuve associée à la vérification de débordement dans `abs`

C'est l'obligation de preuve que génère WP par rapport à notre propriété et notre programme, il n'est pas nécessaire de comprendre tout ce qui 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 ne sont généralement pas capables 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 obligations de preuve 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 obligations de preuve 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 obligations de preuve (bouton encadré 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 pré-condition d'appel.

3.1.2 Pré-condition

Les pré-conditions de fonctions sont introduites par la clause `requires`, de la même manière qu'avec `ensures`, nous pouvons composer nos expressions logiques et mettre plusieurs

pré-conditions :

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

}
```

Les pré-conditions 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 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'arrivera pas. Nous l'ajoutons donc comme pré-condition (à noter : il faut également inclure le header 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;
}
```

Attention

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

Information

Avec les versions de Frama-C NEON et plus anciennes, le pré-processing des annotations n'était pas activé par défaut. Il faut donc lancer Frama-C avec l'option `-pp-annot` :

```
$ frama-c-gui -pp-annot file.c
```

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 (voir figure 3.7).

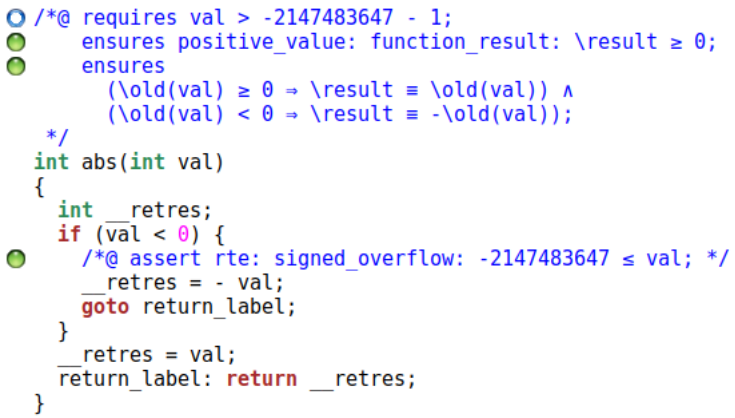


Figure 3.7 – Preuve de `abs` effectuée

Nous pouvons également vérifier qu'une fonction qui appellerait `abs` respecte bien la pré-condition qu'elle impose :

```
void foo(int a){
   int b = abs(42);
   int c = abs(-42);
   int d = abs(a);       // Faux : "a", vaut peut être INT_MIN
   int e = abs(INT_MIN); // Faux : le paramètre doit être strictement supérieur à INT_MIN
}
```

(Le résultat est dans la figure 3.8).

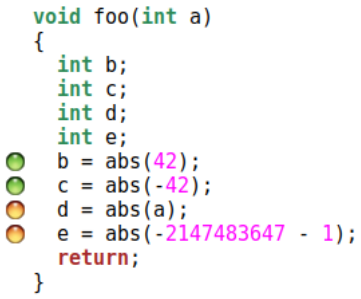


Figure 3.8 – Vérification du contrat à l'appel de `abs`

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 pré-conditions sont vérifiées et que le calcul termine alors la post-condition est vérifiée.

Si nous donnons à notre fonction une valeur qui viole ses pré-conditions, alors nous en déduisons que la post-condition est fausse. À partir de là, nous pouvons prouver tout ce que nous voulons car ce « false » 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 obligations de preuve générées par WP pour l'appel fautif et l'appel prouvé par conséquent (figures 3.9 et 3.10).

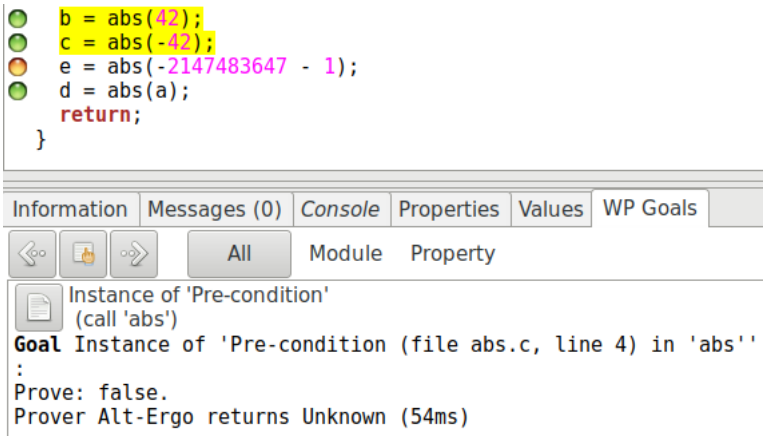


Figure 3.9 – Obligation générée pour l'appel fautif

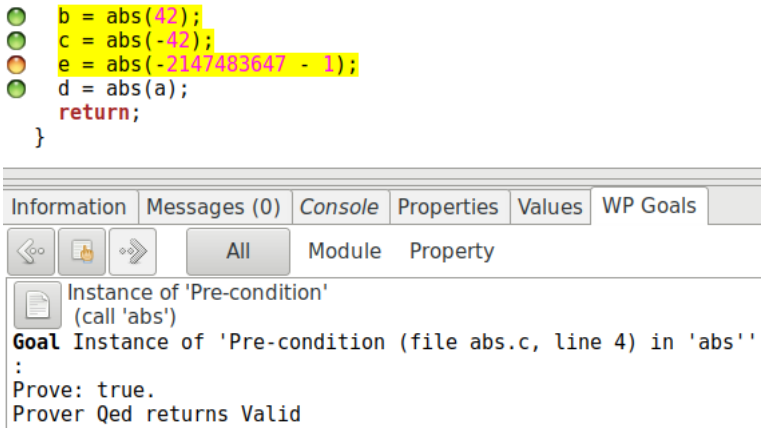


Figure 3.10 – Obligation générée pour l'appel qui suit

Nous pouvons remarquer que pour les appels de fonctions, l'interface graphique nous surligne le chemin d'exécution suivi avant l'appel dont nous cherchons à vérifier la pré-condition. Ensuite, si nous regardons l'appel `abs(INT_MIN)`, nous pouvons remarquer 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 pré-condition sur le deuxième ? La réponse est simplement que `abs(a)` peut, ou ne peut pas, provoquer une erreur, alors que `abs(INT_MIN)` provoque forcément l'erreur. 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)` qui peut aussi ne pas échouer.

