Département de Formation en Informatique

Cours sémantique et preuve de programmes

Jacques JULLIAND

Année 2014-2015

Sommaire

Table des matières

Table des définitions

Table des figures

Table des exemples

Table des exercices

Table des idées clés

Table des notations

Avant Propos

Ce cours est une partie des enseignements qui font l'objet de l'ouvrage suivant [Julliand 10] intitulé "**Cours et exercices corrigés d'algorithmique - Vérifier, tester et concevoir des programmes en les modélisant**" et édité par Vuibert. Le cœur de ce cours est constitué des six premières leçons de cet ouvrage. Les deux leçons suivantes sont des applications. Puis, la leçon 9 présente une approche de conception de programmes guidée par la vérification. Enfin, l'ouvrage contient trois autres leçons orientées sur la modélisation pour la génération automatique de tests appliquée avec le langage de modélisation B [Abrial 96].

Motivations

La partie *Sémantique et Preuve* est une introduction d'éléments de connaissances permettant d'effectuer des raisonnements rigoureux pour analyser la cohérence *sémantique* d'algorithmes. La partie *Théorie des Langages* présente un ensemble de connaissances qui permettent d'analyser la cohérence *syntaxique* d'algorithmes. L'analyse syntaxique est automatisable, l'analyse sémantique est un problème indécidable, donc non entièrement automatisable, c'est-à-dire pour lequel il n'existe pas d'algorithmes dans le cas général. Dans cette situation, il existe deux sortes de solutions, soit développer des algorithmes pour des cas particuliers, soit développer une méthode générale mais partiellement automatisée. Une méthode partiellement automatisée est mise en œuvre par des algorithmes ne répondent pas toujours au problème car il sont interrompus sur des critères comme un temps d'exécution limité. Quand ils ne répondent pas, l'utilisateur peut, soit résoudre le problème à la main, soit utiliser des outils d'assistance dans un processus interactif. Dans le cas de la vérification sémantique, on dispose de vérificateurs interactifs.

Les éléments de formation à l'analyse sémantique contribuent à la formation en algorithmique et en développement de programmes. La méthode présentée permet de concevoir des algorithmes et des programmes en posant et en résolvant des instances du problème générique suivant :

Est-ce que le fragment d'algorithme A réalise la tâche T ?

La méthode de résolution de ce problème est une *preuve* que l'algorithme A effectue la tâche T. Le fragment d'algorithme A est défini par une composition d'opérations élémentaires transformant des variables d'état V. La tâche T est définie par deux conditions appelées annotations, sur les variables V :

- l'une, appelée *pré-condition,* définissant l'état des variables V avant l'exécution de T,
- l'autre, appelée *post-condition,* définissant l'état des variables V après l'exécution de T.

Un tel problème étant résolu, le concepteur à la garantie que l'algorithme qu'il propose exécute correctement la tâche T.

Des environnements de programmation actuels, autour des langages JAVA et C# par exemple, proposent d'utiliser cette démarche pour aider à concevoir des programmes corrects. Pour cela aux langages de description d'algorithmes ont été associés des langages de description de la tâche à réaliser. Ceux-ci s'appellent respectivement ACSL (ANSI/ISO C Specification Language (voir http://frama-c.com/acsl.html)), JML (Java Modeling Language (voir http://www.cs.iastate.edu/leavens/JML)) [Burdy, Leavens] et Spec# (voir http://research.microsoft.com/specsharp/)

Les environnements de programmation associés, par exemple Frama-C (voir http://frama-c.com/) pour C et Keys (voir http://www.key-project.org/) pour Java, intègrent des outils pour résoudre les différentes instances du problème générique présenté ci-dessus. Parmi ces outils ou trouve notamment :

- des vérifieurs d'assertions à l'exécution appelé « Run Time Assertion Checker »,
- des outils de vérification des assertions à la compilation appelé « Static Assertion Checker »,
- des outils de génération de documentation.

Les éléments dispensés dans ce cours sont une présentation, d'une partie des concepts utilisés par ces outils, indépendante des langages C/ACSL, JAVA/JML et C#/SPEC#. En ce sens, ce cours prépare l'informaticien des années 2010 à l'utilisation des outils de développement de programmes des années 2020 et 2030. La logique de HOARE, conçue en 1969 [Hoare 69], constitue l'un des fondamentaux de la programmation sur lesquels sont construits les environnements de développement de programmes tels que B, C/ACSL, JAVA/JML, C#/SPEC#,

EIFFEL (voir http://www.eiffel.com/developers/design_by_contract.html), etc. La méthode est d'un grand intérêt mais le problème générique présenté ci-dessus étant indécidable, l'automatisation de sa résolution nécessite beaucoup d'ingéniosité pour que les outils soient d'un intérêt pratique. Avec ACSL, JML et Spec# de réels outils sont apparus pour C, JAVA et C#. De plus, l'appropriation de la méthode permet sans aucun doute d'aborder la conception de programmes avec un autre point de vue. Cet autre point de vue résulte d'une plus grande maîtrise de la sémantique des divers fragments d'un programme. D'une part, le concepteur est amené à exprimer par des annotations synthétiques et précises ce que doit faire (quoi ?) chaque fragment de programme. D'autre part il doit justifier pourquoi le fragment de programme proposé, aussi compliqué soit-il, réalise bien la tâche exprimée par ses annotations. Il s'assure ainsi que le "quoi faire ?" décrit par les annotations, et le" comment faire ?" décrit par le fragment de programme sont cohérents. Noter que l'expression des annotations appelées pré et post conditions sont une manière de mettre en œuvre la programmation dite par contrats (voir https://en.wikipedia.org/wiki/Design_by_contract) [Mitchell 02].

En résumé l'activité traditionnelle de conception de programmes consiste, étant donné un problème, à proposer un programme et le tester pour s'assurer qu'il résout bien le problème sur les quelques cas testés. La méthode proposée ici consiste, étant donné un problème, à le définir précisément par des annotations, à proposer un programme solution, puis à vérifier que le programme proposé automatise les réponses au problème défini par les annotations. Remarquons que cette méthode est applicable de manière incrémentale, un problème peut-être décomposé en plusieurs sous-problèmes chacun étant traités par cette méthode. Remarquons également que cette méthode doit être complétée par une phase de test qui conserve son intérêt malgré la certitude que le fragment d'algorithme résout le problème défini par les annotations car on n'est jamais sûr que les annotations expriment exactement le problème que l'usager de départ ait posé informellement.

Leçon 1 : Test et vérification de programmes

Pré requis

. comprendre un programme simple exprimé dans un langage inspiré de Pascal, C et Java,
. savoir lire et comprendre une expression logique ; voir cours MF (Méthodes Formelles).

Objectifs

. Montrer la différence entre la notion de test et de vérification,
. Définir les problèmes de vérification et de test de programmes.
. Introduire intuitivement par un exemple le système de vérification de la logique de HOARE.

Dans cette leçon nous présentons la notion de vérification formelle qu'un programme est correct relativement à une définition de ce qu'il doit calculer appelée *spécification*. Cet objectif est décomposé en deux leçons. Dans la première, nous présentons le concept de *vérification* en comparaison avec celui, plus familier, de *test*. Puis dans la seconde, nous présentons la notion de vérification de programmes comme son exécution symbolique de programme. Cette présentation est effectuée sur un exemple. C'est une introduction dans la leçon deux où nous définirons formellement la notion de vérification par exécution symbolique par le système de preuve de la logique de HOARE. Nous illustrons l'intérêt de la méthode dans la leçon 7 par un exemple qui illustre que la preuve permet de découvrir une incohérence entre un programme donné et sa spécification alors que l'erreur était passée inaperçue à 90 % des étudiants ayant eu à décrire formellement la spécification du problème.

La leçon 1 est décomposée en six sections. La section 1 présente la démarche de modélisation pour vérifier ou pour tester un programme. La section 2 présente la notion de test et la section 3 rappelle quelques éléments de stratégie de test. La section 4 répond à la question «comment établir un test de programme ? ». La section 5 explique pourquoi il faut modéliser (on dira également spécifier) un programme pour le tester. La section 6 introduit la notion de vérification relativement au concept de test. Elle indique que pour vérifier il faut modéliser comme pour tester et que les démarches sont semblables bien que la démarche de vérification soit beaucoup plus puissante. Puis la section 7 définit schématiquement une démarche de vérification de programmes. Enfin la section 8 est un résumé et une transition vers la leçon 2 qui introduit la méthode de vérification sur un exemple simple.

1. Présentation de la démarche de modélisation pour la vérification et la validation

Dans cette partie, nous présentons dans la section 1.1, d'une part la démarche de *modélisation* et de *vérification* de la cohérence d'un programme et d'un modèle et d'autre part la démarche de *validation* d'un programme par génération automatique de tests. Puis, dans la section 1.2, nous présentons, sur un exemple, les principales idées sous-jacentes à la vérification.

1.1. Démarche : modéliser pour vérifier ou tester

La modélisation n'a d'intérêt que si elle a un objectif. Parmi les objectifs d'une modélisation, on peut citer :

- la vérification de la cohérence (on dit aussi consistance ou conformité) du modèle et de son implémentation.
- la validation d'une implémentation appelée Implémentation Sous Test (**IST**), en la soumettant à des campagnes de tests, générés automatiquement à partir d'un modèle cohérent,
- la dérivation d'une implémentation (un programme) à partir du modèle,
- l'évaluation des performances du système,
- etc.

Dans cette section, nous nous intéressons aux deux premiers points qui sont utilisables séparément : la génération automatique de tests est effectuée à partir d'un modèle. Elle n'est utile que si la vérification de

cohérence n'a pas été totalement effectuée. La Figure 1 illustre un processus de développement de logiciels combinant les deux techniques de vérification et de test. La démarche proposée confronte un modèle et une implémentation issus d'un cahier des charges par deux processus de développement indépendants. La confrontation est réalisée en faisant exécuter les tests issus du modèle par l'implémentation IST issue du développement.

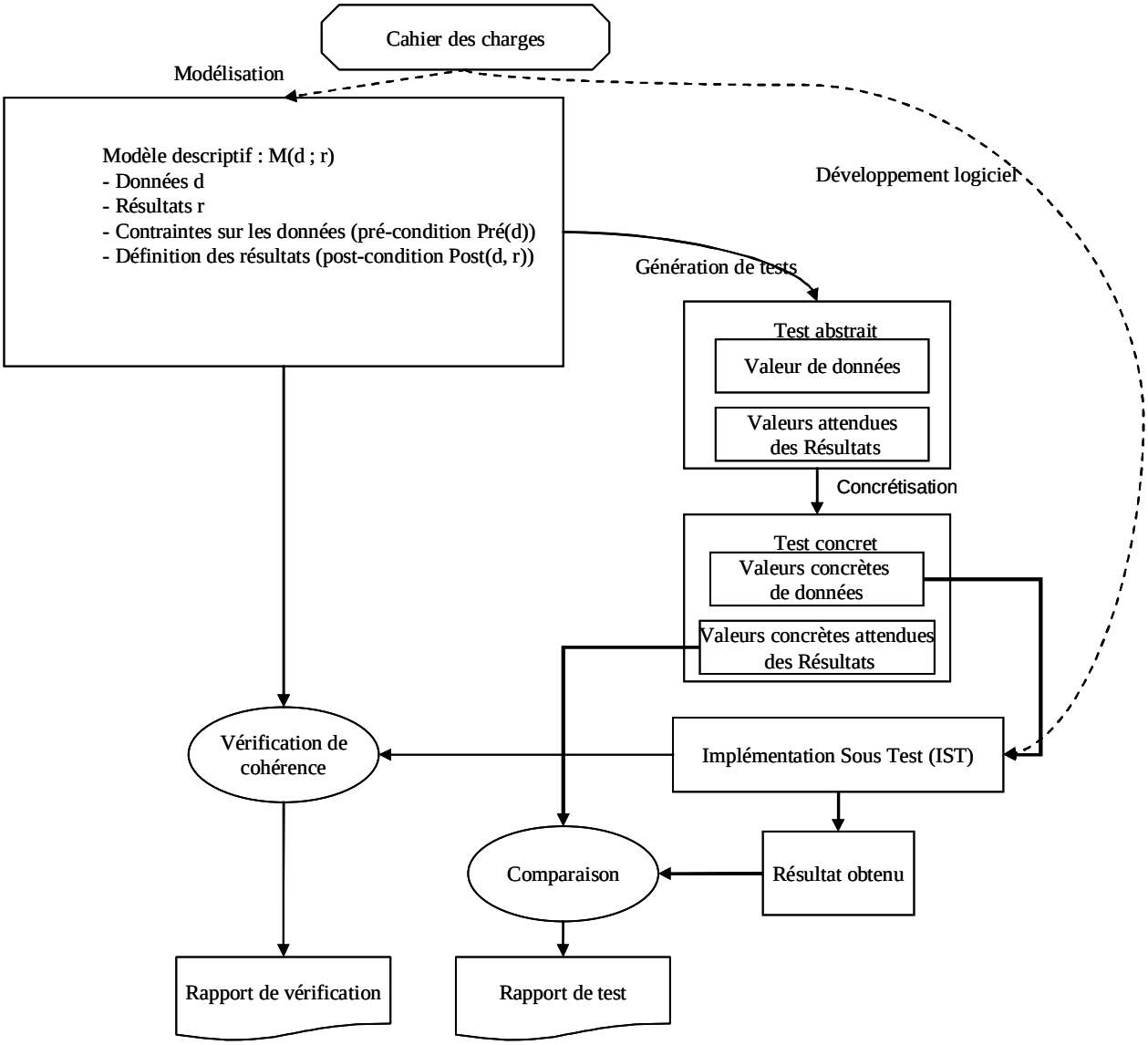


Figure 1 : Modéliser pour vérifier et/ou tester

Le modèle est constitué de trois parties :

- un modèle de données qui décrit les variables données et résultats du système en les typant,
- la description des contraintes sur les données qui ne sont pas spécifiables avec le typage ; elles sont décrites par une expression logique appelée pré condition dénotée `Pré(d)`,
- la définition des résultats en fonction des données ; c'est une relation décrite par une expression logique appelée post-condition dénotée $Post_a(d, r)$.

La Figure 3 est un exemple de modèle du problème de calcul de la racine carrée entière par défaut. Par exemple, pour n=17, valeur satisfaisant la pré condition, ce modèle définit r=4 qui est la seule valeur satisfaisant la post condition.

La vérification de cohérence entre le modèle et l'implémentation est représentée Figure 1 et illustrée par la Figure 2. Elle consiste à s'assurer que l'exécution symbolique de l'implémentation à partir d'un ensemble de valeurs des données définis par la pré condition `Pré(d)` aboutit à un ensemble de résultats obtenus $Post_o(d, r)$ qui est inclus dans l'ensemble des résultats attendus défini par la post-condition $Post_a(d, r)$. Notons

qu'il n'est pas nécessaire que l'ensemble de résultats obtenus soit strictement égal à l'ensemble de résultats attendus. Mais tous les résultats obtenus doivent être un des résultats attendus. Autrement dit la question « Est-ce que l'implémentation est cohérente avec le modèle ? » se formalise par l'expression ensembliste suivante : $Post_o(d, r) \subseteq Post_a(d, r)$? La réponse est oui si cette formule est vraie. Elle se formalise également par l'expression logique suivante : $Post_o(d, r) \Rightarrow Post_a(d, r)$ si $Post_o(d, r)$ et $Post_a(d, r)$ sont les expressions logiques qui définissent les ensembles de résultats respectivement obtenus et attendus. La réponse est oui si la formule est valide, non sinon.

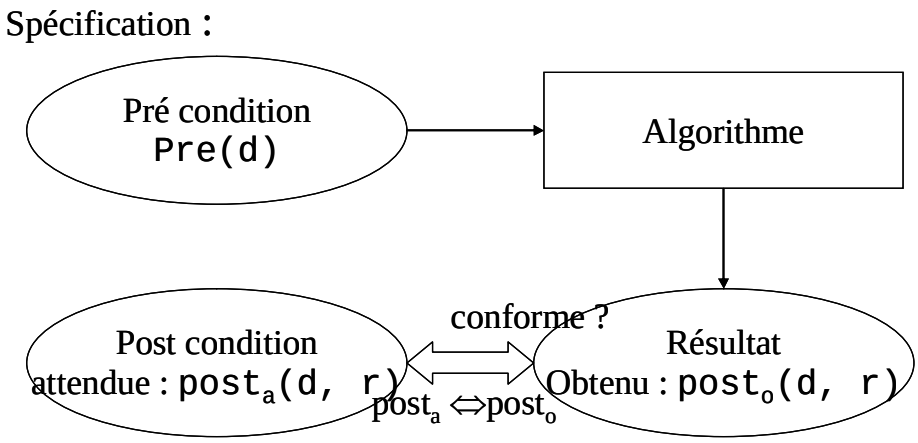


Figure 2 : Vérification de cohérence entre un modèle (spécification) et un algorithme

A partir d'un modèle, il est également possible de générer automatiquement des tests comme l'illustre la Figure 1 à condition de disposer d'un programme capable de déterminer des couples de valeurs *(d, r)* qui satisfont la pré condition et la post condition. Dans le cas particulier du test d'un programme formé d'un seul module *M(d ; r)*, un test est composé d'un appel du module M muni de valeurs de ses données *d* et de ses résultats attendus *r*. Les valeurs des tests générés sont définies au niveau d'abstraction déterminé par les types des données et des résultats du modèle. En général, il est nécessaire de les concrétiser au niveau des types des données et des résultats de l'implémentation pour pouvoir exécuter l'IST sur ces données. Par exemple, pour un programme de tri, si la donnée qui est la séquence à trier est définie comme une séquence et que celle-ci est représentée par un tableau dans le programme, il faudra transformer la séquence en tableau pour la concrétiser. L'exécution d'un test concret sur l'IST permet d'obtenir un résultat qui est comparé avec le résultat attendu pour décider si le test est un succès ou un échec. Le rapport de test indique ce résultat et fait des synthèses pour l'ensemble des tests d'une campagne de test.

Quand la cohérence entre modèle et implémentation a été établie, les tests issus du modèle n'ont plus aucun intérêt. Par contre, il est toujours utile de tester l'IST par des jeux de test issu du cahier des charges. En effet rien ne garantit qu'il n'y ait pas eu d'erreurs d'interprétation du cahier des charges dans l'obtention du modèle et par conséquent les mêmes dans la conception de l'implémentation puisque toutes les deux sont cohérentes. Mais la génération de ces tests n'est pas automatisable, sauf s'ils sont issus d'un autre modèle défini par une équipe de validation dont le seul but est d'engendrer des tests. Des tests *manuels* sont utiles dans tous les cas, car il n'est pas possible de garantir que le modèle est *complet*, c'est-à-dire qu'il spécifie toutes les exigences du cahier des charges. Il est donc utile de tester les exigences qui n'auraient pas été formalisées dans le modèle.

Notons que les approches C/ACSL, JAVA/JML (voir http://www.cs.iastate.edu/leavens/JML) [Burdy 05, Leavens 00] et C#/SPEC# (voir http://research.microsoft.com/specsharp/) consistent à décrire une implémentation en C, JAVA ou C# munie d'un modèle, défini par un ensemble d'annotatiSL, JML ou SPEC#. Les environnements de programmation sont munis d'outils, soit appelés RAC (Runtime Assertion Checker), soit appelé Prover (Static Assertion Checker). Les RAC vérifient la cohérence du programme avec le modèle à chaque exécution. Les Prouveurs vérifient statiquement que le programme est cohérent avec le modèle pour toutes les exécutions possibles.

1.2. Vérification de la cohérence entre les modèles descriptif et l'implémentation

Idée Clé 1 ***(le modèle descriptif décrit le QUOI ?)***

Le modèle décrit le QUOI, on l'appelle *Modèle Descriptif (ou Spécification)* (au sens où il décrit ce que le système doit faire sans rien dire du comment le faire). ♦

Exemple 1 ***(modèle descriptif)*** voir la Figure 3. ♦

Idée Clé 2 ***(le programme décrit le COMMENT ?)***

Le second énoncé décrit le COMMENT, on l'appelle donc programme (programme ou *Modèle Opérationnel,* au sens où il décrit comment "opérer" pour résoudre le problème). ♦

Exemple 2 ***(programme)*** voir la Figure 4. ♦

Idée Clé 3 ***(cohérence modèle, programme)***

Qui dit vérification dit "comparaison" de 2 énoncés pour s'assurer que le second (implémentation) réalise bien ce que décrit le premier (modèle descriptif). ♦

Exemple 3 ***(cohérence modèle, programme)***

Comparaison du couple d'expressions logiques décrit dans la Figure 3 avec le programme décrit dans la Figure 4. La comparaison consiste à vérifier que l'exécution symbolique de A à partir d'une donnée *n* positive ou nulle aboutit à un résultat *r* appartenant à l'ensemble des valeurs de r défini ainsi $\{r \mid (r^2 \leq n) \wedge (n<(r+1)^2) \wedge r \geq 0\}$. Cette vérification est notée par la formule suivante : $\{n \geq 0\}$ A $\{(r^2 \leq n) \wedge (n<(r+1)^2) \wedge r \geq 0\}$. C'est une formule de la *Logique de Hoare* dont la forme générale est $\{Pré(d)\}$ A $\{Post(d, r)\}$ pour exprimer que le programme A est cohérent avec le modèle formé de la pré-condition $Pré(d)$ et de la post-condition $Post(d, r)$ où d sont les données de l'algorithme A et r ses résultats. ♦

Idée Clé 4 ***(les 3 parties d'une méthode formelle de vérification)***

Une méthode formelle est fondée sur :

- un langage d'expression de modèles descriptifs (ici un langage d'annotations qui sont des expressions logiques),
- un langage de description de programmes,
- une méthode de vérification qui est une procédure (de décision) pour établir si les 2 énoncés précédents sont cohérents. ♦

Idée Clé 5 ***(2 approches de la vérification)***

Il y a 2 approches de la vérification :

- la preuve ; par exemple, en Logique de Hoare une preuve est une sorte d'exécution symbolique (sans connaître la valeur des données) sur des ensembles de valeurs,
- la vérification algorithmique de programmes (appelée *model-checking*) ; celle-ci est un ensemble d'exécutions pour toutes les valeurs de données possibles. Elle n'est possible que si toutes les variables ont un nombre fini de valeurs, si toutes les itérations terminent et si toutes les exécutions peuvent être réalisées dans un temps compatible avec la durée de vie humaine. ♦

Partie informelle (modèle de données)
donnée : n : nombre dont on veut calculer la racine (entier positif ou nul)
résultat : r : racine carrée entière par défaut (entier positif ou nul)

Partie formelle
Pré-condition : $\{n \geq 0\}$
Post-condition : $\{r \times r \leq n \wedge n<(r+1) \times (r+1) \wedge r \geq 0\}$

Figure 3 : Modèle du problème de calcul de la racine carrée

La méthode que nous présentons dans ce cours est une méthode de preuve qui permet de traiter des systèmes à nombre infini d'états. Elle permet également de traiter des programmes à nombre fini d'états, mais suffisamment grand pour que l'énumération de tous les cas ne soit pas réalisable par une machine. Pour qu'un programme ait un nombre infini d'états, il suffit qu'il ait une donnée de type entier et qu'on ne considère pas de borne supérieure pour les entiers. En pratique, les entiers dans les langages de programmation ont un nombre fini de valeurs, par exemple 65536 pour des *short* en Java et 2^{32} pour des *int*. Par conséquent, un programme très simple ayant 3 données de type *int* a un ensemble potentiel de 2^{96} états, ce qui est bien trop grand pour être énumérable en pratique. La vérification par preuve est alors indispensable. La vérification algorithmique est

adaptée à des programmes de contrôle-commande dont le nombre d'états est limité. Dans l'état de l'art actuel, un model-checker comme SMV traite, dans certains cas, des programmes ayant jusqu'à 10^{20} états [Clarke 01].

```
A ≝ procédure racine (donnée n : entier ; résultat r : entier) ;
    var Rmax, Rmin, M : entier ;
    début
        Rmax := n+1 ; Rmin := 0 ;
        tantque Rmin+1≠Rmax faire
            M := (Rmin+Rmax+1) div 2 ;
            si M*M≤n alors Rmin := M
            sinon   Rmax := M      /* cas M*M>n */
            finsi
        fait ;
        r := Rmin ;
    fin
```

Figure 4 : Programme de calcul de la racine carrée par dichotomie

2. Notion de test

Définition 1 (*test*) :

Soit un programme A qui calcule des résultats *r* à partir des données *d*. Un test du programme A est composé d'un couple (v_d, v_r) où v_d est une valeur des données *d* et v_r est la valeur des résultats *r* attendue pour les données v_d. ♦

L'application d'un test (v_d, v_r) sur un programme A consiste à exécuter A en lui fournissant la valeur v_d pour *d* et à vérifier que la valeur v_o de *r* obtenue par l'exécution (appelée valeur obtenue) est identique à la valeur attendue v_r.

