Conception Formelle

M1 informatique

Université de Bordeaux

Vincent Penelle

2023-2024

Table des matières

Chapitre 1

Introduction

Version du 17 janvier 2022. Cette introduction est probablement très imparfaite.

1.1 Bugs

Les bugs, c'est le mal, j'espère que tout le monde en est convaincu. Quand il y en a un dans un programme qu'on utilise tous les jours (traitement de texte, jeu, etc), c'est pénible. Mais quand il y en a un dans le système d'un pilotage d'un avion, d'une fusée, dans le protocole SSL ou dans un appareil médical, cela devient beaucoup plus fâcheux (dit clairement il peut y avoir des morts ou des catastrophes). On aimerait donc dans la mesure du possible pouvoir assurer qu'un programme (surtout quand il s'exécute sur un système critique) n'aura pas de bug.

Ce qui est une belle ambition, mais comment si prend t'on et d'abord, au fond c'est quoi un bug ? Parce que j'ai parlé de bug un peu rapidement, mais est-ce que tous les bugs sont de même nature ? La réponse à cette question est évidemment non, on peut à mon avis distinguer deux types principaux de bugs : les bugs d'exécution et les bugs de spécification.

Pour montrer l'exemple d'un bug à l'exécution, prenons simplement le court code suivant :

```
1 int nani(unsigned int a, unsigned int b)
2 {
3     return 4 / (a + b + 1);
4 }
```

Intuitivement, cette fonction ne devrait jamais déclencher d'exception (puisque $a+b+1$ vaut au moins 1). Cependant, c'est ce qui arrivera lorsque, par exemple on appelle `nani(UINT_MAX,0)`. En effet, l'instruction `UINT_MAX+1` fait un overflow qui donne `0`, et on fera donc une division par `0`, ce qui est un comportement indéterminé en C (i.e., non spécifié par la norme – donc le programme peut faire n'importe quoi – en l'occurence, avec gcc, il plante). Pourtant, si on raisonne sur des entiers – ce qui rappelons-le

n'est pas ce que font les machines – un bug semble impossible ici. Un bug à l'exécution sera généralement de cette forme : le déclenchement d'un comportement indéterminé (soit une division par `0` comme ici, soit – souvent – une lecture dans une zone mémoire non allouée, ou également un overflow sur les entiers signés, son comportement n'étant pas défini par la norme). Ce type de bug, quoique facile à oublier pour un développeur, est en réalité assez aisé à détecter automatiquement. En effet, il suffit de demander à un outil de traquer les instructions de base pouvant générer un comportement indéterminé et pour chacune d'entre elle mettre un assert dans le code qui sera violé si le comportement indéterminé est déclenché ou non. Charge ensuite à un outil ou au développeur de déterminer si cela peut être le cas et de prendre les mesures nécessaires pour l'éviter. À noter qu'une source de bug commune est également d'oublier que les calculs sur des entiers font des overflows (que leur comportement soit déterminé ou non). Le programme ci-dessus en est d'ailleurs un parfait exemple, puisque c'est exactement ça qui fait que ce n'est pas complètement évident.

Le second type de bug est plus subtil, puisqu'il va dépendre de ce que le programmeur attend que sa fonction fasse. Par exemple, dans la fonction suivante :

```
1 int min(int a, int b){
2     if(a > b) return a;
3     return b;
4 }
```

Aucun bug à l'exécution ne peut arriver puisqu'on ne fait que renvoyer l'un des arguments. Cependant, si on regarde le nom de la fonction on se doute qu'il y a un problème, doute qui est renforcé si on regarde le commentaire de la fonction que je n'ai pas mis et qui était le suivant :

```
1 /** @brief Returns the lowest of the values given in argument.**/
```

Ce qui n'est pas ce que la fonction fait. Et donc si un utilisateur appelle cette fonction en se basant sur le commentaire de la fonction, il n'aura pas le résultat attendu. Ce type de bug ne peut donc pas se voir avec le code seul : c'est une inadéquation entre le code de la fonction et sa spécification (qu'on a représentée ici par son commentaire mais que nous allons formaliser un peu plus dans ce cours). Évidemment, cela veut aussi dire qu'un bug de ce type peut venir du code, mais également de la spécification (si on a mal décrit le comportement de la fonction). Son but est de décrire *ce que fait* la fonction et non *comment elle le fait* (ce qui est le rôle du code, particulièrement dans le cadre d'un langage impératif comme le C). La spécification est en général plus courte et exprimée de manière plus compréhensible que le code donc c'est peut-être moins courant, mais il reste possible de mal décrire ce que fait une fonction. L'un des thèmes principaux du présent court sera justement de voir ce qu'est une bonne spécification.

1.2 Contre-mesures

Bon, maintenant on voit un peu ce qu'est un bug, quelles sont les options qu'on a pour les traquer ? L'option la plus courante, ce sont les tests. Cela consiste à déterminer un ensemble de scénarios qui sont représentatifs du comportement de la fonction et à lancer ces tests et à constater (ou non) que la fonction se comporte correctement dans ces scénarios là. On parle de *validation*. C'est pas mal, ça permet de détecter un certain nombre de bugs (et soyons honnêtes, dans pas mal de cas pas critiques, ça peut tout à fait être satisfaisant), mais ça ne permettra jamais de certifier qu'une fonction est correcte. Un test peut être un très bon moyen d'exhiber un bug mais ne garantira jamais l'absence d'un bug dans un cas qu'on aura pas testé (et il y en a beaucoup). De plus, l'exhaustivité des tests reposera beaucoup sur le programmeur et le fait qu'il pense réellement à tous les scénarios représentatifs (ce qui est dur), et surtout, il faudra à chaque fonction refaire le travail intégral de déterminer les tests, ce qui, mine de rien, est assez coûteux en temps. Dans des programmes peu critiques, ça peut suffire (un bug étant juste gênant, on pourra toujours attendre une remontée des utilisateurs pas contents et le corriger à la version suivante), mais imaginons qu'on parle du logiciel de pilotage automatique d'un avion de chasse, et que, la première fois qu'il passe l'équateur, l'avion se met brutalement à piquer vers le sol parce qu'il pense être à -64000km d'altitude (ceci est réellement arrivé), car justement ce cas-là n'a jamais été testé, et on comprendra que ce n'est pas le bug report le plus désirable.

Une autre option possible serait d'essayer de démontrer que la fonction respectera sa spécification. On parle dans ce cas de *vérification*. En gros, on aimerait avoir un programme qui prend en entrée le programme à vérifier et sa spécification et qui en sortie dit «c'est bon, y'a pas de bug», ou «il y en a un : là». Bien évidemment, une solution aussi idéale n'existe pas (sinon tout le monde utiliserait cela, du moins si le temps d'exécution était raisonnable et que spécifier était chose aisée). On se heurte pour ça à deux écueils : pratique et théorique. L'écueil pratique, c'est qu'on pourrait se dire qu'il suffit de tester une à une toutes les exécutions possible de la fonction et on saurait comme ça lesquelles présentent des bugs. Ce qui reviendrait à faire de la validation exhaustive. Évidemment, il y a beaucoup trop d'exécution possible pour une fonction même courte (il y en a par exemple 2^{128} pour la fonction donnée plus haut, ce qui est beaucoup[1]). L'écueil théorique est le suivant (et un peu plus délicat à aborder). On peut se dire «Très bien, il y a beaucoup d'exécutions, mais la plupart sont quand même vachement similaires, il n'y a qu'à abstraire un peu». Et donc on pourrait considérer que nos fonctions utilisent des entiers mathématiques, garder les bugs à l'exécution pour plus tard, et démontrer que sur des variables abstraites mathématique, le programme produit bien le résultat escompté. On considérerait donc une

1. environ 10^{38}, à comparer par exemple avec le nombre d'étoiles dans l'univers observable qui n'est «que» de 10^{24}.

infinité d'exécutions, mais descriptible finiment (avec de plus une taille raisonnable), et il suffirait ensuite de déterminer si un éventuel bug apparaît dans les vraies exécutions. Mais voilà, ce faisant, on se place dans un modèle Turing-complet, et le théorème de Rice s'applique donc, théorème disant que toute propriété non triviale d'un programme est indécidable. Donc l'approche idéale décrite ci-dessus est tout simplement impossible (il n'existe pas de programme la réalisant). En particulier, il n'existe pas de programme déterminant si une fonction C termine (du moins si elle contient un while non trivial évidemment – s'il n'y a pas de while, ça termine trivialement – mais encore une fois, indécidable, ça veut dire qu'il n'y a pas de programme qui répond à la question sur toutes les instances, ça n'interdit pas que sur certaines instances, il soit aisé de répondre).

L'option qu'on vient de décrire a l'air mal barrée, mais ça serait oublier un principe important : le mieux est l'ennemi du bien. Ce n'est pas parce que notre solution idéale est indécidable qu'on ne peut rien faire – comme la parenthèse du paragraphe précédent le note. Deux types d'approche existent :

— Une approche qui lorsqu'elle répond «le programme est correct» certifie que le programme est correct, mais dans le cas contraire ne garantit rien (elle peut répondre que le programme est incorrect alors qu'il l'est).
— Un approche qui lorsqu'elle répond le fait toujours correctement, mais peut ne pas terminer (on la force en général à terminer avec un timeout, auquel cas on a une réponse disant «je ne sais pas»).

Évidemment, dans le deux cas, on souhaitera que les cas où la réponse est incorrect (ou indéterminée) soit le moins souvent possible. On notera un point commun entre les deux approches, c'est que lorsque la réponse est correcte, on a une garantie qu'il l'est. Cette propriété se nomme la «soundness» et est au centre de la vérification. Une technique de vérification qui ne serait pas «sound» ne servirait pas à grand chose (puisque on veut une garantie de correction du programme). Et les tests ne présentent pas cette garantie.

Le présent cours est centré sur un outil appliquant la seconde approche : Frama-C.

1.3 Frama-C

Frama-C est un logiciel co-développé par une équipe de l'INRIA Saclay et le CEA LIST. Il est codé en OCaml. Son but est d'être un analyseur statique de code C et de permettre la déclaration et la vérification formelle de contrats de fonction exprimé dans un langage logique appelé ACSL (Ansi-C Specification Language) qui permet d'exprimer des phrases en logique du premier ordre. On pourra trouver plus d'informations sur le logiciel sur le site web `frama-c.com`. Les TPs du présents cours seront réalisé avec ce logiciel (on trouvera les sujets sur la page du présent cours).

On détaillera un certain nombre de fonctionnalité d'analyse statique fournit par le logiciel dans le premier TP. Ce cours sera centré sur les modules permettant la vérification de fonctions. On utilisera donc essentiellement les modules suivants :

— RTE qui permet d'ajouter automatiquement des assertions empêchant la survenue de comportement indéterminés (i.e., visant à empêcher les erreurs à l'exécution).
— WP qui applique l'algorithme de Weakest Precondition à un contrat de fonction puis détermine la validité de ce qu'il obtient à l'aide d'un solveur SMT externe. Dans ce cours, on utilisera seulement Alt-Ergo, mais Frama-C est parfaitement capable d'utiliser d'autres solveurs SMT (via Why3), tels que Z3, ou également de transmettre les preuves à réaliser à Coq.

Dans ce cours, on étudiera les bases théoriques de cet algorithme de Weakest Precondition sur un langage C simplifié à l'extrême, dans le but de comprendre comment fonctionne WP et comment trouver de bons contrats de fonctions.

Dans les TP, on vérifiera des fonctions écrit en C, mais on se restreindra à un sous-ensemble du langage. En particulier, on ne manipulera que des variable de type entier, les flottants posant beaucoup de problèmes pratiques et théoriques qui obscurciraient notre propos. D'autre part, on abordera les fonctions manipulera des pointeurs, mais on n'attribuera jamais de mémoire à l'intérieur d'une fonction (le contrat de malloc étant compliqué et donnant peu de garanties, cela encore une fois obscurcirait le propos). Mis à part ces deux restrictions, on sera en mesure de vérifier un sous-ensemble du C relativement riche.

Si on se concentre sur le langage C, ce n'est pas pour rien. C est un langage largement utilisé, notamment dans le noyau linux, mais également dans beaucoup de logiciels critiques, du fait qu'il ait aisé d'y optimiser les algorithmes via des astuces de traitement de la mémoire. Le revers de la médaille de cet avantage est qu'il devient donc assez complexe de déterminer à la main l'adéquation d'un algorithme avec sa spécification (contrairement à des langages plus haut niveau tels que OCaml) et qu'introduire un bug est aisé, notamment à cause de l'extrême permissivité du compilateur et de la gestion de la mémoire. Ainsi, un vérificateur de fonctions peut être extrêmement utile pour certifier des fonctions critiques écrites en C (e.g., dans le noyau Linux) et nécessaire pour cela, en particulier pour des implémentations astucieuses. Par ailleurs, ACSL vise à donner un standard de déclaration de contrats de fonctions formels qui en permet la vérification, mais est également utile à une documentation normalisée (et certifiable).

Chapitre 2

Contrats de fonctions

Dans ce chapitre, nous abordons de manière informelle et générale la notion qui sera au centre de ce cours : les *contrats de fonction*. De très loin, un contrat de fonction décrit ce qu'on a en sortie de la fonction si un certain nombre de conditions sont vérifiées lors de l'appel (qui représenteront le domaine de définition). On parle également de la *spécification* d'une fonction. Ces contrats peuvent être décrits de différentes manières, du très formel (ce que l'on verra avec les triplets de Hoare au chapitre 4) au moins formel en langue naturelle en passant par une variante semi-formelle dont nous allons immédiatement parler. La plupart des notions esquissées ici seront formalisées aux chapitres suivants.

2.1 Définition informelle et rapide

Un contrat de fonction contient 3 choses :

— Un domaine de définition $\mathcal{D}$ qui décrit l'ensemble des *états de mémoire* (i.e., valeurs des variables, pointeurs, etc.) dans lesquels la fonction peut être appelée.

— Une valeur de retour $\mathcal{V}$ qui décrit ce que vaudra la valeur retournée par la fonction (par rapport à ses arguments).

— Une explication des *effets de bord* qui décrit quel sera l'effet sur l'environnement mémoire auquel a accès la fonction appelante. En C, cela comprendra essentiellement les arguments passés par référence (les pointeurs, dont la valeur peut être modifiée) et les variables globales.

Le premier point sera appelé la *précondition* de la fonction, et les deux suivants les *postconditions*. En effet, le premier point parle de l'environnement de la fonction appelante avant l'appel, et les deux suivants de l'environnement après l'appel. En particulier, un contrat ne peut pas parler des variables locales à une fonction : le code est vu comme une boîte noire qui remplit le contrat.

Le sens d'un tel contrat peut être résumé dans la phrase suivante : «Si j'appelle la fonction f avec des valeurs qui respectent $\mathcal{D}$, alors ma valeur de retour sera $\mathcal{V}$, et j'aurais les effets de bords suivants.»

Un point important à noter : un contrat de fonction ne dit strictement rien sur ce qu'il se passe si on appelle la fonction en dehors de son domaine de définition. Ainsi, en toute rigueur, pour n'importe quelle fonction f, si je l'appelle sur un environnement où $0 = 1$, alors à la sortie de la fonction, la valeur de retour vaudra à la fois 1 et 4, toutes les cases de la mémoire auront été modifiées, et je serai riche à millions. La fonction f satisfait bien ce contrat, puisque le contrat spécifie qu'il n'est jamais possible de le tester.

Il sera donc important de donner des contrats pertinents, c'est-à-dire qui décrivent tout le comportement de la fonction, car comme l'exemple précédent le montre de manière absurde, sur un mauvais domaine de définition, on a un contrat qui n'a aucun sens, tout en étant satisfait.

Le point crucial à retenir est à mon avis le suivant : un contrat de fonction décrit ce que fait la fonction, et pas comment elle le fait.

2.2 Commentaires et contrats de fonction

Que ce soit en C avec doxygen ou en java avec la javadoc, vous avez déjà croisé des contrats de fonction. En effet, le format de commentaire d'une fonction qui est préconisé dans ces formalismes est en réalité un contrat de fonction.

Prenons par exemple le commentaire de fonction C suivant (venant du cours de projets technologiques de L2 de 2020-2021) :

```
1  /**
2   * @brief Creates a new game with default size and initializes it.
3   * @param squares an array describing the initial square values
4   * (row-major storage)
5   * @param nb_tents_row an array with the expected number of tents
6   * in each row
7   * @param nb_tents_col an array with the expected number of tents
8   * in each column
9   * @pre @p squares must be an initialized array of default size squared.
10  * @pre @p nb_tents_row must be an initialized array of default size.
11  * @pre @p nb_tents_col must be an initialized array of default size.
12  * @return the created game
13  **/
14 game game_new(square *squares, uint *nb_tents_row, uint *nb_tents_col);
```

Dans ce contrat, on peut facilement identifier les préconditions (qui sont notées @pre dans le format doxygen) et la valeur de retour (noté @return dans le format doxygen). Le brief donne une description informelle de ce que fait la fonction. Il n'y a ici pas d'effets de bords. Ce contrat, s'il respecte le format doxygen, reste assez informel puisqu'il n'explicite pas ici ce que veut dire «initialiser le jeu». C'est un contrat qui sera donc approprié pour une documentation, mais pas pour une vérification formelle.

Donnons un autre exemple, construit pour ce cours donnant plus de détails :

```
1  /**
2   * @brief Sorts the values of *ptr1, *ptr2 and *ptr3 so that
3   * after the call we have *ptr1 <= *ptr2 <= *ptr3 and returns
4   * the product of their values.
5   * @pre @p ptr1, @p ptr2 and @p ptr3 must be valid pointers.
6   * @pre the product of the arguments must be in the range of
7   * machine integers.
8   * @return the product of *ptr1, *ptr2 and *ptr3
9  **/
10 int reorder(int* ptr1, int* ptr2, int* ptr3);
```

Nous voyons ici à nouveau les trois éléments du contrats. La fonction triera les valeurs des pointeurs passés en argument, aura comme valeur de retour leur produit, et fera cela lorsque les pointeurs sont valides et que le produit des valeurs ne cause pas d'overflow. Le comportement en cas d'overflow des valeurs n'est ici pas spécifié. on peut également noté que la description de son comportement, bien que détaillée est assez informelle, particulièrement dans la section @brief.

2.3 Utilité des contrats de fonctions

Un contrat de fonction a deux utilités principales :

— Décrire le comportement de la fonction pour qu'un utilisateur sache comment l'appeler correctement et ce qu'il obtiendra après l'appel.

— Permettre au développeur de la fonction de certifier que le code qu'il a produit correspond bien à la spécification.

Dans le cadre de ce cours, on se concentrera essentiellement sur le deuxième point. Mais il faut garder à l'esprit que le premier point (qu'on pourrait appeler documentation, puisque c'est bien de cela qu'il s'agit) est également crucial. En réalité, l'un des objectifs principaux de ce cours est que vous compreniez comment on fait le second point dans le but que vous fassiez correctement le premier par la suite (oui ça a l'air tordu, mais pas tant que cela : si votre contrat n'est pas celui que réalise votre implémentation, votre bibliothèque perd en utilité (ou plutôt, elle contient un bug – et c'est mal)). En effet, vous vous apercevrez vite que les contrats que l'on arrive bien à prouver sont ceux qui sont utiles à son utilisateur (du moins dans la plupart des cas), et c'est heureux.