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

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

}
```

Si nous demandons à WP de prouver cette fonction. Il l'acceptera sans rechigner car la supposition que nous lui donnons en entrée est nécessairement fausse. Par contre, nous aurons bien du mal à lui donner une valeur en entrée qui respecte la pré-condition, nous pourrons donc nous en apercevoir. En regardant pourquoi nous n'arrivons pas à transmettre une valeur valide en entrée.

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.

3.1.2.1 Trouver les bonnes pré-conditions

Trouver les bonnes pré-conditions à une fonction est parfois difficile. Le plus important est avant tout de déterminer ces pré-conditions sans prendre en compte le contenu de la fonction (au moins

dans un premier temps) afin d'éviter de construire, par erreur, une spécification qui contiendrait le même bug qu'un code fautif, par exemple en prenant en compte une condition faussée. 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 ces pré-conditions posées, alors seulement, nous nous intéressons aux spécifications dues au fait que nous sommes soumis aux contraintes de notre langage et notre matériel. Par exemple, la fonction valeur absolue n'a, au fond, pas vraiment de pré-condition à respecter, c'est la machine cible qui détermine qu'une condition supplémentaire doit être respectée en raison du complément à deux.

3.1.3 Quelques éléments sur l'usage de WP et Frama-C

Dans les deux sous-sections précédentes, nous avons vu un certain nombre d'éléments à propos de l'usage de la GUI pour lancer les preuves. En fait, il est possible de demander immédiatement à WP d'effectuer les preuves pendant le lancement de Frama-C avec la commande :

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

Cela demande à WP d'immédiatement faire les calculs de plus faible pré-condition et de lancer les prouveurs sur les buts générés.

Concernant les contrôles des RTE, il est généralement conseillé de commencer par vérifier le programme sans mettre les contrôles de RTE. Et ensuite seulement de générer les assertions correspondantes pour terminer la vérification avec WP. Cela permet à WP de se « concentrer » dans un premier temps sur les propriétés fonctionnelles sans avoir la connaissance de propriétés purement techniques dues à C, qui chargent inutilement la base de connaissances. Une nouvelle fois, il est possible de produire ce comportement directement depuis la ligne de commande en écrivant :

```
$ frama-c-gui file.c -wp -then -rte -wp
```

« Lancer Frama-C avec WP, puis créer les assertions correspondant aux RTE, et lancer à nouveau WP ».

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 juste ? 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;
}
```

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;
}
```

3.2.2 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.

Prenons par exemple une fonction `swap` pour les entiers.

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

3.2.2.1 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 c'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 post-condition 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 :

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

  //@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, mais tous ne sont pas supportés par WP :

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

Information

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 post-conditions 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 post-conditions 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).

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

3.2.2.2 Validité de pointeurs

Si nous essayons de prouver le fonctionnement de `swap` (en activant la vérification des RTE), notre post-condition 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 :

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

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 que le pointeur peut être déréférencé mais en lecture seulement. 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 :

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

int const value = 42;

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

Le déréférencement de `p` est valide, pourtant la pré-condition 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`.

3.2.2.3 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) :

```
int h = 42; //nous ajoutons une variable globale

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

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

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

Le résultat n'est pas vraiment celui escompté, comme nous pouvons le voir dans la figure 3.11.

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

Figure 3.11 – Échec de preuve sur une globale non concernée par l'appel à `swap`

En effet, nous n'avons pas spécifié les effets de bords autorisés pour notre fonction. Pour spécifier les effets de bords, nous utilisons la clause `assigns` qui fait partie des post-conditions 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 la fonction `swap` :

```
/*@
  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 bords. Ce cas est précisé en donnant `\nothing` à `assigns` :

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

  assigns  \nothing;

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

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

3.2.2.4 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, nous obtenons le résultat présenté par la figure 3.12.

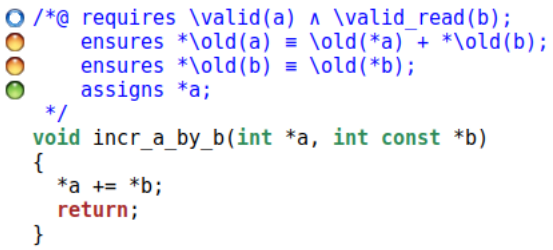


Figure 3.12 – Échec de preuve : risque d'aliasing

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 assign 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 va nous assurer qu'ils sont deux à deux disjoints. Ici, nous spécifierions :

```
#include <limits.h>

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

Et cette fois, la preuve est effectuée, comme le montre la figure 3.13.

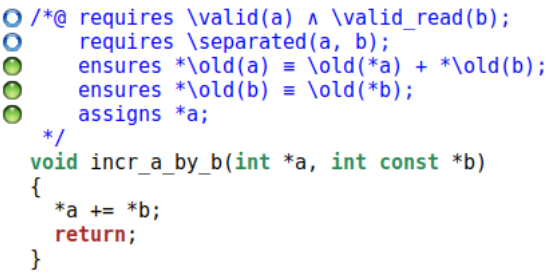


Figure 3.13 – Résolution des problèmes d'aliasing

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 pré-conditions qui nous garantissent l'absence de débordement en conséquence.

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 allons la reprendre 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 post-conditions. Nous les introduisons avec le mot-clé `behavior`. Chaque comportement se voit attribué :

- un nom ;
- les suppositions du cas que nous traitons, introduites par le mot clé `assumes` ;
- la post-condition associée à ce comportement.

Finalement, nous pouvons également demander à WP de vérifier le fait que les comportements sont disjoints (pour garantir le déterminisme) et complets.

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

Par exemple pour `abs` :

```
/*@
  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){
  if(val < 0) return -val;
  return val;
}
```

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

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

Attention

Même si `assigns` est une post-condition, 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 post-condition 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.

3.4 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 allons en profiter 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 nos précédentes fonctions en couples headers/source pour `abs` et `max`. Cela donne pour `abs` les fichiers présentés en figures 3.14 et 3.15.

Nous découpons en mettant le contrat de la fonction dans le header. Le but de ceci est de pouvoir, lorsque nous aurons besoin de la fonction dans un autre fichier, importer la spécification en même temps que la déclaration de celle-ci. En effet, WP en aura besoin pour montrer que les appels à cette fonction sont valides.

Nous pouvons créer un fichier sous le même formatage pour la fonction `max`. Dans les deux cas, nous pouvons ré-ouvrir 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.

```
#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
```

Figure 3.14 – Header pour `abs`

```
#include "abs.h"

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

Figure 3.15 – Sources pour `abs`

```c
#ifndef _MAX_ABS
#define _MAX_ABS

int max_abs(int a, int b);

#endif
```

Figure 3.16 – Header pour `max_abs`

```c
#include "max_abs.h"
#include "max.h"
#include "abs.h"

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

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

Figure 3.17 – Sources pour `max_abs`

Maintenant, nous pouvons préparer le terrain pour la fonction `max_abs` avec le header présenté en figure 3.16 accompagné de son fichier source en figure 3.17.

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), comme illustré par la figure 3.18.

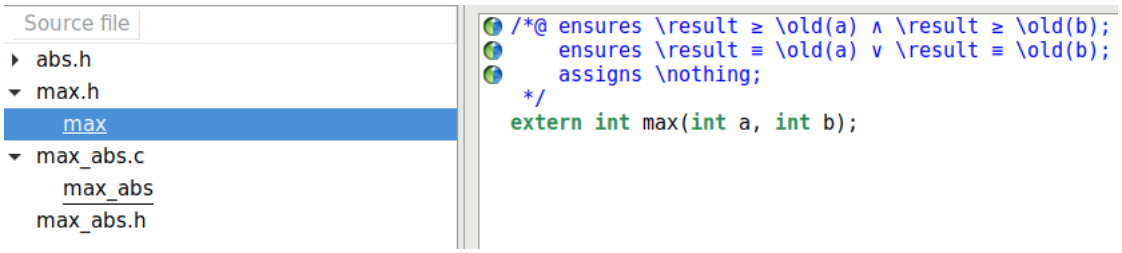


Figure 3.18 – Le contrat de `max` est valide par hypothèse

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 pré-conditions 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 (j'ai mis l'implémentation avec pour rappel) :

```
/*@
  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 abs_max(int a, int b){
  int abs_a = abs(a);
  int abs_b = abs(b);

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

Pendant cette partie, nous avons vu comment spécifier les fonctions par l'intermédiaire de leurs contrats, à savoir leurs pré et post-conditions, 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 de ne pas surcharger l'écriture pour autant.

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

Information

Cette partie est plus formelle que ce 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 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 pré-conditions 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 :

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

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 post-conditions 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 post-condition Q et d'une commande ou d'une liste de commandes C, la pré-condition minimale assurant Q après C. Nous pourrions donc, dans le programme précédent, calculer la propriété `P` qui nous donne les garanties voulues.

Nous allons tout au long de cette partie présenter les différents cas de la fonction wp qui, à une post-condition voulue et un programme ou une instruction, nous associe la plus faible pré-condition qui permet de l'assurer. Nous utiliserons cette notation pour définir le calcul correspondant à une/des instructions :

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

Et 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 :