Définition 2 *(test réussi, test en échec)*

Soit (v_d, v_r) un test. Soit v_o la valeur du résultat obtenu en exécutant un programme A à partir des valeurs de données v_d. Soit C la fonction de concrétisation du modèle vers le programme, on dit que le test (v_d, v_r) est réussi si $v_o = C(v_r)$. Si $v_o \neq C(v_r)$ alors on dit que le test a échoué et que le programme A contient une erreur. ♦

Exemple 4 *(cas de test)*

Soit le programme A de la Figure 4 qui calcule la racine carrée entière *r* d'un nombre donné *n* par recherche dichotomique.

Les couples *(n, r)* de l'ensemble suivant {(0, 0), (1, 1), (2, 1), (3, 1), (4, 2), (5, 2), (6, 2), (7, 2), (8, 2), (9, 3)} sont des tests de ce programme qui réussissent. Par exemple, en exécutant ce programme pour n=4, il termine avec r=2. Donc le couple (4, 2) est un test réussi. Par contre le couple (4, 3) serait un test qui échouerait. Notons qu'ici la fonction de concrétisation est l'identité car les valeurs des tests sont de même type que les valeurs en entrée et en sortie du programme car les entiers sont identiques dans le modèle et dans le programme.♦

Un test (v_d, v_r) permet de vérifier que le programme A calcule le résultat attendu v_r pour la donnée v_d. Un seul test ne permet pas de conclure que A calcule le résultat attendu pour toutes les données. Pour obtenir cette certitude, il faudrait exécuter A pour toutes les valeurs possibles des données. Ceci n'étant pas possible en général, il est conseillé d'établir un ensemble significatif et judicieusement choisi de tests afin d'acquérir une plus grande confiance dans A. Pour établir cet ensemble de tests, on utilise des stratégies de test et on défini un plan de test. Notons que cette question du test n'est pas l'objectif du cours. Elle est utilisée à titre pédagogique car la démarche de vérification est similaire, mais plus générale. On va donc s'appuyer sur la démarche de test, plus connue et plus simple, pour faire comprendre la démarche de vérification.

3. Quelques éléments de stratégie de test

Pour tester un programme A, il est nécessaire d'effectuer plusieurs tests selon plusieurs points de vue. Cet ensemble de tests découle d'un plan de test. Sur un exemple simple comme le précédent, on distingue deux sortes de tests :

- des **tests fonctionnels** dit « test boîte noire » qui sont établis sans connaître le programme A, mais uniquement à partir de la connaissance de ce qu'il doit faire (décrit dans le cahier des charges ou le modèle descriptif),
- des **tests structurels** qui sont établis à partir du programme afin d'exécuter toutes ses instructions ou tous ses chemins d'exécution.

Chaque cas de test d'un plan de test doit être justifié. Par exemple, pour déterminer un ensemble de tests d'un programme, il faut établir un plan de test en déterminant les différents cas à tester et en instanciant au moins un test par cas.

Exemple 5 ***(Choix de cas de test)***

Par exemple, pour tester les fonctionnalités du programme de calcul de la racine carrée entière par défaut défini dans la Figure 4 spécifié dans la Figure 3, on peut utiliser le plan de test suivant :

- cas d'un carré exact,
- cas qui n'est pas un carré exact,
- cas limite où le nombre est immédiatement inférieur au carré exact,
- cas limite où le nombre est immédiatement supérieur au carré exact,
- cas médian où le nombre est au milieu de l'intervalle entre deux carrés exacts successifs,
- cas limite où *n* est égal à zéro qui est la plus petite valeur possible.

L'ensemble suivant de tests est un ensemble de tests fonctionnels établit selon ce plan de test :

- (16, 4) est un représentant du cas d'un carré exact,
- (21, 4) est un représentant du cas qui n'est pas un carré exact,
- (15, 3) est un représentant du cas limite où le nombre est immédiatement inférieur au carré exact,
- (17, 4) est un représentant du cas limite où le nombre est immédiatement supérieur au carré exact,
- (20, 4) est un représentant du cas médian où le nombre est au milieu de l'intervalle entre deux carrés exacts successifs,
- (0, 0) est le cas limite où *n* est égal à zéro qui est la plus petite valeur possible.

L'ensemble suivant de tests est un ensemble de tests structurels :

- (0, 0) pour le cas où aucun pas d'itération ne serait exécuté,
- (1, 1) pour le cas où un seul pas d'itération serait exécuté,
- (25, 5) pour le cas où plusieurs pas d'itération seraient exécutés,
- (35 000, 187) pour le cas où un grand nombre d'itérations seraient nécessaires. ♦

Pour des exemples plus compliqués où le programme est constitué de plusieurs modules, chaque module est testé, puis d'autres tests, appelés **tests d'intégration**, sont nécessaires pour tester la combinaison des modules. Enfin, en général le client fourni avec le problème des tests appelés **tests de recettes**.

Cette présentation étant très succincte, nous proposons au lecteur intéressé de se reporter à un cours de génie logiciel sur le test. Citons les ouvrages *The Art of Software Testing* [Myers 79] et *Practical Model-Based Testing* [Utting 06]. Cette section est une simple sensibilisation au problème du test sur un exemple simpliste. Dans la suite, nous parlerons de tests fonctionnels boîte noire établis à partir d'un modèle descriptif.

4. Comment établir un test de programme ?

Est-ce que les couples (3, 2), (5,3) ou (16, 5) sont des tests de l'exemple de racine carrée ?

Si c'était des tests, ce serait des tests qui ne réussiraient pas sur le programme de la Figure 4, c'est à dire qui mettraient en évidence des erreurs dans A. En réalité, ce ne sont pas des tests pour des raisons différentes :

- (16, 5) de toute évidence n'est pas un test car la racine carrée de 16 est 4,

- par contre, (3, 2) et (5, 3) ne sont pas des tests car le programme A calcule la racine carrée entière par défaut (approximation inférieure). Ce serait des tests si le problème posé avait été de calculer la racine carrée entière par excès.

Ceci met en évidence une difficulté pour constituer des tests à partir d'une présentation informelle du problème à résoudre comme nous l'avons présentée ci-dessus dans la section 1 : « calculer la racine carrée entière *r* d'un nombre donné *n* ». Cette définition ne permet pas de dire si (3, 2) et (3, 1) sont ou ne sont pas des tests. Seule une définition formelle du problème permet d'établir un test sans ambiguïté. Nous appelons une telle définition formelle une **spécification.**

Par exemple, une spécification qui permet de définir le problème du calcul de la racine carrée entière par défaut est décrite dans la Figure 3. Celle-ci ne dit pas comment calculer *r*, elle exprime simplement que seuls les nombres entiers positifs ou nuls sont admis comme valeur de donnée pour *n* et que le résultat *r* doit satisfaire la post-condition.

A partir de cette spécification, on est capable de définir des tests ainsi (voir Figure 5)

- choisir une valeur pour *n* respectant la pré condition et réalisant un des objectifs de test,
- puis déterminer une valeur de *r* qui vérifie la post-condition avec la valeur de *n*.

Par exemple, choisissons n = 45, alors il faut choisir r = 6 car $6\times6\leq45 \wedge 45<(6+1)\times(6+1) \wedge 6\geq0$. Pour *n=3*, il faut choisir r=1 car $1\times1\leq3 \wedge 3<(1+1)\times(1+1) \wedge 1\geq0$.

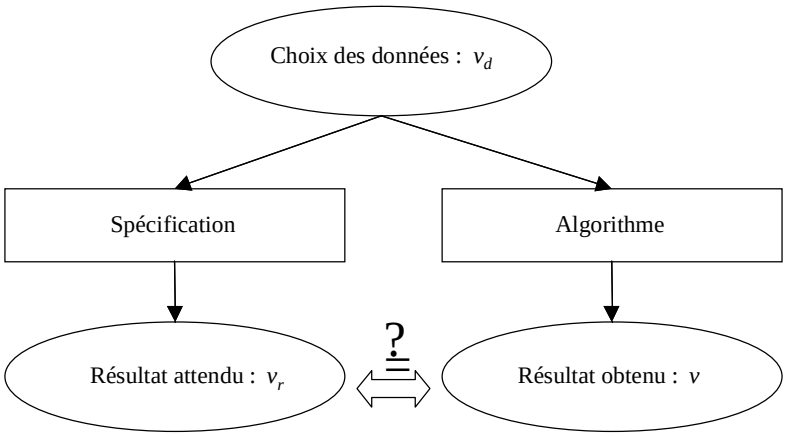


Figure 5 : démarche de test

Notons que cette démarche permet d'établir des tests sans aucune connaissance sur le programme, ce qui paraît tout à fait indispensable pour tester le programme indépendamment de lui-même. Le contraire serait très dangereux, puisqu'il permettrait de prendre ses désirs pour des réalités en établissant un test en choisissant une valeur de donnée, puis en appliquant à la main le programme pour connaître le résultat. Il est alors certain que l'exécution automatique du programme sur machine donnerait le résultat attendu sauf si l'application manuelle comportait des erreurs. Cette démarche est erronée, elle compare le programme avec lui-même, ce qui n'a aucun intérêt. Par contre, les méthodes de test structurel utilisent la connaissance de la structure de contrôle du programme pour établir les valeurs des données qui conduiront à passer par telle ou telle partie du programme. Le résultat attendu est prédit soit par l'ingénieur validation, soit à partir d'un modèle. Pour les mêmes raisons que précédemment, il ne peut pas être produit à partir du programme.

5. Modéliser pour tester des programmes

Définition 3 *(spécification)* :

On appelle spécification d'un programme A un quadruplet formé de :

- une désignation et un typage des données,
- une désignation et un typage des résultats,
- une *pré-condition* définissant des contraintes sur les données restreignant les valeurs admises par le typage,

- une *post-condition* définissant le résultat en fonction des données. ♦

Exemple 6 ***(spécification de racine carrée entière par défaut)***

La Figure 3 définit la spécification de la racine carrée entière par défaut de *n*. La Figure 6 définit si deux mots m_1 et m_2 sont égaux. ♦

Données	m_1, m_2 : tableau[1..100] de caractères (*mots à comparer *) n_1, n_2 : 0..100 ; (* nombre de lettres de m_1 et m_2 *)
Résultat	eg : booléen ; (* vrai si les 2 mots sont identiques *)
Pré-condition :	`vrai`
Post-condition :	$eg \Leftrightarrow \forall i.(i \in [1..n_1] \Rightarrow m_1[i]=m_2[i]) \wedge n_1=n_2$

Figure 6 : spécification de deux mots égaux

Remarque : Les pré et post conditions sont exprimées chacune par une expression en logique des prédicats du premier ordre. Si la pré condition est le prédicat *vrai*, c'est qu'il n'y a aucune contrainte particulière sur les données, ici on compare deux mots quelconques.

Pour tester un programme, il faut établir 3 énoncés et les utiliser comme l'indique la Figure 5 :
- un programme A exprimé dans un langage de programmation,
- une spécification S exprimée en logique des prédicats du premier ordre,
- un ensemble de tests qui sont définis en choisissant des valeurs des données v_d en fonction d'un objectif de test et en évaluant manuellement, ou mieux à l'aide d'outils, les valeurs des résultats attendus v_r à partir de la spécification S.

Puis, pour chaque valeur de données v_d, A est exécuté automatiquement afin de calculer le résultat obtenu. Si celui-ci est identique au résultat attendu v_r, alors le test est réussi, sinon le test a permis de détecter une anomalie.

6. Notion de vérification, différence entre vérification et test et limites des méthodes de test

Vérifier un programme consiste à s'assurer que pour chaque valeur possible des données, la valeur attendue du résultat est identique à la valeur obtenue par le programme (voir Figure 5).

La première méthode qui vient à l'esprit est de déterminer tous les tests possibles :
- la pré-condition permet de déterminer toutes les valeurs de données,
- la post-condition permet d'établir le résultat attendu pour chacune.

Sur l'exemple du calcul de la racine carrée, cette méthode n'est pas applicable car le nombre de valeurs de *n* positives ou nulles pour les entiers naturels est infini. Si on considère des entiers représentés, en machine, sur 32 bits, le nombre de valeurs est 2^{32}, ce qui est bien trop grand pour être réalisable en pratique.

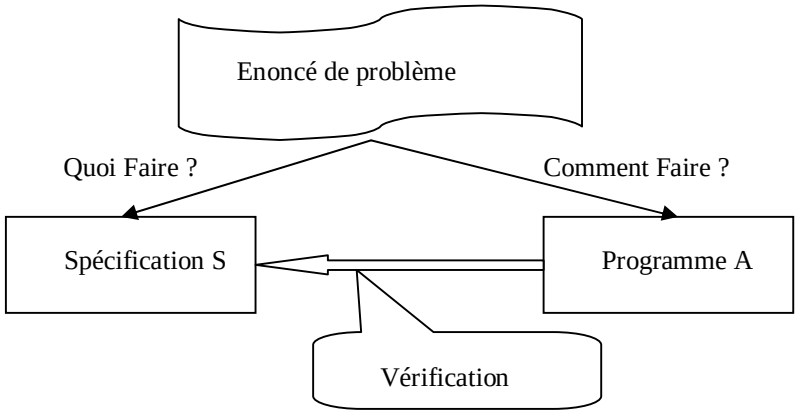


Figure 7 : démarche de vérification de programme

Pour résoudre ce problème de l'infinité ou du trop grand nombre de tests, d'autres méthodes sont proposées. C'est le cas de la Logique de HOARE qui permet d'effectuer une exécution du programme sur des valeurs symboliques représentant des ensembles de valeurs. Sur l'exemple, l'idée intuitive est d'exécuter une seule fois le programme de calcul de la racine carrée sur l'ensemble des valeurs de *n* positives ou nulles. Cette exécution aboutit à un ensemble de valeurs pour le résultat *r*. Il faut ensuite vérifier que cet ensemble de valeurs est inclus dans celui qui est défini par la post-condition de la spécification. Cette technique peut-être appelée *exécution symbolique*. Dans le cas de la preuve de programmes, elle est mise en œuvre par une technique de preuve utilisant deux sortes de systèmes formels. La *Logique de Hoare* est utilisée pour les règles sur le langage de programmation et la *déduction naturelle*, par exemple, est utilisée pour les preuves de validité de prédicats en logique des prédicats du premier ordre.

7. Démarche de vérification

La démarche de vérification est illustrée par la Figure 7. Celle-ci met en évidence que vérifier consiste à *comparer* deux énoncés, un programme et une spécification. Ces deux énoncés décrivent le même problème. La solution calculée par le programme A doit être identique à la solution définie par la spécification S. Comme les deux descriptions de la solution sont exprimées dans des termes différents, la comparaison doit être effectuée grâce à un système qui met en relation les deux énoncés. Dans notre cas, le système qui met en relation les deux énoncés s'appelle Logique de HOARE. Le système associe à tout programme un sens sous forme d'une spécification (réduite à sa partie formelle composée de la pré-condition et de la post-condition). Par exemple, le sens de l'affectation $x := x+1$ est un couple de formules logiques $P(x)$, $P(x-1)$ qui indique que si la condition $P(x)$ est vraie avant d'exécuter $x := x+1$, alors la condition $P(x-1)$ sera vraie après. Par exemple si $x=4$ est vraie avant, alors $x-1=4$, c'est à dire $x=5$, sera vraie après avoir exécuté $x := x+1$. On dit que ce système définit la *sémantique* du langage de programmation en associant à chaque fragment de programme un sens sous la forme d'une spécification. Cette sémantique permet de vérifier qu'un programme calcule un résultat défini par sa spécification. Dans le cas de la logique de HOARE, la vérification est réalisée par preuve.

Définition 4 *(vérification)* :

Vérifier, c'est comparer deux énoncés, un programme A et une spécification S ; A satisfait S si A calcule les résultats définis par S. ♦

Idée Clé 6 *(Condition importante pour faciliter la démarche de vérification)*

Pour faire facilement la preuve d'un programme, il faut respecter les consignes suivantes d'écriture :

- ne pas modifier les données dans le programme, donc séparer données et résultats, ne pas utiliser des *données-résultats* comme c'est souvent le cas en programmation pour économiser de la mémoire. ♦

Cette contrainte permet de réutiliser la post condition telle quelle. En effet, les occurrences des données dans la post condition désignent les données à l'entrée du programme et pas des données modifiées. C'est pourquoi, la confrontation du programme et de la spécification ne peut se faire que si les variables partagées entre ces deux énoncés (données et résultats sont partagés) ont la même signification dans les deux.

Il est facile de satisfaire cette contrainte. Si le programme doit modifier les données au fur et à mesure de ses calculs, il faut les dupliquer dans des variables internes au programme dès le début du programme, modifier ces variables internes et ne pas modifier les données. Ensuite, la vérification étant faite, le programme exploité pourra être optimisé en supprimant ces variables. Noter que tous les exemples du cours respectent cette consigne. Par exemple le programme de tri à bulle de la Figure 16 respecte cette consigne en recopiant le tableau *td* dans le tableau *tr*. En pratique, ce programme sera implémenté avec un seul tableau *t* qui sera la donnée en entrée et le résultat en sortie. Mais dans le modèle descriptif, le même tableau *t* ne peut pas être non triée comme donnée et trié comme résultat. C'est pourquoi non introduisons deux variables distinctes *td* et *tr*, *td* non trié et *tr* trié.

8. Résumé

Spécification et programme sont deux énoncés du même problème :

- la *spécification* S définit le problème (cet énoncé répond à la question suivante : quel est le résultat attendu ?),
- *le programme* A est une procédure de calcul du résultat (cet énoncé répond à la question suivante : comment calculer ce résultat ?).

Un test est un couple de valeurs des données et des résultats (v_d, v_r) établit à partir de la spécification S.

L'exécution du programme A pour une valeur des données v_d permet de valider A pour le test (v_d, v_r) si le résultat obtenu est $v = v_r$.

Pour valider un programme A, il faut déterminer un ensemble de tests élaboré avec une stratégie qui a pour but de couvrir le plus large spectre de données possible. La stratégie habituelle est d'effectuer un partitionnement des données en un ensemble de classes d'équivalence et de choisir un représentant de chaque classe.

Mais quelques tests ne suffisent pas pour garantir qu'un programme donnera toujours le résultat attendu. Les tests fonctionnels établit à partir du modèle doivent être complétés par des tests structurels, des tests d'intégration et des tests de recette.

Un test exhaustif n'est en général pas réalisable en pratique, soit parce que le nombre de tests est infini, soit parce qu'il est trop grand.

La notion de vérification répond à ce problème en permettant en quelque sorte de faire toutes les exécutions en une seule fois ; cette notion est appelée *exécution symbolique*, c'est à dire exécution avec des valeurs symboliques représentant toutes les valeurs possibles à la place d'une exécution avec une seule valeur.

Leçon 2 : Vérification de programmes par exécution symbolique

Objectifs

. Montrer intuitivement sur un exemple comment vérifier un programme par une exécution symbolique,
. Introduire intuitivement le système de vérification de la logique de HOARE.

Cette leçon présente la notion de *valeur symbolique* de variables en section 1. La section 2 est consacrée à la notion *d'exécution symbolique.* La section 3 donne un exemple et enfin la section 4 résume cette leçon et effectue une transition vers les leçons 3, 4 et 5.

Notation 1 ***(comparaison, affectation, définition, équivalence)***

Pour bien distinguer des concepts différents, nous notons :

. = l'opérateur de comparaison ; exemple a=b est un prédicat vrai si *a* est égal à *b*,
. := l'opérateur d'affectation ; c'est la notation Pascal, la notation C et JAVA étant = ; exemple : a := b/2+1
. $\stackrel{def}{=}$ le symbole de définition ; exemple $p \stackrel{def}{=} (a=b)$ est la définition du prédicat p dont la valeur est l'expression a=b, ceci permet de ne pas confondre l'opérateur de comparaison et la notation de définition.
. ≡ le symbole d'équivalence sémantique entre deux expressions logiques ; celui-ci est utilisé quand on substitue un terme d'une expression par un terme équivalent. Par exemple, $r^2 \leq n \wedge y=r^2+2r+1$ est équivalent à $r^2 \leq n \wedge y=(r+1)^2$ sera noté $r^2 \leq n \wedge y=r^2+2r+1 \equiv r^2 \leq n \wedge y=(r+1)^2$. Il ne faut pas confondre ce symbole avec l'opérateur logique « si et seulement si » noté ⇔. Cet opérateur s'applique entre deux expressions logiques a et b ainsi : a⇔b. L'expression logique a⇔b est vraie si et seulement si les valeurs de *a* et *b* sont identiques. L'expression a⇔b est équivalente à a⇒b ∧ b⇒a (a⇔b ≡ a⇒b ∧ b⇒a). Autrement dit a≡b indique la réécriture de l'expression a par une expression b équivalente, alors que a⇔b représente une expression logique, soit vraie, soit fausse.
. les expressions logiques sont représentées en caractères courriers.