Par exemple, pour la fonction suivante :

```
1 int stupid(int a, int b){
2     if(a == b) return 0;
3     while(a == b) {
4         a++;
5         b++;
6         if(a > 42) return 42;
```

```
7       }
8       if (a == b + 2) return 2;
9       int c = a + b;
10      if(c != a + b) return 12;
11      if( a > b) return a - b;
12      return b - a;
13  }
```

Un bon contrat sera simplement que cette fonction retourne la valeur absolue de $a - b$. Certes cette fonction contient du code mort et calcule la valeur absolue de manière inutilement compliquée. Mais on s'en fiche, le contrat considère la fonction comme une boîte noire. Par ailleurs, toute autre fonction faisant la même chose aura le même contrat (encore une fois, on ne dit pas comment la fonction calcule son résultat) :

```
1  int notStupid(int a, int b){
2      if(b > a) return b - a;
3      return a - b;
4  }
```

Mentionnons enfin qu'il n'y a pas un seul contrat vrai pour une fonction, mais plusieurs. Par exemple, dans les fonctions ci-dessus, un contrat tout à fait vrai est que la fonction renvoie `0` si on lui fourni `0` et 4 si on lui fourni -4. C'est un contrat vrai, mais inutile, car la fonction fait d'autres choses. Dans la majorité des cas, on cherchera à ce que le contrat de fonction décrive son comportement sur son domaine de définition, ou du moins sur son domaine d'utilisation (qui peut être différent). Par exemple, dans le cas suivant :

```
1  int fact(int a){
2      int result = 1;
3      for (int i = 1; i <= a; i++){
4          result = result * i;
5      }
6      return result;
7  }
```

Le domaine de définition de `fact` est [`INT_MIN`,c] (pour une certaine valeur c que j'ai la flemme de calculer, mais qui correspond à la valeur où il y a un overflow). Cependant, le domaine d'utilisation normal de cette fonction est [`0`,c], puisque la factorielle n'est pas définie pour les nombres négatifs (et que du coup, on se moque qu'elle renvoie 1, on n'est pas vraiment censé utiliser cette fonction sur un nombre négatif). On préférera donc ici d'un contrat parlant du comportement de la fonction dans le cas des nombres positifs. Pour être tout à fait précis, le «bon» contrat ici serait le même que celui de la fonction suivante :

```
1  int fact_rec(int a){
2      if(a == 0) return 1;
3      else return (a * fact_rec(a-1));
4  }
```

Cette fonction n'aura évidemment pas le même comportement sur les nombres négatifs. Par contre, elle calcule bien la factorielle sur [0,c], et comme c'est bien sur ce domaine qu'est définie la factorielle, il est normal que le contrat parle uniquement de cette portion des entiers. Un bon contrat représente le comportement de la fonction dans le domaine où elle est censée être utilisée, ce qui n'est pas toujours la même chose que le domaine où elle renvoie une valeur sans erreur.

L'un des points les plus important de ce cours (particulièrement en TP) sera de bien formuler des contrats de fonction pour être capable de démontrer que le code de la fonction les respecte. Pour cela, on le fera en logique du premier ordre (à partir du chapitre suivant), et on étudiera l'algorithme permettant de semi-décider (ça marche pas toujours évidemment) la validité d'un contrat de fonction, mais en gardant bien à l'esprit que l'objectif principal est d'en tirer comment formuler de bons contrats de fonction, utiles pour l'utilisateur (pour des fonctions compliquées, la preuve est en général hors de portée).

2.4 Que mettre dans un contrat de fonction ?

À titre de résumé de ce chapitre, et pour servir de «cheat sheet», cette section va résumer ce dont on cherche à parler dans un contrat de fonction.

2.4.1 Préconditions

Les préconditions décrivent l'ensemble des états de mémoire dans lequel la fonction peut être appelée. À ce titre, elle ne peut parler que des paramètres de la fonction, des valeurs pointées par ces paramètres, ainsi que des éventuelles variables globales et ce AVANT l'appel de la fonction. On ne peut en aucun cas parler du résultat (ou plus généralement des effets de la fonction) dans ce bloc.

Le rôle des préconditions étant de déterminer les conditions dans laquelle se comporte correctement, on va typiquement avoir des propriétés du genre :

- n est strictement positif.
- n appartient à l'intervalle I.
- n est pair.
- t est un pointeur valide (alloué).
- t est un tableau de taille n (i.e., les pointeurs t à $t+n-1$ sont des pointeurs valides, en C, c'est pareil).
- la somme de a et de $*b$ ne causera pas d'overflow.
- le tableau t est trié.

Ce ne sont que des exemples et certainement pas une liste exhaustive. En particulier, si on ne parle pas directement du code, on peut observer

que la plupart de ces propriété permettent en fait simplement d'éviter des bugs à l'exécution (overflow ou lecture dans des zones non allouée). C'est normal, c'est l'un de leur intérêt principal. Mais d'autres désigneront une propriété que les données doivent satisfaire pour que l'algorithme de la fonction fonctionne correctement – par exemple, la recherche dichotomique a besoin d'un tableau triée.

On utilisera ce type de propriétés dans la suite (et on les formalisera pour Frama-C).

2.4.2 Postconditions

Les post-conditions décrivent les effets de la fonction sur l'état de mémoire après son exécution.

Elle ne peuvent également parler que des paramètres de la fonction, des zones pointées par ces dernier et des variables globales, mais elle peuvent parler de ces valeurs avant ET après l'exécution de la fonction. Elle peut également parler du résultat de la fonction (elle devrait d'ailleurs).

Un point important : en C, la stratégie de passage des arguments est un passage par valeur. Il n'y a donc aucun sens à parler de la valeur des paramètre après l'appel de la fonction (puisqu'aucun effet de bord n'est possible sur eux). Par contre, cela n'est vrai ni pour les valeurs pointées par eux ni pour les variables globales, et pour ceux-ci, on parlera bien des valeurs avant et après l'exécution de la fonction[1].

Avec une stratégie de passage des arguments différente (par référence par exemple), on ferait évidemment différemment.

On aura donc des propriété du style :

— Le résultat est la somme de a, de l'ancienne valeur de $*b$ et de la nouvelle valeur de $t[4]$.
— Le tableau t est trié.
— Le tableau q n'est pas modifié.
— Le résultat est le minimum des éléments du tableau t (valeurs avant l'appel de la fonction).
— Les valeurs de $*a$ et $*b$ ont été échangées.
— Si a est positif, le résultat est la factorielle de a, mais si a est négatif, le résultat est 1.

À nouveau cette liste n'est pas exhaustive, mais on aura des propriétés de la sorte. À noter, on pourrait aussi avoir des propriétés certifiant des allocation (si on voulait certifier malloc par exemple, mais c'est complexe à faire et on ne le fera pas).

On notera également (dernière propriété) qu'on peut décrire dans ces propriétés des *comportements* différents de la fonction en fonction des arguments fournis.

1. En Frama-C, on aura une convention précise pour cela, mais restons informels pour le moment.

De la même manière, on formalisera plus précisément ces propriétés pour parler à Frama-C, mais à ce stade, l'important est que vous ayez un aperçu de ce qu'on peut exprimer.

2.4.3 Bon, au final, que mettre dans un contrat de fonction ?

Un bon contrat de fonction décrira précisément le comportement de la fonction vu de l'extérieur.

Il doit contenir les préconditions nécessaires au bon comportement de la fonction. Tous les effets de la fonction (effets de bords) doivent être décrits le plus précisément possible. Le résultat de la fonction doit être décrit (s'il y en a un).

Il ne doit pas décrire comment la fonction réalise cette tâche.

Enfin un point à garder en tête : un contrat de fonction ne dit pas ce qu'il se passe lorsqu'on appelle la fonction en-dehors du domaine de définition. Aussi, si vous êtes capable de déterminer ce que fait la fonction dans certains cas, il peut être pertinent de le mettre dans le contrat, de manière à en informer l'utilisateur (développeur).

Si vous laissez un cas en dehors du domaine de définition, vous dites simplement que vous ne promettez strictement rien (ce qui est bien sûr ce qu'il faut faire dans certains cas).

2.4.4 Contrats plus généraux

Au fond, la notion de contrat est assez versatiles, et il n'y a aucune raison de la limiter à des fonction. On peut faire un contrat pour toute séquence d'instructions. Avec évidemment les mêmes restrictions, à savoir qu'on ne peut parler que des variables qui existent avant l'exécution du dit bloc de code, et pas de celles n'existant qu'à l'intérieur.

En particulier, on aura dans la suite besoin de contrats de boucles (qu'on nommera invariants de boucle).

Chapitre 3

Modèle de programmation

3.1 Logique et états de mémoire

3.1.1 États de mémoire

Avant d'introduire notre modèle de programmation, on introduit un *domaine abstrait* permettant de représenter l'état de mémoire d'un ordinateur, ainsi qu'une logique permettant d'en parler.

L'état de mémoire d'un ordinateur est considéré comme étant composée de variables, toutes du même type (des entiers)[1]. Formellement, on appelle Var l'ensemble de toutes les variables acceptables en C, qu'on considérera fixé dans le reste du cours. On considère également dispose d'un espace mémoire pour les pointeurs, indexé par des entiers. La mémoire indexée par Var correspond à la mémoire locale d'une fonction (i.e., qui ne peut être sujette à des effets de bord), alors que la mémoire indexée par $\mathbb{Z}$ correspond à la mémoire globale d'un programme, et peut être modifiée par effet de bord.

Définition 3.1.1. *L'ensemble des états de mémoire* $\mathbb{M}$ *est l'ensemble des fonctions de* $\mathrm{Var} \uplus \mathbb{Z}$ *dans* $\mathbb{Z}$.

On note $\mathcal{M}(x)$ la valeur d'une variable $x \in \mathrm{Var}$ dans un état de mémoire $\mathcal{M}$, et similairement $\mathcal{M}(z)$ la valeur pointée par $z \in \mathbb{Z}$ dans ce même état. Note, ceci est bien évidemment une simplification par rapport à la mémoire d'un ordinateur, dans laquelle il n'y a pas de séparation stricte entre la

1. Cette restriction est là pour des raisons de simplification : le calcul sur les flottants pose de nombreux problèmes théoriques qui vont bien au-delà du champ de ce cours et qui rendraient toute preuve de fonction manipulant des flottants impossible. On pourrait retrouver des preuves faisables en manipulant à la place des nombres réels, mais cela complexifierait notre propos, en plus d'assez mal modéliser ce qui se passe en réalité avec des flottants.

De même, on regarde $\mathbb{Z}$ à la place des entiers machines pour simplifier les preuves. Frama-C considère lui bien des entiers machines. On peut cela dit simuler des propriétés sur des entiers machines en fixant des conditions sur les valeurs des entiers manipulés (ce qui est grosso modo ce que fait Frama-C – et on comprendra plus en détail avec la formalisation des contrats dans le chapitre 4).

mémoire dévolue aux variables et celle dévolue aux pointeurs, et évidemment plusieurs types occupant un espace mémoire différent, et donc une gestion plus complexe des recouvrements de pointeurs. La modélisation présentée ici permet tout de même de représenter assez fidèlement le comportement d'un programme (simple) pour vérifier sa spécification, et permettra de manipuler beaucoup plus simplement les preuves que nous effectuerons.

On distingue une variable particulière `\result` qui contiendra la valeur de retour d'une fonction. On considère que cette variable particulière ne peut pas apparaître telle quelle dans les expressions définies ci-après, et ne peut être manipulée que par les instructions `return` et les appels de fonctions.

Enfin, rappelons qu'il faut voir cet état de mémoire comme une affectation des variables à un moment donné, qui pourra être modifié par des instructions (i.e. programmes). On considère que la notion présentée ici représente l'intégralité d'une configuration du système (encore une fois, c'est une simplification).

3.1.2 Arithmétique

Avant de définir la logique permettant de parler des états de mémoire, il nous faut définir les termes arithmétique et leur sémantique dans un état de mémoire. En effet, nos programmes seront capable de faire des opérations arithmétique, il faut donc être capable de les manipuler.

On considère l'ensemble des expressions arithmétiques Arith défini ainsi :

$$t \quad ::= \quad x \in \mathsf{Var} \;\mid\; z \in \mathbb{Z} \;\mid\; t+t \;\mid\; t-t \;\mid\; t \times t \;\mid\; t/t \;\mid\; t\%t \;\mid\; *t$$

où % représente l'opérateur du reste. On considèrera que les définitions du quotient et du reste sont celles du C, et non la division euclidienne classique dans toute la suite du cours, pour éviter la surcharge de notations. On pourra (notamment dans le langage de programmation utilisé) utiliser la notation `t1 * t2` à la place de $t1 \times t2$.

Étant donnés $\mathcal{M} \in \mathbb{M}$ et $t \in \mathsf{Arith}$, la *valeur* de t dans $\mathcal{M}$, $[\![t]\!]_{\mathcal{M}}$ est définie récursivement comme suit :

- $[\![x]\!]_{\mathcal{M}} = \mathcal{M}(x)$
- $[\![z]\!]_{\mathcal{M}} = z$
- $[\![t_1 + t_2]\!]_{\mathcal{M}} = [\![t_1]\!]_{\mathcal{M}} + [\![t_2]\!]_{\mathcal{M}}$
- $[\![t_1 - t_2]\!]_{\mathcal{M}} = [\![t_1]\!]_{\mathcal{M}} - [\![t_2]\!]_{\mathcal{M}}$
- $[\![t_1 \times t_2]\!]_{\mathcal{M}} = [\![t_1]\!]_{\mathcal{M}} \times [\![t_2]\!]_{\mathcal{M}}$
- $[\![t_1/t_2]\!]_{\mathcal{M}} = [\![t_1]\!]_{\mathcal{M}} / [\![t_2]\!]_{\mathcal{M}}$
- $[\![t_1 \% t_2]\!]_{\mathcal{M}} = [\![t_1]\!]_{\mathcal{M}} \mod [\![t_2]\!]_{\mathcal{M}}$
- $[\![*t]\!]_{\mathcal{M}} = \mathcal{M}([\![t]\!]_{\mathcal{M}})$.

Étant donnés $\mathcal{M} \in \mathbb{M}$, $x \in$ Var et $t \in$ Arith, on définit la *substitution* de x par t dans $\mathcal{M}$ comme l'état de mémoire $\mathcal{M}[x \leftarrow t]$ tel que, pour toute variable y et tout entier z,

$$\mathcal{M}[x \leftarrow t](y) = \begin{cases} [\![t]\!]_{\mathcal{M}} \text{ si } x = y. \\ \mathcal{M}(y) \text{ sinon.} \end{cases} \qquad \mathcal{M}[x \leftarrow t](z) = \mathcal{M}(z)$$

Étant donné un entier $z \in \mathbb{Z}$, la substitution de la valeur pointée par z par t, $\mathcal{M}[z \leftarrow t]$ est défini de manière similaire.

On considère l'ensemble des *opérateurs de comparaisons* usuels sur $\mathbb{Z}$, Comp $:= \{<, >, ==, \leq, \geq, \neq\}$ (de même que pour la multiplication, on pourra remplacer ces symboles par leur équivalent en C). Une *garde* est une comparaison entre deux termes arithmétiques, *i.e.*, t_1 op t_2, pour $t_1, t_2 \in$ Arith et op $\in$ Comp. On note Guard l'ensemble de ces gardes. Leur sémantique est naturelle : $[\![t_1 \text{ op } t_2]\!]_{\mathcal{M}}$ est vraie si $[\![t_1]\!]_{\mathcal{M}}$ op $[\![t_2]\!]_{\mathcal{M}}$.

3.1.3 Logique du premier ordre

Nous pouvons enfin définir formellement la logique du premier ordre dont nous nous servirons pour décrire les contrats de fonctions (ce qui est le but, ne l'oublions pas). Une formule du premier ordre pourra être vue comme la définition d'un ensemble d'états de mémoires (ceux qui la satisfont).

On définit la *logique du premier ordre* FO, comme suit :

$$\varphi \quad ::= \quad \text{guard} \in \text{Guard} \;\mid\; \varphi \wedge \varphi \;\mid\; \varphi \vee \varphi \;\mid\; \neg\varphi \;\mid\; \exists x, \varphi \;\mid\; \forall x, \varphi$$

On considère les macros usuelles $\Rightarrow$, $\Leftrightarrow$, etc. On pourra utiliser, notamment dans le langage de programmation définit ci-après, une syntaxe C pour les connecteurs logique, indifféremment de leur syntaxe définie ici. Pour éviter les confusions, on notera $\equiv$ l'égalité entre deux formules de FO.

L'ensemble Test des *tests arithmétiques* est défini comme le fragment sans quantificateur de FO (c'est-à-dire sans la partie en rouge ci-dessus).

Étant donné une formule φ de FO, une variable $x \in$ Var et un terme arithmétique $t \in$ Arith, on définit la *substitution* de x par t dans φ, $\varphi[x \leftarrow t]$, comme la formule φ dans laquelle toute occurence de x est remplacée par t[1]. Cette réécriture est purement *syntaxique*, c'est-à-dire qu'on n'a pas à se préoccuper du sens, simplement à remplacer le symbole x par la séquence de symboles t. Par exemple $(x = 3 * x - y \vee y = 3 * x)[x \leftarrow 2 * x - z] \equiv (2 * x - z = 3 * (2 * x - z) - y \vee y = 3 * (2 * x - z))$. Notez qu'ici je n'ai pas simplifié l'expression. En toute rigueur simplifier une expression ne donne pas la même formule (syntaxiquement). Cependant, on s'autorisera à simplifier des expressions arithmétiques tant qu'on ne change pas leur sémantique, par lisibilité et habitude. Ici, on obtiendrait donc $(y = 4 * x - z \vee y = 6 * x - 2 * z)$.

1. Attention, t peut bien évidemment contenir x, ce qui n'est pas un problème.

On définit maintenant récursivement la sémantique de FO. On dit qu'une formule φ est vraie sur un état de mémoire $\mathcal{M}$ (ou de manière équivalent que $\mathcal{M}$ juge φ), noté $\mathcal{M} \models \varphi$ si :

— $\mathcal{M} \models t_1$ op t_2 si $[\![t_1]\!]_{\mathcal{M}}$ op $[\![t_2]\!]_{\mathcal{M}}$, pour tout op $\in$ Comp.
— $\mathcal{M} \models \varphi_1 \wedge \varphi_2$ si $\mathcal{M} \models \varphi_1$ et que $\mathcal{M} \models \varphi_2$.
— $\mathcal{M} \models \varphi_1 \vee \varphi_2$ si $\mathcal{M} \models \varphi_1$ ou que $\mathcal{M} \models \varphi_2$.
— $\mathcal{M} \models \neg\varphi$ si $\mathcal{M} \not\models \varphi$.
— $\mathcal{M} \models \exists x, \varphi$ si il existe $z \in \mathbb{Z}$ tel que $\mathcal{M} \models \varphi[x \leftarrow z]$.
— $\mathcal{M} \models \forall x, \varphi$ si pour tout $z \in \mathbb{Z}$, $\mathcal{M} \models \varphi[x \leftarrow z]$.

Une formule φ est dite satisfaisable si il existe un état de mémoire $\mathcal{M}$ tel que $\mathcal{M} \models \varphi$. Elle est dite valide si pour tout état de mémoire $\mathcal{M}$, $\mathcal{M} \models \varphi$.

Par exemple, la formule $x = y$ est satisfaisable : en effet, l'état de mémoire $\mathcal{M}$ dans lequel $\mathcal{M}(x) = 4$ et $\mathcal{M}(y) = 4$ satisfait cette formule. La formule $\exists x, x < 4$ est valide (il existe bien un entier strictement inférieur à 4 (par exemple 0)).