```
//@ 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 (j'ai omis `P` car sa valeur est en fait simplement « vrai ») :

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

  return res;
}
```

4.1 Affectation, séquence et conditionnelle

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 associé 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 post-condition $Post$ où x a été remplacé par E ». Dans l'idée, pour que la formule en post-condition d'une affectation de x à E soit vraie, il faut qu'elle soit vraie en remplaçant chaque occurrence de x dans la formule par E. 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 pré-condition 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 pré-condition de l'affectation, nous avons remplacé chaque occurrence de x dans la post-condition, 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 pré-condition :

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

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 pré-condition est respectée par le programme où se trouve l'instruction, ni que cette pré-condition 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.

Information

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 toujous 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.

4.1.2 Séquence d'instructions

Pour qu'une instruction soit valide, il faut que sa pré-condition nous permette, par cette instruction, de passer à la post-condition voulue. Maintenant, nous avons besoin d'enchaîner ce processus d'une instruction à une autre. L'idée est alors que la post-condition assurée par la première instruction soit compatible avec la pré-condition 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 pré-condition de $S2$ et la post-condition de $S1$. Cependant, rien ne nous indique pour l'instant comment obtenir les propriétés P et R.

Le calcul de plus faible pré-condition wp nous dit simplement que la propriété intermédiaire R est trouvée par calcul de plus faible pré-condition de la deuxième instruction. Et que la propriété P est trouvée en calculant la plus faible pré-condition de la première instruction. La plus faible pré-condition 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.

```
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 précède forment la première « instruction ». De cette manière nous remontons bien les instructions une à une dans notre raisonnement, par exemple avec le programme précédent :

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

Nous pouvons par calcul de plus faibles pré-conditions 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 post-condition soit atteignable par les deux banches, depuis la même pré-condition, 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 $if\ B\ then\ S1\ else\ S2$) :

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

Nos deux prémisses sont donc la vérification que lorsque nous avons la pré-condition et que nous passons dans la branche `if`, nous atteignons bien la post-condition, et que lorsque nous avons la pré-condition et que nous passons dans la branche `else`, nous obtenons bien également la post-condition.

Le calcul de pré-condition 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 pré-condition la plus faible de $S1$, pour pouvoir l'exécuter sans erreur vers la post-condition, et que $\neg B$ doit impliquer la pré-condition 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 post-condition 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 :

```
int c;

// ... du code ...

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

4.2 [Bonus Stage] Conséquence et constance

4.2.1 Règle de conséquence

Parfois, il peut être utile pour la preuve de renforcer une post-condition ou d'affaiblir une pré-condition. 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'issu du calcul de plus faible pré-condition.

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

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

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

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

Inversement, le calcul de pré-condition générera généralement une formule compliquée et souvent plus faible que la pré-condition 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 conséquences.

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

  int res = a * 10;

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

  return res;
}
```

(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 post-condition avec une assertion que `res`, le résultat, est compris entre 0 et 90 à la fin. Le calcul de plus faible pré-condition, sur cette propriété, et avec l'affectation `res = 10*a` nous produit une plus faible pré-condition `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 post-conditions 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.

4.2.2 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 que nous 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 concerne 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'informations, 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) :

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

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 :

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

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 :

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

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 strict 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 $Q \Rightarrow Q \wedge R$.

4.3 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 ?

4.3.1 Induction et invariance

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

Nous allons donc procéder 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).

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. Par exemple, pour la boucle :

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

La propriété $0 <= i <= 10$ est un invariant de la boucle. La propriété $-42 <= i <= 42$ est également un invariant de la boucle (qui est nettement plus imprécis néanmoins). La propriété $0 < i <= 10$ n'est pas un invariant, elle n'est pas vraie à l'entrée de la boucle. La propriété $0 <= i < 10$ **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 I est vrai avant de commencer un tour, auquel cas I est vrai après (préservation).

4.3.1.1 [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 pré-condition 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 « pré-condition » 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 pré-condition que nous connaissons du corps de la boucle (notons KWP, « *Known WP* ») , c'est ($KWP = B \wedge I$). Soit le fait que nous sommes rentrés dedans (B est vrai), et que l'invariant est respecté à ce moment (I, qui est vrai avant de commencer la boucle par 1, et dont veut vérifier qu'il sera vraie en fin de bloc de la boucle (2)),
  — (2) ce qu'il nous reste à vérifier c'est que KWP implique la pré-condition 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 post-condition 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.

4.3.1.2 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 pré-condition 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 :

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

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

Attention

RAPPEL : L'invariant est bien : i <= 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 pré-conditions sur les boucles est que lorsque l'on invalide la condition de la boucle, par la connaissance de l'invariant, on peut prouver la post-condition (Formellement : $(\neg B \wedge I) \Rightarrow Post$).

La post-condition de notre boucle est `i == 30` et doit être impliquée par ¬ `i < 30` ∧ `0 <= i <= 30`. Ici, cela fonctionne bien : `i >= 30 && 0 <= i <= 30 ==> i == 30`. Si l'invariant excluait l'égalité à 30, la post-condition ne serait pas atteignable.

Une nouvelle fois, nous pouvons jeter un œil à la liste des buts dans « *WP Goals* » (montrée dans la figure 4.1).

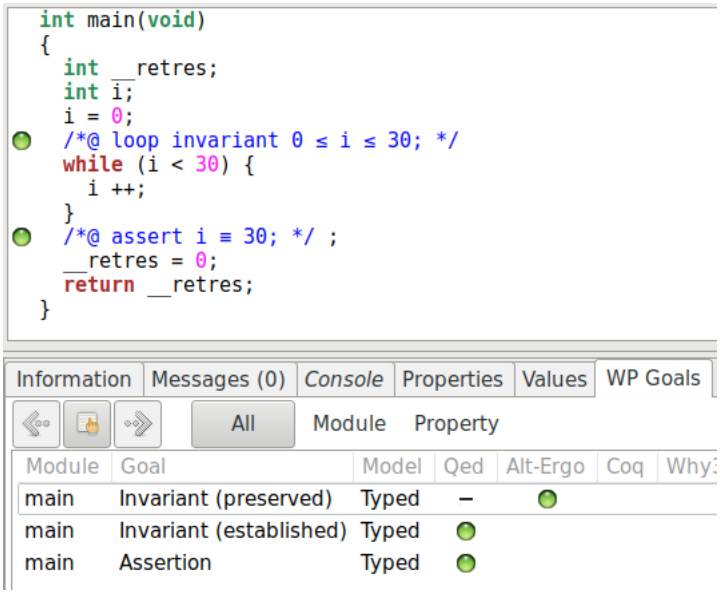


Figure 4.1 – Obligations générées pour prouver notre boucle

Nous remarquons bien 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 l'obligation de preuve générée ne le montre pas (affichant simplement « *True* »). En utilisant l'option `-wp-no-simpl` au lancement, nous pouvons quand même voir l'obligation correspondante (montrée dans la figure 4.2).