Par contre la notation […] est utilisée avec un grand nombre de sens différents qu'il est facile de distinguer selon le contexte :

. t[i] est la désignation du $i^{ème}$ élément du tableau *t*,
. k ∈ [1..n] est le prédicat qui est vrai si l'entier *k* appartient à l'intervalle des entiers compris entre 1 et *n* compris. Nous notons quelques fois plus simplement k ∈ 1..n.
. [nom_auteur année] est une référence bibliographique à un ouvrage écrit par un auteur de nom *nom_auteur*,
. [f_i] définit f_i comme étiquette d'une formule ou d'une règle pour pouvoir y faire référence dans le texte,
. [x := e]p est le prédicat égal au prédicat p dans lequel toutes les occurrences libres de *x* ont été substituées par *e*. Ici la notation "[x := e]" représente la substitution de variable x par l'expression e. ♦

1. Notion de valeur symbolique de variables

Définition 5 ***(valeur symbolique)***

La *valeur symbolique* d'une variable est un ensemble de valeurs défini par une expression logique. ♦

Dans la suite, nous confondons la valeur symbolique et l'expression qui la définit.

Par extension, la valeur symbolique de deux variables est un ensemble de couples de valeurs défini par une expression logique. On peut généraliser cette notion à un nombre quelconque de variables.

Exemple 7 ***(valeurs symboliques)***

- l'expression logique n≥0 définit l'ensemble infini des valeurs entières positives ou nulles,

- L'expression logique $n \geq r^2$ définit l'ensemble des couples de valeurs entières *(x, y)* où *x* est la valeur de *n* et *y* la valeur de *r* telles que $x \geq y^2$. Cet ensemble est partiellement représenté par tous les points de la Figure 8 dans un repère cartésien d'abscisse *n* et d'ordonnée *r*.
- L'expression logique $n \geq r^2 \wedge n < (r+1)^2$ définit l'ensemble des couples (nombre, racine carrée du premier nombre) représenté sur la Figure 8 par les points qui sont sur les lignes de l'escalier. ♦

On notera que l'expression qui définit une valeur symbolique est l'expression caractéristique de la définition en intention (on dit également en compréhension) de l'ensemble des valeurs que représente la valeur symbolique.

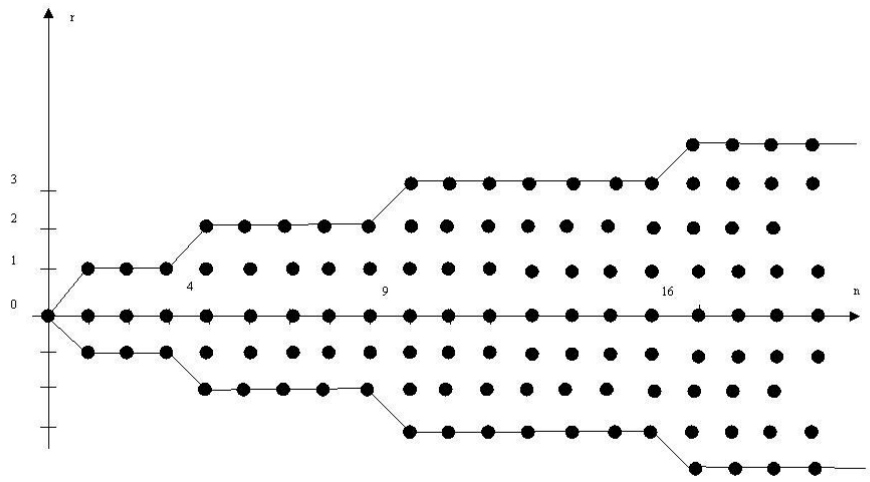


Figure 8 : représentation des valeurs symboliques $n \geq r^2$ **et** $n \geq r^2 \wedge n < (r+1)^2$

Par exemple l'ensemble des couples suivant :
{(0,0) , (1, 1), (2, 1), (3, 1), (4, 2), (5, 2), (6, 2), (7, 2), (8, 2), (9, 3), (10, 3), ..., (15, 3), (16, 4), (17, 4), ...}
peut être défini ainsi en intention : $\{(n, r) \mid n \geq 0 \wedge r \geq 0 \wedge n \geq r^2 \wedge n < (r+1)^2\}$.

L'expression caractéristique de cet ensemble est $n \geq 0 \wedge r \geq 0 \wedge n \geq r^2 \wedge n < (r+1)^2$. Par abus de notation, nous noterons $\{n \geq 0 \wedge r \geq 0 \wedge n \geq r^2 \wedge n < (r+1)^2\}$ la valeur symbolique des variables *n* et *r*. Nous nous servons de ces valeurs symboliques comme des annotations dans le programme.

Une autre manière de voir les choses est de dire qu'un *n-uplet de valeurs des variables* d'un programme représente un *état mémoire*. Une valeur symbolique des variables d'un programme est donc un ensemble d'états mémoire.

Par exemple le couple *(n, r) = (3, 0)* est un état de la mémoire du programme présenté dans la Figure 10. La valeur symbolique $\{n \geq 0 \wedge r \geq 0 \wedge n \geq r^2 \wedge n < (r+1)^2\}$ est l'ensemble des états mémoires *(n, r)* suivants : {(0,0) , (1, 1), (2, 1), (3, 1), (4, 2), (5, 2), (6, 2), (7, 2), (8, 2), (9, 3), (10, 3), ..., (15, 3), (16, 4), (17, 4), ...} qui est représenté par l'escalier de la Figure 8 qui est au-dessus de l'axe des abscisses.

2. Notion d'exécution avec des valeurs symboliques

L'exécution d'un programme A sur une valeur symbolique de données définies par l'expression $\{p\}$ qui donne une valeur symbolique résultat définie par l'expression $\{q\}$ est notée $\{p\}A\{q\}$. Cette notation signifie que si on exécute le programme A sur n'importe quelle donnée de l'ensemble $\{p\}$, on obtiendra une valeur résultat appartenant à l'ensemble $\{q\}$. Elle peut être interprétée comme une formule logique qui est vraie si le

programme A appliqué sur n'importe quelle valeur de donnée de l'ensemble {p}, conduit à un résultat de l'ensemble {q}. La formule {p}A{q} est formée d'un programme A et de deux annotations p et q.

Exemple 8 ***(formules de Hoare)***

- {n≥0} r := 0 {n≥0 ∧ r^2≤n} signifie que l'exécution de l'instruction *r := 0* transforme n'importe quel état mémoire *(n, r)* avec *n* positif ou nul et *r* quelconque en un état mémoire *(n, r)* avec *n* positif ou nul et *r* élevé au carré inférieur ou égal à *n*. Cette formule est vraie. Elle est démontrable.
- {$(r+1)^2$≤n} r := r+1 {r^2≤n} signifie que l'exécution de l'instruction *r := r+1* transforme n'importe quel état mémoire *(n, r)* avec *n* supérieur ou égal à $(r+1)^2$ en un état mémoire *(n, r)* avec *n* supérieur ou égal à r^2. Cette formule est vraie.
- {n≥0} rmin :=0 ; rmax:=n+1 {$rmin^2$≤n ∧ n<$rmax^2$} signifie que l'exécution séquentielle des 2 instructions transforme n'importe quel état mémoire *(n, rmin, rmax)* avec *n* positif ou nul, *rmin* et *rmax* quelconques en un état mémoire *(n, rmin, rmax)* avec $rmax^2$ supérieur strictement à *n* et *n* supérieur ou égal à $rmin^2$, c'est à dire où *rmin* et *rmax* sont un encadrement de la racine carrée de *n*. Cette formule est également vraie.
- {n≥0} rmax:=n {n<$rmax^2$} est une formule de la logique de HOARE qui est fausse quand on l'interprète dans les entiers naturels car pour n=0, il n'est pas vrai que n soit inférieur à $rmax^2$ après l'affectation *rmax:=n* car 0<0^2est une expression logique fausse. Elle est donc non démontrable. ♦

3. Exemple d'exécution symbolique

Pour illustrer la notion d'exécution symbolique, nous présentons l'exemple simple du calcul de la racine carrée entière par défaut. La spécification est identique à celle que nous avons présentée dans la leçon 1 dans la Figure 3. Par contre le programme de la Figure 9 est différent et plus simple que la dichotomie appliquée dans le programme de la leçon 1 Figure 4. Le calcul de la racine est effectué par incrémentations successives. L'idée consiste à choisir une approximation par valeur inférieure de la racine, puis à incrémenter cette valeur par pas de un jusqu'à ce que la valeur suivante élevée au carré soit plus grande que *n*.

Début r :=0 ; **tantque** n≥$(r+1)^2$ **faire** r := r+1 **fait fin**.

Figure 9 : programme racine par incrémentation

L'exécution symbolique de ce programme est représentée par le programme annoté décrit dans la Figure 10.

Début
{0≤n}
r :=0 ;
{r^2≤n}
tantque n≥$(r+1)^2$ **faire**
{(r^2≤n) ∧ (n≥$(r+1)^2$)}
r := r+1
{r^2≤n}
fait
{(r^2≤n) ∧ (n<$(r+1)^2$)}
fin.

Figure 10 : programme de racine annoté par des valeurs symboliques

Ces annotations indiquent qu'on débute l'exécution avec une valeur quelconque pour la donnée *n* positive ou nulle. Notons que cette annotation est exactement la pré-condition énoncée dans la spécification. Ce n'est pas un hasard, mais un choix délibéré puisque nous voulons montrer que l'exécution du programme transforme toutes données satisfaisant la pré-condition en un résultat satisfaisant la post-condition.

L'exécution de *r := 0* conduit à un état mémoire qui satisfait la condition r^2≤n. On peut justifier cette transformation en raisonnant en avant ou en arrière. Le raisonnement en avant est le suivant : étant donné que *r*

désigne la valeur *0* après exécution de l'instruction *r := 0*, on peut remplacer *0* de l'expression $0 \leq n$ qui était vraie avant par r^2, qui est égal à 0, et on obtient l'expression après $r^2 \leq n$. Mais ce raisonnement en avant ne justifie pas le choix de l'expression après $r^2 \leq n$. On aurait pu appliquer le même raisonnement avec d'autres expressions comme $r \leq n$ ou $r^3 \leq n$ ou même $0 \leq n$. Le choix de l'expression après a été effectué, pour aboutir au résultat final du programme. Nous verrons plus tard comment. En admettant que nous avons choisi l'expression $r^2 \leq n$ après et que nous savons pourquoi, le raisonnement en arrière est le suivant : pour aboutir à la condition $r^2 \leq n$ après avoir exécuté l'instruction *r := 0*, il suffit que l'expression $[r := 0]r^2 \leq n$ soit satisfaite avant. Par substitution de *r* par zéro, cette expression est équivalente à $0 \leq n$.

Si la condition $r^2 \leq n$ est vraie avant le *tantque* alors elle reste vraie à l'intérieur juste après l'évaluation de la condition $n \geq (r+1)^2$ puisque celle-ci ne modifie pas les valeurs des variables. Comme cette condition doit être vraie pour entrer dans le *tantque*, la condition $(r^2 \leq n) \wedge (n \geq (r+1)^2)$ est satisfaite au début de l'exécution du premier pas d'itération.

Le raisonnement en avant est le suivant : le pas d'itération exécute *r := r+1* qui aboutit à la condition $(n \geq r^2)$ car, la condition $(n \geq (r+1)^2)$ étant vraie avant et la valeur de *r* devenant celle de *r+1*, on peut substituer *r* par *r-1* dans la condition $(n \geq (r+1)^2)$ pour obtenir la condition $(n \geq (r-1+1)^2)$ équivalente à $(n \geq r^2)$. Le raisonnement en arrière est que pour aboutir à $n \geq r^2$, l'affectation *r := r+1*doit partir d'un ensemble d'états défini par la substitution suivante : $[r := r+1](n \geq r^2)$ qui est équivalente à l'expression $n \geq (r+1)^2$.

La condition $(n \geq r^2)$ étant vraie à la fin du premier pas d'itération, l'évaluation de la condition s'effectue donc dans les mêmes conditions que la première fois et l'exécution de n'importe quel pas d'itération aboutit toujours au même ensemble d'états mémoire jusqu'à ce que la condition d'entrée dans l'itération soit fausse. Alors, après l'itération, deux conditions sont satisfaites :

- $(n \geq r^2)$ qui reste toujours vraie après chaque pas d'itération, donc qui est vraie après le dernier ; on appelle cette condition un *invariant*.
- $n < (r+1)^2$ qui est la condition d'arrêt (de terminaison) de l'itération (négation de $n \geq (r+1)^2$).

La condition finale obtenue étant exactement identique à la post-condition énoncée dans la spécification, cette exécution symbolique montre que le programme est correct vis à vis de sa spécification, c'est à dire que partant d'un état mémoire qui vérifie les conditions fixées sur les données (c'est à dire la pré condition) dans la spécification, l'exécution aboutit à un état mémoire où le résultat répond à la définition énoncée dans la spécification (c'est à dire la post-condition).

Revenons sur la notion d'invariant d'une itération. Une itération est un programme de la forme suivante : *Init ;* ***tantque*** *B* ***faire*** *A* ***fait***. Celui-ci calcule un résultat par une recherche dans un espace de recherche défini par un prédicat I. La recherche est en trois étapes :

1. Il détermine une première approximation du résultat qui appartient à l'espace de recherche I en appliquant le programme *Init*,
2. puis, il détermine une meilleure approximation appartenant toujours à l'espace I en appliquant le programme A,
3. l'étape numéro 2 est répétée jusqu'à ce que l'approximation trouvée satisfasse $\neg B$, c'est à dire appartienne à l'ensemble $I \wedge \neg B$. Dans ce cas, l'itération termine et le résultat a été trouvé.

Idée Clé 7 ***(itération annotée par un invariant)***

Ce processus de calcul est modélisé par le programme annoté suivant :

Init ;$\{I\}$ ***tantque*** *B* ***faire*** $\{I \wedge B\}$ *A* $\{I\}$ ***fait*** $\{I \wedge \neg B\}$. ♦

Noter que toute itération est caractérisée par des invariants. Noter également que cette notion nous permet de raisonner sur un nombre quelconque de pas d'itération en raisonnant sur l'exécution symbolique d'un seul pas.

Exemple 9 ***(itération annotée par un invariant)***

L'itération de l'algorithme de la Figure 4 est annotée selon le schéma ci-dessus ainsi :

Rmax := n+1 ; Rmin := 0 ;$\{Rmin^2 \leq n \wedge n < Rmax^2\}$

tantque $Rmin+1 \neq Rmax$ **faire**

$\{Rmin^2 \leq n \wedge n < Rmax^2 \wedge Rmin+1 \neq Rmax\}$

M := (Rmin+Rmax+1) **div** 2 ; **si** M*M≤n **alors** Rmin := M **sinon** Rmax := M **finsi**

{$Rmin^2 \leq n \wedge n < Rmax^2$}

fait {$Rmin^2 \leq n \wedge n < Rmax^2 \wedge Rmin+1=Rmax$} ♦

4. Résumé

Une expression logique p représente un ensemble d'états mémoire appelé valeur symbolique des variables.

Une exécution symbolique est une exécution qui transforme une valeur symbolique des variables en une autre valeur symbolique ; On note {p}A{q} une exécution symbolique qui transforme la valeur symbolique p en une valeur symbolique q.

Une exécution symbolique est donc une transformation d'annotations (donc de prédicats).

La vérification qu'un programme A qui résout un problème spécifié par un couple (p, q), où p est une pré-condition et q une post-condition, consiste à vérifier que l'exécution symbolique de A à partir de p conduit à q, c'est à dire que {p}A{q} est une formule vraie.

Pour définir un système d'exécution symbolique, il faut définir des règles qui indiquent comment chaque forme de programme transforme des prédicats.

En résumé, pour vérifier des programmes relativement à des spécifications, il faut trois éléments parfaitement définis :

- un langage de spécification pour décrire les prédicats pré et post-conditions,
- un langage de description de programmes,
- un ensemble de règles d'exécution symbolique qui transforment des prédicats.

C'est ce que nous décrivons respectivement dans les leçons 3, 4 et 5.

Leçon 3 : Le langage de programmation

Objectifs

. Décrire le langage de programmation utilisé pour la vérification dans la logique de HOARE,
. Définir les limites de ce langage par rapport à un langage de programmation réel comme C ou Java et justifier ses limites par rapport à la difficulté de définir les règles de transformation de prédicats.

La leçon 3 est décomposée en quatre sections. La section 1 présente la syntaxe du langage de programmation. La section 2 décrit les restrictions de ce langage par rapport à un langage de programmation à la Pascal. La section 3 présente la sémantique du langage de programmation et enfin la section 4 donne quelques exemples.

1. Syntaxe du langage de programmation

Ce langage est appelé langage des *tantques programmes* par HOARE.

Soit <Var> un ensemble de noms de variables. Les types de ces variables sont limités aux suivants : *entier*, *booléen* et *ensembles énumérés* définis par le programmeur. Notons que nous ne prenons pas en compte des données de type caractère et chaîne de caractères pour simplifier. Leur prise en compte ne poserait pas de difficultés particulières. Les seules structures de données prises en compte sont les tableaux dont l'utilisation est légèrement restreinte dans les expressions. Les déclarations de variables apparaissent au début du programme dans la clause *variables*. Par exemple dans un programme on déclare les trois variables de nom *g*, *d* et *m* ainsi : **variables** g, d, m : 0..MAX. Ces variables sont de type entier compris entre 0 et MAX, MAX étant une constante définie précédemment.

Soit <Expr> l'ensemble des expressions que l'on peut former avec ces types de variables, les parenthèses et les opérateurs suivants munis des priorités habituelles :

- +, -, *, *div* et *mod* sur les entiers,
- *non*, *et*, *ou* sur les booléens,
- <, >, =, ≠, ≥, ≤ comme opérateurs de comparaison entre valeurs de même type (entier, booléen et ensemble énumérés).

Définition 6 ***(tantque programmes)***

Un *tantque* programme est une séquence d'instructions entre les mots clés *début* et *fin* notée : **début** <Séquence d'Instructions> **fin** où :

- si A_1, A_2, …, A_n sont des instructions (appartenant à l'ensemble <Instruction>) alors A_1 ; A_2 ; …; A_n est une <Séquence d'Instructions>. ♦

Définition 7 ***(Instructions des tantques programmes)***

Une instruction est l'un des éléments de l'ensemble <Instruction> composé des quatre instructions suivantes :

- <Var> := <Expr>, appelée affectation,
- **si** <Expr> **alors** <Séquence d'Instructions> **finsi**, appelée conditionnelle,
- **si** <Expr> **alors** <Séquence d'Instructions> **sinon** <Séquence d'Instructions> **finsi**, appelée conditionnelle,
- **tantque** <Expr> **faire** <Séquence d'Instructions> **fait**, appelée *itération.* ♦

Les programmes peuvent être commentés par des textes compris entre "(*" et "*)" ou entre "/*" et "*/".

Exemple 10 ***(Tantque programmes)***

Voir les exemples Figure 12, Figure 14 et Figure 16 section 4 de cette leçon.

2. Restrictions du langage de programmation

C'est un sous-ensemble des langages impératifs comme C, Java ou Pascal. Les principales restrictions sont :

- l'absence de données de type pointeur (donc pas de prise en compte de la notion d'objet),
- l'utilisation restreinte des tableaux (pas d'utilisation de plusieurs indices pouvant être égaux),
- l'absence d'appel de procédures.

Ces constructions invalident dans certains cas la théorie de la vérification de la logique de HOARE à cause du problème de *synonymie* des variables rendues possible par ces constructions. Sous certaines conditions on peut étendre la théorie pour traiter ces constructions. C'est ce que font les environnements existants comme Frama-C et Keys. Pour l'extension aux procédures, voir [Berlioux 83] où sont présentées les conditions à vérifier par les paramètres et les appels de procédure afin de pouvoir appliquer une règle d'appel de procédure par substitution des paramètres formels par leurs valeurs effectives.

Définition 8 ***(variables synonymes)***

Deux variables sont *synonymes* si elles désignent le même emplacement mémoire. ♦

Exemple 11 ***(variables synonymes, les pointeurs)***

Par exemple, l'affectation *p := q* où *p* et *q* sont des pointeurs crée une synonymie entre la valeur pointée par *p* (dénotée par *p^* en Pascal ou par *p** en C) et celle pointée par *q*. Les variables *p** et *q** sont synonymes. ♦

Le problème de la synonymie est le suivant : si une affectation (par exemple : *p* := 4*) a lieu alors non seulement celle-ci modifie la valeur désignée par *p**, mais également celle désignée par la variable synonyme *q**. Cette modification de *q** est totalement invisible dans l'affectation *p* := 4* car *q** n'apparaît pas dans l'instruction d'affectation *p* := 4*. On dit que l'affectation *p* := 4* a un effet de bord sur *q**.

Définition 9 ***(Régle d'affectation)***

La règle d'exécution symbolique d'une affectation *x := e* est la suivante : {[x :=e]P} x :=e {P}.♦

Elle signifie que pour que le prédicat P soit vrai après l'exécution de *x := e*, il faut que le prédicat {[x :=e]P soit vrai avant l'affectation. Le prédicat représente le prédicat P dans lequel on a substitué toutes les occurrences libres de *x*.

Exemple 12 ***(Application de la règle d'affectation)***

Exemples : $\{0^2 \le n\}$ r :=0 $\{r^2 \le n\}$, $\{(r+1)^2 \le n\}$ r :=r+1 $\{r^2 \le n\}$, $\{r^2 \le n\}$ p :=r+1 $\{r^2 \le n\}$, $\{aux+2=3 \wedge \forall k.(k \in 1..i-1 \Rightarrow t[k]<aux+2\}$ t[i] :=aux+2 $\{t[i]=3 \wedge \forall k.(k \in 1..i-1 \Rightarrow t[k]<t[i]\}$♦

Idée Clé 8 ***(les variables synonymes invalident la règle d'affectation)***

Les variables synonymes mettent en défaut le système de vérification car celui-ci utilise la substitution qui ne prend en compte que les modifications qui apparaissent explicitement dans le texte du *tantque* programme. ♦

Exemple 13 ***(autres variables synonymes)***

La notion de paramètre passé par adresse rend également synonymes le paramètre effectif qui apparaît dans l'instruction d'appel et le paramètre formel qui apparaît dans l'entête et le corps de la procédure ou de la fonction.