Une formule peut être vue comme un ensemble d'états de mémoire (ceux qui les satisfont). On écrira dans la suite cet ensemble $[\![\varphi]\!] = \{\mathcal{M} \mid \mathcal{M} \models \varphi\}$.

Par exemple, $[\![x = y]\!]$ désigne l'ensemble des états de mémoire $\mathcal{M}$ dans lesquels $\mathcal{M}(x) = \mathcal{M}(y)$ (bon, là c'est pas très malin, mais vous voyez l'idée).

On notera en particulier la propriété suivante : $\varphi \Rightarrow \psi$ est valide si et seulement si $[\![\varphi]\!] \subseteq [\![\psi]\!]$. La raison est que c'est simplement une conséquence de la définition de l'implication : on dit simplement que si $\varphi \Rightarrow \psi$ est valide, alors tout état de mémoire satisfaisant φ satisfait également ψ (ce qui est la définition de l'implication).

3.2 Programmes

On travaille, pour les parties théoriques de ce cours, sur un langage qui peut être vu comme un sous-ensemble du C (Frama-C travaille lui sur du C). Plus précisément, dans les chapitres suivants, on travaillera sur 2 langages, selon que l'on inclue ou non les boucles. Chaque instruction aura une sémantique qui représentera son effet sur un état de mémoire. Cette sémantique ne sera pas nécessairement complète : si l'effet d'une instruction n'est pas défini sur un état de mémoire donné, on est en présence d'un comportement indéterminé. Cela signifie qu'on considère, dans le domaine abstrait que le calcul n'est pas défini (donc que ce cas ne peut pas arriver), mais dans un programme sur une machine concrète, il se passera bien quelque chose, mais la sémantique ne précise pas quoi (ça peut dépendre de la machine, du compilateur, etc). Ce sera en particulier le cas des boucles infinies qui n'auront pas de sémantique définie (puisqu'on ne sort jamais de la boucle, il est impossible de définir l'état de mémoire en sortie).

3.2.1 Programmes sans boucles

On considère C_1 un langage de programmation simpliste ne contenant pas de boucles. Ce langage sera utilisé pour introduire le calcul de weakest precondition sans s'embarrasser de détails.

Les programmes sont construits récursivement de la manière suivante, où `x`, `tab` ∈ Var, `t`, `p` ∈ Arith, et `test` ∈ Test et `P1` et `P2` sont des programmes :

```
1 skip;
2 x = t;
3 P1; P2;
4 if(test) P1; else P2;
5 *(p) = t;
6 tab[i] = t;
7 return t;
```

On verra `tab[i]` comme un raccourci pour `*(tab+i)`, on ne définira donc pas sa sémantique.

On notera l'absence du constructeur `&` par rapport au C. Étant donné qu'on sépare les notions de mémoires locales et globale, l'adresse d'une variable n'est en effet pas un concept que l'on peut manipuler dans notre modèle. Bien évidemment, Frama-C est lui capable de manipuler ce constructeur, mais pour les parties théoriques, nous simplifions ce point.

Pour simplifier, on considèrera que `return t` ne peut apparaître que comme *dernière* instruction d'un programme et une seule fois (tous les exemples que nous manipuleront vérifirons cette propriété). C'est une simplification pour les calculs de weakest precondition que nous effectuerons, qui ne fait pas perdre de généralité au langage de programmation (on peut simuler une sortie prématurée du programme avec cette restriction). On pourrait définir tout ce que l'on fera sans cette simplification, mais les définitions en seraient plus complexes.

La sémantique d'un programme est défini comme une fonction des états de mémoire dans les états de mémoire. Elle est définie récursivement comme suit, pour tout $\mathcal{M} \in \mathbb{M}$:

- $\texttt{skip}(\mathcal{M}) = \mathcal{M}$
- $(\texttt{x = t})(\mathcal{M}) = \mathcal{M}[\texttt{x} \leftarrow \texttt{t}]$
- $(\texttt{P1;P2})(\mathcal{M}) = \texttt{P2}(\texttt{P1}(\mathcal{M}))$
- $(\texttt{if(test) P1; else P2})(\mathcal{M}) = \begin{cases} \texttt{P1}(\mathcal{M}) \text{ si } \mathcal{M} \models \texttt{test} \\ \texttt{P2}(\mathcal{M}) \text{ sinon.} \end{cases}$
- $(\texttt{*(p) = t})(\mathcal{M}) = \mathcal{M}[[\![\texttt{p}]\!]_{\mathcal{M}} \leftarrow \texttt{t}]$
- $(\texttt{return t})(\mathcal{M}) = \mathcal{M}[\backslash\texttt{result} \leftarrow [\![t]\!]_{\mathcal{M}}]$

Cette fonction s'étend naturellement aux états de mémoire : Étant donné un ensemble d'états de mémoire E, et un programme `P`, $\texttt{P}(E) = \{\texttt{P}(\mathcal{M}) \mid \mathcal{M} \in E\}$.

3.2.2 Boucles

On définit maintenant C_2 qui contient additionnellement des boucles `while`. On ne considèrera pas les boucles `for`, celles-ci pouvant s'exprimer à partir des boucles `while`. Concrètement, on ajoute la construction suivante, pour `test`$\in$ Test, et `P` un programme :

```
1 while(test) P;
```

La sémantique d'une boucle `while` est définie inductivement comme suit :

— $\texttt{while(test) P;}(\mathcal{M}) = \begin{cases} \mathcal{M} & \text{si } \mathcal{M} \not\models test \\ \texttt{while(test)P;}(P(\mathcal{M})) & \text{sinon.} \end{cases}$

Notez que la sémantique d'une boucle `while` n'est bien définie que si elle termine sur l'état de mémoire considéré !

3.2.3 Appels de fonction

Jusqu'ici nous avons éludés les appels de fonction, mais il est bien évidemment possible de modéliser de tels appels. Mais cela vient au prix de complexifier légèrement le modèle (surtout pour distinguer les effets de bords). J'en parle donc ici, et a priori, la plupart des exemples n'incluront pas de tels appels (pour la partie théorique, pour Frama-C, on les utilisera).

On ajoute les instructions suivantes aux programmes :

```
1 x = f(a1,...,ak)
2 *p = f(a1,...,ak)
```

où `x` $\in$ Var, `p`,`a1`,...,`ak` $\in$ Arith, et `f` est une fonction ayant k arguments (on considère que les variables manipulées par `f` pour stocker ces arguments sont nommées `x1`,...,`xk` dans la suite).

Pour définir la sémantique d'un tel appel, il nous faut découper la représentation de l'état de mémoire entre sa partie sur Var (mémoire locale) et celle sur $\mathbb{Z}$ (mémoire globale). Dans cette partie, un état de mémoire est donc vu comme un couple $\mathcal{M} = (\mathcal{M}_{\mathsf{Var}}, \mathcal{M}_{\mathbb{Z}})$. De plus on note entre $<>$, avec des associations de certaines variables à une valeur, un état de mémoire explicite (qui est simplement l'état de mémoire d'appel de la fonction où on a évalué les arguments). La sémantique des appels de fonctions est donc définie ainsi :

— $(\texttt{x = f(a1,...,ak)})(\mathcal{M}) = (\mathcal{M}_{\mathsf{Var}}[\texttt{x} \leftarrow [\![\texttt{\result}]\!]_{\mathcal{M}'}], \mathcal{M}'_{\mathbb{Z}})$, où $\mathcal{M}' = \texttt{f}(<\texttt{x1} : [\![\texttt{a1}]\!]_{\mathcal{M}}, ..., \texttt{xk} : [\![\texttt{ak}]\!]_{\mathcal{M}} >, \mathcal{M}_{\mathbb{Z}})$

— $(\texttt{*p = f(a1,...,ak)})(\mathcal{M}) = (\mathcal{M}_{\mathsf{Var}}, \mathcal{M}'_{\mathbb{Z}}[[\![\texttt{p}]\!]_{\mathcal{M}} \leftarrow [\![\texttt{\result}]\!]_{\mathcal{M}'}])$, où $\mathcal{M}' = \texttt{f}(<\texttt{x1} : [\![\texttt{a1}]\!]_{\mathcal{M}}, ..., \texttt{xk} : [\![\texttt{ak}]\!]_{\mathcal{M}} >, \mathcal{M}_{\mathbb{Z}})$

Informellement, cela signifie que l'état de mémoire qu'on passe au début de l'appel de `f` contient pour les variables les valeurs qu'on lui a passé par argument (et uniquement celles-ci, les variables du programmes appelant ne sont pas présentes dans cette mémoire), et pour les pointeurs l'état correspondant au moment de l'appel de la fonction. À la sortie de la fonction, l'état de mémoire des variables de la fonction appelante n'est

pas modifié (sauf par l'affectation éventuelle du résultat), alors que celui des pointeurs est modifié par l'appel de la fonction. Pour info, cette sémantique correspond à une sémantique *call by value*, c'est-à-dire que les valeurs des arguments sont calculés lors de l'appel de la fonction (c'est bien le cas en C).

Notez également que dans le cas où on stocke la valeur de retour d'une fonction dans un pointeur l'adresse de stockage est évaluée dans le contexte d'avant l'appel de la fonction (cela correspond encore une fois au comportement du C).

Dans la suite, on ne fera attention à cette distinction que lorsqu'on parlera d'appels de fonctions.

Finalement, remarquez qu'avec le formalisme qu'on a choisi, on interdit de passer en argument d'une fonction l'adresse d'une variable pour qu'elle soit modifiée par effet de bord, puisque l'adresse d'une variable n'existe pas. On peut bien évidemment simuler cela facilement (quitte à rajouter des instructions qui utilisent un pointeur frais pour récupérer cet effet de bord). C'est encore une fois une simplification pour ce cours qui vise à nous simplifier la vie.

Chapitre 4

Weakest Precondition

Dans ce chapitre on s'intéresse au langage de programmation C_1 sans boucles défini au chapitre 3.

4.1 Triplets de Hoare

Définition 4.1.1. *Un triplet de Hoare est un triplet de la forme $\{\varphi\}P\{\psi\}$, où φ et ψ sont des formules de FO, et P est un programme.*

Définition 4.1.2. *Un triplet de Hoare $\{\varphi\}P\{\psi\}$ est dit valide si pour tout état de mémoire $\mathcal{M}$, si $\mathcal{M} \models \varphi$, alors $P(\mathcal{M}) \models \psi$, ou de manière équivalente, si $P([\![\varphi]\!]) \subseteq [\![\psi]\!]$.*

Un triplet de Hoare n'est rien d'autre qu'un contrat de fonction dont les pré et post-conditions sont exprimées en logique du premier ordre. En effet, la définition 4.1.2 dit bien que si la fonction est appelée dans un contexte où φ est vraie, alors ψ sera vrai après l'appel. Il suffira donc que φ exprime la précondition de la fonction, et ψ la valeur de retour et les effets de bord.

À noter que, comme évoqué au chapitre 2, les formules φ et ψ ne peuvent faire référence qu'à des valeurs définies dans l'environnement mémoire appelant la fonction ($\mathcal{M}$ dans la définition 4.1.2), pas à celui dans la fonction. À noter que cela correspond bien au formalisme que nous utilisons (cf. fin du chapitre 3). Il est donc évidemment impossible de parler des variables internes de `P` (et ce point restera vrai quand on aura des contrats de boucle). Frama-C vous empêchera d'ailleurs de le faire.

Un dernier point à noter. Il découle de cette définition deux propriétés un peu contre-intuitives auxquelles il faudra faire parfois attention. Tout d'abord, pour tout programme `P` et toute formule ψ, le triplet $\{\bot\}P\{\psi\}$ est valide. Par ailleurs, étant donné que si un programme `P` ne termine pas sur un état de mémoire $\mathcal{M}$, $P(\mathcal{M})$ n'est pas défini, un triplet valide ne décrit le comportement d'une fonction que dans le cas où elle termine (et ne peut dire si c'est bien le cas). On appellera cela de la *correction partielle*, et on développera cela quand on abordera les boucles.

4.2 Rapide historique de la notion

Comme son nom l'indique, le triplet de Hoare a été introduit par Tony Hoare, en 1969[4]. Il l'a été dans le but de démontrer la correction de programme, accompagné d'un système de dérivation logique, que l'on appelle logique de Floyd-Hoare, car Robert W. Floyd a parallèlement introduit une notion très similaire (mais sur des flowcharts)[2].

L'idée de ce système de dérivation est de présenter un *système de preuve* (très similaire à ceux que vous avez vu dans le cours de logique et preuve de L3) qui démontre la validité d'un triplet grâce à des axiomes (pour les constructions de base du langage) et à règles d'inférences (pour les constructeurs tels que le if then else, le séquencement, etc), mais également une règle dite de conséquence qui en gros permet d'introduire des formules logiques (qui affaiblissent les préconditions ou renforcent les postconditions). Cette dernière règle rend tout le procédé de preuve très non-déterministe (mais c'est souvent le cas dans les systèmes logiques).

Originellement ce système a été défini sur un langage impératif minimaliste (très proche de celui considéré dans ce cours), mais il a été étendu à de nombreux langages concrets (le C par exemple, et d'une certaine manière, c'est ce que Frama-C utilise).

Néanmoins, la faiblesse de l'approche est que du fait de la règle de conséquence et qu'il existe donc de nombreuses preuves d'un triplet, automatiser directement le procédé est peu aisé, en l'état.

En 1975, Edsger W. Dijkstra [1] propose une reformulation de la logique de Floyd-Hoare vue maintenant comme un système déterministe de calcul de prédicat (qu'on appelle aussi sémantique de transformation de prédicats) qui automatise tout le procédé de calcul (pour les programmes sans boucles du moins, nous reviendrons là-dessus plus loin), qui permet de ramener la détermination de la preuve d'un triplet à la preuve de la validité d'une formule logique de FO. Ce calcul nommé le calcul de weakest precondition est ce que nous allons voir et appliquer dans ce cours, et est ce que réalise Frama-C : l'idée est que le calcul de weakest precondition est automatisé par Frama-C, et la preuve de l'implication résultante est déléguée à un outil spécialisée dans cette tâche (vous vous souvenez de Z3 ?).

4.3 Calcul de Weakest Precondition

Le calcul de Weakest Precondition est un algorithme qui permet, étant donné un programme $\mathtt{P}$ et une formule logique ψ de trouver la formule la plus faible au sens de l'implication $\mathbf{WP}(\mathtt{P}, \psi)$ telle que $\{\mathbf{WP}(\mathtt{P}, \psi)\}\mathtt{P}\{\psi\}$ est un triplet de Hoare valide. C'est-à-dire que pour tout triplet de Hoare valide $\{\varphi\}\mathtt{P}\{\psi\}$, on a $\varphi \Rightarrow \mathbf{WP}(\mathtt{P}, \psi)$, ou de manière équivalente $[\![\varphi]\!] \subseteq [\![\mathbf{WP}(\mathtt{P}, \psi)]\!]$.

Ce calcul permet de donner un algorithme simple pour tester la validité d'un triplet. En effet, la principale difficulté pour évaluer un triplet consiste en ce que la précondition et la postcondition ne s'évaluent pas sur le même état de mémoire. Le calcul de Weakest Precondition permet de calculer une formule qui permet de déterminer la validité d'un triplet en déterminant la validité d'une formule.

On peut tout à fait voir ce calcul comme une *sémantique* différente du langage de programmation (d'où le nom alternatif de sémantique par transformation de prédicat) : dans ce cas, on considère que les programmes agissent sur des formules logiques (représentant des ensembles d'états de mémoire, ce qui n'est donc pas très éloigné de la réalité, même si c'est évidemment assez abstrait). Avec tout de même la particularité amusante que les programmes s'exécutent de la fin vers le début dans cette définition.

Définition 4.3.1 (Weakest Precondition)**.** *La weakest precondition d'un couple* (P, ψ) *est définie récursivement comme suit :*

- $\mathbf{WP}(\texttt{skip}, \psi) \equiv \psi$
- $\mathbf{WP}(x = t, \psi) \equiv \psi[x \leftarrow t]$
- $\mathbf{WP}(P\_1;P\_2, \psi) \equiv \mathbf{WP}(P\_1, \mathbf{WP}(P\_2, \psi))$
- $\mathbf{WP}(\texttt{if(test) P\_1; else P\_2}, \psi) \equiv \texttt{test} \Rightarrow \mathbf{WP}(P\_1, \psi) \wedge \neg \texttt{test} \Rightarrow \mathbf{WP}(P\_2, \psi)$.
- $\mathbf{WP}(*p = t, \psi) \equiv \psi[*p \leftarrow t]$.
- $\mathbf{WP}(\texttt{return } t, \psi) \equiv \psi[\backslash\texttt{result} \leftarrow t]$.

Théorème 4.3.2. *Pour toutes formules* φ *et* ψ *et tout programme* P*,* $\{\varphi\}P\{\psi\}$ *est un triplet de Hoare valide si et seulement si* $\varphi \Rightarrow \mathbf{WP}(P, \psi)$ *est une formule valide, i.e., pour tout état de mémoire* $\mathcal{M}$*,* $\mathcal{M} \models \varphi \Rightarrow \mathbf{WP}(P, \psi)$*, ou* $[\![\varphi]\!] \subseteq [\![\mathbf{WP}(P, \psi)]\!]$.

Démonstration. On va démontrer cette propriété par induction sur la structure du programme.

Cas de bases :

- Supposons que $\{\varphi\}\texttt{skip}\{\psi\}$ est un triplet valide. Par définition, on a $\texttt{skip}([\![\varphi]\!]) \subseteq [\![\psi]\!]$, ce qui donne, par définition de la sémantique de skip, $[\![\varphi]\!] \subseteq [\![\psi]\!]$. Comme $\mathbf{WP}(\texttt{skip}, \psi) \equiv \psi$, on a bien $[\![\varphi]\!] \subseteq [\![\mathbf{WP}(\texttt{skip}, \psi)]\!]$.
- Supposons que $\{\varphi\}\texttt{x=t}\{\psi\}$ est un triplet valide. Par définition, on a $\texttt{x=t}([\![\varphi]\!]) \subseteq [\![\psi]\!]$, ce qui donne, par définition de la sémantique de x=t, $\{\mathcal{M}[x \leftarrow t] \mid \mathcal{M} \models \varphi\} \subseteq [\![\psi]\!]$. On obtient $\{\mathcal{M}[x \leftarrow t] \mid \mathcal{M} \models \varphi\} \subseteq \{\mathcal{M}[x \leftarrow t] \models \psi\}$, puisque l'ensemble des états de mémoire qui ne sont pas de la forme $\mathcal{M}[x \leftarrow t]$ ne peuvent pas être dans le membre de gauche de l'inclusion. On considère maintenant $\texttt{x=t}([\![\mathbf{WP}(\texttt{x=t}, \psi)]\!]) = \{\mathcal{M}[x \leftarrow t] \mid \mathcal{M} \models \mathbf{WP}(\texttt{x=t}, \psi)\} = \{\mathcal{M}[x \leftarrow t] \mid \mathcal{M} \models \psi[x \leftarrow t]\}$ (en appliquant les définitions). On peut montrer que $\{\mathcal{M}[x \leftarrow t] \mid \mathcal{M} \models \psi[x \leftarrow t]\} = \{\mathcal{M}[x \leftarrow t] \models \psi\}$ (par une induction sur les formules, que l'on laisse au lecteur), ce qui conclue la preuve.
- Les cas $*\texttt{p=t}$ et $\texttt{return t}$ sont similaires au cas d'affectations.