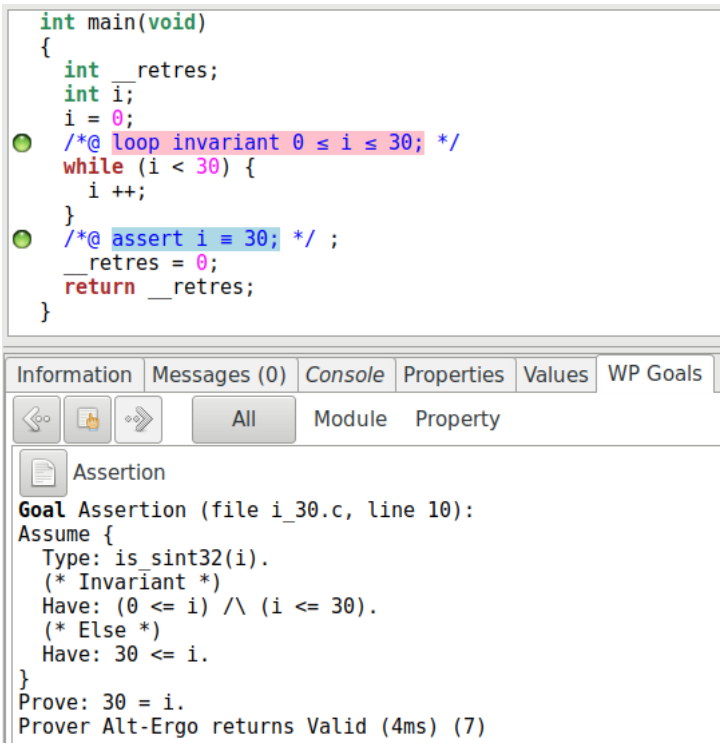


Figure 4.2 – Preuve de l'assertion par connaissance de l'invariant et l'invalidation de la condition de boucle

Mais notre spécification est-elle suffisante ?

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

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

Il semble que non (voir figure 4.3).

4.3.2 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

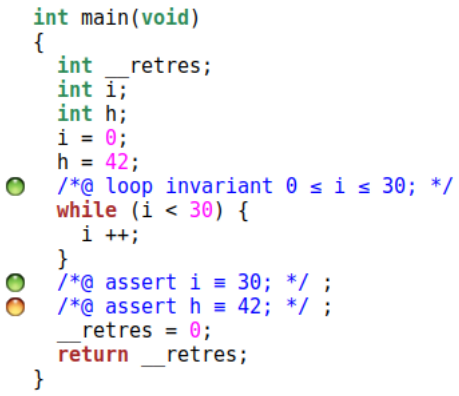


Figure 4.3 – Encore des effets de bord

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 :

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

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

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 :

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

}
```

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

4.3.3 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 pré-condition est validée et **si** le calcul termine, alors la post-condition est validée ». Dans le second cas, « si la pré-condition est validée, alors

le calcul termine et la post-condition est validée ». WP s'intéresse par défaut à de la preuve de correction partielle :

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

Si nous demandons la vérification de ce code en activant le contrôle de RTE, nous obtenons le résultat présenté en figure 4.4.

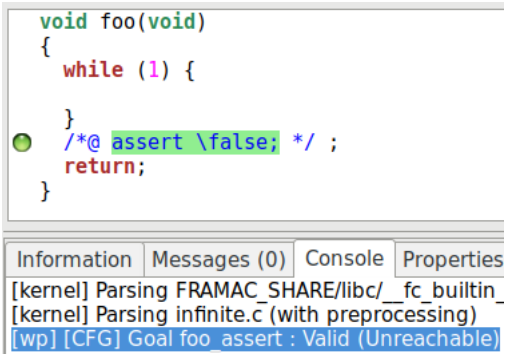
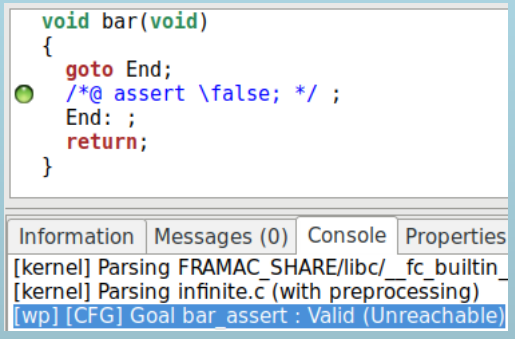


Figure 4.4 – Preuve de faux par non-terminaison de boucle

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.

Information

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



Et c'est également le cas lorsque l'on sait trivialement qu'une instruction produit nécessairement 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`.

Pour prouver la terminaison d'une boucle, nous utilisons la notion de variant de boucle. Le variant de boucle n'est pas une propriété mais une valeur. C'est 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 (cela sera également vérifié par induction par WP).

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

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

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

Une nouvelle fois nous pouvons regarder les buts générés (voir figure 4.5).

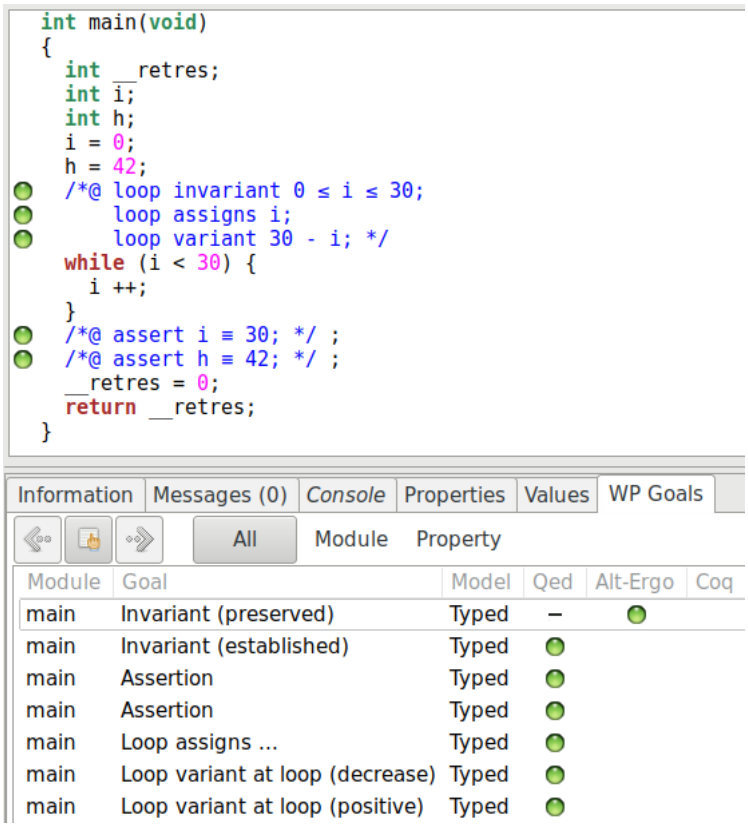


Figure 4.5 – Notre simple boucle complètement spécifiée et prouvée

Le variant nous génère bien deux obligations au niveau de la vérification : assurer que la valeur est positive, et assurer qu'elle décroît strictement pendant l'exécution de la boucle. Et si nous supprimons la ligne de code qui incrémente `i`, WP ne peut plus prouver que la valeur `30 - i` décroît strictement.

Il est également bon de noter qu'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 undetermined_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.

Le variant n'est donc bien qu'une borne supérieure sur le nombre d'itérations de la boucle.

4.3.4 Lier la post-condition 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.

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

En remontant les instructions depuis la post-condition, 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 `plus_dix` ne peut pas être prouvée en l'état : l'invariant ne mentionne rien à propos de `a`. Pour lier notre post-condition à l'invariant, il faut ajouter une telle information. Par exemple :

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

    return a;
}
```

Remarque

Ce besoin peut semble être une contrainte très forte. Elle 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 spécifications, 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 différentes qui évoluent dans le même programme, par exemple l'égalité mentionnée par notre invariant ajouté.