Les expressions de tableaux de la forme *t[i]* et *t[j]* permettent également de réaliser des synonymies si *i=j*. C'est pourquoi, nous prendrons des exemples de programmes avec des tableaux, mais dans lesquels, nous n'autorisons plusieurs expressions de la forme *t[i], t[j]* qu'à condition que *i* soit différent de *j*. ♦

Exemple 14 ***(variables synonymes, substitution et invalidation de la règle d'affectation)***

L'expression $\{p^*=4 \wedge q^*=2\}$ n'est pas vraie après exécution du programme suivant : p := q ; q* := 2 ; p* := 4 alors qu'on peut démontrer que toutes les valeurs symboliques qui décorent ce programme ainsi : {vrai} p := q ; {vrai} q* := 2 ; {q*=2}; p* := 4 $\{p^*=4 \wedge q^*=2\}$ sont vraies par substitution. L'erreur vient du fait que l'instruction q* := 2 n'engendre une substitution que sur q* alors que cette instruction modifie également la variable synonyme p*. Il en est de même sur le programme suivant *i:=4 ; j:=4 ; t[i]:=5 ; t[j]:=4*

qui permet d'établir par substitution {vrai} i:=4 ; j:=4 ; t[i]:=5 ; t[j]:=4 {t[j]=4 ∧ t[i]=5}. La substitution étant établie à partir de la syntaxe, elle ne répercute pas l'effet de bord de l'instruction sur les variables synonymes. ♦

3. Sémantique du langage de programmation

Le langage des *tantques* programmes possède la sémantique habituelle des langages de programmation impératifs :

- *x := e* signifie que la valeur de l'expression *e* est affectée à la variable *x*,
- A_1 ; A_2 signifie que A_1 est exécutée, puis A_2,
- ***Si*** *e* ***alors*** *A* ***finsi*** signifie que si *e* a pour valeur de vérité vrai, *A* est exécutée, sinon aucune instruction n'est exécutée et l'exécution se poursuit par l'instruction qui suit le *si … alors … finsi*.
- ***Si*** *e* ***alors*** A_1 *sinon* A_2 ***finsi*** signifie que si *e* a pour valeur de vérité vrai, A_1 est exécutée, sinon A_2 est exécutée.
- ***Tantque*** *e* ***alors*** *A* ***fait*** signifie que A est exécuté tant que *e* a pour valeur de vérité vrai ; lorsque *e* a pour valeur de vérité faux, l'exécution se poursuit par l'instruction qui suit le *tantque*.

4. Exemples de programmes

Dans la Figure 4 et la Figure 9, nous avons présenté deux *tantque* programmes réalisant le calcul de la racine carrée entière par défaut d'un nombre *n* positif ou nul.

Ci-dessous, nous donnons trois autres exemples :

- le calcul de factorielle *n*,
- la recherche dichotomique de la position d'un élément dans une séquence,
- le tri d'une séquence de nombres entiers par la technique appelée «tri bulle».

4.1 factorielle n

Soit $f \stackrel{def}{=} n\,!$. *n* est un nombre entier positif ou nul donné, *f* est le résultat. Si *n = 0* alors $f \stackrel{def}{=} 1$ sinon $f \stackrel{def}{=} 1 * 2 * 3 * ... * n$. Cette spécification est décrite dans la Figure 11. La Figure 12 décrit un programme qui calcule *f* égal à factorielle *n* selon sa définition donnée en Figure 11. Noter que ce programme utilise la donnée *n* pour laquelle on fait l'hypothèse qu'elle satisfait la pré-condition posée dans la Figure 11.

Données *n* : entier,
Résultat *f* : entier, factorielle *n*.
Pré-condition : $n \geq 0$
Post-condition : $f = \prod_{j=1}^{n} j$

Figure 11 : spécification de factorielle

```
Procédure factorielle (données n : entier ; resultat f : entier) ;
Variable i : entier ;
Début
    i := 0 ; f := 1 ; /* f=factorielle 0 */
    tantque i≠n faire
            i := i+1 ; f := f*i /* f=factorielle i */
    fait
    /* i=n et f=factorielle i, donc f=factorielle n */
fin.
```

Figure 12 : programme de calcul de factorielle

4.2 Recherche dichotomique

Ce problème consiste à calculer la position où insérer un élément *x* dans une séquence de *n* entiers, représentée par les éléments d'un tableau *t* entre les indices 1 et *n* ; t[1..n] est classé par ordre croissant. La spécification de ce problème est décrite dans la Figure 13. La Figure 14 décrit un programme de calcul de *posx* par recherche dichotomique de *x* dans t[1..n].

Constante Max =100, nombre maximum d'éléments dans la tableau

Données

t : tableau de *n* entiers classés en ordre croissant

x : entier dont on recherche la position d'insertion dans la séquence *t[1..n]*

Résultat *posx* : entier compris entre *1* et *n+1* ; position d'insertion de *x* dans *t*.

Pré-condition : *t* est un tableau éventuellement vide trié en ordre croissant ; ceci est exprimé formellement ainsi :

$n \geq 0 \wedge n \leq MAX \wedge \forall i.\forall j.((i \in [1..n] \wedge j \in [1..n] \wedge i<j) \Rightarrow t[i] \leq t[j])$

Post-condition : tous les éléments de *t* de position comprise entre 1 et *posx-1* sont inférieurs ou égaux à *x*, ceux de position comprise entre *posx* et *n* sont supérieurs à *x*. Ceci s'exprime formellement ainsi :

$\forall k.(k \in [1..posx-1] \Rightarrow t[k] \leq x) \wedge \forall k.(k \in [posx..n] \Rightarrow t[k]>x)$

Figure 13 : spécification de la recherche dichotomique

```
Procédure Dichotomie (données n : 0..MAX ; t : tableau[1..MAX] d'entier, x : entier ;
                      résultat posx : 1..MAX+1) ;
Variables min, max, m : 0..MAX ; (* m : milieu de l'intervalle min..max.*)
   (* min : indice tel que t[1..min] est composé d'éléments inférieurs ou égaux à x *)
   (* max : indice tel que t[max+1..n] est composé d'éléments supérieurs à x *)
Début
   min := 0 ; max := n ;
   /* I  def=  t[j]≤x pour j ∈ [1..min] et t[j]>x pour j ∈ [max+1..n] */
   tantque min≠max faire
          /* min≠max et I */
          m := (min+max+1) div 2 ;
          si t[m]≤x alors min := m
          sinon max := m-1 finsi
          /* I */
   fait ;
   /* min=max et I, donc la position d'insertion et min+1 ou max+1 qui
   sont l'indice du premier élément de t supérieur à x*/
   posx := min+1
fin.
```

Figure 14 : programme de recherche dichotomique

Le programme de la Figure 14 calcule le bon résultat car à chaque étape de l'itération il établit la condition I. Si il termine, I étant vrai, on en déduit facilement que le résultat est *min+1* ou *max+1*. Par contre pour justifier que le programme s'arrête toujours, c'est plus difficile. En effet, pour qu'il s'arrête, il faut justifier qu'à chaque étape, soit *min* augmente, soit *max* diminue de telle sorte que min=max devienne vrai, c'est-à-dire que l'intervalle de recherche *min+1..max* devienne vide. Or le programme affecte *min* par *m* et *max* par *m-1*. Autrement dit, étant donné le calcul de *m*, est-on sûr que *m* est toujours plus grand que *min* ou que *m-1* est toujours plus petit que *max* ?

Exercice 1 ***(arrêt de programme)***

Justifier que le programme de la Figure 14 termine toujours. ♦

4.3. Tri bulle

Ce problème consiste à classer par ordre croissant une séquence de *n* entiers, représentée par les éléments d'un tableau *tr* entre 1 et *n*, à partir d'une séquence, représentée par le tableau *td*, composée des mêmes éléments qui sont dans un ordre quelconque. Soit *max* le nombre maximum d'éléments des tableaux *td* et *tr*.

Constante Max =100, nombre maximum d'éléments dans la tableau

Données td : tableau de *n* entiers classés dans un ordre quelconque.

n : nombre de valeurs dans *td*

Résultat tr : tableau de *n* entiers classés en ordre croissant

Pré-condition : *td* est un tableau éventuellement vide ; ceci est exprimé formellement ainsi : $n \geq 0 \wedge n \leq max$

Post-condition : *tr* est un tableau trié par ordre croissant et contient les éléments de *td* ; ceci s'exprime formellement ainsi : $\forall k.\forall u.((k \in [1..n] \wedge u \in [1..n] \wedge k<u) \Rightarrow tr[k] \leq tr[u])$

$\wedge\ ran(tr)=ran(td)$

Figure 15 : spécification du tri bulle

```
Procédure Tri_Bulle (donnée n : 0..MAX ; donnée td : tableau[1..MAX] d'entier ;
                     résultat tr : tableau[1..MAX] d'entier) ;
Variables i, j : 0 .. max ; (* i est un indice tel que t[1..i] est trié *)
          (* j est un indice de parcours de t *)
      aux : entier ;
Début
   i := 0 ; tr := td ; /* copie des éléments du tableau */
   /*I ≝ tr[1..i] trié et tr[j]≥tr[i] pour j ∈ [i+1..n] */
   tantque i<n-1 faire
          /* i<n-1 et I */
          /* placement du minimum de tr[i+1..n] en position i+1 */
          j := n ;
          /* I' ≝ tr[j] est le minimum de tr[j..n] et tr[j]≥tr[i] */
          tantque j-1>i faire
                /* j>i et I' */
                si tr[j]<tr[j-1] alors
                      /* échanger tr[j] et tr[j-1] */
                      aux := tr[j] ; tr[j] := tr[j-1] ; tr[j-1] := aux
                finsi ;
                j := j-1 ;
                /* I' */
          fait ;
          /*j-1=i et I' donc tr[1..i+1] trié ; fin du placement du minimum */
          i := i+1
   fait
   /* i=n-1 et I, donc tr[1..n] trié */
fin.
```

Figure 16 : programme du tri bulle

5. Résumé

Nous avons défini un langage de programmation simplifié permettant d'exprimer des programmes de calcul de valeurs et de collections de valeurs de type booléen et entier.

Pour représenter les collections, nous utilisons des tableaux à condition de ne jamais avoir des éléments de tableau synonymes.

Pour raisonner sur ces programmes, nous définissons un langage d'expressions logiques à la leçon suivante. Ces expressions permettront de spécifier et d'annoter les programmes par des valeurs symboliques.

6. Exercices

Exercice 2 ***(Et bit à bit)***

Décrire un programme qui calcule le résultat *n* de l'opérateur Et bit à bit entre deux entiers n_1 et n_2. Par exemple EtbitAbit (11, 7)=3. En binaire, cette opération réalise le Et bit à bit entre 1011 et 111 qui est égal à 11. Noter que cet opérateur est dénoté & en java et se distingue de l'opérateur booléen dénoté && en java. L'opérateur & permet de réaliser très rapidement l'opérateur d'intersection de deux ensembles représentés par une séquence de bits. Par exemple, un ensemble de nombres compris entre 0 et 15 peut être représenté par un nombre binaire *b* de 16 bits tels que b[i]=1 si *i* appartient à l'ensemble et b[i]=0 si *i* n'appartient pas à l'ensemble. Avec cette représentation 11 (1011 en binaire) représente l'ensemble {3, 1, 0} et 7 (111 en binaire) représente l'ensemble {2, 1, 0}. EtbitAbit(11, 7) = 3 (11 en binaire) qui représente l'intersection de ces deux ensembles, {1, 0}.♦

Exercice 3 ***(pgcg de a et b)***

Décrire un programme qui calcule le pgcd de deux nombres entiers positifs ou nuls *a* et *b*. Le pgcd est défini ainsi :

- pgcd(a, 0) $\stackrel{def}{=}$a,
- pgcd (a, b) $\stackrel{def}{=}$pgcd (b, a mod b). ♦

Exercice 4 ***(tri bulle)***

Est-ce que le programme de la Figure 16 termine toujours. Justifier ♦

Leçon 4 : La logique des prédicats du premier ordre

Pré requis

. Savoir lire et interpréter les expressions logiques telles qu'elles ont été introduites en Méthodes Formelles.

Objectifs

. S'approprier le langage de la logique des prédicats du premier ordre (LP1) pour décrire des spécifications et des annotations de programmes définissant des valeurs symboliques. Le but est de maîtriser la formalisation de propriétés exprimées informellement et de maîtriser l'interprétation des expressions formelles décrites en logique des prédicats du premier ordre.

. S'approprier quelques techniques de réécriture de propriétés en propriétés équivalentes.

. S'approprier quelques méthodes (ou stratégies) pour vérifier la validité de formules de LP1.

La leçon 4 est décomposée en huit sections. La section 1 présente la syntaxe de la *logique des prédicats du premier ordre*. La section 2 décrit sa sémantique et quelques propriétés. La section 3 présente des exemples d'utilisation pour décrire des spécifications. La section 4 présente quelques techniques de réécriture de prédicats en prédicats équivalents. Ce sont les techniques les plus couramment utilisées pour pouvoir appliquer des substitutions de variables par des expressions et pour justifier qu'un prédicat est valide. La section 5 présente les trois stratégies les plus couramment employées pour justifier qu'un prédicat est valide. Enfin la section 6 est un résumé des principales idées de ce chapitre. Elle est suivie d'exercices en section 7 et d'explications plus détaillées sur les notions de formules satisfiables et valides en section 8.

Cette leçon n'est pas un cours de logique des prédicats du premier ordre. Son objectif est essentiellement de maîtriser l'interprétation d'une formule en un énoncé informel et réciproquement. Le lecteur intéressé pourra consulter le chapitre 11 de [Dehornay 00].

1. Syntaxe de la logique des prédicats du premier ordre

Nous faisons un bref rappel de cette notion en supposant que le lecteur a suivi un cours de Méthodes Formelles introduisant le calcul des propositions et le calcul des prédicats.

Notation 2 ***(prédicats)***

Les formules de logique des prédicats du premier ordre sont établies à partir :

- de termes qui sont des formules élémentaires à valeur booléenne (exemples : $r^2 \leq n$, $n \geq (r+1)^2$, $i \in [1..n]$, $min = max+1$, $m \neq 0$, *vrai*, *faux*, …),
- d'opérateurs logiques *non, et, ou, implique* et *si et seulement si* dénotés : $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$, …
- de quantificateurs universels et existentiels dénotés respectivement ainsi : $\forall$ et $\exists$,
- de variables dénotées *x, y, z,* …,
- de symboles de liaison comme le point et de délimitation de portée comme les parenthèses.

Définition 10 ***(syntaxe des prédicats)***

Les formules du premier ordre, appelées *prédicats*, sont définies inductivement ainsi, soit *x* une variable, soit *t* un terme et soit p, p_1, p_2 des prédicats :

- t est un prédicat,
- $\neg p$, $p_1 \wedge p_2$, $p_1 \vee p_2$, $p_1 \Rightarrow p_2$, $p_1 \Leftrightarrow p_2$ sont des prédicats,
- $\forall x.p$ et $\exists x.p$ sont des prédicats,
- (p) est un prédicat. ♦

Remarques *:*

Les parenthèses sont utilisées pour lever toute ambiguïté dans les formules. En l'absence de parenthèse, les règles de priorité des opérateurs qui sont utilisées sont les suivantes :

- les éléments les plus prioritaires sont les *termes*,
- puis l'opérateur ¬,
- puis les opérateurs ∧ et ∨,
- puis les opérateurs ⇒ et ⇔.

Les formules avec quantificateurs suivantes ∀x.(d⇒p) et ∃x.(d∧p) de la forme ∀x.p' et ∃x.p' où, en général d est un terme de la forme x∈D. Elles sont souvent notées respectivement ainsi : ∀x:D.p et ∃x:D.p ou ∀x∈D.p et ∃x∈D.p dans les cours de Mathématiques pour l'Informatique. Dans ces notations, le prédicat *d* définit le domaine de la variable *x*, c'est à dire le type (l'ensemble des valeurs, noté D) de cette variable et le prédicat p la condition que *x* doit vérifier.

Par exemple, le prédicat suivant définit que tous les éléments d'un tableau *t* indicés par une valeur comprise entre 1 et *posx* sont inférieurs ou égaux à *x* : ∀i.(i∈[1..posx] ⇒ t[i]≤x). Mot à mot, ce prédicat signifie que pour tout indice *i*, si *i* est l'une des valeurs comprises entre 1 et *posx* alors *t[i]* doit être inférieur ou égal à *x* pour que le prédicat soit vrai sur les objets *t*, *x* et *posx*.

Il est quelques-fois décrit ainsi : ∀i:[1..posx].t[i]≤x avec :

- d $\stackrel{def}{=}$ (i ∈ [1..posx]), D $\stackrel{def}{=}$ [1..posx],
- et p $\stackrel{def}{=}$ (t[i]≤x).

Notation 3 ***(prédicats quantifiés)***

Dans la suite, nous utilisons toujours les notations de prédicat quantifié sous la forme de la définition de la Définition 10. Nous adoptons cette notation, qui est celle du langage B, qui remplace l'opérateur " ." par les opérateurs booléens ⇒ et ∧ et qui remplace l'opérateur " :" par " ∈" :

- dans ∀i:d.p, il a la signification de l'opérateur *implique* (∀i:d.p signifie ∀i.((i∈d)⇒p),
- dans ∃i:d.p, il a la signification de l'opérateur *et* (∃i:d.p signifie ∃i.((i∈d)∧p).

Nous pouvons voir dans la Figure 17 que la valeur de vérité de (i∈d)⇒p est différente de celle de (i∈d)∧p.

Notons que dans un prédicat peuvent apparaître deux sortes de variables :

- des variables liées par un quantificateur (exemple : *i*) ; ces variables sont appelées *variables liées*,
- des variables non liées par un quantificateur (exemple : *t*, *posx*, *x*) ; ces variables sont appelées *variables libres* ; dans notre contexte, ce sont toujours des variables du programme qui est annoté par le prédicat.

Notation 4 ***(Règles que les prédicats doivent respecter)***

- toutes les variables du programme doivent être libres,
- les autres variables doivent être liées par l'un des quantificateur ∀ ou ∃.

La présence d'une variable libre qui ne serait pas une variable du programme révélerait une erreur de spécification. Soit cette variable doit disparaître, soit elle doit être quantifiée. De manière générale, ce sont des variables qui modélisent l'objet à propos duquel le prédicat décrit une propriété.

2. Sémantique et propriétés de la logique des prédicats du premier ordre

Nous décrivons l'interprétation des prédicats dans le domaine des booléens {Vrai, Faux}.

2.1. Interprétation des prédicats

Sur le domaine {Vrai, Faux}, les opérateurs booléens dénotés ¬, ∧, ∨, ⇒, ⇔ sont définis par les tables de vérité de l'algèbre de Boole. Par exemple, soit *I(p)* la valeur de vérité du prédicat p_i, la table de vérité des opérateurs *et* et *implique* noté ⇒ est décrite dans la Figure 17.

On notera que $p \Rightarrow q$ est faux dans un seul cas, *Vrai implique Faux* (*Vrai*⇒*Faux*). On notera également que :

- Faux⇒q ≡ Vrai pour q prédicat quelconque soit Vrai, soit Faux.
- p⇒Vrai ≡ Vrai pour p prédicat quelconque soit Vrai, soit Faux.
- p∧q⇒q ≡ Vrai pour p et q prédicats quelconques soit Vrai, soit Faux

I(p)	*I(q)*	*I(p ∧ q)*	*I(p ⇒ q)*	*I(p∧q⇒ q)*
Vrai	Vrai	Vrai	Vrai	Vrai
Vrai	Faux	Faux	Faux	Vrai
Faux	Vrai	Faux	Vrai	Vrai
Faux	Faux	Faux	Vrai	Vrai

Figure 17 : table de vérité de l'opérateur implique

La maîtrise de l'opérateur *implique* est très importante pour la suite et, en général, pose quelques difficultés pour spécifier et pour calculer la valeur de vérité d'un prédicat. Il ne faut jamais oublier que l'interprétation intuitive de $p \Rightarrow q$ est «si p est vrai alors q est vrai». Autrement dit «pour que $p \Rightarrow q$ soit vrai, soit p est vrai et alors q doit être vrai, soit p est faux» qui est formalisée par l'équivalence $p \Rightarrow q \equiv \neg p \vee q$.

Nous ne définissons pas l'interprétation des termes utilisant les symboles comme +, -, *, =, ≠, ≤, <, ∈, ∉, … auxquels nous donnons respectivement l'interprétation usuelle : addition, soustraction, multiplication, égalité, différence, inférieur ou égal, inférieur, appartient, n'appartient pas, … L'interprétation des termes dépend uniquement de la valeur des variables qu'ils contiennent. Si les variables sont libres, leurs valeurs sont déterminées par un environnement. Si elles sont liées, leurs valeurs sont choisies judicieusement pour rendre vraie la formule où elles sont liées existentiellement et leur valeur est quelconque si elles sont quantifiées universellement. Dans ce cas, on fera une analyse par cas selon la forme de la formule.

Définition 11 ***(interprétation des prédicats)***

On note *I(t)* la valeur de vérité du terme *t* supposée définie et I(p), la valeur de vérité du prédicat p. La sémantique des prédicats est définie ainsi :

- $I(\neg p)$=Vrai si I(p)=Faux,
- $I(p_1 \wedge p_2)$=Vrai si $I(p_1)$=Vrai et $I(p_2)$=Vrai,
- $I(p_1 \vee p_2)$= Vrai si $I(p_1)$=Vrai ou $I(p_2)$=Vrai,
- $I(p_1 \Rightarrow p_2)$=Vrai si $I(p_1)$=Faux ou $I(p_2)$=Vrai (utilise la propriété $p_1 \Rightarrow p_2 \equiv \neg p_1 \vee p_2$),
- $I(p_1 \Leftrightarrow p_2)$=Vrai si $I(p_1)$=$I(p_2)$
- $I(\forall x.p)$=Vrai si pour toute valeur de *x*, I(p)=Vrai, sinon $I(\forall x.p)$=Faux
- $I(\exists x.p)$=Vrai si pour au moins une valeur de x, I(p)=Vrai, sinon $I(\exists x.p)$=Faux. ♦

2.2. Validité d'un prédicat

Cette sémantique permet de déterminer la *validité* d'un prédicat. Nous parlons alors de *prédicat valide*. Nous définissons la validité d'un prédicat en utilisant le vocabulaire défini dans un cours de Mathématiques pour l'Informatique : formule close, formule ouverte, environnement et validité. Un prédicat est en général une *formule ouverte*, c'est à dire une formule qui contient des variables libres dont la valeur est déterminée dans un *environnement*. Par exemple, le prédicat suivant est une formule ouverte :

$\forall i.\forall j.((i \in [1..n] \wedge j \in [1..n] \wedge i<j) \Rightarrow t[i] \leq t[j])$.