Cas d'inductions

- Supposons que $\{\varphi\}$`P1;P2`$\{\psi\}$ est un triplet valide. Par définition, on a `P1;P2`$([\![\varphi]\!]) \subseteq [\![\psi]\!]$, et donc `P2`$($`P1`$([\![\varphi]\!])) \subseteq [\![\psi]\!]$, par définition de la séquence. Par hypothèse d'induction (sur `P2`), on a `P1`$([\![\varphi]\!]) \subseteq [\![WP(\texttt{P2}, \psi)]\!]$, et donc $\{\varphi\}$`P1`$\{$WP$(\texttt{P2}, \psi)\}$ est un triplet valide. Par hypothèse d'induction (sur `P1`), on a $[\![\varphi]\!] \subseteq [\![WP(\texttt{P1},$ WP$(\texttt{P2}, \psi))]\!] = [\![WP(\texttt{P1;P2}, \psi)]\!]$, par définition de WP, ce qui conclue la preuve.
- Supposons que $\{\varphi\}$`if(test) P1 else P2`$\{\psi\}$. Par définition, on a `if(test) P1 else P2`$([\![\varphi]\!]) \subseteq [\![\psi]\!]$, et donc $\{P1(\mathcal{M}) \mid \mathcal{M} \models test \wedge \varphi\} \cup \{P2(\mathcal{M}) \mid \mathcal{M} \models \neg test \wedge \varphi\} \subseteq [\![\psi]\!]$, par définition du if then else. On a donc $\{\varphi \wedge test\}$`P1`$\{\psi\}$ qui est un triplet valide (similairement pour `P2`). Par hypothèse d'induction, on a $[\![\varphi \wedge test]\!] \subseteq [\![WP(\texttt{P1}, \psi)]\!]$ (de même pour `P2`). Cela est équivalent au fait que $(\varphi \wedge test) \Rightarrow$ WP$(\texttt{P1}, \psi)$ est valide. Cette dernière formule est logiquement équivalente à $\varphi \Rightarrow (test \Rightarrow$ WP$(\texttt{P1}, \psi))$, et en combinant avec `P2`, on obtient que la formule suivante est valide : $\varphi \Rightarrow (test \Rightarrow$ WP$(\texttt{P1}, \psi) \wedge \neg test \Rightarrow$ WP$(\texttt{P2}, \psi))$. Par définition de WP, on en déduit que $\varphi \Rightarrow$ WP$($`if (test) P1; else P2`$, \psi)$ est une formule valide, ce qui conclue la preuve.

□

La proposition suivante permet de simplifier le calcul de weakest precondition en la découpant en sous-formule (et justifie les différentes clauses en Frama-C) :

Proposition 4.3.3. *Pour tout programme* `P` *et formules* φ, ψ, WP$(P, \varphi \wedge \psi) \equiv$ WP$(P, \varphi) \wedge$ WP(P, ψ).

Démonstration. On démontre la proposition par induction sur `P`. On a WP$(\texttt{skip}, \varphi \wedge \psi) \equiv \varphi \wedge \psi$.

On a WP$(\texttt{x = t}, \varphi \wedge \psi) \equiv (\varphi \wedge \psi)[\texttt{x} \leftarrow \texttt{t}] \equiv \varphi[\texttt{x} \leftarrow \texttt{t}] \wedge \psi[\texttt{x} \leftarrow \texttt{t}]$, puisque cela consiste à remplacer `x` par `t` dans la formule. Les cas pour l'affectation de pointeur et le return sont identiques et donc omis.

Nous passons maintenant aux cas d'induction. L'hypothèse d'induction est que pour tous les programmes avec strictement moins d'instruction que celui considéré, la propriété est démontrée. On suppose donc, pour le `if` et la séquence que WP$(\texttt{P_1}, \varphi \wedge \psi) =$ WP$(\texttt{P_1}, \varphi) \wedge$ WP$(\texttt{P_1}, \psi)$, et de même pour `P_2`. On a donc :

$$\begin{aligned}
\text{WP}(\texttt{P_1;P_2}, \varphi \wedge \psi) &\equiv \text{WP}(\texttt{P_1}, \text{WP}(\texttt{P_2}, \varphi \wedge \psi)) \\
&\equiv \text{WP}(\texttt{P_1}, \text{WP}(\texttt{P_2}, \varphi) \wedge \text{WP}(\texttt{P_2}, \psi)) \\
&\equiv \text{WP}(\texttt{P_1}, \text{WP}(\texttt{P_2}, \varphi)) \wedge \text{WP}(\texttt{P_1}(\text{WP}(\texttt{P_2}, \psi)) \\
&\equiv \text{WP}(\texttt{P_1;P_2}, \varphi) \wedge \text{WP}(\texttt{P_1;P_2}, \psi)
\end{aligned}$$

Et pour le if-then-else :

$$
\begin{aligned}
&\mathbf{WP}(\texttt{if(test) P\_1;else P\_2}, \varphi \wedge \psi)\\
&\quad \equiv \texttt{test} \Rightarrow \mathbf{WP}(\texttt{P\_1}, \varphi \wedge \psi) \wedge \neg\texttt{test} \Rightarrow \mathbf{WP}(\texttt{P\_2}, \varphi \wedge \psi)\\
&\quad \equiv \texttt{test} \Rightarrow (\mathbf{WP}(\texttt{P\_1}, \varphi) \wedge \mathbf{WP}(\texttt{P\_1}, \psi))\\
&\qquad \wedge \neg\texttt{test} \Rightarrow (\mathbf{WP}(\texttt{P\_2}, \varphi) \wedge \mathbf{WP}(\texttt{P\_2}, \psi))\\
&\quad \equiv \texttt{test} \Rightarrow \mathbf{WP}(\texttt{P\_1}, \varphi) \wedge \neg\texttt{test} \Rightarrow \mathbf{WP}(\texttt{P\_2}, \varphi))\\
&\qquad \wedge (\texttt{test} \Rightarrow \mathbf{WP}(\texttt{P\_1}, \psi) \wedge \neg\texttt{test} \Rightarrow \mathbf{WP}(\texttt{P\_2}, \psi))\\
&\quad \equiv \mathbf{WP}(\texttt{if(test) P\_1; else P\_2}, \varphi)\\
&\qquad \wedge \mathbf{WP}(\texttt{if(test) P\_1; else P\_2}, \psi)
\end{aligned}
$$

□

Attention : il n'y a pas de propriété équivalente pour les connecteurs $\vee$ et $\neg$, car la preuve d'induction ne fonctionne pas pour le cas if-then-else !

4.4 Exemples et méthodologie

4.4.1 Conventions

Dans tous les exemples de programmes, on considèrera que les programmes ont lignes numérotées, et qu'il n'y a qu'une seule instruction par ligne. Cela permettra facilement de se repérer dans le programme et dans le calcul de WP. On notera en particulier, quand il n'y a pas d'ambiguité, $\mathbf{WP}(i-j, \psi)$ pour désigner le sous-programme commençant à l'instruction i et terminant après l'instruction j (à condition bien sûr que cela soit un sous-programme bien formé).

On considèrera qu'un `if` contient toujours un bloc `else`. C'est en particulier pour cela qu'on dispose de l'instruction `skip`.

4.4.2 Un exemple très simple

On considère la fonction `f` suivante :

```
1 int f(int a, int b){
2     return -b/a;
3 }
```

Considérons la formule du premier ordre $\psi_1 \equiv$ `\result`< 1. On va calculer $\mathbf{WP}(\texttt{f}, \psi_1)$. Comme c'est simplement un des cas de base, on applique simplement la règle correspondante, et on obtient :

$$\mathbf{WP}(\texttt{f}, \psi_1) \equiv \psi_1[\texttt{\result} \leftarrow -b/a] \equiv -b/a < 1 \equiv b > -a \wedge a \neq 0.$$

La dernière étape est simplement obtenue par des règles de l'arithmétique (on ne peut pas diviser par 0, donc pour ne pas perdre d'information, on précise donc dans la formule que $a \neq 0$). Le triplet de Hoare suivant est donc valide : $\{b > -a \wedge a \neq 0\}\texttt{f}\{\phi_1\}$. Autre exemple, $\{b == 2 \wedge a == 14\}\texttt{f}\{\phi_1\}$ est un triplet valide, puisque $(b == 2 \wedge a == 14) \Rightarrow (b > -a \wedge a \neq 0)$ est une formule valide.

Si on s'intéresse à une autre formule $\psi_2 \equiv \texttt{\result} == -b/a$, on aura de même :

$$\mathsf{WP}(\texttt{f}, \psi_2) \equiv \psi_2[\texttt{\result} \leftarrow -b/a] \equiv -b/a == -b/a \equiv \top.$$

4.4.3 Un deuxième exemple très simple

On considère la fonction `g` suivante :

```
1 int g(int a){
2     if(a > 2){
3         res = a;
4     }
5     else {
6         res = -a;
7     }
8     return res;
9 }
```

Considérons $\psi_1 \equiv \texttt{\result} \geq 0$. Pour calculer la précondition de cette formule, on effectue les étapes suivantes :

$$\begin{aligned}
\mathsf{WP}(\texttt{g}, \psi_1) &\equiv \mathsf{WP}(2-7, \mathsf{WP}(8, \psi_1)) \\
&\equiv \mathsf{WP}(2-7, \psi_1[\texttt{\result} \leftarrow \texttt{res}]) \\
&\equiv a > 2 \Rightarrow \mathsf{WP}(3, \psi_1[\texttt{\result} \leftarrow \texttt{res}]) \\
&\quad \wedge \neg(a > 2) \Rightarrow \mathsf{WP}(6, \psi_1[\texttt{\result} \leftarrow \texttt{res}]) \\
&\equiv a > 2 \Rightarrow \psi_1[\texttt{\result} \leftarrow \texttt{res}][\texttt{res} \leftarrow a] \\
&\quad \wedge \neg(a > 2) \Rightarrow \psi_1[\texttt{\result} \leftarrow \texttt{res}][\texttt{res} \leftarrow -a] \\
&\equiv a > 2 \Rightarrow (\texttt{res} \geq 0)[\texttt{res} \leftarrow a] \\
&\quad \wedge \neg(a > 2) \Rightarrow (\texttt{res} \geq 0)[\texttt{res} \leftarrow -a] \\
&\equiv (a > 2 \Rightarrow a \geq 0) \wedge (\neg(a > 2) \Rightarrow -a \geq 0) \\
&\equiv \top \wedge a > 2 \vee a \leq 0 \\
&\equiv a > 2 \vee a \leq 0
\end{aligned}$$

Pour que le résultat soit positif, il suffit donc que a soit négatif ou strictement plus grand que 2, *i.e.*, $\{a > 2 \vee a \leq 0\}\texttt{g}\{\texttt{\result} \geq 0\}$ est valide.

Si on considère une autre formule $\psi_2 \equiv \texttt{\result} \leq 42$. On peut remarquer que le début du calcul va être le même, puisqu'on n'avait pas simplifié ψ_1. Inutile donc de le refaire. Une idée importante, lorsqu'on a plusieurs formules à prouver sur le même programme, on peut effectuer une fois le

calcul sur la formule abstraite, puis partir de cet état pour chaque formule pour adapter le calcul. Par exemple, pour ψ_2, par le calcul précédent, on partir de la situation suivante :

$$\begin{aligned}
\mathbf{WP}(\mathtt{g}, \psi_2) &\equiv a > 2 \Rightarrow \psi_2[\backslash\mathtt{result} \leftarrow \mathtt{res}][\mathtt{res} \leftarrow a] \\
&\quad \wedge \neg(a > 2) \Rightarrow \psi_2[\backslash\mathtt{result} \leftarrow \mathtt{res}][\mathtt{res} \leftarrow -a] \\
&\equiv a > 2 \Rightarrow (\mathtt{res} \leq 42)[\mathtt{res} \leftarrow a] \\
&\quad \wedge \neg(a > 2) \Rightarrow (\mathtt{res} \leq 42)[\mathtt{res} \leftarrow -a] \\
&\equiv (a > 2 \Rightarrow a \leq 42) \wedge (\neg(a > 2) \Rightarrow -a \leq 42) \\
&\equiv (a \leq 2 \vee a \leq 42) \wedge (a > 2 \vee -a \leq 42) \\
&\equiv (a \leq 42) \wedge (a \geq -42) \\
&\equiv (-42 \leq a \leq 42)
\end{aligned}$$

On a donc $\{-42 \leq a \leq 42\}$ `g` $\{\backslash\mathtt{result} \leq 42\}$ qui est un triplet valide.

Supposons maintenant que l'on souhaite à la fois que le résultat soit compris entre 0 et 42, c'est-à-dire, calculer $\mathbf{WP}(\mathtt{g}, \psi_1 \wedge \psi_2)$. Il est inutile de refaire un calcul compliqué, on peut couper le calcul de WP grâce à la proposition 4.3.3. On a :

$$\begin{aligned}
\mathbf{WP}(\mathtt{g}, \psi_1 \wedge \psi_2) &\equiv (a > 2 \vee a \leq 0) \wedge (-42 \leq a \leq 42) \\
&\equiv (-42 \leq a \leq 0) \vee (2 < a \leq 42)
\end{aligned}$$

4.5 Positions dans le programme et valeurs des variables/pointeurs

4.5.1 Old et At

Jusqu'à présent, dans les programmes que nous avons présenté, nous n'avions pas de pointeurs et les valeurs passées en arguments n'étaient pas modifiées (ce qui est autorisé en C). Cependant la valeur des arguments passés en paramètres n'est pas modifié par un appel de fonction *vus de l'extérieur*. Cela veut dire que lorsqu'on fait référence à la valeur d'un argument dans une post-condition, on fait implicitement référence à sa valeur *avant l'appel de la fonction*. Lorsqu'on manipule des pointeurs par contre, leur valeur peut être modifiée par la fonction, et on va vouloir parler de la valeur *avant* l'appel et *après* l'appel. Cependant, si on reprend la définition de FO, on ne peut parler de la valeur d'un terme arithmétique que dans l'état de mémoire dans lequel la formule est évaluée. Cela ne permettra pas d'exprimer des contrats pourtant désirables tels que le swap des valeurs contenues dans deux pointeurs (et plus généralement, pour toutes fonction qui modifierait la valeur des arguments qu'elle a reçue, son contrat ne pourrait être exprimer correctement, ou plutôt, le calcul de Weakest Precondition n'aurait pas le comportement attendu).

Pour cela, on dispose (en Frama-C, mais également ici) de la notation old qui va modifier le statut du terme arithmétique qu'il contient : dans une *post-condition*, old(t) désignera la valeur d'un terme arithmétique *avant* l'appel de la fonction. On n'aura pas de notation spéciale pour les valeurs *après* l'appel de la fonction : dans les post-conditions, la sémantique par défaut d'un terme arithmétique est déjà cette valeur là, et dans la pré-condition, il est interdit d'en parler (puisqu'elle n'existe pas encore). Dans une précondition, on n'utilisera pas la notation old : la sémantique des termes les évaluera bien dans le contexte d'appel de la fonction. Dans les calculs de WP, ces termes seront traités comme des constantes (c'est-à-dire qu'il ne seront jamais modifiés en cours de calcul, puisqu'ils font référence à des valeurs présentes avant l'appel).

Cela permet d'exprimer la post-condition du contrat de `swap` dont on parlait plus haut : old$(*a) == *b \wedge$ old$(*b) == *a$. On donnera un exemple d'utilisation dans la suite de cette section.

En réalité, on va autoriser les formules à faire référence à la valeur d'un terme à un autre point du programme (sous certaines conditions qu'on détaille juste après). On dispose pour cela de la notation at(t, i) qui désignera la valeur de t à la ligne i du programme où l'annotation apparaît (en Frama-C, on désignera des labels, et pas des numéros de ligne). Comme old(t), at(t, i) est vu comme une constante pour WP et est remplacé par t lorsque le calcul atteint la ligne i (c'est-à-dire que WP$(i - j, \psi))$ ne contiendra pas at(t, i)). En particulier, old(t) est un raccourci pour at$(t, 0)$.

Dans une annotation du programme (un contrat ou un assert), on autorisera uniquement à faire référence à une position du programme située plus haut dans l'arbre de syntaxe abstraite (ou plus prosaïquement, plus haut dans le programme, et pas dans un sous-bloc de celui où l'annotation apparaît). L'idée est qu'on peut uniquement faire référence aux positions que toutes les branches du programmes empruntent pour arriver à l'endroit où l'annotation apparaît. Il est donc impossible de faire référence à une ligne dans un `if` ou dans une boucle depuis l'extérieur.

Cette notation at n'est évidemment pas autorisée dans un contrat de fonction (puisque le code n'est pas «visible» depuis l'extérieur), mais elle deviendra plus utile lorsqu'on écrira des invariants de boucle au chapitre suivant, ou pour des annotations locale (avec un assert dans Frama-C).

Formellement, il faut simplement modifier la Définition 4.3.1 de manière à avoir WP$(i - j, \phi) \equiv$ WP$'(i - j, \phi)[$at$(t, i) \leftarrow t]$, où WP$'$ est la formule donnée en Définition 4.3.1. De même WP$(P, \phi) \equiv$ WP$'(P, \phi)[$old$(t) \leftarrow t]$. Encore plus formellement, on obtient la définition suivante, qui sera la définition complète de WP :

Définition 4.5.1 (Weakest Precondition). *La weakest precondition d'un couple* (P, ψ) *est définie récursivement comme suit :*

- WP$(i : \texttt{skip}, \psi) \equiv \psi[at(i, s) \leftarrow s \mid s \in$ Arith$]$
- WP$(i : \texttt{x = t}, \psi) \equiv \psi[\texttt{x} \leftarrow \texttt{t}][at(i, s) \leftarrow s \mid s \in$ Arith$]$
- WP$(i : \texttt{P_1;}j : \texttt{P_2}, \psi) \equiv$ WP$(i :$ `P_1`, WP$(j :$ `P_2`$, \psi))$
- WP$(i :$ `if(test)`$j :$ `P_1; else` $k :$ `P_2`$, \psi) \equiv (\texttt{test} \Rightarrow$ WP$(j :$ `P_1`$, \psi) \wedge \neg\texttt{test} \Rightarrow$ WP$(k :$ `P_2`$, \psi))[$at$(i, s) \leftarrow s \mid s \in$ Arith$]$.
- WP$(i : \texttt{*p = t}, \psi) \equiv \psi[\texttt{*p} \leftarrow t][at(i, s) \leftarrow s \mid s \in$ Arith$]$.

— $\mathsf{WP}(i : \texttt{return } t, \psi) \equiv \psi[\backslash\texttt{result} \leftarrow t][\mathsf{at}(i, s) \leftarrow s \mid s \in \mathsf{Arith}]$.

où i, j, k désignent les positions du code (ou label) où apparaissent ces instructions.

Pour une fonction complète P on a :

$$\mathsf{WP}(P, \psi) = \mathsf{WP}(0 : P, \psi)[\mathsf{old}(s) \leftarrow s \mid s \in \mathsf{Arith}]$$

Cette définition est celle que nous appliquerons en présence de old et de at, sa formalisation est là pour vous montrer comment elle est censée s'appliquer. En l'absence de ces notations, la définition 4.3.1 sera bien évidemment suffisante.