4.4 Les boucles - Exemples

4.4.1 Exemple avec un tableau read-only

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 vont nous permettre d'introduire des constructions très importantes d'ACSL.

Prenons par exemple la fonction qui cherche une valeur dans un tableau :

```
#include <stddef.h>

/*@
  requires 0 < length;
  requires \valid_read(array + (0 .. length-1));

  assigns  \nothing;

  behavior in:
    assumes \exists size_t off ; 0 <= off < length && array[off] == element;
    ensures array <= \result < array+length && *\result == element;

  behavior notin:
    assumes \forall size_t off ; 0 <= off < length ==> array[off] != element;
    ensures \result == NULL;

  disjoint behaviors;
  complete behaviors;
*/
int* search(int* array, size_t length, int element){
  /*@
    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];
  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 :

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

Cette pré-condition 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 :

```
/*@
//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 respecte une première propriété (`p`), alors 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 respecte 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 respectant 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.

Cette partie de l'invariant mérite une attention particulière :

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

En effet, c'est la partie qui définit l'action de notre boucle, elle indique à WP ce que la boucle va faire (ou apprendre 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 (voir figure 4.6).

Nous pouvons constater 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 pouvons voir 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;
```

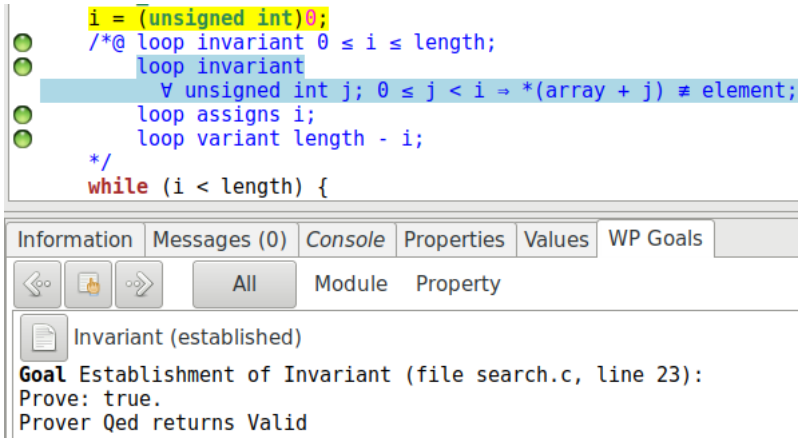


Figure 4.6 – But trivial

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

4.4.2 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.

4.4.2.1 Remise à zéro

Regardons la fonction effectuant la remise à zéro d'un tableau (voir figure 4.7).

```
#include <stddef.h>

/*@
  requires \valid(array + (0 .. length-1));
  assigns  array[0 .. length-1];
  ensures  \forall size_t i; 0 <= i < length ==> array[i] == 0;
*/
void raz(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;
}
```

Figure 4.7 – Fonction de remise à zéro d'un tableau

Les seules parties sur lesquelles nous pouvons nous attacher ici sont les `assigns` de la fonction et de la boucle. À nouveau, nous pouvons utiliser la notation `n .. m` pour indiquer les parties du tableau qui sont modifiées.

4.4.2.2 Chercher et remplacer

Le dernier exemple qui nous intéresse est l'algorithme « chercher et remplacer ». C'est donc un algorithme qui va sélectivement modifier 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 que nous pouvons réaliser pour cette fonction est présentée dans la figure 4.8.

```
#include <stddef.h>

/*@
  requires \valid(array + (0 .. length-1));
  assigns array[0 .. length-1];

  ensures \forall size_t i; 0 <= i < length && \old(array[i]) == old
             ==> array[i] == new;
  ensures \forall size_t i; 0 <= i < length && \old(array[i]) != old
             ==> array[i] == \old(array[i]);
*/
void search_and_replace(int* array, size_t length, int old, int new){
  /*@
    loop invariant 0 <= i <= length;
    loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) == old
                     ==> array[j] == new;
    loop invariant \forall size_t j; 0 <= j < i && \at(array[j], Pre) != old
                     ==> array[j] == \at(array[j], Pre);
    loop assigns i, array[0 .. length-1];
    loop variant length-i;
  */
  for(size_t i = 0; i < length; ++i){
    if(array[i] == old) array[i] = new;
  }
}
```

Figure 4.8 – Fonction chercher et remplacer dans un tableau

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 :

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

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. En fait, si nous essayons de prouver l'invariant, WP n'y parvient pas. Dans ce genre de cas, le plus simple est encore d'ajouter diverses assertions exprimant les propriétés intermédiaires que nous nous attendons à voir facilement prouvées et impliquant

l'invariant. En fait, nous nous apercevons rapidement que WP n'arrive pas à maintenir le fait que nous n'avons pas encore modifié la fin du tableau :

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

Nous pouvons donc ajouter cette information comme invariant :

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

  //La fin du tableau n'a pas été modifiée :
  loop invariant \forall size_t j; i <= j < length
                    ==> 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. À noter que si nous tentons la preuve directement avec la vérification des RTE, il est possible qu'Alt-Ergo n'y parvienne pas (CVC4 décharge l'ensemble sans problème). Dans ce cas, nous pouvons faire séparément les deux preuves (sans, puis avec RTE) ou encore ajouter des assertions permettant de guider la preuve dans la boucle :

```
for(size_t i = 0; i < length; ++i){
  if(array[i] == old) array[i] = new;

  /*@ assert \forall size_t j; i < j < length
              ==> array[j] == \at(array[j], Pre);                    */
  /*@ assert \forall size_t j; 0 <= j <= i && \at(array[j], Pre) == old
              ==> array[j] == new;                                   */
  /*@ assert \forall size_t j; 0 <= j <= i && \at(array[j], Pre) != old
              ==> array[j] == \at(array[j], Pre);                    */
}
```

À mesure que nous cherchons à prouver des propriétés plus compliquées et notamment dépendantes de boucles, il va y avoir une part de tâtonnement pour comprendre ce qui manque au prouveur pour réussir la preuve.

Ce qui peut lui manquer, ce sont des hypothèses. Dans ce type de cas, nous pouvons tenter d'ajouter des assertions au code pour guider le prouveur. Avec de l'expérience, nous pouvons regarder le contenu des obligations de preuve ou tenter de commencer la preuve avec Coq pour voir si la preuve semble réalisable. Parfois le prouveur manque juste de temps, auquel cas, il suffit d'augmenter (parfois de beaucoup) la durée du *timeout*. Finalement, la propriété peut également être hors de portée du prouveur. Auquel cas, il faudra écrire une preuve à la main avec un prouveur interactif.

Enfin, il reste le cas où l'implémentation est effectivement fausse, et dans ce cas, il faut la corriger. Et c'est là que nous utiliserons plutôt le test que la preuve, car le test permet de prouver la présence d'un bug. ;)

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 allons nous attarder 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 n'allons pas les présenter 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 nous 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 pour nous : 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 nous propose deux types qui vont nous permettre d'écrire des propriétés ou des fonctions sans avoir à se préoccuper 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 propriété 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 cours d'écriture en parle : effectuer des calculs numériques précis.

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 nous permet de créer des versions nommées de ces prédicats, à la manière d'une fonction booléenne en C par exemple. À la différence près tout de même 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 bords 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 vont permettre d'établir des relations entre divers points du code.

5.2.1 Syntaxe

Les prédicats sont, comme les spécifications, introduits au travers d'annotations. La syntaxe est la suivante :