Les variables libres *n* et *t* sont associées à des valeurs dans un environnement. Dans notre cas l'environnement est le programme au point de contrôle où apparaît le prédicat. Les valeurs des variables libres sont alors leurs valeurs à ce point de contrôle. Par exemple, un environnement peut associer la valeur 7 à *n* et la valeur suivante au tableau *t* :

0	4	4	6	8	12	15

Rappelons la définition d'un prédicat valide :

Définition 12 ***(prédicat valide)***

Un prédicat p est valide si et seulement si sa valeur de vérité est Vrai quel que soit l'environnement : *I(p) = Vrai.* ♦

Par exemple, le prédicat ci-dessus de la forme ∀i.∀j.(d⇒p) est vrai dans l'environnement précédent, par contre il ne l'est pas dans un environnement où *n* est associé à la valeur 7 et où le tableau *t* est associé à la valeur suivante :

10	4	4	8	6	15	12

Ce prédicat n'est donc pas valide. On dit qu'il est satisfiable, c'est à dire qu'il existe un environnement qui lui donne pour valeur de vérité Vrai.

Définition 13 ***(prédicat satisfiable)***

Un prédicat p est satisfiable si et seulement si il existe un environnement dans lequel sa valeur de vérité est Vrai : *I(p) = Vrai.* ♦

Le prédicat suivant est valide :

∀i.∀j.((i∈[1..1] ∧ j∈[1..1] ∧ i≤j) ⇒ t[i]≤t[j]) .

La seule variable libre est *t*. Examinons la valeur de vérité de ce prédicat pour une valeur de *t* quelconque dans deux cas, le cas i=j=1 et le cas i≠1 ou j≠1.

- Dans le cas i=j=1, par substitution, le prédicat est de la forme ((1∈[1..1] ∧ 1∈[1..1] ∧ 1≤1) ⇒ t[1]≤t[1]) qui est équivalent à Vrai⇒Vrai, équivalent à Vrai par définition de l'opérateur implique.
- Dans le cas i≠1 ou j≠1, par substitution, le prédicat est de la forme (Faux ⇒ t[i]≤t[j]) qui est équivalent à Vrai car le terme t[i]≤t[j] a deux interprétations Vrai et Faux, et Faux ⇒ Vrai = Vrai et Faux ⇒ Faux = Vrai par définition de l'opérateur implique.

Définition 14 ***(Propriétés)***

Nous rappelons ci-dessous quelques-unes des principales propriétés des prédicats que nous utilisons pour effectuer des vérifications de programmes. :

- d⇒p ≡ ¬d∨p ,
- p ≡ ¬¬p,
- ¬(p ∧ q) ≡ ¬p ∨ ¬q,
- ¬(p ∨ q) ≡ ¬p ∧ ¬q,
- d∧p⇒p ≡ Vrai. C'est un prédicat valide quelque soit les valeurs de vérité des prédicats d et p ; c'est également une *tautologie* (voir commentaires en fin de chapitre sur formules satisfiables, valide et tautologies) car sa valeur de vérité ne dépend pas de l'interprétation des opérateurs apparaissant dans les termes de d et p.
- p⇒Vrai ≡ Vrai et Faux⇒p ≡ Vrai. Ce sont des prédicats valides quelle que soit la valeur de vérité de p. ♦

Dans la leçon suivante, pour faire des preuves de programmes, nous verrons que nous avons, certaines fois, à prouver la validité d'expressions logiques de la forme p ⇒ q. Pour cela, nous essaierons, par réécriture équivalente, de mettre ces formules sous l'une des 3 formes de formules valides énoncées ci-dessus.

3. Exemples de prédicats

Pour faire une exécution symbolique du programme de dichotomie (voir Figure 14), il faut l'annoter avec les prédicats explicités dans la Figure 18.

Le prédicat qui est en 4ème ligne de la Figure 18 contient la variable liée *k* (quantifiée universellement) et les cinq variables libres du programme *min* et *max* (variables de calcul), *x*, *n* et *t* (données). Sa valeur de vérité s'établit à partir d'un état mémoire où les valeurs des cinq variables libres du programme sont définies ainsi : *min=0, max=n, x* est un entier quelconque donné, *n* est un entier vérifiant 0≤n et *t* est un tableau quelconque

mais trié par ordre croissant d'après la pré-condition en 2ème ligne. Dans ce contexte, ce prédicat est vrai car pour tous *k*, *k∈[1..0*] est faux et *k∈[n+1..n]* et faux également avec *n* positif ou nul. Il est donc de la forme Faux⇒p pour tous *k*.

Début
{n≥0 ∧ ∀i.∀j.((i∈[1..n] ∧ j∈[1..n] ∧ i<j) ⇒ t[i]≤t[j])}
min := 0 ; max := n ;
{∀k.(k∈[1..min] ⇒ t[k]≤x) ∧ ∀k.(k∈[max+1..n] ⇒ t[k]>x)}
tantque min ≠ max **faire**
{∀k.(k∈[1..min]⇒t[k]≤x) ∧ ∀k.(k∈[max+1..n]⇒t[k]>x) ∧ min≠max }
m := (min+max+1) **div** 2 ; **si** t[m] ≤ x alors min := m **sinon** max := m-1 **finsi**
{∀k.(k∈[1..min] ⇒ t[k]≤x) ∧ ∀k.(k∈[max+1..n] ⇒ t[k]>x)}
fait ;
{∀k.(k∈[1..min] ⇒ t[k]≤x) ∧ ∀k.(k∈[max+1..n] ⇒ t[k]>x) ∧ min=max }
posx := min
{∀k.(k∈[1..posx]⇒t[k]≤x) ∧ ∀k.(k∈[posx+1..n]⇒t[k]>x)}
fin.

Figure 18 : programme de dichotomie annoté

Exercice 5 ***(Variables libres et liées)***

Quels sont les variables libres et liées du prédicat de la ligne 12 de la Figure 18 ? Quel est son contexte ? ♦

4. Réécriture de prédicats quantifiés

Les prédicats quantifiés peuvent poser des difficultés pour effectuer la substitution. En effet, il est possible que l'élément substitué soit concerné par une propriété quantifiée sans que celui-ci apparaisse explicitement. Par exemple, l'action T[g] := X concerne le prédicat $P \stackrel{def}{=} \forall k.(k\in[1..g] \Rightarrow T[k]\leq X)$. Pour calculer la substitution [T[g] := X]P il faut transformer le prédicat P pour mettre en évidence l'élément substitué T[g]. On utilise l'équivalence suivante : P ≡ T[g]≤X ∧ ∀k.(k∈[1..g-1] ⇒ T[k]≤X).

On a sorti le terme T[g] de la quantification en retirant l'indice *g* de l'intervalle [1..g] et on ajoute la propriété T[k]≤X pour k=g en conjonction. Il est alors possible de substituer T[g] par X et on obtient le prédicat suivant :

$[T[g] := X]P \stackrel{def}{=} X\leq X \wedge \forall k.(k\in[1..g-1] \Rightarrow T[k]\leq X)$

$\equiv \forall k.(k\in[1..g-1] \Rightarrow T[k]\leq X)$ car $X\leq X \equiv$ vrai et vrai ∧ p ≡ p.

Quelques-fois, il est nécessaire de faire l'opération inverse qui consiste à faire entrer une propriété pour un indice particulier dans une formule quantifiée. Par exemple si on a la formule $P' \stackrel{def}{=} \forall k.(k\in[1..g-1] \Rightarrow T[k]\leq X) \wedge T[g]\leq X$, celle-ci peut se réécrire ainsi en ajoutant l'indice *g* en fin de l'intervalle [1..g-1] :

$P' \stackrel{def}{=} \forall k.(k\in[1..g] \Rightarrow T[k]\leq X)$. Cette réécriture est fondée sur l'équivalence suivante : $\forall i.(i\in[1..n] \Rightarrow P(i))$

$\equiv P(1) \wedge P(2) \wedge \ldots \wedge P(n)$

$\equiv P(1) \wedge.\forall i.(i\in[2..n] \Rightarrow P(i))$.

5. Stratégies de preuve de validité de prédicats

Pour effectuer des preuves en logique de HOARE, les prédicats dont nous aurons à justifier la validité sont toujours de la forme p⇒q. Nous verrons que ces prédicats apparaissent pour appliquer les régles *Pré* ou *Post* de cette logique. Il y a cinq stratégies courantes pour justifier de la validité de ces formules. Pour les quatre premières, il faut justifier les équivalences suivantes :

- p⇒q ≡ faux⇒q, donc justifier p ≡ faux,

- $p \Rightarrow q \equiv p \Rightarrow vrai$, donc justifier $q \equiv vrai$,
- $p \Rightarrow q \equiv p \Rightarrow p$, donc justifier $q \equiv p$,
- $p \Rightarrow q \equiv p' \wedge q \Rightarrow q$, donc justifier $p \equiv p' \wedge q$.

En effet, les formules $faux \Rightarrow q$ et $p \Rightarrow vrai$ sont toujours vraies par définition de l'opérateur implique (voir Figure 17). C'est également le cas des formules de la forme $p \Rightarrow p$ et $p' \wedge q \Rightarrow q$ (voir Figure 17).

La cinquième consiste à faire une analyse par cas en justifiant que q est vrai si p l'est. Autrement dit on montre que le seul cas où $p \Rightarrow q$ est faux, p vrai et q faux est impossible.

Exemple 15 ***(justification de validité de prédicats de la forme $p \Rightarrow q \equiv faux \Rightarrow q$)***

Par exemple pour l'exemple de la recherche dichotomique (voir Figure 14), pour justifier que

$(n \geq 0 \wedge n \leq MAX \wedge \forall i.(i \in [1..n-1] \Rightarrow t[i] \leq t[i+1])) \Rightarrow$

$(\forall i.(i \in [1..0] \Rightarrow t[i] \leq x) \wedge \forall i.(i \in [n+1..n] \Rightarrow x < t[i]))$ est valide,

il sufffit justifier que les deux prédicats suivants sont équivalents à vrai, c'est-à-dire valide : $\forall i.(i \in [1..0] \Rightarrow t[i] \leq x)$ et $\forall i.(i \in [n+1..n] \Rightarrow x < t[i])$. Ces prédicats sont des conjonctions pour tout *i* des prédicats suivants : $i \in [1..0] \Rightarrow t[i] \leq x$ et $i \in [n+1..n] \Rightarrow x < t[i]$. Pour que la conjonction soit vraie, il faut et il suffit que chacun de ses membres soit vrai. Pour cela, on justifie que ces 2 prédicats se mettent respectivement sous la forme $faux \Rightarrow t[i] \leq x$ et $faux \Rightarrow x < t[i]$. C'est le cas car pour tout *i*, le prédicat $i \in [1..0]$ est faux car l'intervalle $[1..0]$ est vide. On applique la même justification pour le second cas où le prédicat $i \in [n+1..n]$ est faux. ♦

Exemple 16 ***(justification de validité de prédicats de la forme $p \Rightarrow q \equiv p \Rightarrow vrai$)***

Par exemple pour l'exemple de factorielle (voir Figure 12), pour justifier que $n \geq 0 \Rightarrow 1 = \prod_{j=1}^{0} j$, il suffit de constater que $1 = \prod_{j=1}^{0} j \equiv vrai$, car l'élément neutre du produit est un, pour obtenir un prédicat de la forme $p \Rightarrow vrai$.♦

Exemple 17 ***(justification de validité de prédicats de la forme $p \Rightarrow q \equiv p \Rightarrow p$)***

Par exemple pour l'exemple de racine carrée par incrémentation (voir Figure 10), pour justifier que $n \geq 0 \Rightarrow 0^2 \leq n$, il suffit de constater que $0^2 \leq n \equiv 0 \leq n$ pour obtenir un prédicat de la forme $p \Rightarrow p$.

Par exemple pour justifier que $n \geq 0 \Rightarrow 0^2 \leq n \wedge n < (n+1)^2$ est un prédicat valide, ♦

Exemple 18 ***(justification de validité de prédicats de la forme $p \Rightarrow q \equiv p' \wedge q \Rightarrow q$)***

Par exemple pour l'exemple de factorielle (voir Figure 12), pour justifier que $i \neq n \wedge f = \prod_{j=1}^{i} j \Rightarrow f*(i+1) = \prod_{j=1}^{i+1} j$, il suffit d'appliquer la stratégie en constatant que $f*(i+1) = \prod_{j=1}^{i+1} j \equiv f = \prod_{j=1}^{i} j$ en divisant chaque membre de l'égalité par *i+1*.♦

Exemple 19 ***(justification de validité de prédicats de la forme $p \Rightarrow q$ par analyse par cas)***

Par exemple pour justifier que $n \geq 0 \Rightarrow 0^2 \leq n \wedge n < (n+1)^2$ est un prédicat valide, on examine le cas où $n \geq 0$ et vrai. Dans ce cas le prédicat $0^2 \leq n \wedge n < (n+1)^2$ est vrai car pour tout *n* est positif ou nul, $(n+1)^2$ est supérieur à *n*. ♦

6. Résumé

La logique des prédicats du premier ordre est le langage de spécification des programmes et de description des valeurs symboliques.

Ayant parfaitement défini les langages de description des programmes et celui de description des spécifications, nous pouvons maintenant définir le système de vérification qu'un programme satisfait une spécification. Il est appelé logique de HOARE.

7. Exercices

Exercice 6 ***(Et bit à bit et pgcd - spécification)***

Décrire la spécification des problèmes posés dans l'Exercice 2. ♦

8. Formules Satisfiables, formules Valides versus Tautologies – commentaires

On peut distinguer 2 cas, logique mathématique et logique informatique.

C'est le deuxième cas qui nous intéresse. En logique pour l'informatique, le raisonnement logique vient après le typage. Le typage définit des domaines pour les variables, donc une interprétation. Ensuite, on ne raisonne que pour cette interprétation. En mathématiques, on raisonne pour n'importe quelle interprétation.

Plaçons-nous en informatique. Les formules portent sur les variables du programme qui sont libres dans les formules logiques. Leurs valuations sont fixées dans un environnement. Une formule **valide** est alors une formule vraie quelque-soit la valuation des variables du programme dans l'environnement. Par contre une formule **satisfiable** est une formule vraie pour au moins une valeur des variables du programme.

En pratique les formules dont on doit montrer qu'elles sont valides sont de la forme $A \Rightarrow B$. Ce sont les formules nécessaires pour appliquer les règles *Pré* ou *Post*. Souvent (pas toujours) elles sont telles que, par équivalence, on peut les abstraire par des formules propositionnelles :

- soit de la forme $P \Rightarrow Vrai$ ou $Faux \Rightarrow P$,
- soit de la forme $P \Rightarrow P$, soit de la forme $P \wedge P' \Rightarrow P$.

Ces formules sont valides en calcul propositionnel. On dit également que ce sont des tautologies (vraies quelque soit les interprétations booléennes de P et P'). Donc en pratique, souvent, quand on a à prouver que $A \Rightarrow B$ est valide, il faut essayer, par des équivalences sur A et B, de mettre la formule sous l'une des 4 formes ci-dessus.

Voici un exemple qui rentre dans le cas $P \wedge P' \Rightarrow P$:

$(y=(r+1)^2 \wedge z=2r+1 \wedge r^2 \leq n \wedge (r+1)^2 \leq n) \Rightarrow (y=(r+1)^2 \wedge z=2r+1 \wedge r^2 \leq n)$.

Cette formule peut s'abstraire sous la forme $P \wedge P' \Rightarrow P$ où $P \stackrel{def}{=} y=(r+1)^2 \wedge z=2r+1 \wedge r^2 \leq n$ et $P' \stackrel{def}{=} (r+1)^2 \leq n$.

Un exemple sur lequel cette méthode ne s'applique pas est de prouver que $n \geq 0 \Rightarrow (0^2 \leq n \wedge n<(n+1)^2)$. *n* est une variable de l'environnement. *n* est typé dans les entiers naturels, qui est donc le domaine d'interprétation. Pour cette interprétation, pour toutes les valeurs de *n* (indépendamment de l'environnement) la formule est vraie. Elle est donc valide.

7. Notations Equivalences – commentaires

Le symbole "$\Leftrightarrow$" est un des 16 opérateurs booléens possibles comme "$\wedge$" (et), "$\vee$" (ou), "$\Rightarrow$" (implique), etc. "$a \Leftrightarrow b$" se lit "a équivalent à b" ou "a si et seulement si b". Donc "$a \Leftrightarrow b$" est vrai si les valeurs booléennes de a et b sont identiques et faux sinon.

Le symbole "≡" est l'équivalence sémantique utilisée quand on veut remplacer une expression par une autre qui a les mêmes valeurs booléennes. Par exemple l'expression booléenne "$a \Rightarrow b$" est équivalente à l'expression "$\neg a \vee b$". Ceci se notera "$a \Rightarrow b$" ≡ "$\neg a \vee b$".

Pour embrouiller un peu, on peut dire que si P est une expression booléenne qui peut être remplacée par l'expression équivalente Q ($P \equiv Q$), alors l'expression booléenne "$P \Leftrightarrow Q$" est une tautologie (toujours vraie).

Alors que pour 2 expressions quelconques P et Q, l'expression booléenne "$P \Leftrightarrow Q$" est vraie ou fausse selon les valeurs de vérité de P et de Q.

Par exemple : $(a \Rightarrow b) \Leftrightarrow (\neg a \vee b)$ est toujours vraie quelque soit les valeurs booléennes de a et b. Par contre $a \Leftrightarrow b$ peut être vraie ou fausse. Elle est satisfaite quand $a = b$ et non satisfaite quand $a \neq b$.

Leçon 5 : La logique de HOARE

Pré requis

. savoir lire et utiliser la définition d'un système formel, notion introduite dans le cours MF,
. savoir faire des preuves formelles en utilisant un système formel,
. savoir lire une grammaire définissant un langage algébrique.

Objectifs

. S'approprier la logique de HOARE pour savoir comprendre des vérifications (appelées preuves) de programmes.

La leçon 5 est décomposée en six sections. La section 1 est un rappel de la notion de système formel vue en MF. La section 2 est un rappel de la notion de preuve. La section 3 discute deux manières de présenter les preuves. La section 4 décrit le système formel appelé logique de Hoare. La section 5 donne quelques éléments sur ses propriétés. Enfin la section 6 donne l'interprétation des axiomes et des règles de la logique de HOARE.

1. Rappel de la notion de système formel

Définition 15 ***(système formel)***

Un système formel est un triplet <L, Ax, R> où :
- L est un langage définissant un ensemble de formules,
- Ax est un sous-ensemble de L ; chaque formule de Ax est appelée axiome,
- R est un ensemble de règles de déduction de formules à partir d'autres formules ; une règle qui permet de déduire la formule f à partir des formules $f_1, f_2, \ldots, f_n$ est notée ainsi : $\frac{f_1, f_2, \ldots, f_n}{f}$. ♦

Dans une règle, les formules $f_1, f_2, \ldots, f_n$ sont appelées *prémisses* et la formule f est appelée *conclusion*.

Exemple 20 ***(système formel)***

L'exemple suivant est issu du livre [Hofstadter 98] p 83-84.

Soit l'alphabet de quatre symboles suivant : {~, NDP, SD, P}. Soit le langage L défini par la grammaire algébrique suivante :

<L> ::= P <Nbre> | <Nbre> NDP <Nbre> | <Nbre> SD <Nbre>

<Nbre> ::= ~<Nbre> | ~

L'ensemble <Nbre> contient les formules suivantes : ~, ~~, ~~~, ~~~~, ...

L contient les formules suivantes :
- P~, P~~, P~~~, …
- ~NDP~, ~~NDP~, ~NDP~~, ~~NDP~~, …
- ~SD~, ~~SD~, ~SD~~, ~~SD~~, …
- ...

L'ensemble d'axiomes $Ax \stackrel{def}{=} \{P\sim\sim\} \cup \{x\ y\ NDP\ x \mid x \text{ et } y \text{ sont des éléments de <Nbre>}\}$.

x et y étant des éléments de <Nbre>, l'ensemble R est formé des quatre règles suivantes numérotées R_1, R_2, R_3 et R_4 :

$$[R_1]\ \frac{x\ NDP\ y}{x\ NDP\ xy} \qquad [R_2]\ \frac{\sim\sim\ NDP\ x}{x\ SD\ \sim\sim} \qquad [R_3]\ \frac{x\sim\ NDP\ y,\ y\ SD\ x}{y\ SD\ x\sim} \qquad [R_4]\ \frac{y\sim\ SD\ y}{P\ y\sim}.$$

Un système formel est conçu pour raisonner syntaxiquement, c'est-à-dire indépendamment de la signification de ses formules. Cependant, nous donnons maintenant l'interprétation de ce système formel pour le rendre intelligible. Les symboles et les formules de ce système formel s'interprètent ainsi :

- une séquence de *n* tirets (~) représente l'entier naturel *n*, par exemple ~~~ représente l'entier 3,
- si *P suivi de n* ~ est un théorème, alors *n est un nombre premier*, par exemple P~~~ indique que 3 est un nombre premier, cette formule est un théorème car elle est vraie,
- si *x NDP y* est un théorème, alors l'entier *x ne divise pas l'entier y*, par exemple ~~~ NDP ~~~~~ signifie que 3 ne divise pas 5, ce qui est vrai,
- si *x SD y* est un théorème, alors l'entier *x est sans diviseur jusqu'à l'entier y*. ♦

2. Rappel de la notion de preuve et de théorème

Définition 16 ***(preuve)***

Une preuve de la formule f_n d'un système formel <L, Ax, R> est une séquence de formules du langage L f_1, f_2, …, f_n telle que pour tout *i* appartenant à 1..*n* :

- soit f_i est un axiome ($\in$ Ax),
- soit f_i est déduit à partir de certaines des formules de f_1, f_2, …, f_{i-1} en utilisant une règle de R. ♦

Idée Clé 9 ***(preuve)***

La notion de preuve est formelle au sens où le mécanisme est totalement syntaxique. Une formule *f* est déduite, soit à partir des axiomes qui sont considérés comme les vérités de base, soit en appliquant des règles uniquement à partir de formules précédentes déjà prouvées. ♦

Définition 17 ***(théorème)***

Une formule *f* qui apparaît dans une preuve est un théorème. ♦

Exemple 21 ***(Preuve de théorème)***

Par exemple la séquence de formules dans la Figure 19 est une preuve de la formule P~~~ dans le système formel présenté précédemment.

[f_1]	~~ NDP ~	Axiome (avec $x \stackrel{def}{=} ~$ et $y \stackrel{def}{=} ~$)
[f_2]	~~ NDP ~~~	$R_1(f_1)$
[f_3]	~~~ SD ~~	$R_2(f_2)$
[f_4]	P ~~~	$R_4(f_3)$

Figure 19 : preuve de la formule P~~~ sous forme de séquence de théorèmes

Les formules ~~ NDP ~, ~~ NDP ~~~, ~~~ SD ~~ et P ~~~ sont des théorèmes. ♦

Notation 5 ***(preuve)***

Chaque ligne de la Figure 19 est constituée de trois parties :

- à gauche entre crochets une référence à la formule qui la suit,
- au centre la formule elle-même,
- à droite la raison pour laquelle cette formule est un théorème. La notation $R_i(f_j, \ldots, f_k)$ signifie que la formule est établie en utilisant la règle R_i à partir des prémisses $f_j, \ldots, f_k$.