En particulier, on remarquera que le remplacement des at arrive *après* l'application de la règle de la définition 4.3.1. Comme $\mathsf{at}(t, i)$ désigne la valeur de t *avant* l'instruction i, il est normal d'enlever la notation sur la formule obtenue avant cette instruction (ce qui ne serait pas le cas si on inversait l'ordre).

À noter également que dans cette définition, on suppose implicitement que la première ligne d'une fonction s'appelle toujours 0. Ce ne sera pas nécessairement le cas dans nos exemples où nous utiliserons des numéros de lignes à la place du numéro d'instruction dans la linéarisation du programme (ce qu'on devrait avoir en toute rigueur). On considèrera donc que dans la définition précédente il faut remplacer 0 par la première ligne du programme (par exemple, 4 dans l'exemple qui suit).

4.5.2 Un exemple : `swap`

Pour illustrer ce dont on vient de parler, prouvons le contrat de la fonction `swap`, en appliquant la nouvelle définition 4.5.1.

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

On va donc calculer $\mathsf{WP}(\texttt{swap}, \psi)$:

$$\begin{aligned}
\mathsf{WP}(\texttt{swap}, \psi) &\equiv \mathsf{WP}(4, \mathsf{WP}(5, \mathsf{WP}(6, \psi))) \\
&\equiv \mathsf{WP}(4, \mathsf{WP}(5, *a == \mathsf{old}(*b) \wedge aux == \mathsf{old}(*a))) \\
&\equiv \mathsf{WP}(4, *b == \mathsf{old}(*b) \wedge aux == \mathsf{old}(*a)) \\
&\equiv (*b == \mathsf{old}(*b) \wedge *a == \mathsf{old}(*a))[\mathsf{old}(s) \leftarrow s \mid s \in \mathsf{Arith}] \\
&\equiv *b == *b \wedge *a == *a \\
&\equiv \top
\end{aligned}$$

Et donc $\{\top\}\texttt{swap}\{\psi\}$ est un triplet de Hoare valide.

À noter que je n'ai indiqué explicitement la notation venant de la définition 4.5.1 que là où elle était utile (et ce pour éviter la surcharge de notation). Dans les calculs postérieurs, on pourra s'autoriser à faire les deux étapes de simplifications d'un coup (on passerait directement de la ligne 3 à la ligne 5).

On va, pour insister et montrer un exemple avec les at refaire le calcul avec la formule suivante : $\psi' \equiv *a == \mathsf{old}(b) \wedge *b \neq \mathsf{at}(*b, 5)$.

$$\begin{aligned}
\mathsf{WP}(\texttt{swap}, \psi') &\equiv \mathsf{WP}(4, \mathsf{WP}(5, *a == \mathsf{old}(*b) \wedge aux \neq \mathsf{at}(*b, 5))) \\
&\equiv \mathsf{WP}(4, *b == \mathsf{old}(*b) \wedge aux \neq \mathsf{at}(*b, 5))[\mathsf{at}(s, i) \leftarrow s \mid s \in \mathsf{Arith}]) \\
&\equiv \mathsf{WP}(4, *b == \mathsf{old}(*b) \wedge aux \neq *b)) \\
&\equiv *b == \mathsf{old}(*b) \wedge *a \neq *b[\mathsf{old}(s) \leftarrow s \mid s \in \mathsf{Arith}] \\
&\equiv *b == *b \wedge *a \neq *b \\
&\equiv *a \neq *b
\end{aligned}$$

Ce qui donne que $\{*a \neq *b\}\texttt{swap}\{\psi'\}$ est aussi un triplet valide. Évidemment, il faut se souvenir qu'il est normalement impossible d'utiliser $\mathsf{at}(*b, 5)$ dans une postcondition, j'ai pris ça pour garder un exemple simple. L'intérêt des at se fera plus crucialement ressentir en présence de boucle (on aura naturellement à écrire des formules à l'intérieur du programme).

4.6 Validité et séparation des pointeurs

Dans nos parties théoriques, on supposera par défaut que tous les pointeurs sont toujours bien définis, contrairement à Frama-C où il faut le supposer explicitement (avec \valid, cf. les TPs). Mais pour ce point, il y aura dans le cadre de ce cours (et dans la plupart des cas concret de vérification), une bonne heuristique qui consistera simplement à mettre comme précondition que tous les pointeurs manipulés par une fonction sont valide.

Un autre point auquel faire attention pour les pointeurs, c'est la séparation des zones mémoires. En effet, il n'y a a priori aucune raison de supposer qu'un fonction n'a pas reçu en argument deux fois le même pointeur. Cependant, quand ce cas arrive, la fonction peut avoir un comportement assez différent du cas où les zones pointées sont séparées.

Donnons un court exemple de fonction dans lequel on va toucher le problème du doigt :

```
1 /*@ ensures Psi: *a == *b+\old(*a);*/
2 void sumPoint(int* a, int* b){
3     *a = *a+*b;
4 }
```

Cette fonction semble avoir un triplet de Hoare complètement évident, $\{\top\}\texttt{sumPoint}\{\psi\}$, mais une réflexion moins immédiate vous montrera que si j'appelle la fonction dans le contexte suivant :

```
1 int *a = 4;
2 sumPoint(a,a);
```

alors la post-condition est fausse, puisque la formule $*a == *b + \mathsf{old}(*a)$ est équivalente à $8 == 8 + 4$ (ce qui est bien sûr faux).

Pourtant, si on applique la définition 4.5.1, on obtient bien $\top$ comme plus faible précondition :

$$\begin{aligned}
\mathsf{WP}(3, \psi) &\equiv ((*a == *b + \mathsf{old}(*a))[*a \leftarrow *a + *b])[\mathsf{old}(s) \leftarrow s \mid s \in \mathsf{Arith}] \\
&\equiv (*a + *b == *b + \mathsf{old}(*a))[\mathsf{old}(s) \leftarrow s \mid s \in \mathsf{Arith}] \\
&\equiv (*a + *b == *b + *a) \\
&\equiv \top
\end{aligned}$$

Dans cet exemple précis, il y aurait évidemment un remède très simple pour avoir une post-condition correcte : l'écrire $*a == \mathsf{old}(*b + *a)$, mais ce n'est d'abord pas la même (et de toutes façons, on ne pourrait pas prouver que la valeur de $*b$ reste inchangée), et on aura évidemment des problèmes bien plus sérieux quand on va parler de tableaux.

Quel est le problème ? Hé bien c'est «simplement» que notre logique est incapable en l'état de modéliser le fait que des pointeurs stockées dans des zones différentes peuvent pointer dans la même zone mémoire. Ce n'est pas un problème simple à résoudre et sa résolution est bien au delà du champ de ce présent cours. Mais il existe bien une solution : la logique de séparation.

À notre niveau, dans les preuves papier, nous ferons donc les mêmes simplifications que notre logique (c-à-d, les fonctions seront toujours appelées sur des zones mémoire séparées), et dans le cadre des TPs, comme Frama-C manipule partiellement la logique de séparation, on prêtera attention à spécifier chaque fois que nécessaire que les zones mémoires manipulées sont séparées grâce au prédicat \separated. D'une manière générale, dans la plupart des cas concrets, le comportement attendu d'une fonction a souvent besoin de supposer que les pointeurs qu'elle a reçu en argument sont séparés, donc cette discipline qu'on adoptera est une bonne règle générale (et cela permet par ailleurs de penser à le spécifier ce qui peut être utile à rappeler à un programmeur étourdi).

Chapitre 5

Les boucles : correction partielle

5.1 Introduction

Les boucles (ou de manière équivalente pour les langages fonctionnels, la récursion) augmentent strictement l'expressivité d'un langage. Si on prend notre langage C_1, il est par exemple évident que tout programme y termine, alors que vous savez que la terminaison d'un programme est en général indécidable. Mais si on considère le langage C_2, la terminaison d'un programme n'est plus systématique, comme le montre par exemple le programme suivant : `while(0 == 0) skip;`. Le langage C_2 étant Turing-complet, la terminaison d'un programme écrit avec est indécidable.

Concrètement, cela veut dire que la sémantique d'un programme de C_2 (cf. chapitre 3) n'est pas aussi facile à déterminer qu'un programme de C_1, puisque le sémantique d'une boucle n'est pas assurée d'être bien définie dans tous les états de mémoire : la sémantique du programme précédent n'est jamais définie. En revanche, la sémantique de `while(i < 0) i = i-1;` est définie pour les états de mémoire où `i` est supérieur à 0, mais pas pour ceux où il est négatif.

En terme de vérification des triplets de Hoare (qu'on peut toujours définir sans problème), cela va nous amener à distinguer deux possibilités :

- La *correction totale* (i.e., ce qu'on avait au chapitre 4) qui consiste à déterminer si un triplet est valide ET que le programme termine. C'est-à-dire, pour tout programme `P` et formule ψ, pour tout état de mémoire $\mathcal{M}$, si $\mathcal{M} \models \mathbf{WP}(\mathtt{P}, \psi)$, alors `P` termine et $\mathtt{P}(\mathcal{M}) \models \psi$.
- La *correction partielle* qui consiste à déterminer si un triplet est valide SI le programme termine. C'est-à-dire, pour tout programme `P` et formule ψ, pour tout état de mémoire $\mathcal{M}$, si $\mathcal{M} \models \mathbf{WP}(\mathtt{P}, \psi)$, alors, si `P` termine sur $\mathcal{M}$, on aura $\mathtt{P}(\mathcal{M}) \models \psi$.

Dans ce chapitre, nous nous concentrerons sur la correction partielle.

5.2 Weakest liberal precondition et invariants de boucles

Contrairement aux instructions de C_1, il n'est pas possible de déterminer directement la weakest precondition d'un programme constitué d'une boucle. Très grossièrement, on pourrait dire que cela est dû au fait que l'on ne connaît (a priori) pas le nombre d'itérations de la boucle nécessaires pour obtenir un certain état de mémoire. Considérons par exemple le programme `while(i > 0) i = i-1;`, si on considère la formule $\psi = (i == 0)$, alors, si la boucle est exécutée k fois, on a en début de boucle un état de mémoire vérifiant $i == k$, et ce évidemment pour tout k. Dans ce cas, on peut facilement voir que la plus faible précondition de ψ est en réalité $i \geq 0$, et on pourrait imaginer un procédé calculatoire pour l'obtenir (du genre passage à la limite), mais d'autre boucles ont des comportement plus complexes. Par exemple, considérons le programme `while(i<n) {res = res + i; i = i+1;}`, Si on veut calculer la précondition de $\psi \equiv (res == n \times (n+1)/2)$, qui se trouve être $(res == n \times (n+1)/2 - (n-i) \times (n-i+1)/2) \wedge i \leq n) \vee (res == n \times (n+1)/2 \wedge i > n)$, alors il faudrait un procédé qui rajoute i dans les formules alors qu'elle n'est pas mentionnée dans la postcondition, et on peut se douter qu'on obtiendrait en réalité difficilement une telle forme. En général, il n'est pas possible d'obtenir calculatoirement la weakest precondition d'une boucle.

Pour contourner cette difficulté, il va falloir aider «à la main» le calcul de weakest precondition, en fournissant pour chaque boucle des *invariants de boucle*.

Définition 5.2.1. *Étant donné un programme L = `while(test) P;` un invariant de boucle de L est une formule I telle que $\{I \wedge test\}P\{I\}$ est un triplet de Hoare valide.*

Informellement, un invariant de boucle est une formule qui, si elle est vraie au début d'un tour de boucle, sera vraie après un tour de boucle (avec la restriction qu'on ne nécessite que le triplet soit valide que si le test d'entrée de la boucle est satisfait).

Pour déterminer la validité d'un triplet de Hoare, l'idée va donc être d'indiquer pour chaque boucle d'un programme, de prouver qu'il s'agit bien d'un invariant, que cet invariant ainsi que la condition de sortie de boucle implique la weakest precondition de la post-condition par la fin du programme, et que la weakest precondition de l'invariant par le début du programme est impliqué par la précondition. Pour cela (et pour clarifier, car la phrase précédente est probablement très dure à parser), on va définir le concept de *weakest liberal precondition*. On va considérér dans la suite que les boucles while sont annotées par des invariants. On notera cela `while(test)@I@ P;` pour une formule I dans les définitions pour simplifier (dans les exemples, nous les expliciterons séparémment, ou utiliserons la syntaxe de Frama-C).

Définition 5.2.2. *Étant donné un programme P de C_2 annoté par des invariants de boucles, et une formule ψ, on définit la weakest liberal pre-*

condition de ψ par P, $\mathbf{WLP}(P, \psi)$ inductivement comme le calcul de WP (cf. Definition 4.5.1), plus le cas suivant :

$\mathbf{WLP}(\texttt{while(test)@I@ P;}, \psi)) = I$, si les deux propriétés suivantes sont vraies :

— Pour tout état de mémoire $\mathcal{M}$, $\mathcal{M} \models \texttt{test} \wedge I \Rightarrow \mathbf{WLP}(P, I)$
— Pour tout état de mémoire $\mathcal{M}$, $\mathcal{M} \models \neg\texttt{test} \wedge I \Rightarrow \psi$.

Dit autrement, I est vrai en début de boucle, si I est vrai au début d'un tour de boucle, alors il est vrai à la fin du tour, et I et la condition de fin de boucle sont suffisants pour avoir ψ.

Si les deux conditions précédentes ne sont pas vraies, alors $\mathbf{WLP}(\texttt{while(test)@I@P;}, \psi$ n'est pas défini.

À noter que contrairement à WP sur C_1, où obtenir le résultat de WP nécessite un peu de calcul, mais est toujours défini, le résultat de WLP sur une boucle est connu d'avance (c'est son annotation), mais ce qui est à prouver, c'est que ce résultat existe bien.

On parle de weakest liberal precondition car, contrairement au cas sans boucle, on sera incapable de démontrer qu'on a bien la plus faible précondition pour le programme et la post-condition : cela dépend des annotations.

On a par contre le théorème suivant :

Théorème 5.2.3. *Étant donné un programme P de C_2, une formule ψ et P' le programme P avec des annotations de boucles. Si $\mathbf{WLP}(P', \psi)$ est défini, alors $\{\mathbf{WLP}(P', \psi)\}P\{\psi\}$ est valide.*

Dit autrement, si des annotations permettent de construire une weakest liberal precondition, on obtient un triplet de Hoare valide quelles que soient les annotations (ce qui n'est pas surprenant, les annotations ne modifient pas le comportement du programme, elles sont simplement une aide à la preuve).

Par contre, on ne peut pas avoir d'équivalent au théorème 4.3.2. En effet, puisque la précondition obtenue dépend des annotations, on n'a pas de garantie qu'il n'existe pas d'annotation donnant une précondition plus faible pour satisfaire cette postcondition. Pire, on n'est même pas sûre qu'il existe des annotations permettant de déterminer la plus faible précondition, ni même qu'il existe une plus faible précondition (il pourrait exister une séquence infinie de précondition toutes plus faibles les unes que les autres mais que cette séquence n'ait pas de limite, ou que la limite ne satisfasse pas la postcondition). Mais bon, le cas précédent est pathologique et pas très intéressant dans le cadre de ce cours. On notera juste que le résultat est un peu moins fort que sans boucle, mais que cela nous permet tout de même de démontrer la validité de contrats de fonctions, ce qui est bien ce que l'on cherche.

Déterminer des invariants de boucles est une tâche complexe, au centre de ce qu'est la vérification de programmes. On pourra consulter [3] pour une discussion de ce concept et des problèmes liés.

5.3 Exemples et méthodologie

On utilisera les mêmes conventions qu'au chapitre 4. De plus, on pourra inclure, par lisibilité, les annotations de boucles et les contrats de fonction avec une syntaxe Frama-C au-dessus des boucles annotées (ou la syntaxe décrite plus haut dans ce chapitre).

À noter que dans la plupart de ces exemples, on fournira les invariants de boucle, et les contrats de fonction. Bien évidemment, vous pourrez avoir des fonctions à vérifier où ils ne vous sont pas fournis. Il faudra alors déterminer ceux qui vous aiderons à prouver la spécification de la fonction.

5.3.1 Addition à la con

On considère le programme suivant :

```
1  /*@ ensures Psi: \result == x + y;
2  */
3  int stupidAdd(int x, int y){
4      /*@ loop invariant I1: y >= 0;
5          loop invariant I2: x + y == \old(x) + \old(y);
6      */
7      while(y > 0){
8          x = x+1;
9          y = y-1;
10     }
11     return x;
12 }
```

Ici, on utilise les même conventions que Frama-C. À savoir :
— La postcondition est en réalité : $\psi \equiv \texttt{\textbackslash result} == \mathsf{old}(x) + \mathsf{old}(y)$.
— On met les noms des invariants/contrats.
— Le fait d'avoir deux invariants signifie que l'invariant est $I_1 \wedge I_2$.

On souhaite déterminer $\mathsf{WLP}(\texttt{stupidAdd}, \psi)$.

Pour cela, on commence par appliquer ce qu'on a vu au chapitre 4, la construction de plus haut niveau étant une séquence, ici. On a :

$$\begin{aligned}\mathsf{WLP}(\texttt{stupidAdd}, \psi) &= \mathsf{WLP}(4-10, \mathsf{WLP}(11, \psi)) \\ &= \mathsf{WLP}(4-10, x == \mathsf{old}(x) + \mathsf{old}(y))\end{aligned}$$

$4-10$ étant une boucle, on applique maintenant la construction vue plus haut : on a $\mathsf{WLP}(4-10, x == \mathsf{old}(x) + \mathsf{old}(y)) \equiv x + y == \mathsf{old}(x) + \mathsf{old}(y) \wedge y \geq 0$. Comme on va l'évaluer au début du programme, on a $\mathsf{old}(x) == x$ (par définition), et donc $\mathsf{WLP}(4-10, x == \mathsf{old}(x) + \mathsf{old}(y)) \equiv y \geq 0$. Et on doit prouver deux contraintes de plus pour que cela soit défini :

Lemme 5.3.1. $\neg y > 0 \wedge I_1 \wedge I_2 \Rightarrow x == \mathsf{old}(x) + \mathsf{old}(y)$.

Démonstration. On a :

$$\begin{aligned}
&(\neg y > 0 \wedge y \geq 0 \wedge x + y == \mathsf{old}(x) + \mathsf{old}(y)) \Rightarrow x == \mathsf{old}(x) + \mathsf{old}(y) \\
\equiv\ &(y == 0 \wedge x + 0 == \mathsf{old}(x) + \mathsf{old}(y)) \Rightarrow x == \mathsf{old}(x) + \mathsf{old}(y) \\
\equiv\ &\top
\end{aligned}$$

Cette formule est donc bien vraie dans tous les état de mémoire, donc l'invariant suffit bien pour prouver $\mathsf{WLP}(11, \psi)$ à la sortie de la boucle. □

Lemme 5.3.2. $y > 0 \wedge I_1 \wedge I_2 \Rightarrow \mathsf{WLP}(8 - 9, I_1 \wedge I_2)$.