```
/*@
  predicate nom_predicat { Lbl0, Lbl1, ..., LblN }(type0 arg0, type1 arg1, ..., typeN argN) =
    //une relation logique entre toutes ces choses.
*/
```

Nous pouvons par exemple définir le prédicat nous disant qu'un entier en mémoire n'a pas changé entre deux points particuliers du programme :

```
/*@
  predicate unchanged{L0, L1}(int* i) =
    \at(*i, L0) == \at(*i, L1);
*/
```

Attention

Gardez bien en mémoire que le passage se fait, comme en C, par valeur. Nous ne pouvons pas écrire ce prédicat en passant directement `i` :

```
/*@
  predicate unchanged{L0, L1}(int i) =
    \at(i, L0) == \at(i, L1);
*/
```

Car i est juste une copie de la variable reçue en paramètre.

Nous pouvons par exemple vérifier ce petit code :

```
int main(){
  int i = 13;
  int j = 37;

 Begin:
  i = 23;

  //@assert ! unchanged{Begin, Here}(&i);
  //@assert   unchanged{Begin, Here}(&j);
}
```

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 d'écrire 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.

```
/*@
  predicate valid_range_rw(int* t, integer n) =
    n >= 0 && \valid(t + (0 .. n-1));

  predicate valid_range_ro(int* t, integer n) =
    n >= 0 && \valid_read(t + (0 .. n-1));
*/

/*@
  requires 0 < length;
  requires valid_range_ro(array, length);
  //...
*/
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 va automatiquement en ajouter 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.

5.2.2 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 respecter 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 de spécifications. Par exemple, nous pourrions poser les spécifications nécessaires à la sûreté d'une pile de taille limitée. Et cela donnerait quelque chose comme :

```
struct stack_int{
  size_t top;
  int    data[MAX_SIZE];
};

/*@
  predicate valid_stack_int(struct stack_int* s) = // à définir ;
  predicate empty_stack_int(struct stack_int* s) = // à définir ;
  predicate full_stack_int(struct stack_int* s) =  // à définir ;
*/

/*@
  requires \valid(s);
  assigns *s;
  ensures valid_stack_int(s) && empty_stack_int(s);
*/
void initialize(struct stack_int* s);

/*@
  requires valid_stack_int(s) && !full_stack_int(s);
  assigns  *s;
```

```
  ensures valid_stack_int(s);
*/
void push(struct stack_int* s, int value);

/*@
  requires valid_stack_int(s) && !empty_stack_int(s);
  assigns \nothing;
*/
int  top(struct stack_int* s);

/*@
  requires valid_stack_int(s) && !empty_stack_int(s);
  assigns *s;
  ensures valid_stack_int(s);
*/
void pop(struct stack_int* s);

/*@
  requires valid_stack_int(s);
  assigns \nothing;
  ensures \result == 1 <==> empty_stack_int(s);
*/
int  is_empty(stack_int_t s);


/*@
  requires valid_stack_int(s);
  assigns \nothing;
  ensures \result == 1 <==> full_stack_int(s);
*/
int  is_full(stack_int_t s);
```

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 que nous 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).

5.3 Fonctions logiques

Les fonctions logiques nous permettent de décrire des fonctions qui 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;
}
```

Figure 5.1 – Les débordements semblent pouvoir survenir

Le code est bien prouvé mais les contrôles d'*overflow*, eux, ne le sont pas (voir figure 5.1). Nous pouvons à nouveau définir des fonctions logiques générales, qui vont, du point de vue de la logique, nous fournir les bornes en fonction des valeurs que nous donnons en entrée. Cela nous permet ensuite d'ajouter nos contrôles de bornes en pré-condition de fonction :

```
/*@
  logic integer limit_int_min_ax_b(integer a, integer b) =
    (a == 0) ? (b > 0) ? INT_MIN : INT_MIN-b :
    (a <  0) ? (INT_MAX-b)/a :
               (INT_MIN-b)/a ;

  logic integer limit_int_max_ax_b(integer a, integer b) =
    (a == 0) ? (b > 0) ? INT_MAX-b : INT_MAX :
    (a <  0) ? (INT_MIN-b)/a :
               (INT_MAX-b)/a ;
*/
```

```
/*@
  requires limit_int_min_ax_b(3,4) < x < limit_int_max_ax_b(3,4);
  assigns \nothing ;
  ensures \result == ax_b(3,x,4);
*/
int function(int x){
  return 3*x + 4;
}
```

Information

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 avec les calculs de `INT_MIN-b` ou `INT_MAX-b`.

Et cette fois tout est prouvé. Évidemment, nous pourrions fixer ces valeurs en dur chaque fois que nous avons besoin d'une nouvelle fonction affine du côté de la logique mais en posant ces fonctions, nous obtenons directement ces valeurs sans avoir besoin de les calculer nous même, ce qui est assez confortable.

5.3.2 Récursivité et limites

Les fonctions logiques peuvent être définie récursivement. Cependant, une telle définition va très rapidement montrer 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, 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 :

```
/*@
  logic integer factorial(integer n) = (n <= 0) ? 1 : n * factorial(n-1);
*/

/*@
  assigns \nothing ;
  ensures \result == factorial(n) ;
*/
unsigned facto(unsigned n){
  return (n == 0) ? 1 : n * facto(n-1);
}
```

Sans contrôle de borne, cette fonction se prouve rapidement. Si nous ajoutons le contrôle des RTE, le vérification de débordement sur l'entier non-signé n'est pas ajoutée, car c'est un comportement déterminé d'après la norme C. Pour ajouter une assertion à ce point, nous pouvons demander à WP de générer ses propres vérifications en faisant un clic droit sur la fonction puis « insert WP-safety guards ». Et dans ce cas, le non-débordemement n'est pas prouvé.

Sur le type `unsigned`, le maximum que nous pouvons calculer est la factorielle de 12. Au-delà, cela produit un dépassement. Nous pouvons donc ajouter cette pré-condition :