3. Comment présenter les preuves ?

Ci-dessus, nous présentons les preuves sous la forme de séquences de formules munies d'une référence et d'une justification. Dans le cours de MF, vous avez peut être appris à présenter les preuves sous la forme d'un arbre dont la racine est en bas comme dans la Figure 20.

La présentation arborescente est beaucoup plus adaptée à la construction de la preuve. Elle permet de partir du but (démontrer la formule P~~~ sur l'exemple) et de remonter progressivement aux axiomes en appliquant à chaque étape une règle qui a pour conclusion (la formule en dessous de la barre de fraction) le but à atteindre. Par exemple, seule la Règle R_4 est applicable pour déduire P~~~ avec y $\stackrel{def}{=}$ ~~. La prémisse à démontrer est la formule ~~~ SD ~~. Sur cette formule, on peut appliquer la règle R_2 avec x$\stackrel{def}{=}$~~~. Noter qu'on aurait pu également appliquer la règle R_3. Mais comme l'application de R_2 permet de continuer la preuve avec succès, nous n'avons pas exploré cette possibilité.

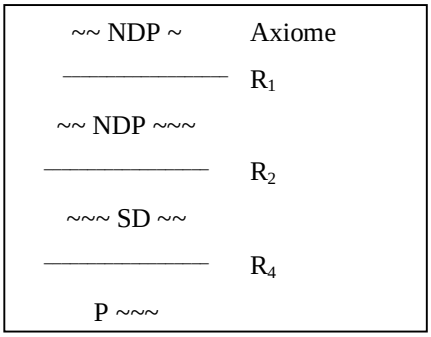


Figure 20 : arbre de preuve de la formule P~~~

Mais les arbres de preuve ne sont pas toujours facilement représentables sur une feuille de papier dans le cas où la longueur des formules et le nombre des prémisses fait exploser l'arbre en largeur. C'est pourquoi nous utilisons également la présentation linéaire tout en raisonnant à partir du but comme nous le faisons en construisant l'arbre de preuve. Noter qu'en MI, vous avez peut-être appris à présenter les preuves sous la forme de tableaux à 4 colonnes. Ce tableau est une représentation d'une séquence de formules comme dans la Figure 19 avec deux différences :

- l'ordre est inverse, dans le tableau, la première formule est le but et dans la séquence c'est la dernière,
- il n'y a pas l'équivalent de la colonne hypothèse car les preuves sont sans hypothèse.

Sur l'exemple, la présentation en tableau est donnée dans la Figure 21. On part du *But*. Sur celui-ci, on ne peut appliquer que la règle R_4 à partir de la prémisse [1] que l'on pose en dessous. Pour prouver celle-ci, on applique la règle R_2 à partir de la prémisse [2] que l'on pose en dessous. On répète ce processus jusqu'à ce que toutes les prémisses soient prouvées à partir d'axiomes.

[*But*]	P ~~~	R_4(1)
[1]	~~~ SD ~~	R_2(2)
[2]	~~ NDP ~~~	R_1(3)
[3]	~~ NDP ~	Axiome (avec x$\stackrel{def}{=}$~ et y$\stackrel{def}{=}$~)

Figure 21 : tableau de preuve de la formule P~~~

4. Logique de HOARE

Celle-ci est définie dans la Définition 18.

Les instructions d'affectation font l'objet d'axiomes. La notation `[x:=e]p` est celle de la *substitution*. `[x:=e]p`[1] désigne le prédicat *p* dans lequel toutes les occurrences libres de la variable *x* ont été substituées par *e*. Par exemple, le prédicat suivant `[x :=x+1](x<y` $\wedge$ $\forall$`x.(x`$\in$`[1..n]` $\Rightarrow$ `t[x]=3)` est égal à `(x+1<y` $\wedge$ $\forall$`x.(x`$\in$`[1..n]` $\Rightarrow$ `t[x]=3))` qui est obtenu en substituant la seule occurrence libre de *x* qui est dans `x<y` par *x+1*. Les autres occurrences sont liées par le quantificateur universel et ne sont pas substituées car le prédicat aurait pu être noté ainsi, par renommage de la variable liée, de telle sorte qu'il n'y ait plus d'ambiguïté : `[x :=x+1](x<y` $\wedge$ $\forall$`z.(z`$\in$`[1..n]` $\Rightarrow$ `t[z]=3))`.

Il a autant de règles que de structures de composition des instructions dans le langage de programmation :

[1] La notation adoptée est celle utilisée dans la méthode B. Dans certains ouvrages, la substitution se note en inversant les deux termes ainsi : `p[x := e]` ou ainsi `p[x\e]`.

- La règle [Seq] définit la sémantique de la composition séquentielle de deux instructions A_1 et A_2,
- Les règles [Si] et [Sinon] définissent la sémantique de la conditionnelle dans ses deux formes possibles, avec ou sans *sinon*,
- La règle [Tantque] définit la sémantique de l'itération.

Définition 18 ***(logique de Hoare)***

La logique de HOARE est un triplet <L, Ax, R> avec :

- L est l'ensemble des formules {p} A {q} où p et q sont des prédicats rédigés dans la syntaxe de la section 1 de la leçon 4 et A est un fragment de programme décrit dans la syntaxe de la leçon 3,
- Ax est l'ensemble des axiomes de la forme suivante pour toute affectation :
  . [Aff] {[x := e]p} x := e {p}
- R est l'ensemble de règles de déduction suivant :
  . [Seq] $\dfrac{\{p\}\ A_1\ \{q'\},\ \{q'\}\ A_2\ \{q\}}{\{p\}\ A_1\ A_2\ \{q\}}$ /* ajouter un « ; » entre A_1 et A_2 au dénominateur */
  . [Si] $\dfrac{\{p \wedge e\}\ A\ \{q\},\ p \wedge \neg e \Rightarrow q}{\{p\}\ \textbf{Si}\ e\ \textbf{alors}\ A\ \textbf{finsi}\ \{q\}}$ [Sinon] $\dfrac{\{p \wedge e\}\ A_1\ \{q\},\ \{p \wedge \neg e\}\ A_2\ \{q\}}{\{p\}\ \textbf{Si}\ e\ \textbf{alors}\ A_1\ \textbf{sinon}\ A_2\ \textbf{finsi}\ \{q\}}$
  . [Tantque] $\dfrac{\{I \wedge e\}\ A\ \{I\}}{\{I\}\ \textbf{Tantque}\ e\ \textbf{faire}\ A\ \textbf{fait}\ \{I \wedge \neg e\}}$
  . [Pré] $\dfrac{\{p'\}\ A\ \{q\},\ p \Rightarrow p'}{\{p\}\ A\ \{q\}}$ [Post] $\dfrac{\{p\}\ A\ \{q'\},\ q' \Rightarrow q}{\{p\}\ A\ \{q\}}$
  . [Et] $\dfrac{\{p\}\ A\ \{q\},\ \{p\}\ A\ \{q'\}}{\{p\}\ A\ \{q \wedge q'\}}$ [Ou] $\dfrac{\{p\}\ A\ \{q\},\ \{p'\}\ A\ \{q\}}{\{p \vee p'\}\ A\ \{q\}}$ ♦

Ces quatre règles permettent de démontrer des théorèmes formés de composition d'instructions à partir de théorèmes sur des instructions plus simples. Par exemple, à partir des deux axiomes suivants :

- $\{n \geq 0^2 \wedge n < (n+1)^2\}$rmin := 0 $\{n \geq rmin^2 \wedge n < (n+1)^2\}$
- $\{n \geq rmin^2 \wedge n < (n+1)^2\}$rmax := n+1 $\{n \geq rmin^2 \wedge n < rmax^2\}$

on peut démontrer le théorème suivant :

$\{n \geq 0^2 \wedge n < (n+1)^2\}$rmin := 0 ; rmax := n+1 $\{n \geq rmin^2 \wedge n < rmax^2\}$

en appliquant la règle [Seq]. Notons que le prédicat après l'instruction *rmin := 0* doit être le même que le prédicat avant l'instruction *rmax := n+1*. Cette contrainte est indiquée dans la règle [Seq] par le prédicat q' satisfait après A_1 et avant A_2.

Les règles [Pré] et [Post] permettent d'obtenir une formule avec le prédicat attendu, respectivement p et q alors qu'on a démontré un théorème avec des prédicats "voisins" p' et q'. La notion la plus forte de prédicats "voisins" est un prédicat équivalent ($p \Leftrightarrow p'$). Ici, la contrainte est plus faible, on a seulement besoin d'une implication : $p \Rightarrow p'$ pour la règle *Pré* et $q' \Rightarrow q$ pour la règle *Post*.

Enfin, les règles [Et] et [Ou] permettent de décomposer la preuve d'un théorème où l'un des prédicats est une conjonction ou une disjonction de prédicats en une preuve de deux théorèmes plus simples. Ces deux règles ne sont pas indispensables, elles sont rarement utilisées dans les exemples. Ce sont des règles de confort pour décomposer les preuves de formules trop grandes.

5. Propriétés de la logique de HOARE

Un système formel est une modélisation mathématique d'un problème réel de décision. Dans notre cas, la logique de HOARE permet de décider si un programme A satisfait une spécification formée de la pré-condition p et de la post-condition q en prouvant que la formule suivante {p} A {q} est un théorème.

Cette modélisation doit donc être conforme au problème réel qu'elle modélise. Dans le cas de la logique de HOARE, le problème réel est celui de l'exécution d'un programme A sur une machine. La conformité a été démontrée par HOARE. Elle recouvre deux propriétés :

- la *correction* qui signifie que si on démontre le théorème {p}A{q}, alors l'exécution du programme A à partir de n'importe quelles données qui satisfont le prédicat p aboutit, si il termine, à des résultats qui satisfont le prédicat q,
- la *complétude* qui signifie que si le programme A aboutit à des résultats qui satisfont le prédicat q à partir de n'importe quelles données qui satisfont le prédicat p, alors il existe une preuve du théorème {p}A{q}.

Ces deux propriétés garantissent que la vérification en utilisant la logique de HOARE correspond à l'interprétation du monde réel des machines bien qu'elle soit un processus mécanique qui fait abstraction complètement du sens des formules manipulées. Il est cependant intéressant de comprendre ces règles car leur appropriation donne un autre point de vue sur la construction de programmes. C'est pourquoi nous expliquons leurs interprétations dans la section 6.

6. Interprétation des règles de la logique de HOARE

Dans cette section, nous expliquons en quoi les axiomes et les règles de la logique de HOARE définissent une sémantique, en termes de transformation d'ensembles d'états mémoire, qui correspond à la réalité de l'exécution en machine. Ce type de sémantique est appelée « sémantique axiomatique ». Ce type de sémantique est adapté à la preuve de programmes. Notons qu'il y a d'autres sortes de sémantiques. Par exemple, des sémantiques qualifiées d'opérationnelles adaptées à la réalisation d'un interpréteur ou à des raisonnements algorithmiques sur les programmes.

En préliminaires de ces explications, nous définissons quelques «images» pour raisonner sur les prédicats de manière plus intuitive à travers des schémas.

Un état mémoire d'un programme A (voir leçon 2 section 1) est formé d'une valeur pour chaque variable de A. Par exemple, $n \stackrel{def}{=} 4$, $i \stackrel{def}{=} 3$, $f \stackrel{def}{=} 6$ est un état mémoire du programme de calcul de factorielle décrit section 4.1 de la leçon 3 (voir Figure 12). Autre exemple : $n \stackrel{def}{=} 16$, $r \stackrel{def}{=} 2$ est un état mémoire du programme de calcul de la racine carrée entière par défaut de la section 3 leçon 2 (voir Figure 10).

L'exécution d'un programme A transforme un état mémoire en un autre. Par exemple l'incrémentation *r := r+1* transforme l'état mémoire $n \stackrel{def}{=} 16$, $r \stackrel{def}{=} 2$ en $n \stackrel{def}{=} 16$, $r \stackrel{def}{=} 3$.

Rappelons qu'un prédicat p définit une valeur symbolique qui représente un ensemble de valeurs des variables utilisées dans p. Cet ensemble de valeurs correspond à l'ensemble des états mémoire de la machine qui exécute le programme agissant sur la valeur symbolique p. Par exemple, le prédicat $r^2 \leq n \wedge n \geq 0 \wedge r \geq 0$ représente l'ensemble d'états mémoire suivant :

- {($n \stackrel{def}{=} 0$, $r \stackrel{def}{=} 0$), ($n \stackrel{def}{=} 1$, $r \stackrel{def}{=} 0$), ($n \stackrel{def}{=} 1$, $r \stackrel{def}{=} 1$), ($n \stackrel{def}{=} 2$, $r \stackrel{def}{=} 0$), ($n \stackrel{def}{=} 2$, $r \stackrel{def}{=} 1$), ($n \stackrel{def}{=} 3$, $r \stackrel{def}{=} 0$), ($n \stackrel{def}{=} 3$, $r \stackrel{def}{=} 1$), ($n \stackrel{def}{=} 4$, $r \stackrel{def}{=} 0$), ($n \stackrel{def}{=} 4$, $r \stackrel{def}{=} 1$), ($n \stackrel{def}{=} 4$, $r \stackrel{def}{=} 2$), ($n \stackrel{def}{=} 5$, $r \stackrel{def}{=} 0$), ...}.

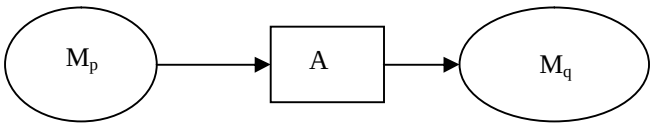


Figure 22 : exécution symbolique représentée par {p}A{q}.

Notation 6 ***(schéma des règles de preuves)***

- Nous notons M_p l'ensemble d'états mémoire qui satisfait le prédicat p.
- Le prédicat $p \Rightarrow q$ signifie que M_p est inclus dans M_q ($M_p \subseteq M_q$) par définition de l'implication : si un état mémoire satisfait p (donc appartient à M_p), alors il satisfait q (donc appartient à M_q). Par exemple, la partie gauche de la Figure 29 illustre que $M_p \subseteq M_{p'}$, c'est-à-dire que $p \Rightarrow p'$.
- Le théorème {p}A{q} représente l'exécution symbolique de A à partir de M_p donnant pour résultat M_q comme l'illustre la Figure 22.
- Les ellipses séparées en deux parties représentent un ensemble décomposé en deux sous-ensembles.
- Les textes dans les ellipses ou les parties d'ellipses indiquent ce que représente la partie où il apparaît.
- Les bulles sur les ensembles partitionnés indiquent ce que représente l'ellipse dans sa totalité.
- Les flèches pointillées indiquent les prémisses et les flèches pleines la conclusion. ♦

Règle de l'affectation

Les axiomes d'affectation peuvent s'expliquer par la formule suivante illustrée par la Figure 23 :

{P(e)} x:=e {P(x)}.

P(e) est un prédicat qui dépend de l'expression *e*. Par exemple le prédicat n+1≥0 dépend des expressions 0 et *n+1*. L'axiome indique que si P(e) est vrai et que si on affecte la valeur de *e* à la variable *x* alors le prédicat P(x) est vrai après l'affectation. Par exemple, {n+1≥0} n := n+1 {n≥0} est un axiome pour $e \stackrel{def}{=} n+1$ et $x \stackrel{def}{=} n$, {k+1≥0} n := 0 {k+1≥n} est également un axiome pour $e \stackrel{def}{=} 0$ et $x \stackrel{def}{=} n$. C'est exactement ce que formalise les axiomes de la logique de HOARE : pour que le prédicat p soit vrai après avoir exécuté *x:=e*, il faut que le prédicat p dans lequel on a substitué *x* par *e* soit vrai avant. On notera, qu'en général, cet axiome est défini d'arrière en avant : à partir du prédicat P(x), positionné après l'instruction d'affectation x := e, on calcule P(e) positionné avant l'instruction en substituant *x* par *e* dans P.

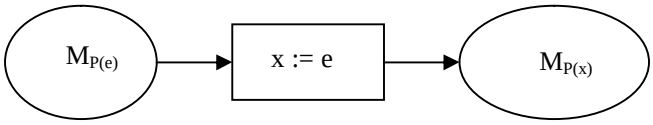


Figure 23 : Exécution symbolique {p(e)} x := e {p(x)}.

Exemple 22 ***(Application de la règle {p(e)} x := e {p(x)})***

{n+m*2≥0} n := n+m*2 {n≥0} où p(n) $\stackrel{def}{=}$ n≥0 et p(e) $\stackrel{def}{=}$ n+m*2≥0 obtenu en substituant *n* par *n+m*2* dans le prédicat avant l'instruction. ♦

On notera que des formules d'affectation qui ne sont pas des axiomes existent. Par exemple la formule {x=4} x := x+1 {x=4} n'est pas un axiome. Par contre {x+1=4} x := x+1 {x=4} est un axiome. Les formules qui ne sont pas des axiomes représentent des exécutions qui ne peuvent pas avoir lieu en réalité. De manière générale, toute formule qui n'est pas un théorème représente une exécution qui ne peut pas avoir lieu. Ces exécutions ne pouvant avoir lieu, elles sont modélisées par des formules qui ne sont pas des théorèmes. On dit qu'elles sont fausses. Par contre, la propriété de correction implique que tout théorème représente une exécution qui peut avoir lieu. La complétude implique que toute exécution qui peut avoir lieu est modélisable par un théorème prouvable.

Dans le cas particulier où l'affectation est de la forme x := f(x) et où la fonction *f* a une fonction inverse f^{-1}, l'axiome d'affectation peut être appliqué d'avant en arrière comme l'illustre la Figure 24 : connaissant P(x) et *f*, on calcule $P(f^{-1}(x))$.

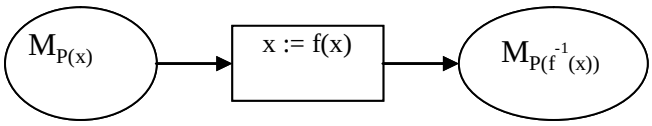


Figure 24 : Exécution symbolique {p(x)} x := f(x) $\{p(f^{-1}(x))\}$

Exemple 23 ***(Application de la règle {p(x)} x := f(x) $\{p(f^{-1}(x))\}$)***

{n≥0} n := n-2 {n+2≥0} où p(n) $\stackrel{def}{=}$ n≥0 et $p(f^{-1}(x)) \stackrel{def}{=}$ n+2≥0 obtenu en substituant *n* par *n+2* dans le prédicat après l'instruction. *n+2* réalise la fonction inverse de *n-2* ♦

Règle de la séquence

La règle de la séquence décrit la sémantique illustrée dans la Figure 25. Si l'instruction A_1 modifie les états mémoire M_p en $M_{q'}$ et si l'instruction A_2 modifie les états mémoire $M_{q'}$ en M_q, alors l'instruction A_1 ; A_2 modifie les états mémoire M_p en M_q.

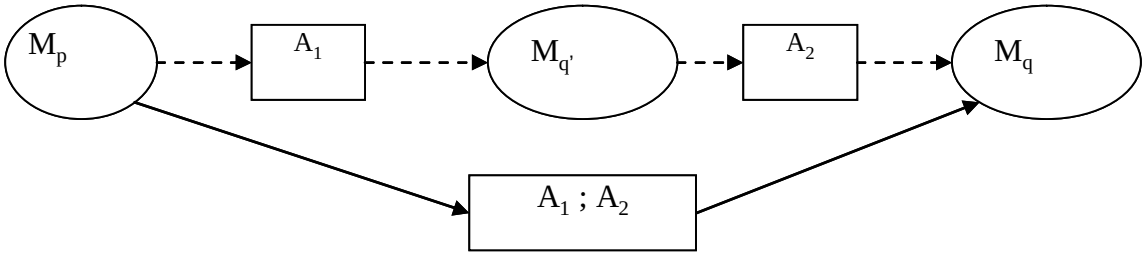


Figure 25 : exécution symbolique {p} A_1 ; A_2 {q}.

Règles de la conditionnelle

La règle de la conditionnelle décrit la sémantique illustrée dans la Figure 26. Si l'instruction A_1 modifie les états mémoire $M_{p\wedge e}$ en M_q et si l'instruction A_2 modifie les états mémoire $M_{p\wedge\neg e}$ en M_q, alors l'instruction ***Si e alors*** A_1 ***sinon*** A_2 ***finsi*** modifie les états mémoire M_p en M_q.

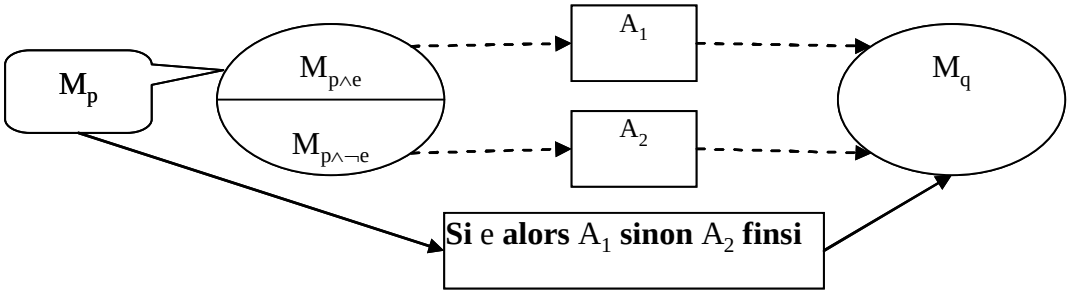


Figure 26 : exécution symbolique {p} *Si* ***e*** *alors* ***A_1*** *sinon* ***A_2*** *finsi* **{q}.**

La règle de la conditionnelle sans partie *sinon* décrit la sémantique illustrée dans la Figure 27. Si l'instruction A modifie les états mémoire $M_{p\wedge e}$ en M_q et si les états mémoire $M_{p\wedge\neg e}$ sont inclus dans M_q, alors l'instruction ***Si e alors*** *A* ***finsi*** modifie les états mémoire M_p en M_q.