Démonstration. On va procéder comme au chapitre 4 puisqu'il s'agit d'un programme sans boucle. On a :

$$\begin{aligned}
\mathsf{WLP}(8 - 9, I_1 \wedge I_2) &\equiv (I_1 \wedge I_2)[\mathtt{y} \leftarrow y - 1][\mathtt{x} \leftarrow x + 1] \\
&\equiv y - 1 \geq 0 \wedge x + 1 + y - 1 == \mathsf{old}(x) + \mathsf{old}(y) \\
&\equiv y > 0 \wedge x + y == \mathsf{old}(x) + \mathsf{old}(y)
\end{aligned}$$

Il reste donc maintenant à prouver qu'on a bien tout le temps le point 2 valide :

$$(y > 0 \wedge y \geq 0 \wedge x + y == \mathsf{old}(x)) \Rightarrow (y > 0 \wedge x + y == \mathsf{old}(x))$$

On laissera au lecteur le soin de prouver le fait que cette formule est bien une tautologie :-). □

On en déduit que $\mathsf{WLP}(\mathtt{stupidAdd}, \psi) \equiv (I_1 \wedge I_2)[\mathsf{old}(s) \leftarrow s \mid s \in \mathsf{Arith}] \equiv y \geq 0$, avec ces annotations de boucles. On a donc que $\{y \geq 0\}\mathtt{stupidAdd}\{\psi\}$ est un triplet de Hoare valide (et ce quelles que soient les annotations de boucles).

5.3.2 Somme d'entiers

Regardons un exemple un peu moins trivial, une fonction calculant la somme des entiers de 1 à n. Ici n n'étant pas modifié par la fonction, on s'autorise à éluder la notation old (que l'on devrait en toute rigueur utiliser).

```
1  /*@ ensures Psi: \result == n * (n+1) / 2;
2  */
3  int integerSum(int n){
4      int sum = 0;
5      int i = 0;
6      /*@ loop invariant I1: i <= n;
7          loop invariant I2: sum == i * (i+1) / 2;
8      */
9      while(i < n){
10         i = i+1;
11         sum = sum + i;
```

```
12      }
13      return sum;
14  }
```

Calculons $\mathbf{WLP}(\texttt{integerSum}, \psi)$. On a :

$$\begin{aligned}
\mathbf{WLP}(\texttt{integerSum}, \psi) &\equiv \mathbf{WLP}(4-5, \mathbf{WLP}(9-12, \mathbf{WLP}(13, \psi))) \\
&\equiv \mathbf{WLP}(4-5, \mathbf{WLP}(9-12, sum == n \times (n+1)/2)) \\
&\equiv \mathbf{WLP}(4-5, i \leq n \wedge sum == i \times (i+1)/2) \\
&\equiv 0 \leq n \wedge 0 == 0 \times (0+1)/2 \\
&\equiv 0 \leq n
\end{aligned}$$

La plus faible précondition avec ces annotations est donc, si elle est définie, $0 \leq n$.

Il nous reste à démontrer que $\mathbf{WLP}(9-12, sum == n \times (n+1)/2)$ est bien définie. Pour cela, il nous faut prouver les deux lemmes qui suivent.

Lemme 5.3.3. $\neg i < n \wedge I_1 \wedge I_2 \Rightarrow sum == n \times (n+1)/2$.

Démonstration. On a :

$$\begin{aligned}
&\neg i < n \wedge I_1 \wedge I_2 \Rightarrow sum == n \times (n+1)/2 \\
\equiv\ & i \geq n \wedge i \leq n \wedge sum == i \times (i+1)/2 \Rightarrow sum == n \times (n+1)/2 \\
\equiv\ & i == n \wedge sum == i \times (i+1)/2 \Rightarrow sum == n \times (n+1)/2 \\
\equiv\ & i == n \wedge sum == n \times (n+1)/2 \Rightarrow sum == n \times (n+1)/2 \\
\equiv\ & \top
\end{aligned}$$

On a, entre la troisième et la quatrième ligne substitué i par n, ce qu'on peut faire puisqu'ils sont égaux (on est sur une équivalence de formule, pas une égalité syntaxique). On est arrivé à une tautologie, puisque que pour tous p, q, $p \wedge q \Rightarrow p$ est une tautologie. □

Lemme 5.3.4. $i < n \wedge I_1 \wedge I_2 \Rightarrow \mathbf{WLP}(10-11, I_1 \wedge I_2)$.

Démonstration. Commençons par calculer $\mathbf{WLP}(10-11, I_1 \wedge I_2)$:

$$\begin{aligned}
\mathbf{WLP}(10-11, I_1 \wedge I_2) &\equiv \mathbf{WLP}(10, i \leq n \wedge sum + i == i \times (i+1)/2) \\
&\equiv i+1 \leq n \wedge sum + i + 1 == (i+1) \times (i+1+1)/2 \\
&\equiv i+1 \leq n \wedge sum == ((i+1) \times (i+2) - 2 \times (i+1))/2 \\
&\equiv i+1 \leq n \wedge sum == ((i+1) \times i)/2
\end{aligned}$$

Remarquons que $i < n \Rightarrow i+1 \leq n$.
Remarquons aussi que $I_2 = sum == i \times (i+1)/2$.
On a donc bien $i < n \wedge I_1 \wedge I_2 \Rightarrow i+1 \leq n \wedge sum == (i \times (i+1))/2$. □

Avec ces deux lemmes, on peut donc conclure que $\mathbf{WLP}(10-11, sum == n \times (n+1)/2)$ est bien défini, et donc que le calcul de weakest liberal precondition effectué plus haut est correct.

On peut en conclure que $\{0 \leq n\}\texttt{integerSum}\{\psi\}$ est un triplet de Hoare valide.

5.3.3 Multiplication russe

On regarde maintenant le programme suivant, qui implémente une technique de multiplication (dite russe) qui est efficace à la main (on ne parlera pas de son utilité en tant que programme).

```
1  /*@ ensures Psi: \result == p * q;
2  */
3  int russianMult(int p, int q){
4      int r = 0;
5      /*@ loop invariant I1: q >= 0;
6          loop invariant I2: r + p * q == \old(p) * \old(q);
7      */
8      while(q>0){
9          if(q%2 == 1){
10             r = r + p;
11         }
12         else;
13         p = p + p;
14         q = q / 2;
15     }
16     return r;
17 }
```

Comme lors de l'exemple précédent, on va calculer $\mathbf{WLP}(\texttt{russianMult}, \psi)$. On a :

$$\begin{aligned}
\mathbf{WLP}(\texttt{russianMult}, \psi) &\equiv \mathbf{WLP}(4, \mathbf{WLP}(5-15, \mathbf{WLP}(16, \psi))) \\
&\equiv \mathbf{WLP}(4, \mathbf{WLP}(5-15, r == \mathbf{old}(p) * \mathbf{old}(q))) \\
&\equiv \mathbf{WLP}(4, I_1 \wedge I_2) \\
&\equiv (I_1 \wedge I_2)[r \leftarrow 0] \\
&\equiv q \geq 0 \wedge 0 + p * q == \mathbf{old}(p) * \mathbf{old}(q) \\
&\equiv q \geq 0
\end{aligned}$$

Il faut maintenant démontrer que $\mathbf{WLP}(5-15, r == \mathbf{old}(p) * \mathbf{old}(q))$ est bien défini, puisque $5-15$ est une boucle annotée :

1. $(\neg q > 0 \wedge I_1 \wedge I_2) \Rightarrow (r == \mathbf{old}(p) * \mathbf{old}(q))$
2. $(y > 0 \wedge I_1 \wedge I_2) \Rightarrow (\mathbf{WLP}(9-14, I_1 \wedge I_2))$

Lemme 5.3.5. $(\neg q > 0 \wedge I_1 \wedge I_2) \Rightarrow (r == \mathbf{old}(p) * \mathbf{old}(q))$.

Démonstration. On a :

$$\begin{aligned}
&((\neg q > 0 \wedge I_1 \wedge I2) \Rightarrow (r == \mathbf{old}(p) * \mathbf{old}(q))) \\
\equiv\ &((q \leq 0 \wedge q \geq 0 \wedge (r + p * q == \mathbf{old}(p) + \mathbf{old}(q))) \Rightarrow (r == \mathbf{old}(p) + \mathbf{old}(q))) \\
\equiv\ &((q == 0 \wedge (r + p * 0 == \mathbf{old}(p) + \mathbf{old}(q))) \Rightarrow (r == \mathbf{old}(p) * \mathbf{old}(q))) \\
\equiv\ &\top
\end{aligned}$$

On a dans ce calcul, substitué q par 0 entre la deuxième et la troisième ligne (puisqu'on a au même moment déterminé que $y == 0$), et pour le passage à la quatrième ligne, remarqué que pour tout p, q, $p \wedge q \Rightarrow p$ est une tautologie. □

Lemme 5.3.6. $(y > 0 \wedge I_1 \wedge I_2) \Rightarrow (\mathsf{WLP}(9 - 14, I_1 \wedge I_2))$.

Démonstration. On commence par calculer la weakest precondition du corps de la boucle pour une formule I (qu'on remplacera ensuite par I_1 et I_2) :

$$\begin{aligned}
\mathsf{WLP}(9-14, I) &\equiv \mathsf{WLP}(9-12, \mathsf{WLP}(13, \mathsf{WLP}(14, I))) \\
&\equiv \mathsf{WLP}(9-12, \mathsf{WLP}(13, I[q \leftarrow q/2])) \\
&\equiv \mathsf{WLP}(9-12, I[q \leftarrow q/2][p \leftarrow p+p]) \\
&\equiv (q\%2 == 1) \Rightarrow \mathsf{WLP}(10, I') \wedge (q\%2 \neq 1) \Rightarrow I' \\
&\equiv (q\%2 == 1) \Rightarrow I'[r \leftarrow r+p] \wedge (q\%2 \neq 1) \Rightarrow I'
\end{aligned}$$

où, dans l'expression précédente, $I' = I[q \leftarrow q/2][p \leftarrow p+p]$.

Terminons maintenant le calcul pour I_2. On a :

$$\begin{aligned}
I_2[q \leftarrow q/2] &\equiv r + p * q/2 == \mathsf{old}(p) + \mathsf{old}(q) \\
I_2' \equiv I_2[q \leftarrow q/2][p \leftarrow p+p] &\equiv r + (p+p) * q/2 == \mathsf{old}(p) + \mathsf{old}(q) \\
I_2'' \equiv I_2'[r \leftarrow r+p] &\equiv (r + p + (p+p) * q/2 == \mathsf{old}(p) + \mathsf{old}(q)
\end{aligned}$$

De la même manière (en plus simple, donc éludons), on a $I_1' = I_1'' = q/2 \geq 0$.

Il nous reste à démontre qu'on a bien un invariant maintenant. On le prouve séparémment pour I_1 et I_2 pour simplifier les formules, et alléger les notations. On ne développera dans les calculs que les formules utiles (pour les mêmes raisons).

$$\begin{aligned}
& q > 0 \wedge I_1 \wedge I_2 \Rightarrow \mathsf{WLP}(9-14, I_1) \\
\equiv\ & q > 0 \wedge q \geq 0 \wedge I_2 \Rightarrow ((q\%2 == 1 \Rightarrow q/2 \geq 0) \wedge (q\%2! = 1) \Rightarrow q/2 \geq 0) \\
\equiv\ & q > 0 \wedge I_2 \Rightarrow q/2 \geq 0
\end{aligned}$$

Il suffit maintenant de remarquer que le quotient d'un entier positif par 2 est positif, ce qui permet de dire que la formule précédente est une tautologie. Pour passer de la deuxième à la troisième ligne, on a simplement remarquer que pour tous p, q, $p \Rightarrow q \wedge \neg p \Rightarrow q$ est équivalent à q.

Pour I_2,

$$\begin{aligned}
& q > 0 \wedge I_1 \wedge I_2 \Rightarrow \mathsf{WLP}(9-14, I_2) \\
\equiv\ & q > 0 \wedge I_2 \Rightarrow (((q\%2 == 1 \Rightarrow r + p + (2 * p * q/2) == \mathsf{old}(p) + \mathsf{old}(q)) \\
& \wedge ((q\%2 \neq 1) \Rightarrow r + (2 * p * q/2) == \mathsf{old}(p) + \mathsf{old}(q))) \\
\equiv\ & (q > 0 \wedge I_2 \wedge q\%2 == 1) \Rightarrow r + p + (2 * p * q/2) == \mathsf{old}(p) + \mathsf{old}(q) \\
& (q > 0 \wedge I_2 \wedge q\%2 == 0) \Rightarrow r + (2 * p * q/2) == \mathsf{old}(p) + \mathsf{old}(q)
\end{aligned}$$

Pour conclure ici, on a besoin de faire un peu d'arithmétique : si q est impair, alors $2 * p * q/2 = p * q - p$ (puisque $q = 2 * (q/2) + 1$). Donc, le conséquent de

la première implication devient, en simplifiant $r + p * q == \text{old}(p) + \text{old}(q)$, ce qui est exactement I_2, et donc la première implication est vraie. Si q est pair, alors $2 * p * q/2 = p * q$, et de la même manière, on en déduit la véracité de la deuxième implication.

Ainsi, on a bien $q > 0 \wedge I_1 \wedge I_2 \Rightarrow \mathbf{WLP}(9 - 14, I_1 \wedge I_2)$, et donc on a bien un invariant de boucle. □

On déduit de tout cela que $\mathbf{WLP}(\texttt{russianMult}, \psi)$ est bien défini, et que $\{q \geq 0\}\texttt{russianMult}\{\psi\}$ est un triplet de Hoare valide.

5.3.4 Division euclidienne

Pour cet exemple, on va stocker les résultats dans des pointeurs. Dans cet exemple, on va considérer pour les spécifications les versions euclidiennes du quotient et du reste (notés respectivement $/_e$ et $\%_e$) pour avoir une preuve simple. On rapelle que en C, le quotient et la division ne coincident pas avec la division euclidienne (pour les nombres négatifs, grossièrement).

```
1  /*@ ensures Psi1: *q == a /e b;
2      ensures Psi2: *r == a %e b;
3  */
4  void euclidianDiv(int a, int b,int* q, int* r){
5      *q = 0;
6      *r = a;
7      /*@ loop invariant I1: *r >= 0;
8          loop invariant I2: a == b * *q + *r;
9      */
10     while(*r >= b){
11         *r = *r - b;
12         *q = *q + 1;
13     }
14     return;
15 }
```

On note $\psi \equiv \psi_1 \wedge \psi_2$ On va calculer $\mathbf{WLP}(\texttt{euclidianDiv}, \psi)$. On a :

$$\begin{aligned}
\mathbf{WLP}(\texttt{euclidianDiv}, \psi) &\equiv \mathbf{WLP}(5, \mathbf{WLP}(6, \mathbf{WLP}(10 - 13, \psi))) \\
&\equiv \mathbf{WLP}(5, \mathbf{WLP}(6, I_1 \wedge I2)) \\
&\equiv \mathbf{WLP}(5, a \geq 0 \wedge a == b \times *q + a) \\
&\equiv a \geq 0 \wedge a == b \times 0 + a \\
&\equiv a \geq 0
\end{aligned}$$

Comme 10-13 est une boucle, il reste deux lemmes à démontrer pour pouvoir conclure que $\mathbf{WLP}(\texttt{euclidianDiv}, \psi)$ est bien définie (et donc égale à $a \geq 0$).

Lemme 5.3.7. $\neg(*r \geq b) \wedge I_1 \wedge I_2 \Rightarrow \psi$.

Démonstration. On a :

$$\begin{aligned}&\neg(*r \geq b) \wedge I_1 \wedge I_2 \Rightarrow \psi\\ \equiv\ & 0 \leq *r < b \wedge a == b \times *q + *r \Rightarrow *q == a/_e b \wedge *r == a\%_e r\end{aligned}$$

Ici (allons vite), il suffit de remarquer que par définition de la division euclidienne, le quotient et le reste de a par b sont définis comme les uniques entiers tels que $a = b * a/_e b + a\%_e r$ et $0 \leq *r < b$. Comme c'est exactement ce qu'on a à gauche de l'implication, alors, par définition, la droite de l'implication est vraie. □

Lemme 5.3.8. $(*r \geq b) \wedge I_1 \wedge I_2 \Rightarrow \mathbf{WLP}(11 - 12, I_1 \wedge I_2)$.

Démonstration. Calculons d'abord $\mathbf{WLP}(11 - 12, I_1 \wedge I_2)$.

$$\begin{aligned}\mathbf{WLP}(11 - 12, I_1 \wedge I_2) &\equiv \mathbf{WLP}(11, *r \geq 0 \wedge a == b \times (*q + 1) + *r)\\ &\equiv *r - b \geq 0 \wedge a == b \times (*q + 1) + *r - b\\ &\equiv *r \geq b \wedge a == b \times q + *r\end{aligned}$$

Il reste donc à remarquer que $\mathbf{WLP}(11 - 12, I_1 \wedge I_2)$ est exactement $*r \geq b \wedge I_2$, ce qui rend l'implication à démontrer trivialement vraie. □

5.4 Comment trouver un bon invariant

Cette petite section est là pour vous aider à déterminer les invariants avec des astuces simples. Sachant qu'il n'existe pas de technique universelle connue (sinon on aurait des programmes automatisant leur détermination dans tous les cas – voir [3] pour plus de détails).

La condition de boucle La condition d'un while est en général une très bonne aide pour trouver un premier invariant. En effet, souvent, la valeur de l'expression contrôlant l'arrêt de la boucle est bien déterminée en fin de boucle, et est utile pour démontrer que la post-condition est vraie à ce moment-là.

Attention cependant, un invariant n'est pas la condition du while de la boucle, mais une version élargie de celle-ci (puisqu'il faut que l'invariant soit encore vrai quand la condition du while est devenue fausse). Ainsi, en général, si on a `while(x < n) P;` et que la variable x est augmentée de d à chaque tour de boucle, alors un bon invariant de boucle sera $x < n + d$. Et en fin de boucle, on pourra déduire que $n \leq x < n + d$.

Que doit-on démontrer ? Pour les autres invariants, la règle à garder en mémoire est qu'ils doivent être utiles à la démonstration de la post-condition. Pour cela, il est important de commencer par calculer la weakest precondition de la fin du programme pour savoir quelle formule doit être vraie en fin de boucle. À partir de cette formule, on peut déterminer à quel endroit apparaît le terme de la condition de la boucle (ou quels termes sont en fait égaux à icelui). Puis, en regardant le comportement de la boucle, il faut essayer de déterminer quelles sont les formules qui seraient vraies à chaque tour de boucle et reviendraient à la formule à démontrer à la fin. Par exemple, si en fin de boucle on a une formule disant que toutes les cases d'un tableau sont égales à 0, et que la boucle les remplit une à une, un bonne invariant sera une formule disant que toutes les cases vues lors des tours de boucles précédent sont déjà à 0 (évidemment, on ne peut pas exprimer directement une condition du style «les cases vues lors des tours précédent», cela dépend de comment le tableau est exploré : typiquement si le tableau est lu de gauche à droite, alors ce seront les cases entre 0 et $x-1$ qui sont déjà à 0).

Pour le reste Il n'y a au-delà des deux conseils précédent, pas vraiment de conseil universel. Parfois, il faudra ajouter des invariants qui, s'ils n'aident pas à démontrer la post-condition, sont utiles uniquement pour prouver les autres invariants. La seule règle dans ce cas-là est de regarder ce qu'on doit démontrer, d'analyser le comportement de la boucle et de déterminer les régularités qui pourraient aider à démontrer ce que l'on cherche.

Un point extrêmement important à garder en mémoire quand vous déterminez des invariants (et quand vous les testez sur Frama-C) : quand on prouve qu'un invariant en est un, on suppose *uniquement* les invariants qu'on a écrit. Ainsi, si vous écrivez ce qui suit :

```
1 j = 0;
2 /*@ loop invariant I1: 0 <= i <= n;
3     loop invariant I2: j <= 2 * n;
4 */
5 while (i<n){
6     i = i-1;
7     j = j+2;
8 }
```

Il sera impossible de prouver que I_2 est un invariant ! En effet, si on essaie, on obtient que $\text{WLP}(6-7, j \leq 2\times n) \equiv j+2 \leq 2\times n$. Et la formule $i < n \wedge 0 \leq i \leq n \wedge j \leq 2\times n \Rightarrow j+2 \leq 2\times n$ est fausse (si $j == 2n-1$, par exemple). Pensez à avoir des invariants qui sont suffisamment précis.