```
/*@
  requires n <= 12 ;
  assigns \nothing ;
  ensures \result == factorial(n) ;
*/
unsigned facto(unsigned n){
  return (n == 0) ? 1 : n * facto(n-1);
}
```

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. Nous pouvons par exemple changer la valeur maximale de `n` pour voir comment se comporte les différents prouveurs. Avec un `n` maximal fixé à 9, Alt-ergo produit la preuve en moins de 10 secondes, tandis que pour une valeur à 10, même une minute ne suffit pas.

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 fait, ce qui limite nos possibilités de preuve.

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 une fois et la propriété en question pourra être utilisée par les prouveurs, leur permettant ainsi de ne pas reproduire les étapes de preuve nécessaires à chaque fois qu'une propriété équivalente intervient dans une preuve plus longue sur une propriété plus précise.

Les lemmes peuvent par exemple nous permettre d'exprimer des propriétés à propos des fonctions récursives pour que les preuves les faisant intervenir nécessitent moins de travail pour les prouveurs.

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 a > 0;
  requires limit_int_min_ax_b(a,4) < x < limit_int_max_ax_b(a,4);
  assigns \nothing ;
  ensures \result == ax_b(a,x,4);
*/
int function(int a, int x){
  return a*x + 4;
}

/*@
  requires a > 0;
  requires limit_int_min_ax_b(a,4) < x < limit_int_max_ax_b(a,4) ;
  requires limit_int_min_ax_b(a,4) < y < limit_int_max_ax_b(a,4) ;
  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.

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 tous cas, si la spécification n'utilise que ce type de constructions et que chaque lemme, chaque pré-condition (aux points d'appels), chaque post-condition, 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 allons voir vont nous ajouter 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

Dans cette partie nous allons voir deux notions importantes d'ACSL :

— les définitions axiomatiques,
— le code fantôme.

Dans certaines configurations, ces deux 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 deux notions est qu'elles peuvent rendre notre preuve inutile si nous faisons une erreur dans leur usage. La première en nous autorisant à introduire des hypothèses fausses ou des définitions trop imprécises. La seconde en nous ouvrant le risque de modifier le programme vérifié, nous faisant ainsi prouver un autre programme que celui que nous voulons prouver.

6.1 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 vont pouvoir 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.

6.1.1 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 (voir figure 6.1).

```
/*@
axiomatic lt_plus_lt {
  axiom always_true_lt_plus_lt: ∀ ℤ i, ℤ j; i < j ⇒ i + 1 < j + 1;

  }

*/
```

Figure 6.1 – Premier axiome, supposé vrai par Frama-C

Hors sujet

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 le statuer comme axiome ! Reste à comprendre la question pour savoir où ce résultat peut-être utile ...

```
/*@
  axiomatic Ax_answer_to_the_ultimate_question_of_life_the_universe_and_everything {
    logic integer the_ultimate_question_of_life_the_universe_and_everything{L} ;

    axiom answer{L}:
      the_ultimate_question_of_life_the_universe_and_everything{L} = 42;

*/
```

6.1.1.1 Lien avec la notion de lemme

Les lemmes et les axiomes vont nous permettre 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.

6.1.2 Définition de fonctions ou prédicats récursifs

Les définitions axiomatiques de fonctions ou de prédicats récursifs sont particulièrement utiles car elles vont permettre 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) ;
  }
*/
```

Dans cette définition axiomatique, notre fonction n'a pas de corps. Son comportement étant défini par les axiomes ensuite définis.

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.

Si par exemple, nous voulons définir qu'un tableau ne contient que des 0, nous pouvons le faire de cette façon :

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

Et nous pouvons à nouveau prouver notre fonction de remise à zéro avec cette nouvelle définition :

```
#include <stddef.h>

/*@
  requires \valid(array + (0 .. length-1));
  assigns  array[0 .. length-1];
  ensures  zeroed(array,0,length);
*/
void raz(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 pas induction. Pour les curieux, la (très simple) preuve en Coq est ci-dessous :

Preuve Coq

```
(* Généré par WP *)
Definition P_same_elems (Mint_0 : farray addr Z) (Mint_1 : farray addr Z)
    (a : addr) (b : Z) (e : Z) : Prop :=
    forall (i : Z), let a_1 := (shift_sint32 a i%Z) in ((b <= i)%Z) ->
      ((i < e)%Z) -> (((Mint_0.[ a_1 ]) = (Mint_1.[ a_1 ]))%Z).
Goal
  forall (i_1 i : Z), forall (t_1 t : farray addr Z), forall (a : addr),
  ((P_zeroed t a i_1%Z i%Z)) ->
    ((P_same_elems t_1 t a i_1%Z i%Z)) ->
      ((P_zeroed t_1 a i_1%Z i%Z)).
(* Notre preuve *)
Proof.
  intros b e.
  (* par induction sur la distance entre b et e *)
  induction e using Z_induction with (m := b) ; intros mem_l2 mem_l1 a Hz_l1 Hsame.
  (* cas de base : Axiome "empty" *)
  + apply A_A_all_zeros.Q_zeroed_empty ; assumption.
  + replace (e + 1) with (1 + e) in * ; try omega.
    (* on utilise l'axiome range *)
    rewrite A_A_all_zeros.Q_zeroed_range in * ; intros Hinf.
    apply Hz_l1 in Hinf ; clear Hz_l1 ; inversion_clear Hinf as [Hlast Hothers].
    split.
    (* sous plage de Hsame *)
    - rewrite Hsame ; try assumption ; omega.
    (* hypothèse d'induction *)
    - apply IHe with (t := mem_l1) ; try assumption.
```

```
      * unfold P_same_elems ; intros ; apply Hsame ; omega.
Qed.
```

Dans le cas présent, utiliser une axiomatique 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 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(){
  // Exemples de propriétés prouvées

  //@ assert \false;
  //@ assert \forall integer x; x > x;
  //@ assert \forall integer x,y,z ; x == y == z == 42;
  return *(int*) 0;
}
```

Et tout est prouvé, y compris que le déréférencement de l'adresse 0 est valable (voir figure 6.2).

```
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;
}
```

Figure 6.2 – 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 connaitrait 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.

6.1.4 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;
*/
```

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.

6.1.5 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 ci-dessous et allons nous concentrer plutôt sur la spécification et la preuve de la fonction de tri qui sont une illustration intéressant de l'usage des axiomatiques.

Solution

```
/*@
  requires \valid_read(a + (beg .. end-1));
  requires beg < end;

  assigns  \nothing;

  ensures  \forall integer i; beg <= i < end ==> a[\result] <= a[i];
  ensures  beg <= \result < end;
*/
size_t min_idx_in(int* a, size_t beg, size_t end){
  size_t min_i = beg;

  /*@
    loop invariant beg <= min_i < i <= end;
    loop invariant \forall integer j; beg <= j < i ==> a[min_i] <= a[j];
    loop assigns min_i, i;
    loop variant end-i;
  */
  for(size_t i = beg+1; i < end; ++i){
    if(a[i] < a[min_i]) min_i = i;
  }
  return min_i;
}

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

En effet, une erreur commune que nous pourrions faire dans le cas de la preuve du tri est de poser la spécification (qui est vraie !) illustrée en figure 6.3.

Cette spécification est vraie. 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, la fonction de la figure 6.4 remplit pleinement la spécification.

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 axiomatique. En effet, pour déterminer qu'un tableau est la permutation d'un

```
/*@
  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){
  /*@ //annotation de l'invariant */
  for(size_t i = beg ; i < end ; ++i){
    size_t imin = min_idx_in(a, i, end);
    swap(&a[i], &a[imin]);
  }
}
```

Figure 6.3 – Spécification insuffisante de `sort`

```
/*@
  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;
}
```

Figure 6.4 – Fonction remplissant la spécification de la figure 6.3

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 que nous créons la permutation p_3 de p_2, alors par transitivité p_3 est une permutation de p_1.