Exemple 24 ***(Application de la règle {p}*** *Si* ***e*** *alors* ***A_1*** *sinon* ***A_2*** *finsi* ***{q})***

La formule {$Rmin^2 \leq N \wedge N < Rmax^2$} **Si** $M*M \leq N$ **alors** Rmin := M **sinon** Rmax := M **finsi** {$Rmin^2 \leq N \wedge N < Rmax^2$} est un théorème si les deux formules suivantes sont des théorèmes : {$Rmin^2 \leq N \wedge N < Rmax^2 \wedge M*M \leq N$} Rmin := M {$Rmin^2 \leq N \wedge N < Rmax^2$} et {$Rmin^2 \leq N \wedge N < Rmax^2 \wedge M*M > N$} Rmax := M {$Rmin^2 \leq N \wedge N < Rmax^2$}♦

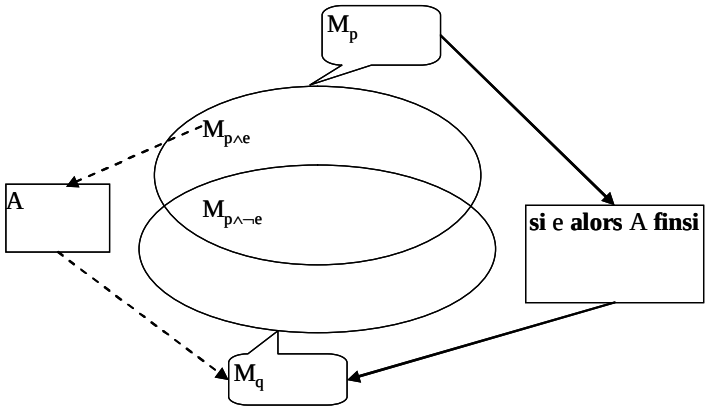


Figure 27 : exécution symbolique {p} *Si* ***e*** *alors* ***A*** *finsi* **{q}.**

Règle du tantque

La règle [Tantque] de l'itération décrit la sémantique illustrée dans la Figure 28. Si l'instruction A modifie les états mémoire $M_{p\wedge e}$ en M_p, alors l'instruction ***Tantque*** *e* ***faire*** *A* ***fait*** modifie les états mémoire M_p en $M_{p\wedge\neg e}$.

La règle [Tantque], illustrée par la Figure 28, est remarquable puisqu'elle met en évidence le fait que chaque pas d'itération maintient un invariant noté `I` dans la règle. En effet, d'après la règle, le prédicat `I` décore l'instruction *tantque* ainsi :

`{I}`
Tantque e **faire**
  `{I∧e}`
  A
  `{I}`
fait
`{I∧¬e}`

Une itération est une forme de calcul par approximation successive, c'est à dire de recherche, d'une solution dans un ensemble. Donc le prédicat `I` représente l'ensemble dans lequel est recherchée la solution. Cet espace de recherche déterminé au départ ne doit jamais changer. C'est pourquoi ce prédicat `I` est invariant. Et le prédicat `I∧¬e` représente l'ensemble de solutions. Par exemple, le calcul de la racine carrée par incrémentation (voir Figure 9) recherche une solution dans l'espace défini par le prédicat $r^2 \leq n$, c'est à dire l'espace 0.. ∞, car la plus petite valeur de *n* est 0. La condition $(r+1)^2 \geq n$ définit la solution donc la condition d'arrêt de l'itération.

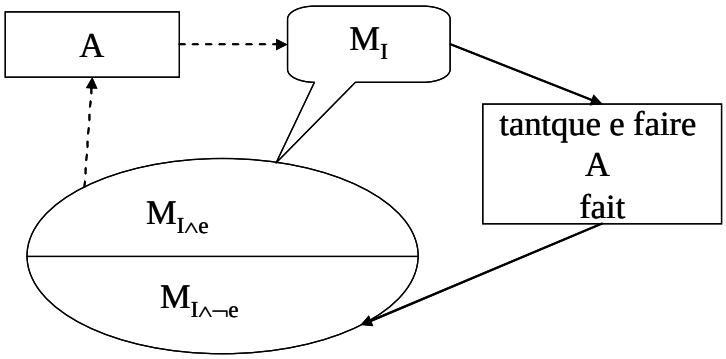


Figure 28 : exécution symbolique `{I}` *Tantque e faire A fait* `{I∧¬e}`.

Exemple 25 ***(Application de la règle {I} Tantque e faire A fait {I∧¬e})***

La formule $\{Rmin^2 \leq N \wedge N < Rmax^2\}$ **Tantque** $Rmin+1 \neq Rmax$ **faire** M := (Rmin+Rmax+1) div 2 ; **Si** $M*M \leq N$ **alors** Rmin := M **sinon** Rmax := M **finsi fait** $\{Rmin^2 \leq N \wedge N < Rmax^2 \wedge Rmin+1=Rmax\}$ est un théorème si la formule suivante est un théorème : $\{Rmin^2 \leq N \wedge N < Rmax^2 \wedge Rmin+1 \neq Rmax\}$ M := (Rmin+Rmax+1) div 2 ; **Si** $M*M \leq N$ **alors** Rmin := M **sinon** Rmax := M **finsi** $\{Rmin^2 \leq N \wedge N < Rmax^2\}$ ♦

Règles Pré et Post

La règle [Pré] est illustrée dans la Figure 29. Elle permet de dire que si l'instruction A modifie les états mémoire $M_{p'}$ en M_q ($\{p'\}A\{q\}$) et que M_p est inclus dans $M_{p'}$ ($p \Rightarrow p'$) alors l'instruction A modifie les états mémoire M_p en M_q ($\{p\}A\{q\}$).

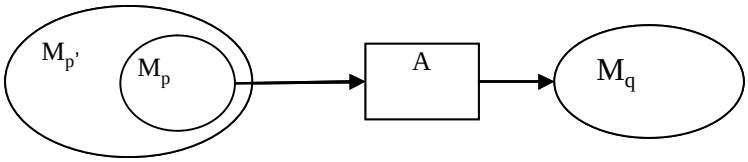


Figure 29 : règle Pré, « *qui part de plus peut partir de moins* »

La règle [Post] est illustrée dans la Figure 30. Elle permet de dire que si l'instruction A modifie les états mémoire M_p en $M_{q'}$ ($\{p\}A\{q'\}$) et que $M_{q'}$ est inclus dans M_q ($q' \Rightarrow q$) alors l'instruction ***A*** modifie les états mémoire M_p en M_q ($\{p\}A\{q\}$).

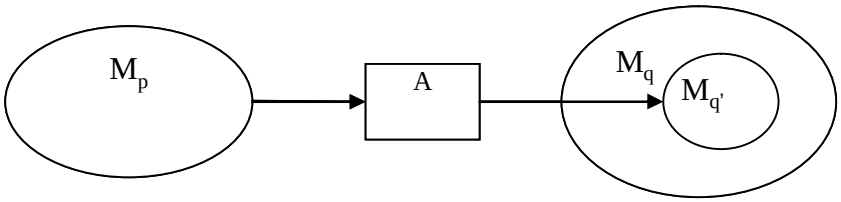


Figure 30 : règle Post « *qui arrive à plus peut arriver à moins* »

La règle *Pré* est utile lorsqu'on a établi le théorème ({p'}A{q}) et que le but cherché est le théorème ({p}A{q}). Il faut démontrer que la formule $p \Rightarrow p'$ est valide. La règle *Post* est utile lorsqu'on a établi le théorème ({p}A{q'}) et que le but cherché est le théorème ({p}A{q}). Il faut démontrer que la formule $q' \Rightarrow q$ est valide. On notera que souvent il est possible de démontrer l'équivalence $p \equiv p'$ ou $q' \equiv q$ ou encore que les formules suivantes sont valides : $p \Leftrightarrow p'$ ou $q' \Leftrightarrow q$.

Exemple 26 ***(Application de la règle Pré)***

La formule $\{Rmin^2 \leq N \wedge N < Rmax^2 \wedge Rmin+1 \neq Rmax\}$ M := (Rmin+Rmax+1) div 2 $\{Rmin^2 \leq N \wedge N < Rmax^2\}$ est un théorème si la formule suivante est un théorème : $\{Rmin^2 \leq N \wedge N < Rmax^2\}$ M := (Rmin+Rmax+1) div 2 $\{Rmin^2 \leq N \wedge N < Rmax^2\}$ et si celle-ci est une formule valide : $(Rmin^2 \leq N \wedge N < Rmax^2 \wedge Rmin+1 \neq Rmax) \Rightarrow (Rmin^2 \leq N \wedge N < Rmax^2)$. La première est un axiome par la règle d'affectation et la seconde est une formule valide car de la forme $p \wedge p' \Rightarrow p$ avec $p \stackrel{def}{=} (Rmin^2 \leq N \wedge N < Rmax^2)$. ♦

7. Résumé

Les axiomes et les règles de la logique de HOARE constituent un système formel.

Vérifier qu'un programme A satisfait une spécification formée de la pré-condition p, et de la post-condition q consiste à prouver que la formule {p}A{q} est un théorème de la logique de HOARE. Notons qu'il s'agit ici d'une preuve de correction partielle. Pour prouver la correction totale, il faut prouver la terminaison.

Nous avons indiqué «comment présenter les preuves ? ». Il reste à indiquer « comment trouver les preuves ? ». C'est ce que nous faisons dans la leçon suivante.

Leçon 6 : Quelques éléments de stratégie de preuve de programmes

Pré requis

. savoir utiliser les axiomes et les règles de la logique de HOARE,
. savoir lire des preuves effectuées à partir d'un système formel,

Objectifs

. S'approprier la logique de HOARE pour effectuer des vérifications (appelées preuves) de programmes,
. savoir construire des preuves formelles en logique de HOARE.

La leçon 6 est décomposée en huit sections. La section 1 est un rappel que vérifier c'est faire une preuve dans la logique de HOARE. La section 2 explique une stratégie de preuve d'un fragment de programme constitué d'une séquence de deux instructions. La section 3 explique une stratégie de preuve d'une conditionnelle. La section 4 décrit une stratégie de preuve d'une itération. La section 5 décrit une stratégie de preuve de programmes. La section 6 donne quelques éléments d'une méthode pour trouver les invariants d'itération. La section 7 explique la différence entre les notions de preuve de correction partielle et totale de programmes. Enfin la section 8 fait un bilan de la première partie de ce cours de «preuve de programmes».

1. Vérifier c'est prouver

Pour vérifier un programme A, il faut effectuer trois tâches :

1. décrire une spécification S du programme A formé d'une pré-condition `p` et d'une post-condition `q` décrite comme des prédicats,
2. décrire le programme A dans le langage des *tantque* programmes,
3. trouver une preuve de la formule `{p}A{q}`.

La stratégie applicable habituellement pour trouver une preuve est une stratégie de chaînage arrière qui consiste à construire l'arbre de preuve en partant du but. Au départ, on connaît la formule `{p}A{q}` à prouver. On détermine une règle qui a pour conclusion une formule de cette forme. On en déduit les prémisses à prouver. Ces prémisses décomposent le problème en plusieurs sous-problèmes plus simples. On applique ce processus jusqu'à ce que toutes les prémisses soient des axiomes.

Cependant, trouver une preuve avec cette méthode n'est pas forcément simple pour plusieurs raisons. La première est que plusieurs règles peuvent être applicables sur la même formule. Se pose alors la question du choix de la règle à appliquer. Avec la logique de Hoare, comme il n'y a qu'une seule règle par forme de structure de contrôle des programmes, il n'y a en général pas le choix sauf pour les règles *Pré, Post, Et* et *Ou* qui peuvent être applicables en même temps qu'une des règles *Seq, Si, Sinon* ou *Tantque*. Une stratégie consiste à appliquer les règles *Pré* et *Post* par nécessité. Quant aux règles *Et* et *Ou*, elles peuvent être appliquées si dans une formule `{p}A{q}` `p` est une disjonction ou `q` est une conjonction. Notons que, quand il y a plusieurs règles applicables, souvent un choix aléatoire convient car il existe plusieurs preuves qui consistent à appliquer les mêmes règles dans un ordre différent. Par exemple, si on a à prouver la formule `{p}` A_1 ; A_2 ; A_3 `{q}` qui est la séquence de trois fragments A_1, A_2 et A_3, il y a deux manières de décomposer le problème en deux sous problèmes en appliquant la règle de la séquence, soit en `{p}` A_1 `{p'}` et `{p'}` A_2 ; A_3 `{q}`, soit en `{p}` A_1 ; A_2`{p"}` et `{p"}` A_3 `{q}` La seconde raison, plus difficile, est que pour progresser, quelques fois, il faut inventer un prédicat (cas des invariants pour les itérations), puis vérifier que ce prédicat convient. Dans ce cas, il est nécessaire de bien comprendre le but à atteindre et le principe de base du programme.

C'est à ces problèmes que nous allons tenter de répondre ci-dessous. Nous utilisons l'exemple du programme de calcul de la racine carrée entière par défaut appliquant la méthode de la dichotomie donnée section 1 leçon 1 dans la Figure 4.

2. Stratégie de preuve d'une séquence de deux instructions

Exemple 27 ***(preuve de séquence)***

Une preuve de la formule suivante :

$\{n \geq 0\}$ rmin := 0 ; rmax := n+1 $\{rmin^2 \leq n \wedge n < rmax^2\}$

est la séquence des cinq formules décrites dans la Figure 31. ♦

$[f_1]$	$\{rmin^2 \leq n \wedge n < (n+1)^2\}$ rmax := n+1 $\{rmin^2 \leq n \wedge n < rmax^2\}$	Axiome
$[f_2]$	$\{0^2 \leq n \wedge n < (n+1)^2\}$ rmin := 0 $\{rmin^2 \leq n \wedge n < (n+1)^2\}$	Axiome
$[f_3]$	$n \geq 0 \Rightarrow (0^2 \leq n \wedge n < (n+1)^2)$	prédicat valide
$[f_4]$	$\{n \geq 0\}$ rmin := 0 $\{rmin^2 \leq n \wedge n < (n+1)^2\}$	Pré(f_2, f_3)
$[f_5]$	$\{n \geq 0\}$ rmin := 0 ; rmax := n+1 $\{rmin^2 \leq n \wedge n < rmax^2\}$	Seq(f_4, f_1)

Figure 31 : exemple de preuve de séquence

Le prédicat f_3 est valide. Justifions le. Si $n \geq 0$ est faux, $Faux \Rightarrow p \equiv Vrai$ quelque soit p. Si $n \geq 0$ est vrai, seul $Vrai \Rightarrow Vrai \equiv Vrai$. On est dans ce cas car le prédicat à droite de l'implication qui est $(0^2 \leq n \wedge n < (n+1)^2)$ est vrai quand $n \geq 0$ est vrai. En effet, quand $n \geq 0$ est vrai, $0^2 \leq n \equiv 0 \leq n$ est vrai et $n < (n+1)^2$ est vrai car pour tout *n* supérieur ou égal à zéro : $0 < (0+1)^2$, $1 < (1+1)^2$, etc.

L'arbre de preuve est représenté dans la Figure 32. Nous pouvons voir sur cet exemple que cette représentation des preuves pose des problèmes de mise en page, c'est pourquoi nous préférons la mise en page sous forme de séquence de formules.

$p \Rightarrow p'$ /* prédicat valide */, $\{p'\}$ A_1 $\{q'\}$ /*Axiome*/

_____ Pré

$\{p\}$ A_1 $\{q'\}$, $\{q'\}$ A_2 $\{q\}$ /* Axiome */

_____Seq

$\{p\}$ A_1 ; A_2 $\{q\}$

Figure 32 : arbre de preuve d'une séquence de 2 instructions

Dans le cas général où le but est la formule $\{p\}$ A_1 ; A_2 $\{q\}$ où A_1 et A_2 sont des affectations, cette preuve s'effectue ainsi :

1. établir la formule f_1, $\{q'\}$ A_2 $\{q\}$ pour qu'elle soit un axiome ; comme q est connu, celle-ci permet de trouver $q' \stackrel{def}{=} [A_2]q$. Dans l'exemple $q \stackrel{def}{=} rmin^2 \leq n \wedge n < rmax^2$ et $A_2 \stackrel{def}{=}$ rmax := n+1, donc $q' \stackrel{def}{=} [A_2]q \stackrel{def}{=} rmin^2 \leq n \wedge n < (n+1)^2$). Ce prédicat est obtenu en substituant les occurrences de *rmax* par l'expression *n+1*.
2. établir la formule f_2, $\{p'\}$ A_1 $\{q'\}$ pour qu'elle soit un axiome ; comme q' a été défini à l'étape 2, celle-ci permet de trouver $p' \stackrel{def}{=} [A_1]q'$. Dans l'exemple $q' \stackrel{def}{=} rmin^2 \leq n \wedge n < (n+1)^2$ et $A_1 \stackrel{def}{=}$ rmin := 0, donc $p' \stackrel{def}{=} [A_1]q' \stackrel{def}{=} 0^2 \leq n \wedge n < (n+1)^2$). Ce prédicat est obtenu en substituant les occurrences de *rmin* par l'expression *0*.
3. montrer que la formule f_3, $p \Rightarrow p'$ est un prédicat valide ; pour cela, faire une analyse par cas, pour justifier que, quand p est vrai, p' est également vrai (pour plus de détails, voir section 7),
4. établir la formule f_4, $\{p\}$ A_1 $\{q'\}$ en appliquant la règle *Pré* entre f_3 et f_2,
5. établir la formule f_5, en appliquant la règle *Seq* entre f_4 et f_1.

Il y a d'autres stratégies de preuve possibles qui conduisent au même résultat, notamment celle qui aboutit à la séquence de formules de la Figure 33. Dans la Figure 33, la stratégie est d'appliquer la règle *Pré* par nécessité. Lorsqu'on a établi la formule $\{p'\}$ A_1 $\{q\}$ et que l'on veut établir la formule $\{p\}$ A_1 $\{q\}$, on essaie de démontrer que la formule $p \Rightarrow p'$ est une formule valide pour pouvoir appliquer la règle *Pré*.

$[f_6]$	$\{rmin^2 \le n \wedge n < (n+1)^2\}$ rmax := n+1 $\{rmin^2 \le n \wedge n < rmax^2\}$	Axiome
$[f_7]$	$\{0^2 \le n \wedge n < (n+1)^2\}$ rmin := 0 $\{rmin^2 \le n \wedge n < (n+1)^2\}$	Axiome
$[f_8]$	$\{0^2 \le n \wedge n < (n+1)^2\}$ rmin:=0 ; rmax:=n+1 $\{rmin^2 \le n \wedge n < rmax^2\}$	Seq(f_7, f_6)
$[f_9]$	$n \ge 0 \Rightarrow (0^2 \le n \wedge n < (n+1)^2)$	prédicat valide
$[f_{10}]$	$\{n \ge 0\}$ rmin:=0 ; rmax:=n+1 $\{rmin^2 \le n \wedge n < rmax^2\}$	Pré(f_8, f_9)

Figure 33 : autre exemple de preuve de séquence

La seule différence entre la preuve de la Figure 32 et celle de la Figure 33 est l'ordre d'application des règles *Pré* et *Seq*.

3. Stratégie de preuve d'une conditionnelle

Exemple 28 ***(preuve de conditionnelle)***

Une preuve de la formule suivante :

$\{rmin^2 \le n \wedge n < rmax^2\}$

si m*m>n **alors** rmax := m **sinon** rmin := m **finsi**

$\{rmin^2 \le n \wedge n < rmax^2\}$

est la séquence de sept formules décrite dans la Figure 34. ♦

$[f_{11}]$	$\{rmin^2 \le n \wedge n < m^2\}$ rmax := m $\{rmin^2 \le n \wedge n < rmax^2\}$	Axiome
$[f_{12}]$	$\{m^2 \le n \wedge n < rmax^2\}$ rmin := m $\{rmin^2 \le n \wedge n < rmax^2\}$	Axiome
$[f_{13}]$	$(rmin^2 \le n \wedge n < rmax^2 \wedge n < m^2) \Rightarrow (rmin^2 \le n \wedge n < m^2)$	prédicat de la forme $p \wedge p' \Rightarrow p$ valide
$[f_{14}]$	$(rmin^2 \le n \wedge n < rmax^2 \wedge m^2 \le n) \Rightarrow (n < rmax^2 \wedge m^2 \le n)$	prédicat de la forme $p \wedge p' \Rightarrow p$ valide
$[f_{15}]$	$\{rmin^2 \le n \wedge n < rmax^2 \wedge n < m^2\}$ rmax := m $\{rmin^2 \le n \wedge n < rmax^2\}$	Pré(f_{11}, f_{13})
$[f_{16}]$	$\{rmin^2 \le n \wedge n < rmax^2 \wedge m^2 \le n\}$ rmin := m $\{rmin^2 \le n \wedge n < rmax^2\}$	Pré(f_{12}, f_{14})
$[f_{17}]$	$\{rmin^2 \le n \wedge \wedge n < rmax^2\}$ **si** m*m>n **alors** rmax := m **sinon** rmin := m **finsi** $\{rmin^2 \le n \wedge n < rmax^2\}$	Sinon(f_{15}, f_{16})

Figure 34 : exemple de preuve de conditionnelle

L'arbre de preuve est représenté dans la Figure 35.

$p \wedge e \Rightarrow p'$, $\{p'\}$ A_1 $\{q\}$ (*Axiome*) — Pré

$p \wedge \neg e \Rightarrow q'$, $\{q'\}$ A_2 $\{q\}$ (*Axiome*). — Pré

$\{p \wedge e\}$ A_1 $\{q\}$, $\{p \wedge \neg e\}$ A_2 $\{q\}$ — Sinon

$\{p\}$ **si** e **alors** A_1 **sinon** A_2 **finsi** $\{q\}$

Figure 35 : arbre de preuve d'une conditionnelle

Dans le cas général où le but est une formule de la forme suivante, $\{p\}$ **si** e **alors** A_1 **sinon** A_2 **finsi** $\{q\}$ où A_1 et A_2 sont des affectations, cette preuve s'effectue ainsi :

1. établir la formule f_{11}, $\{p'\}$ A_1 $\{q\}$ pour que ce soit un axiome ; celle-ci permet de trouver $p' \stackrel{def}{=} [A_1]q$.

2. établir la formule f_{12}, $\{q'\}\ A_2\ \{q\}$ pour que ce soit un axiome ; celle-ci permet de trouver $q' \stackrel{def}{=} [A_2]q$.
3. montrer que la formule f_{13} suivante : $p \wedge e \Rightarrow p'$, est un prédicat valide,
4. montrer que la formule f_{14}, $p \wedge \neg e \Rightarrow q'$, est un prédicat valide,
5. établir la formule f_{15} suivante : $\{p \wedge e\}\ A_1\ \{q\}$, en appliquant la règle *Pré* entre f_{11} et f_{13},
6. établir la formule f_{16}, $\{p \wedge \neg e\}\ A_2\ \{q\}$, en appliquant la règle *Pré* entre f_{12} et f_{14},
7. établir la formule f_{17} en appliquant la règle *Sinon* entre f_{15} et f_{16}.

4. Stratégie de preuve d'une itération

Exemple 29 ***(preuve d'itération)***

Pour prouver l'itération suivante :

Tantque $rmin+1 \neq rmax$ **faire**
m := (rmin+rmax) **div** 2 ;
si $m*m>n$ **alors** rmax := m **sinon** rmin := m **finsi**
fait,

on suppose avoir prouvé les formules suivantes qui ne contiennent pas d'itération en utilisant les stratégies vues précédemment :

$[f_{18}]$ $\{rmin^2 \leq n \wedge n < rmax^2 \wedge rmin+1 \neq rmax\}$
m := (rmin+rmax) **div** 2 ; $\{rmin^2 \leq n \wedge n < rmax^2\}$.