5.5 Loop assigns

Comme on l'a vu plus haut, quand on rencontre une boucle, on est obligé de passer par des invariants pour prouver sa correction, et les invariants doivent être exhaustifs, dans le sens où la propriété à démontrer correcte en début de boucle sera la conjonction des invariants, et seulement celle-ci. De plus, la conjonction des invariants et la condition de fin de boucle doivent pouvoir permettre de démontrer la précondition de la fin du programme.

Dans les exemples que l'on a vu précédemment, cela n'a jamais posé de problème, car ils sont simples, et les variables étaient toutes modifiées par la boucle. Cependant, ça ne sera pas toujours le cas. Par exemple, considérons le bout de code suivant (repris d'un exemple précédent, mais avec un assert modifié) :

```
1 /*@ loop invariant I1: i <= n;
2     loop invariant I2: sum == i * (i+1) / 2;
3 */
4 while(i < n){
5     i = i+1;
6     sum = sum + i;
7 }
8 //@ assert sum = n * (n+1)/2 && a == 42;
```

Comme on le voit, la seule modification que l'on a faite est d'ajouter la contrainte $a == 42$ à la formule à démontrer. En gardant les mêmes invariants, alors on ne peut pas démontrer cette assertion, puisque les invariants ne parlant pas de a, on ne peut pas en déduire que $a == 42$ en fin de boucle. Selon ce qu'on a dit plus haut, le remède est simple : ajouter un invariant de boucle $a == 42$ qui sera immédiat à démontrer et qui demandera donc de démontrer $a == 42$ en début de boucle.

Cependant, cela est lourd à faire, ajoute des invariants qui parlent de variables n'apparaissant pas dans la boucle, et rajoute du travail de preuve inutile. D'un point de vue utilisateur (dans Frama-C), cela a le gros désavantage de demander à l'utilisateur d'ajouter des invariants artificiels, ce qui complique sa tâche (déjà peu simple).

Pour simplifier tout cela, Frama-C permet d'utiliser une clause qui s'appelle *loop assigns*. Cette clause consiste en une liste de variables (ou zones mémoires) qui seront les seules à pouvoir être modifiées par la boucle. Par exemple `loop assigns a,t[1],c;` dira que la boucle ne modifie aucun zone mémoire exceptées les variables locales a et c et la valeur pointée par $(t + 1)$. Attention, cela ne veut pas dire que les dites valeurs seront forcément modifiées, cela veut simplement dire qu'aucune autre ne le sera. On verra plus de détails là-dessus en TP.

Pour prouver qu'une clause loop assigns est satisfaite, on regardera quelles sont les variables qui reçoivent une affectation dans la boucle (ce qui est grossièrement ce que fait Frama-C[1]). Concrètement, si l'on fait des

1. C'est évidemment légèrement plus compliqué que ça, mais ça suffira comme approximation

preuves sur papier, on admettra de telles clauses (ce sera sur des codes simples), si elles sont correctes bien sûr.

L'intérêt principal de ces clauses est qu'elle permettra de ne pas écrire des invariants dont aucune des variables n'est modifiée par la boucle. Concrètement, on pourra considérer qu'une boucle `while` est annotée par un invariant I et par un ensemble de variables et d'adresses mémoire S : `while(test)@I,S@ P`. On pourra considérer une modification de la règle présentée Définition 5.2.2. On conserve évidemment les conditions donnant la bonne définition de WLP (en y ajoutant le fait que le loop assigns soit vrai), mais on va conserver les parties de la formule qui ne contiennent pas de variables dans S. Formellement, on définit alors $\mathbf{WLP}(\texttt{while(test)@I,S@ P;},\varphi)$ inductivement comme suit (en notant `while(test)@I,S@ P;` par `W`) :

- $\mathbf{WLP}(\texttt{W},\varphi) \equiv I$ si $\mathrm{Var}_\varphi \cap S \neq \emptyset$, $I \wedge \neg\texttt{test} \Rightarrow \varphi$ et I est bien un invariant de boucle. Sinon, il n'est pas défini (cf Definition 5.2.2).
- $\mathbf{WLP}(\texttt{W},\varphi) \equiv \varphi$ si $\mathrm{Var}_\varphi \cap S = \emptyset$, si S inclue bien l'ensemble des valeurs modifiées.
- $\mathbf{WLP}(\texttt{W},\phi_1 \wedge \phi_2) \equiv \mathbf{WLP}(\texttt{W},\phi_1) \wedge \mathbf{WLP}(\texttt{W},\phi_2)$, si $\mathrm{Var}_{\phi_1} \cap S = \emptyset$ ou $\mathrm{Var}_{\phi_2} \cap S = \emptyset$
- $\mathbf{WLP}(\texttt{W},\phi_1 \vee \phi_2) \equiv \mathbf{WLP}(\texttt{W},\phi_1) \vee \mathbf{WLP}(\texttt{W},\phi_2)$, si $\mathrm{Var}_{\phi_1} \cap S = \emptyset$ ou $\mathrm{Var}_{\phi_2} \cap S = \emptyset$
- $\mathbf{WLP}(\texttt{W},\phi_1 \Rightarrow \phi_2) \equiv \mathbf{WLP}(\texttt{W},\phi_1) \Rightarrow \mathbf{WLP}(\texttt{W},\phi_2)$, si $\mathrm{Var}_{\phi_1} \cap S = \emptyset$ ou $\mathrm{Var}_{\phi_2} \cap S = \emptyset$

Je laisse un peu sous le tapis que pour avoir la précondition la plus faible possible, il faut appliquer les deux premières règles uniquement aux formules qui ne contiennent pas de sous-formules ne contenant pas de variables de S et construire la précondition par les cas d'inductions[2]. Et évidemment, cela est globalement équivalent à rajouter des invariants (ceux qu'on cherche à éviter d'ajouter).

Dans le cas de la deuxième règle, il est évidemment inutile de démontrer que l'invariant implique φ, d'autant que ce sera très probablement faux. En fait, il est même inutile de vérifier que l'invariant en est bien un dans ce cas (en gros si on a une boucle qui n'a aucun effet sur le contrat qu'on cherche à démontrer, on peut complètement ignorer la boucle).

Si on reprend l'exemple de cette section, on aura $\mathbf{WLP}(4-7, sum == n\times(n+1)/2 \wedge a == 42) \equiv i \leq n \wedge sum == i\times(i+1)/2 \wedge a == 42$.

Bon, globalement, il est peu probable qu'on utilise concrètement cela dans les preuves que l'on fera sur papier, mais c'est je pense utile de l'écrire pour mieux comprendre comment fonctionne Frama-C sur ce point.

5.6 Appel de fonction

Le traitement des appels de fonctions est très similaire à celui des boucles, dans le sens où, de la même manière, le contenu de la fonction

2. En fait, je laisse pas mal sous le tapis, car c'est un peu plus compliqué que ça : ce que j'ai présenté ici ne permet pas de savoir quoi faire quand on compare une valeur de S et une valeur hors de S (alors que c'est possible). Mais passons, cette simplification est suffisante pour comprendre un peu le truc.

est traité comme une boîte noire, et la précondition ne sera définie que si la fonction dispose d'un contrat valide. Cela permet des preuves modulaires, c'est-à-dire qu'on peut prouver une fonction indépendamment de ses contextes d'appels et vice-versa. C'est d'ailleurs bien ce qui se passe avec les boucles. On aura également pour les fonctions une clause assigns qui décrit les zones mémoires pouvant être modifiées par la fonction par effet de bord (la seule différence avec le cas des boucles étant donc que les variables locales de la fonction appelante ne peuvent jamais être modifiées).

Formellement, si on a une fonction `f(x1,...,xk)` dont on a démontré le triplet $\{\varphi\}$`f(x1,...,xk)`$\{\psi\}$ et qu'on a démontré que les seules valeurs modifiées par effet de bord sont dans l'ensemble S, alors on peut définir WLP pour l'appel de fonction de la manière suivante :

- $\mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau) \equiv \varphi$ si $\mathbf{Var}_\tau \cap S \neq \emptyset$ et $\psi \Rightarrow \tau$. Sinon, il n'est pas défini.
- $\mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau) \equiv \tau$ si $\mathbf{Var}_\tau \cap S = \emptyset$, si S inclue bien l'ensemble des valeurs modifiées.
- $\mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau_1 \wedge \tau_2) \equiv \mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau_1) \wedge \mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau_2)$, si $\mathbf{Var}_{\tau_1} \cap S = \emptyset$ ou $\mathbf{Var}_{\tau_2} \cap S = \emptyset$
- $\mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau_1 \vee \tau_2) \equiv \mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau_1) \vee \mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau_2)$, si $\mathbf{Var}_{\tau_1} \cap S = \emptyset$ ou $\mathbf{Var}_{\tau_2} \cap S = \emptyset$
- $\mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau_1 \Rightarrow \tau_2) \equiv \mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau_1) \Rightarrow \mathbf{WLP}(\texttt{f(x1,...,xk)}, \tau_2)$, si $\mathbf{Var}_{\tau_1} \cap S = \emptyset$ ou $\mathbf{Var}_{\tau_2} \cap S = \emptyset$

Pour le cas des fonctions, on peut encore mieux comprendre l'intérêt de ce mécanisme de assigns. Il permet de donner un unique contrat à une fonction, et que celui-ci n'ai pas besoin de parler des valeurs non-modifiées par la fonction pour être utilisé par du code appelant.

Il y a évidemment ici les mêmes simplifications que précédemment (notamment sur le fait que ce que je présente ne permet pas de traiter le cas où on compare une zone mémoire modifiée avec une qui ne l'est pas, alors qu'on sait quoi faire dans ce cas). Mais cette simplification suffit pour comprendre le phénomène. Par ailleurs la discussion concernant la logique de séparation effectuée au chapitre 4 est également valable ici pour les effets de recouvrement de pointeurs. On supposera dans toutes nos preuves que les espaces mémoires sont tous séparés, et on pensera à le préciser en TP.

Chapitre 6

Les boucles : terminaison

6.1 Introduction

Comme on l'a déjà rapellé, la terminaison d'un programme est un problème indécidable. Mais, le fait qu'un problème soit indécidable ne veut pas dire qu'on ne peut pas déterminer son résultat sur certaines instances. De manière caricaturale, il est facile de voir qu'un programme sans boucles termine, et que le programme `while(true) skip;` ne termine jamais. Mais on peut heureusement faire mieux que ça. Le but de ce chapitre est de présenter une condition suffisante pour qu'un programme termine : la présence d'un variant de boucle. Il ne vous surprendra donc pas que l'absence d'un variant ne pourra pas être une garantie de non-terminaison.

6.2 Variants de boucles

6.2.1 Ordres bien fondés

Pour présenter les variants et justifier leur utilité, nous allons utiliser un concept mathématique simple : les ordres bien fondés.

Définition 6.2.1. *Étant donné un ensemble E et un ordre $\leq_E$, on dit que $\leq_E$ est un* ordre bien fondé *si toute séquence strictement décroissante d'éléments de E est finie.*

Ou de manière équivalente, pour toute séquence infinie d'éléments de E, $e_1, e_2, \cdots$, il existe $i < j$ tels que $e_i \leq_E e_j$.

Par exemple, on peut facilement voir que $(\mathbb{N}, \leq)$ est un ordre bien fondé, puisque pour tout entier n, il existe un nombre fini d'entiers plus petits (exactement n). Toute suite strictement décroissante commençant par n a donc au plus $n+1$ éléments.

L'ensemble des paires d'entiers naturels $\mathbb{N} \times \mathbb{N}$ muni de l'ordre lexicographique est également bien fondé.

L'ensemble $(\mathbb{Z}, \leq)$ n'est pas bien fondé. En effet, la suite $0, -1, -2, -3, \cdots$ est strictement décroissante et infinie.

6.2.2 Variants

On va se servir du fait que $(\mathbb{N}, \leq)$ est un ordre bien fondé pour définir les variants.

Définition 6.2.2. *Étant donné une boucle annotée* `while(test)@I@ P;`*, un variant de boucle pour cette boucle est un terme arithmétique* $V \in$ Arith *vérifiant les conditions suivantes :*

— $test \wedge I \Rightarrow V \geq 0$

— $test \wedge I \Rightarrow \mathbf{WLP}(P, V < \mathsf{at}(V, P_0))$*, où* P_0 *désigne la première ligne de* `P`*.*

Dit autrement, un variant est une valeur entière qui reste positive, et décroit strictement à chaque tour de boucle.

Théorème 6.2.3. *Si* $V \in$ Arith *est un variant pour la boucle annotée* `while(test)@I@ P;`*, et que* I *est bien un invariant pour cette boucle, alors* `while(test) P;` *termine sur toutes ses exécutions.*

Démonstration. À faire un peu plus proprement plus tard, mais en gros :

Supposons que la boucle a une exécution infinie et admette un variant V. Alors en notant $V_0, V_1, V_2, \cdots$ la valeur de V après le tour numéro i, on a pour tout i, $V_i \geq 0$ et $V_i > V_{i+1}$. On a donc une suite décroissante infinie dans $\mathbb{N}$, ce qui est impossible. □

Variants généralisés Il est bien évidemment possible de définir une notion de variant généralisé, en donnant une valeur V appartenant à un ensemble E muni d'un ordre bien fondé. V sera un variant d'une boucle s'il vérifie la même propriété de décroissance que sur les entiers.

Pour info, Frama-C donne la possibilité d'écrire de tels variants, mais n'est pas à ce jour capable de prouver des variants généralisés (uniquement des variants sur $\mathbb{N}$).

6.3 Exemples et méthodologie

6.3.1 Comment trouver un bon variant ?

Trouver un variant est en général beaucoup plus aisé que trouver des invariants. En effet, dans la quasi-totalité des boucles, regarder la condition du while est une très bonne indication de la valeur qui va décroître.

Quand cette condition ne donne pas directement la solution, demandez-vous pourquoi la fonction va terminer nécessairement (parcours d'une structure, calculs qui convergent nécessairement, etc).

Mais dans la plupart des cas, soit le variant est évident à trouver, soit il n'y en a pas.

6.3.2 Terminaison de la somme d'entiers

On reprend la somme d'entiers présentée en section 5.3.2. On va montrer qu'elle termine en exhibant un variant de la boucle qui y est présente (10-11). On pose $V = n - i$. Pour montrer que c'est un variant de la boucle 10-11, il nous faut montrer qu'il est toujours positif à l'entrée de la boucle, et qu'il décroit strictement à chaque tour de boucle.

Lemme 6.3.1. $i < n \wedge I_1 \wedge I_2 \Rightarrow n - i \geq 0$.

Démonstration. Trivial. □

Lemme 6.3.2. $i < n \wedge I_1 \wedge I_2 \Rightarrow \mathbf{WLP}(10 - 11, n - i < \mathbf{at}(n - i, 10))$.

Démonstration. Commençons par calculer $\mathbf{WLP}(10 - 11, n - i < \mathbf{at}(n - i, 10))$. On a :

$$\begin{aligned}\mathbf{WLP}(10 - 11, n - i < \mathbf{at}(n - i, 10)) &\equiv n - (i + 1) < \mathbf{at}(n - i, 10) \\ &\equiv n - i - 1 < n - i \\ &\equiv -1 < 0 \\ &\equiv \top\end{aligned}$$

Ce qui permet donc directement de conclure que l'implication est vraie. □

On vient de montrer que $n - i$ est un variant pour la boucle 10-11. Par conséquent, celle-ci termine quelles que soient les valeurs en entrées de la boucle. Et comme c'est la seule boucle du programme, il termine toujours.

6.3.3 Terminaison de la multiplication russe

On reprend à présent la multiplication russe de la section 5.3.3, et on va montrer que cette fonction termine. On pose $V = q$.

On laissera au lecteur le soin de démontrer que V reste toujours positif et qu'il décroît à chaque tour de boucle.

6.3.4 Terminaison de la division euclidienne : où l'on voit qu'il faut parfois rajouter des informations.

On reprend la division euclidienne présentée en section 5.3.4. On aimerai montrer qu'elle termine en exhibant un variant. On pose $V = *r$. Vérifions si c'est bien un variant de la boucle 11-12, en démontrant les deux lemmes suivants.

Lemme 6.3.3. $*r \geq b \wedge I_1 \wedge I_2 \Rightarrow *r \geq 0$.

Démonstration. Il suffit de remarquer que $I_1 = *r \geq 0$. □

Lemme 6.3.4. $*r \geq b \wedge I_1 \wedge I_2 \Rightarrow \mathsf{WLP}(11 - 12, *r < \mathsf{at}(*r, 11))$.

Commençons par calculer la partie droite de l'implication :

$\mathsf{WLP}(11 - 12, *r < \mathsf{at}(*r, 11)) \equiv (*r - b < \mathsf{at}(*r, 11))[\mathsf{at}(t, 11) \leftarrow t \mid t \in \mathsf{Arith}] \equiv *r - b < *r \equiv b > 0$.

On doit donc avoir $*r \geq b \wedge *r \geq 0 \wedge a == b \times *q + *r \Rightarrow b > 0$, ce qui, en général n'est pas vrai !

Que se passe t'il ici ? On vient de s'apercevoir que notre variant n'est décroissant que si b est strictement positif. Et que donc le triplet de Hoare calculé au chapitre précédent est trop faible pour assurer la terminaison de la fonction.

Pour avoir la terminaison, il faut ajouter $I3 : b > 0$ comme invariant de boucle, ce qui donnera donc au final $\mathsf{WLP}(\texttt{divEucl}, \psi) \equiv a \geq 0 \wedge b > 0$, pour avoir la correction totale.

Au passage, il est facile (dans cette exemple) qui si $b \leq 0$, alors la boucle ne terminera jamais : en effet, à chaque tour de boucle $*r$ augmentera (ou ne changera pas de valeur pour $b == 0$). Mais encore une fois, ce n'est pas nécessairement le cas (on pourrait avoir des boucles sans variant identifiable qui terminent).

Il faut donc faire attention : il est parfois possible de prouver la correction partielle d'une fonction dans une hypothèse plus large que celle où elle termine. Ce qu'on a fait au chapitre précédent reste correct : il suffit que a soit positif pour que, si la fonction termine, la fonction calcule la division euclidienne. Ce qu'on a démontré ici, c'est que pour qu'elle termine, il faut de plus que b soit strictement positif.

6.3.5 Un non-exemple : la suite de Syracuse

Considérons la fonction suivante :

```
1  int syracuseSequence(int a){
2      /*@ loop invariant I: a >= 1;
3      */
4      while(a != 1){
5          if(a % 2 == 0){
6              a = a/2;
7          }
8          else {
9              a = 3 * a + 1;
10         }
11     }
12     return a;
13 }
```

Pour cette fonction, on ne démontrera pas de contrat (il n'y en a pas d'intéressant), et on admettra l'invariant I (vous pouvez le démontrer en exercice).

Cette fonction implémente la suite de Syracuse, dont la terminaison pour tout entier est à ce jour inconnue. Ainsi, on ne pourra pas ici donner

de variant : si nous en étions capable, nous démontrerions la conjecture de Collatz qui dit que cette suite termine pour toute valeur initiale, et sans vouloir préjuger de nos capacités, c'est un peu ambitieux.