Ceci est exprimé par le code suivant :

```
/*@
  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);

  axiomatic Permutation{
    predicate permutation{L1,L2}(int* a, integer b, integer e)
     reads \at(*(a+(b .. e - 1)), L1), \at(*(a+(b .. e - 1)), L2);

    axiom reflexive{L1}:
      \forall int* a, integer b,e ; permutation{L1,L1}(a, b, e);

    axiom 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);

    axiom 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 :

```
/*@
  requires beg < end && \valid(a + (beg .. end-1));
  assigns  a[beg .. end-1];
  ensures sorted(a, beg, end);
  ensures permutation{Pre, Post}(a,beg,end);
*/
void sort(int* a, size_t beg, size_t end){
  /*@
    loop invariant beg <= i <= end;
    loop invariant sorted(a, beg, i) && permutation{Begin, Here}(a, beg, end);
    loop invariant \forall integer j,k; beg <= j < i ==> i <= k < end ==> a[j] <= a[k];
    loop assigns i, a[beg .. end-1];
    loop variant end-i;
  */
  for(size_t i = beg ; i < end ; ++i){
    //@ ghost begin: ;
    size_t imin = min_idx_in(a, i, end);
    swap(&a[i], &a[imin]);
    //@ assert swap_in_array{begin,Here}(a,beg,end,i,imin);
  }
}
```

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 prochaine section.

6.2 Code Fantôme

Derrière ce titre faisant penser à un scénario de film d'action se cache en fait un moyen d'enrichir la spécification avec des informations sous la forme de code en langage C. Ici, l'idée va être d'ajouter des variables et du code source qui n'interviendra 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.

6.2.1 Syntaxe

Le code fantôme est ajouté par l'intermédiaire d'annotations qui vont contenir du code C ainsi que la mention `ghost`.

```
/*@
  ghost
  // code en langage C
*/
```

Les seules règles que nous devons respecter dans un tel code est que nous ne devons jamais écrire une portion de mémoire qui n'est pas elle-même définie dans du code fantôme et que le code en question doit terminer tout bloc qu'il ouvrirait. Mis à par cela, tout calcul peut être inséré tant qu'il ne modifie *que* les variables du code fantôme.

Voici quelques exemples pour la syntaxe de code fantôme :

```
//@ ghost int ghost_glob_var = 0;

void foo(int a){
  //@ ghost int ghost_loc_var = a;

  //@ ghost Ghost_label: ;
  a = 28 ;

  //@ ghost if(a < 0){ ghost_loc_var = 0; }

  //@ assert ghost_loc_var == \at(a, Ghost_label) == \at(a, Pre);
}
```

Il faut être très prudent lorsque nous produisons ce genre de code. En effet, aucune vérification n'est effectuée pour s'assurer que nous n'écrivons pas dans la mémoire globale par erreur. Ce problème étant comme la vérification elle-même, un problème indécidable, une telle analyse serait un travail de preuve à part entière. Par exemple, ce code est accepté en entrée de Frama-C, alors qu'il modifie manifestement l'état de la mémoire du programme :

```
int a;

void foo(){
  //@ ghost a = 42;
}
```

Il faut donc faire très attention à ce que nous faisons avec du code fantôme.

6.2.2 Expliciter un état logique

Le but du code *ghost* est de rendre explicite des informations généralement implicites. Par exemple, dans la section précédente, nous nous en sommes servi pour récupérer explicitement un état logique connu à un point de programme donné.

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 }`, ... Notez 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.

```
int max_subarray(int *a, size_t len) {
  int max = 0;
  int cur = 0;
  for(size_t i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0)   cur = 0;
    if (cur > max) max = cur;
  }
  return max;
}
```

Pour spécifier la fonction précédente, nous allons avoir 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 :

```
/*@ axiomatic Sum {
  logic integer sum(int *array, integer begin, integer end) reads a[begin..(end-1)];
}*/
```

Solution

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

La spécification de notre fonction est la suivante :

```
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;  0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;  0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
```

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 va nous manquer 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 post-condition 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 allons donc utiliser du code *ghost* pour conserver ces bornes et exprimer l'invariant de notre boucle.

Nous allons d'abord avoir besoin de 2 variables qui vont nous permettre 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. Le code spécifié est présenté dans la figure 6.5.

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 post-condition 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.

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.

```
/*@
  requires \valid(a+(0..len-1));
  assigns \nothing;
  ensures \forall integer l, h;  0 <= l <= h <= len ==> sum(a,l,h) <= \result;
  ensures \exists integer l, h;  0 <= l <= h <= len &&  sum(a,l,h) == \result;
*/
int max_subarray(int *a, size_t len) {
  int max = 0;
  int cur = 0;
  //@ ghost size_t cur_low = 0;
  //@ ghost size_t low = 0;
  //@ ghost size_t high = 0;

  /*@
    loop invariant BOUNDS: low <= high <= i <= len && cur_low <= i;

    loop invariant REL :   cur == sum(a,cur_low,i) <= max == sum(a,low,high);
    loop invariant POST:   \forall integer l;    0 <= l <= i      ==> sum(a,l,i) <= cur;
    loop invariant POST:   \forall integer l, h; 0 <= l <= h <= i ==> sum(a,l,h) <= max;

    loop assigns i, cur, max, cur_low, low, high;
    loop variant len - i;
  */
  for(size_t i = 0; i < len; i++) {
    cur += a[i];
    if (cur < 0) {
      cur = 0;
      /*@ ghost cur_low = i+1; */
    }
    if (cur > max) {
      max = cur;
      /*@ ghost low = cur_low; */
      /*@ ghost high = i+1; */
    }
  }
  return max;
}
```

Figure 6.5 – Fonction `max_sub_array` spécifiée

7 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 pu voir comment utiliser ces outils pour spécifier ce que nous attendons de nos programmes et vérifier que le code que nous avons produit correspond effectivement à la spécification que nous en avons faite. Cette spécification passe par l'annotation de nos fonctions avec le contrat qu'elle doit respecter, c'est-à-dire les propriétés que doivent respecter 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.

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 débugger 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 qu'est ce qu'il faut 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 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]

Traduction

> L'adoption des méthodes formelles dans différents domaines (notamment la vérification du matériel et des systèmes embarqués, et l'analyse et le test de logiciel) a fortement amélioré la qualité des systèmes informatiques. Nous nous attendons à ce que les méthodes formelles puissent fournir des améliorations similaires pour la sécurité des systèmes informatiques.
> [...]
> **Sans une utilisation plus large des méthodes formelles, la sécurité sera toujours fragile.**

8 Pour aller plus loin

8.1 Avec Frama-C

Frama-C propose divers moyen 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 :

— l'analyse par interprétation abstraite avec Value,
— 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 vrais 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 va contenir les vérifications en question.

Il existe divers autres greffons pour des tâches très différentes :

— analyse d'impact de modifications,
— analyse du flot de données pour le parcourir efficacement,
— ...

Et finalement, la dernière possibilité qui va motiver 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.

8.2 Avec la preuve déductive

Tout au long du tutoriel nous avons utilisé WP pour générer des obligations de preuve 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 que Alt-Ergo, le dialogue avec ceux-ci est assuré par une traduction vers le langage de Why3 qui va ensuite faire 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. WP comme Why3 peuvent extraire 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 obligations 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.