Chapitre 7

Un exemple complet

Dans ce chapitre, on s'intéresse à la fonction qui effectue une recherche dichotomique dans un tableau, et on va la spécifier et la démontrer entièrement.

Point important, dans les exercices, je vous guiderai bien évidemment, et éviterai de vous donner des fonctions aussi complexes.

7.1 Spécification

7.1.1 Spécification formelle de la post-condition : recherche dans un tableau

On souhaite une fonction prenant un entrée un tableau `tab`, la taille de ce table `size`, ainsi qu'une valeur cible `target`, et qui renvoit un entier. Cet entier est :

— une position du tableau `index` telle que `tab[index] == target` s'il existe au moins une telle position.

— -1 sinon.

On peut noter que le comportement souhaité dépend de l'existence ou non d'un élément égal à `target` dans le tableau. On exprime donc d'abord une propriété désignant ce fait en FO :

$$\text{ExistElement}(tab, size, target) ::= \exists x; 0 \leq x < size \land tab[x] == target$$

La traduction en logique des deux proposition précédentes est la suivante :

— Found $::=$ ExistElement(`tab`,`size`,`target`) $\Rightarrow 0 \leq$ `\result` $<$ `size`$\land$`tab[\result]` $==$ `target`

— NotFound $::= \neg$ExistElement(`tab`,`size`,`target`) $\Rightarrow$ `\result` $== -1$.

Pour arriver à cette formalisation, il faut juste écrire formellement les propriétés qui nous intéressent. Ainsi, «le tableau contient une position qui vaut target» s'écrit bien comme le membre de droite de la première implication.

La conjonction des deux implications que l'on cherche est donc la spécification d'une fonction qui renvoie la position d'un élément égal à `target`, et -1 s'il n'y en a pas.

7.1.2 Implémentation : la recherche dichotomique

On propose ci-dessous une implémentation d'une recherche dans un tableau dont on va montrer qu'elle satisfait à la spécification déterminée plus haut. On y écrit celle-ci en syntaxe Frama-C, en privilégiant une syntaxe avec comportements : Ici, c'est pertinent puisque le comportement de la fonction est très différent s'il existe un élément ou pas.

```
1   /*@ behavior Found:
2           assumes ExistElement(tab,size,target);
3           ensures InBound: 0 <= \result < size;
4           ensures IsTarget: tab[\result] == target;
5       behavior NotFound:
6           assumes !ExistElement(tab,size,target);
7           ensures ErrorValue: \result == -1;
8   */
9   int dichotomicSearch(int* t, int size, int target){
10      low = 0;
11      high = size-1;
12      while(low < high){
13          mid = (low+high)/2;
14          if(tab[mid] < target){
15              low = mid+1;
16          }
17          else{
18              high = mid;
19          }
20      }
21      if(tab[low] == target){
22          res = low;
23      }
24      else{
25          res = -1;
26      }
27      return res;
28  }
```

Informellement, cette implémentation recherche l'existence d'un élément par dichotomie : il regarde l'élément médian du tableau, si celui-ci est plus grand que la cible, on réitère le procédé sur la partie gauche du tableau, sinon on réitère sur la partie droite. On itére le procédé jusqu'à ce que le tableau regardé ait taille 1. À la sortie de la boucle, il suffit de regarder la valeur de l'unique case de ce tableau de taille un : si c'est la cible, on le renvoie, sinon, c'est qu'il n'y a pas de telle valeur.

Comme vous le savez sans doute, un tel algorithme ne peut fonctionner que si le tableau est trié. On aura donc très probablement besoin d'un prédicat indiquant qu'un tableau est trié. On définit donc le prédicat Sorted :

$$\mathsf{Sorted}(tab, size) ::= \forall x, y; 0 \leq x < y < size \Rightarrow tab[x] <= tab[y]$$

À noter un point subtil : cette formule n'est pas la seule formule qui définit correctement le prédicat Sorted. En effet, un prédicat disant que pour tout indice x entre 0 et $size-2$, $tab[x] \leq tab[x+1]$ définit la même chose. Par contre, cette dernière définition contient moins d'informations, c'est-à-dire que les preuves avec cette définition demandent un peu plus de déductions qu'avec celles qu'on a choisie. À la main, on ne s'apercevrait sans doute pas de la différence (puisqu'on admettra facilement qu'elles sont équivalentes). Cependant, quand on commence à jouer avec des SMT-solveurs, c'est une autre histoire : ils ont particulièrement des difficultés pour effectuer des preuves par induction, et c'est exactement ce qu'il faut faire pour passer de la seconde version à la première. Et donc si on choisit la seconde, Frama-C n'arrive pas à prouver la fonction dont on discute ici (du moins, avec Alt-Ergo), alors qu'avec la première, on y arrive aisément.

7.1.3 Ce qui doit être vrai en fin de boucle

Pour pouvoir déterminer nos invariants, il faut commencer par déterminer quelles sont les formules qui doivent être vraies en fin de boucle. On calcule donc $\mathsf{WLP}(21-27, \psi)$, pour $\psi \equiv \mathsf{Found} \wedge \mathsf{NotFound}$. On a :

$$\begin{aligned}
\mathsf{WLP}(21-27, \psi) &\equiv \mathsf{WLP}(21-26, \psi[\backslash\texttt{result} \leftarrow res]) \\
&\equiv tab[low] == target \Rightarrow \mathsf{WLP}(22, \psi[\backslash\texttt{result} \leftarrow res]) \\
&\quad \wedge \neg tab[low] == target \Rightarrow \mathsf{WLP}(23, \psi[\backslash\texttt{result} \leftarrow res]) \\
&\equiv tab[low] == target \Rightarrow \psi[\backslash\texttt{result} \leftarrow res][res \leftarrow low] \\
&\quad \wedge \neg tab[low] == target \Rightarrow \psi[\backslash\texttt{result} \leftarrow res][res \leftarrow -1]
\end{aligned}$$

En coupant le calcul en deux, on a les deux formules suivantes :

$$\begin{aligned}
\mathsf{WLP}(21-27, \mathsf{Found}) \equiv\ & tab[low] == target \Rightarrow \mathsf{ExistElement}(tab, size, target) \\
& \Rightarrow 0 \leq low < size \wedge tab[low] == target \\
& \wedge tab[low] \neq target \Rightarrow \mathsf{ExistElement}(tab, size, target) \\
& \Rightarrow 0 \leq -1 < size \wedge tab[-1] == target \\
\equiv\ & tab[low] == target \Rightarrow 0 \leq low < size \\
& \wedge tab[low] \neq target \Rightarrow \neg\mathsf{ExistElement}(tab, size, target)
\end{aligned}$$

La dernière ligne a été obtenue par des simplifications syntaxiques simples.

$$\begin{aligned}\text{WLP}(21-27, \text{NotFound}) \equiv\ & tab[low] == target \Rightarrow \neg\text{ExistElement}(tab, size, target)\\ & \Rightarrow low == -1\\ & \wedge\, tab[low] \neq target \Rightarrow \neg\text{ExistElement}(tab, size, target)\\ & \Rightarrow -1 == -1\\ \equiv\ & tab[low] == target \wedge \neg\text{ExistElement}(tab, size, target)\\ & \Rightarrow low == -1\end{aligned}$$

7.1.4 Des invariants de boucle

Pour continuer la pleuve de la fonction, il nous faut déterminer des invariants de boucle. On explique ici comment les déterminer informellement (bien évidemment, il faudra ensuite les démontrer).

Premier invariant assez immédiat, celui qui concerne la condition d'arrêt de la boucle. Ici la condition d'arrêt de la boucle est `low < high`, et à chaque étape leur différence est divisée par 2 (grossièrement). À la sortie de la boucle, on devrait donc avoir `low == high`, donc notre premier invariant sera : $I_1 = low \leq high$. En regardant la post-condition Found, on peut observer que la valeur de `\result` doit être entre `0` et `size`, et on sait que ce résultat sera la valeur de `low` dans ce cas. On va donc inclure ces bornes dans I_1 pour obtenir $I_1 = 0 \leq low \leq high < size$.

Deuxième invariant qui va être nécéssaire assez facilement : comme discuté en section précédente, pour que l'algorithme fonctionne, le tableau doit être trié. On va donc l'ajouter comme invariant : I_2 : Sorted$(tab, size)$. En réalité ici, comme le tableau n'est pas modifié, cet invariant sera trivial. En Frama-C, on n'aurait même pas besoin de l'écrire (il est capable de faire usage des préconditions pour démontrer des invariants). Dans le formalisme présenté dans ce cours, il est cependant nécessaire de le considérer comme invariant.

Enfin, on va mettre un autre invariant permettant de démontrer qu'en fin de boucle, on est bien sur une position contenant `target`, s'il existe un tel élément. Son but est d'assurer que si il y a un élément, alors il y en a un entre low et $high$. Notre invariant est donc $I_3 = ExistElement(tab, size, target) \Rightarrow tab[low] \leq target \leq tab[high]$.

Le bloc de code à placer comme commentaire de la boucle est donc, en Frama-C :

```
1     /*@ loop invariant I1: 0 <= low <= high < size;
2         loop invariant I2: Sorted(tab,size);
3         loop invariant I3: ExistElement(tab,size,target)
4             ==> tab[low] <= target <= tab[high];
5     */
```

7.1.5 Un variant ?

Ici, on peut voir assez facilement que `high - low` restera toujours strictement positif, et qu'à chaque tour de boucle il décroîtra strictement. C'est donc un bon candidat pour un variant. On pose donc $V = high - low$.

7.2 Preuve de la fonction

On va donc maintenant calculer un triplet de Hoare pour cette fonction avec les annotations que nous venons de déterminer, et en déduire si la fonction est correcte (ou plus précisément, dans quelles hypothèses elle est correcte).

7.2.1 Correction

On va calculer la weakest liberal precondition de la fonction avec ces annotation (si elle est définie). On calcule la dite precondition sur une formule abstraite ψ, qu'on instanciera lorsqu'on en aura besoin dans le calcul.

Par simplification on notera $I \equiv I_1 \wedge I_2 \wedge I_3$.

On a :

$$\begin{aligned}
\mathbf{WLP}(dichotomicSearch, \psi) &\equiv \mathbf{WLP}(10-11, \mathbf{WLP}(12-20, \mathbf{WLP}(21-27, \psi))) \\
&\equiv \mathbf{WLP}(10-11, I) \\
&\equiv I[high \leftarrow size - 1][low \leftarrow 0] \\
&\equiv 0 \leq size - 1 \wedge \mathbf{Sorted}(tab, size) \\
&\quad \wedge (\forall j; 0 \leq j < 0 \Rightarrow tab[j] \leq target) \\
&\quad \wedge (\forall j; size - 1 < j < size \Rightarrow tab[j] \geq target) \\
&\equiv 0 \leq size - 1 \wedge \mathbf{Sorted}(tab, size)
\end{aligned}$$

Pour le passage de la première ligne à la seconde, on a simplement remarqué que comme 12-20 est une boucle, si $\mathbf{WLP}(12{-}20, \varphi)$ est défini, c'est I, quelque soit la formule φ. Pour le passage de la troisième à la quatrième ligne, on peut remarquer que les deux implications ont un membre gauche toujours faux, et sont donc vraies.

On a donc que la weakest liberal precondition, si elle existe est $0 \leq size - 1 \wedge \mathbf{Sorted}(tab, size)$.

On va maintenant démontrer que $\mathbf{WLP}(12{-}20, \mathbf{WLP}(21{-}27, \mathbf{Found} \wedge \mathbf{NotFound}))$ est bien définie. Par convenance, on va couper le lemme montrant que les invariants sont suffisant pour obtenir la post-condition en deux.

Lemme 7.2.1. $\neg low < high \wedge I \Rightarrow \mathbf{WLP}(21 - 27, \mathbf{Found})$.

Démonstration. On coupe encore WLP$(21-27, \text{Found})$ en deux. On commence par remarquer que :

$$low \geq high \wedge I \Rightarrow tab[low] == target \Rightarrow 0 \leq low < size$$

est trivialement vraie puisque $I_1 \equiv 0 \leq low \leq high < size$.

On considère maintenant :

$$low \geq high \wedge I \Rightarrow tab[low] \neq target \Rightarrow \neg\text{ExistElement}(tab, size, target)$$

Pour démontrer que cette formule est vraie, on va utiliser l'invariant I_3, qui nous dit que s'il y a une valeur égale à target, il y en a une entre low et $high$, et I_1 et $low \geq high$ qui nous donne $low == high$. Avec ces deux fait, on n'a pas de valeur égale à $target$ entre low et $high$, et que si il existe une valeur égale à $target$ il y en a une entre low et $high$. On peut en déduire qu'il n'y a pas de valeur égale à $target$, et donc $\neg$ExistElement$(tab, size, target)$ est vraie. □

Lemme 7.2.2. $\neg low < high \wedge I \Rightarrow$ WLP$(21-27, \text{NotFound})$.

Démonstration. Pour démontrer cette propriété, il suffit de remarquer que comme $low >= 0$, (par I_1), alors $tab[low] == target \wedge \neg$ExistElement$(tab, size, target)$ est contradictoire.

Et comme, pour tout p, q, on a $p \Rightarrow \bot \Rightarrow q$ est une tautologie, on peut conclure. □

Lemme 7.2.3. $low < high \wedge I \Rightarrow$ WLP$(12-20, I)$.

Démonstration. Commençons par calculer WLP$(12-20, I)$ (sans instancier I), puis en calculant séparément les trois invariants. On notera, pour simplifier, $test = tab[(low + high)/2] < target$, et EE $=$ ExistElement$(tab, size, target)$ (cette valeur n'étant pas modifiée ici).

$$\begin{aligned}\mathbf{WLP}(12-20, I) &\equiv \mathbf{WLP}(13, tab[mid] < target \Rightarrow I[low \leftarrow mid+1]) \\ &\quad \wedge \neg tab[mid] < target \Rightarrow I[high \leftarrow mid]) \\ &\equiv (tab[mid] < target \Rightarrow I[low \leftarrow mid+1] \\ &\quad \wedge \neg tab[mid] < target \Rightarrow I[high \leftarrow mid])[mid \leftarrow (low+high)/2) \\ &\equiv (test \Rightarrow I[low \leftarrow mid+1][mid \leftarrow (low+high)/2] \\ &\quad \wedge \neg test \Rightarrow I[high \leftarrow mid][mid \leftarrow (low+high)/2])\end{aligned}$$

$$\begin{aligned}\mathbf{WLP}(12-20, I_1) &\equiv ((test \Rightarrow 0 \leq mid+1 \leq high < size) \\ &\quad \wedge (\neg test \Rightarrow 0 \leq low \leq mid < size))[mid \leftarrow (low+high)/2] \\ &\equiv (test \Rightarrow 0 \leq (low+high)/2+1 \leq high < size) \\ &\quad \wedge (\neg test \Rightarrow 0 \leq low \leq (low+high)/2 < size)\end{aligned}$$

$$\begin{aligned}\mathbf{WLP}(12-20, I_2) &\equiv test \Rightarrow \mathbf{Sorted}(tab, size) \wedge \neg test \Rightarrow \mathbf{Sorted}(tab, size) \\ &\equiv \mathbf{Sorted}(tab, size)\end{aligned}$$

$$\begin{aligned}\mathbf{WLP}(12-20, I_2) &\equiv test \Rightarrow \mathbf{EE} \Rightarrow tab[(low+high)/2+1] \leq target \leq tab[high] \\ &\quad \wedge \neg test \Rightarrow \mathbf{EE} \Rightarrow tab[low] \leq target \leq tab[(low+high)/2]\end{aligned}$$

Montrons maintenant le résultat, invariant par invariant.

Par I_1 et $low < high$, on a $0 \leq low < high < size$. On a donc $0 \leq low \leq (low+high)/2 < high < size$. Et donc $I_1 \wedge low < high \Rightarrow \mathbf{WLP}(12-20, I_1)$ est une tautologie.

On a trivialement $I_2 \Rightarrow \mathbf{WLP}(12-20, I_2)$.

On sait que le tableau est trié (I_2), et par I_3 que s'il existe un élément égal à $target$ (EE), alors $tab[low] \leq target \leq tab[high]$. Si maintenant, $tab[(low+high)/2] < target$, alors, comme le tableau est trié, si $tab[(low+high)/2+1] > target$, il n'y a aucun élément égal à $target$ dans le tableau. Donc on a bien $\mathbf{EE} \Rightarrow tab[(low+high)/2+1] \leq target \leq tab[high]$. Si $tab[(low+high)/2] \geq target$, alors on a bien $tab[low] \leq targeq \leq tab[(low+high)/2]$. On vient donc bien de démontrer que $I_2 \wedge I_3 \Rightarrow \mathbf{WLP}(12-20, I_3)$.

On vient donc bien de démontrer que $I_1 \wedge I_2 \wedge I_3$ est un invariant de boucle. □

Ces deux derniers lemmes nous permettent de conclure que $\mathbf{WLP}(12-20, \mathbf{WLP}(21-27, \mathbf{Found} \wedge \mathbf{NotFound}))$ est bien défini, et donc, on peut conclure que le triplet de Hoare suivant est bien valide :

$$\{0 \leq size - 1 \wedge \mathbf{Sorted}(tab, size)\}\texttt{dichotomicSearch}\{\mathbf{Found} \wedge \mathbf{NotFound}\}$$

7.2.2 Terminaison

Il nous reste maintenant à montrer que le triplet de Hoare précédent est bien vrai au titre de la correction totale, et donc que, le terme donné

plus haut $V = high - low$ est bien un variant de boucle. Il nous suffit de montrer les deux lemmes suivants.

Lemme 7.2.4. $low < high \wedge I \Rightarrow high - low \geq 0$.

Démonstration. Trivial. □

Lemme 7.2.5. $low < high \wedge I \Rightarrow \mathsf{WLP}(12 - 20, high - low < \mathsf{at}(high - low, 12))$.

Démonstration. On commence par calculer $\mathsf{WLP}(12-20, high-low < \mathsf{at}(high-low, 12))$. On utilise les calculs abstraits faits à la section précédente.

$$\begin{aligned}\mathsf{WLP}(12 - 20, V < \mathsf{at}(V, 12)) \equiv\ & test \Rightarrow high - ((low + high)/2 + 1) < high - low \\ & \wedge \neg test \Rightarrow (low + high)/2 - low < high - low \\ \equiv\ & test \Rightarrow low \leq (low + high)/2 + 1 \\ & \wedge \neg test \Rightarrow (low + high)/2 < high\end{aligned}$$

De même qu'à la section précédente, on a $low < high$ qui implique que $low \leq (low + high)/2 < high$, ce qui permet donc de conclure que $low < high \Rightarrow \mathsf{WLP}(12 - 20, V < \mathsf{at}(V, 12))$ est une tautologie, et donc que V est bien strictement décroissant à chaque tour de boucle. □

Ainsi, V est bien un variant de la boucle `12-20`, et la fonction termine bien sur toutes les entrées respectant le contrat déterminé à la section précédente.

Index

Bibliographie

[1] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8) :453–457, 1975.

[2] Robert W. Floyd. *Assigning Meanings to Programs*, pages 65–81. Springer Netherlands, Dordrecht, 1993.

[3] Carlo A. Furia, Bertrand Meyer, and Sergey Velder. Loop invariants : Analysis, classification, and examples. *ACM Comput. Surv.*, 46(3) :34 :1–34 :51, 2014.

[4] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.