$[f_{19}]$ $\{rmin^2 \leq n \wedge n < rmax^2 \wedge rmin+1 \neq rmax\}$
m := (rmin+rmax) **div** 2 ; **si** m*m>n **alors** rmax := m **sinon** rmin := m **finsi**
$\{rmin^2 \leq n \wedge n < rmax^2\}$.

La formule f_{19} se prouve en appliquant la stratégie de la séquence à partir des formules f_{17} et f_{18}. f_{18} se prouve à partir d'un axiome d'affectation, d'un prédicat valide et de la règle *Pré* (voir Figure 31).

Pour prouver l'itération, il faut prouver la formule f_{20} suivante :

$[f_{20}]$ $\{rmin^2 \leq n \wedge n < rmax^2\}$
Tantque rmin+1≠rmax **faire**
m := (rmin+rmax) **div** 2 ;
si m*m>n **alors** rmax := m **sinon** rmin := m **finsi**
fait $\{rmin^2 \leq n \wedge n < rmax^2 \wedge rmin+1 = rmax\}$ Tantque(f_{19})

Figure 36 : exemple de preuve d'itération

La preuve consiste simplement à appliquer la règle [Tantque] sur la formule f_{19}. Noter que dans cette preuve, le prédicat invariant est $I \stackrel{def}{=} rmin^2 \leq n \wedge n < rmax^2$. Ici, nous n'avons pas expliqué comment trouver cet invariant. Nous l'expliquons dans la section 6. ♦

Exercice 7 ***(preuve d'affectation)***

Prouver la formule f_{18} en appliquant la règle de l'affectation et la règle [Pré]. ♦

Dans le cas général où le but est une formule $f \stackrel{def}{=} \{p\}$ **tantque** e **faire** A **fait** $\{q\}$ où A est un fragment de programme quelconque, cette preuve s'effectue ainsi :

- Trouver le prédicat invariant I tel que $q \equiv I \wedge \neg e$,
- Prouver la formule $f' \stackrel{def}{=} \{I \wedge e\}\ A\ \{I\}$,
- Prouver que la formule $fv \stackrel{def}{=} p \Rightarrow I$ est valide,

- Etablir la formule $f'' \stackrel{def}{=} \{I\}$ **tantque** e **faire** A **fait** $\{I \wedge \neg e\}$ en appliquant la règle Tantque,
- Etablir la formule *f* en appliquant la règle *Pré* à partir de *f''* et *fv*.

La principale difficulté de la preuve d'une itération est de trouver le prédicat invariant I à chaque pas d'itération. Sur l'exemple, celui-ci est le suivant : $rmin^2 \leq n \wedge n < rmax^2$. Il signifie que *rmin* et *rmax* encadrent le résultat. Quand la condition d'arrêt, rmin+1=rmax, est satisfaite, l'intervalle est de longueur 1 et le résultat étant dans l'intervalle, le résultat est égal à *rmin*. Par conséquent, le prédicat invariant définit l'intervalle de recherche de la solution et on comprend bien pourquoi il est très important de ne pas sortir de cet intervalle. On comprend également que chaque pas d'itération doit, non seulement rester dans l'intervalle et doit, réduire la taille de cet intervalle pour converger vers la solution. C'est le principe d'un calcul par approximations successives. La section 6 donne quelques indications pour trouver les invariants.

5. Stratégie de preuve d'un programme

Prouver un programme se décompose en trois sous-problèmes principaux, décomposer sa preuve en plusieurs preuves de fragments du programme, trouver un ordre de résolution des sous-problèmes et enfin trouver les valeurs des invariants d'itération. Nous traitons le premier point en section 5.1, le second en section 5.2 et le troisième en section 6.

5.1. Décomposition de la preuve d'un programme en preuves de fragments

Ci-dessus, nous avons fait des preuves de fragments de programmes. Le but initial était d'effectuer la preuve du programme complet. Nous allons montrer comment ramener cette preuve à un ensemble de preuves de fragments en annotant le programme. La méthode proposée est en six étapes :

1. Annoter le programme par des annotations inconnues appelées p_0, p_1, p_2, etc. de telle sorte que les règles *Seq, Si, Sinon* et *Tantque*, associées aux structures de contrôle du programme s'appliquent.
2. Définir les valeurs des annotations en fonction de la précondition, de la postcondition, des invariants des diverses itérations et des conditions des itérations et des conditionnelles. Déterminer un ensemble minimal d'annotations inconnues qu'il va falloir définir.
3. Recenser les sous problèmes qui restent à résoudre.
4. Déterminer un ordre de traitement des sous problèmes. Résoudre en priorité les sous problèmes qui permettent de trouver certaines annotations inconnues, en particulier les invariants.
5. Définir des annotations inconnues, en particulier les invariants.
6. Résoudre les sous problèmes restants :

   . soit si c'est un sous problème sans structure de contrôle qui consiste à faire une preuve de validité de formule ou de séquence de quelques affectations, alors le résoudre selon la stratégie de la section 2,

   . soit si c'est un sous problème avec structure de contrôle, alors lui réappliquer la méthode à partir de l'étape 1.

$\{Pre \stackrel{def}{=} n \geq 0\}$
rmin :=0 ; rmax := n+1 ;
$\{p_0\}$
Tantque rmin+1≠rmax **faire**
    $\{p_1\}$
    m := (rmin+rmax) **div** 2 ;
    $\{p_2\}$
    si m*m>n **alors** rmax := m **sinon** rmin := m **finsi**
    $\{p_3\}$
fait ;
$\{p_4\}$
r := rmin
$\{p_5\}$
$\{Post \stackrel{def}{=} r^2 \leq n \wedge n < (r+1)^2\}$

Figure 37 : Programme annoté racine carré par dichotomie annoté

Etape 1 : Annoter le programme

Idée clé 10 ***(Annoter les programmes pour les prouver)***

Pour faire la preuve d'un programme, l'annoter pour que les règles de la logique de HOARE s'appliquent. ♦

Exemple 30 *(Annotation du programme)*Nous annotons le programme comme dans la Figure 37. Nous mettons des annotations inconnues appelées p_i, i=1..n dans l'ordre du programme en les plaçant avant (p_0, p_2) et après (p_3, p_4) les itérations et les conditionnelles, au début (p_1) et à la fin (p_3) des corps des itérations et des conditionnelles et à la fin du programme (p_5).

Etape 2 : Définir les valeurs des annotations

Sur l'exemple, les annotations peuvent se définir ainsi :

$p_0 \stackrel{def}{=} I$, $p_1 \stackrel{def}{=} I \wedge rmin+1 \neq rmax$, $p_3 \stackrel{def}{=} I$, $p_4 \stackrel{def}{=} I \wedge rmin+1 = rmax$. Les annotations inconnues sont I, p_2 et p_5. Les annotations sont définies pour pouvoir appliquer les règles de la logique de HOARE de la manière suivante. Pour appliquer la règle *tantque*, les annotations doivent être les suivantes :

> $\{I\}$ **tantque** e **faire** $\{I \wedge e\}$ A $\{I\}$ **fait** $\{I \wedge \neg e\}$; Le programme étant annoté ainsi : $\{p_0\}$ **tantque** e **faire** $\{p_1\}$ A $\{p_3\}$ **fait** $\{p_4\}$, nous avons défini $p_0 \stackrel{def}{=} I$, $p_1 \stackrel{def}{=} I \wedge e$, $p_3 \stackrel{def}{=} I$, $p_4 \stackrel{def}{=} I \wedge \neg e$ afin de pouvoir appliquer la règle Tantque. C'est ce que nous avons appliqué à l'exemple pour définir p_0, p_1, p_3 et p_4.

Etape 3 : Définir les sous-problèmes restant à résoudre

Il reste cinq sous-problèmes à résoudre qui sont les suivants :

[f_{21}] $\{Pre \stackrel{def}{=} n \geq 0\}$ rmin :=0 ; rmax := n+1 $\{I\}$

[f_{22}] $\{I \wedge rmin+1 \neq rmax\}$ m := (rmin+rmax) **div** 2 $\{p_2\}$

[f_{23}] $\{p_2\}$ **si** m*m>n **alors** rmax := m **sinon** rmin := m **finsi** $\{I\}$

[f_{24}] $\{I \wedge rmin+1=rmax\}$ r := rmin $\{p_5\}$

[f_{25}] $p_5 \Rightarrow r^2 \leq n \wedge n < (r+1)^2\}$ $(p_5 \Rightarrow Post)$.

Ces cinq sous problèmes sont déduits de la Figure 37 de telle sorte que la règle *Seq* s'applique entre tous les fragments de la décomposition et que l'on prouve que si le programme termine dans l'état p_5, alors il satisfasse $Post$.

Etape 4 : Déterminer un ordre de résolution des sous-problèmes

Les quatre premiers sous-problèmes dépendent de l'annotation inconnue I. Sur cet exemple, une stratégie simple consiste à :

1. déterminer p_5 pour que la formule f_{25} soit valide,
2. déterminer I pour que la formule f_{24} soit un théorème,
3. connaissant I, démontrer que la formule f_{21} est un théorème,
4. déterminer p_2, soit à partir de la formule f_{22}, soit à partir de la formule f_{23}. Comme la formule f_{23} est plus complexe car elle utilise une conditionnelle, nous déterminerons p_2 à partir de f_{22}. Il faut trouver p_2 telle que la formule f_{22} soit un théorème.
5. démontrer la formule f_{23} par la même méthode puisque c'est un fragment de programme structuré par une conditionnelle.

Etape 5 : Déterminer des annotations inconnues

Déterminer p_5 pour que la formule f_{25} soit valide. Il suffit de choisir $p_5 \stackrel{def}{=} Post \stackrel{def}{=} r^2 \leq n \wedge n < (r+1)^2$. La formule est alors de la forme $p_5 \Rightarrow p_5$ qui est valide. Le sous-problème f_{25} est résolu.

Etape 6 : Résoudre les sous-problèmes restants

Premièrement, déterminer `I` pour que la formule f_{24} soit un théorème. La résolution de ce sous-problème consiste à calculer la précondition p_6 pour que la formule suivante soit un axiome de la logique de Hoare :

$[f_{26}]$ $\{p_6\}$ r := rmin $\{p_5\}$. Par substitution de *r* par *rmin*, on trouve $p_6 \stackrel{def}{=} rmin^2 \le n \wedge n < (rmin+1)^2$. Pour prouver la formule f_{24}, il faut démontrer que la formule f_{27} suivante est valide $I \wedge rmin+1=rmax \Rightarrow p_6$ et appliquer la règle *Pré*. Il faut trouver `I` pour quelle soit valide. Une solution possible est $I \stackrel{def}{=} rmin^2 \le n \wedge n < (rmax)^2$. On a obtenu cette formule en substituant *rmin+1* par *rmax* comme le permet la condition d'arrêt `rmin+1=rmax`. Ainsi la formule f_{27} est la suivante :

$rmin^2 \le n \wedge n < rmax^2 \wedge rmin+1=rmax \Rightarrow rmin^2 \le n \wedge n < (rmin+1)^2$ qui est valide car de la forme $I \wedge p \Rightarrow I$ si on remplace *rmin+1* par *rmax* dans le terme à droite de l'implique. Le sous problème f_{24} est résolu et nous avons trouvé `I`. Il reste à prouver f_{21}, f_{22} et f_{23}.

Avec cette valeur de `I`, la formule f_{21} est identique à la formule f_5 démontrée ci-dessus section 2. Le sous problème f_{21} est donc résolu.

$[f_{28}]$	$\{rmin^2 \le n \wedge n < rmax^2\}$	
	m := (rmin+rmax) **div** 2 $\{rmin^2 \le n \wedge n < rmax^2\}$	Axiome
$[f_{29}]$	$(rmin^2 \le n \wedge n < rmax^2 \wedge rmin+1 \ne rmax) \Rightarrow (rmin^2 \le n \wedge n < rmax^2)$	$p \wedge p' \Rightarrow p$
$[f_{22}]$	$\{rmin^2 \le n \wedge n < rmax^2 \wedge rmin+1 \ne rmax\}$	
	m := (rmin+rmax) **div** 2 $\{rmin^2 \le n \wedge n < rmax^2\}$	Pré(f_{28}, f_{29})

Figure 38 : Exemple de preuve d'affectation

Puis, on trouve facilement $p_2 \stackrel{def}{=} I$ en résolvant la formule f_{22}. Comme l'instruction *m := (rmin+rmax) **div** 2* ne modifie pas les variables *rmin* et *rmax* du prédicat `I`, il est clair que la formule suivante est un axiome.

$[f_{22}]$ $\{rmin^2 \le n \wedge n < rmax^2 \wedge rmin+1 \ne rmax\}$

m := (rmin+rmax) **div** 2 $\{rmin^2 \le n \wedge n < rmax^2\}$. Cet axiome, permet de prouver la formule f_{22} par la séquence de formules décrite dans la Figure 38.

Enfin, il faut résoudre le dernier sous-problème défini maintenant qu'on connaît `I` et p_2 par la formule suivante :

$\{I \stackrel{def}{=} rmin^2 \le n \wedge n < rmax^2\}$
si m*m>n **alors** $\{I \wedge m*m>n\}$ rmax := m $\{I\}$ **sinon** $\{I \wedge m*m \le n\}$ rmin := m $\{I\}$**finsi**
$\{I \stackrel{def}{=} rmin^2 \le n \wedge n < rmax^2\}$.

Les annotations de la conditionnelle ont été posées pour que la règle *Sinon* s'applique. Il reste alors les deux sous-problèmes suivants à résoudre :

$[f_{30}]$ $\{I \wedge m*m>n\}$ rmax := m $\{I\}$,

$[f_{31}]$ $\{I \wedge m*m \le n\}$ rmin := m $\{I\}$.

Ces deux sous problèmes de preuve d'une simple affectation sont résolus dans la Figure 39 et Figure 40 en établissant un axiome sur l'affectation et en appliquant la règle *Pré*.

$[f_{32}]$	$\{m^2 \le n \wedge n < rmax^2\}$ rmin := m $\{rmin^2 \le n \wedge n < rmax^2\}$	Axiome
$[f_{33}]$	$(rmin^2 \le n \wedge n < rmax^2 \wedge m*m \le n) \Rightarrow$	
	$(m^2 \le n \wedge n < rmax^2)$	valide car de la forme $p \wedge q \Rightarrow p$
$[f_{31}]$	$\{ rmin^2 \le n \wedge n < rmax^2 \wedge m*m \le n \}$ rmin:=m $\{rmin^2 \le n \wedge n < rmax^2\}$	Pré(f_{32}, f_{33})

Figure 39 : Preuve de l'affectation r := rmin

L'une des difficultés de cette preuve était de trouver un ordre de résolution des sous-problèmes. C'est à cette question que nous tentons de répondre dans la section suivante. ♦

$[f_{34}]$	$\{rmin^2 \le n \wedge n < m^2\}$ rmax := m $\{rmin^2 \le n \wedge n < rmax^2\}$	Axiome
$[f_{35}]$	$(rmin^2 \le n \wedge n < rmax^2 \wedge m*m > n) \Rightarrow$	
	$(m^2 > n \wedge rmin^2 \le n)$	valide car de la forme $p \wedge q \Rightarrow p$
$[f_{30}]$	$\{rmin^2 \le n \wedge n < rmax^2 \wedge m*m > n\}$ rmax:=m $\{rmin^2 \le n \wedge n < rmax^2\}$	Pré(f_{34}, f_{35})

Figure 40 : Preuve de l'affectation r := rmax

5.2. Comment trouver l'ordre des preuves des fragments ?

Reprenons l'exemple précédent pour lui appliquer la démarche en six étapes présentée dans la section précédente. Cette reprise permet d'illustrer le fait que l'application de la démarche n'est pas unique et qu'elle laisse une part de choix à l'utilisateur. Si nous ne connaissons rien d'autre que le programme et sa spécification, nous pouvons annoter le programme, y compris sa conditionnelle pour traiter le problème en une fois, comme dans la Figure 41 en utilisant deux annotations inconnues, `I` pour l'invariant d'itération et `P` pour la formule intermédiaire afin que les règles *Seq*, *Tantque* et *Sinon* s'appliquent. On a ainsi effectué en même temps les deux premières étapes de la méthode. Les sous problèmes restant à prouver sont les suivants :

- $[sp_1]$ $\{n \ge 0\}$ rmin :=0 ; rmax := n+1 $\{I\}$,
- $[sp_2]$ $\{I \wedge rmin+1 \ne rmax\}$ m := (rmin+rmax) **div** 2 $\{P\}$,
- $[sp_3]$ $\{P \wedge m*m > n\}$ rmax := m $\{I\}$,
- $[sp_4]$ $\{P \wedge m*m \le n\}$ rmin := m $\{I\}$,
- $[sp_5]$ $\{I \wedge rmin+1 = rmax\}$ r := rmin $\{r^2 \le n \wedge n < (r+1)^2\}$.

L'étape quatre consiste à déterminer un ordre de traitement de ces sous problèmes. L'ordre proposé est le suivant : sp_5 afin de trouver `I`, sp_2 afin de trouver `P`, puis sp_1, sp_3, sp_4 dans un ordre quelconque. Cet ordre a été établi ainsi pour les raisons suivantes. Pour résoudre les cinq sous problèmes, il est nécessaire de connaître `I`. Etant donné l'axiome d'affectation qui permet de calculer la plus faible pré condition d'une affectation, c'est sp_5 qui permet de trouver `I` le plus simplement. Une fois `I` trouvé, il est envisageable d'aborder n'importe lequel des autres sous problèmes. Mais le second permet de calculer `P` en utilisant l'axiome d'affectation qui permet de calculer la plus forte post condition d'une affectation *x := f(x)* dans le cas où *f* est une fonction qui possède une fonction inverse. C'est le cas dans cet exemple, c'est pourquoi nous avons choisi de résoudre sp_2 en second. La solution de ce sous problème nous définira `P`. Puis, ayant défini `P` et `I`, les trois autres sous problèmes sont sans prédicat inconnu et peuvent être traités dans un ordre quelconque.

L'étape cinq consiste à trouver `I` et `P`. Pour trouver `I`, il faut résoudre sp_5. On établit l'axiome suivant sur l'affectation *r := rmin* :

$\{rmin^2 \le n \wedge n < (rmin+1)^2\}$ r := rmin $\{r^2 \le n \wedge n < (r+1)^2\}$. Pour terminer la preuve, il faut établir que la formule logique suivante est valide :

$I \wedge rmin+1 = rmax \Rightarrow rmin^2 \le n \wedge n < (rmin+1)^2$. Il faut donc trouver `I` pour qu'elle soit valide. Une stratégie consiste à paramétrer la partie droite de l'implication par *rmax* qui n'apparaît pas dans cette partie droite, mais qui apparaît dans la condition d'arrêt et qui est l'une des variables d'itération. Sachant que $rmax = rmin+1$, on propose l'expression suivante pour $I \stackrel{def}{=} rmin^2 \le n \wedge n < rmax^2$ qui est obtenue en remplaçant *rmin+1* par *rmax*. On trouve bien l'invariant utilisé pour la preuve dans la section précédente. De `I`, et comme la variable *m* n'apparaît pas dans la pré condition de sp_2, nous déduisons $P \stackrel{def}{=} I \wedge rmin+1 \ne rmax$ pour que sp_2 soit l'axiome suivant :

$\{rmin^2 \le n \wedge n < rmax^2 \wedge rmin+1 \ne rmax\}$ m := (rmin+rmax) **div** 2

$\{rmin^2 \le n \wedge n < rmax^2 \wedge rmin+1 \ne rmax\}$.

Avec ces valeurs pour `I` et `P`, l'étape cinq consiste à résoudre les trois autres sous problèmes qui, sur l'exemple, se prouvent sans difficulté. Nous ne détaillons pas ces preuves ici. sp_1 a été prouvé dans la Figure 31. Les Figure 39 et Figure 40 contiennent des preuves similaires à celles de sp_3 et sp_4.

Les autres difficultés de cette preuve sont d'une part de trouver l'invariant de l'itération et d'autre part de justifier que les formules de la forme $p \Rightarrow q$ sont valides. C'est à ces questions que nous tentons de répondre dans les deux sections suivantes. ♦

```
{n≥0}
rmin :=0 ; rmax := n+1 ;
{I}
Tantque rmin+1≠rmax faire
        {I ∧ rmin+1≠rmax}
        m := (rmin+rmax) div 2 ;
        {P}
        si m*m>n alors {P ∧ m*m>n} rmax := m {I} sinon {P ∧ m*m≤n} rmin := m {I} finsi
        {I}
fait ;
{I ∧ rmin+1=rmax}
r := rmin
{r²≤n ∧ n<(r+1)²}
```

Figure 41 : Programme de racine carrée par dichotomie muni d'annotations inconnues

6. Comment trouver les invariants d'itération ?

Il y a deux manières d'aborder le problème. La première est syntaxique, la seconde est sémantique et utilise la connaissance sur la méthode de calcul par approximation successive que réalise l'itération.

La première méthode consiste à résoudre le problème suivant, connaissant `post`, `e` et A, trouver `I` tel que :

- `I` décore le programme ainsi : `{I}` **tantque** e **faire** `{I∧e}`A `{I}` **fait** `{I∧¬e},`
- et que `I∧¬e ⇒ post.`

Comme on connaît `e` et `post`, inventer `I` revient souvent à paramétrer `post` par les variables intervenant dans `e`.

Exemple 31 ***(recherche d'invariants)***

Par exemple, pour les 2 programmes de calcul de la racine carrée, le problème s'exprime ainsi, trouver `p` tel que :

- `I ∧ n<(r+1)² ⇒ (r²≤n ∧ n<(r+1)²),`
- `I ∧ rmin+1=rmax ⇒ (r²≤n ∧ n<(r+1)²).`

Dans le premier cas, on cherche à rendre les deux membres de l'implication identiques donc on cherche une équivalence, on trouve $I \stackrel{def}{=} r^2 \leq n$. Dans le second cas, on cherche à rendre les deux membres équivalents en paramétrant le prédicat $r^2 \leq n \wedge n<(r+1)^2$ en remplaçant *r* par *rmin* et *r+1* par *rmax* car `rmin+1=rmax` est une sous formule de la partie gauche de l'implication, on trouve $I \stackrel{def}{=} (rmin^2 \leq n \wedge n<(rmax)^2)$.

La seconde méthode consiste à raisonner à partir des choix de calcul par approximation successive effectués.

Dans le cas du calcul par dichotomie, la stratégie de calcul est d'encadrer le résultat dans un intervalle *rmin* .. *rmax* de telle sorte que lorsque l'intervalle est de longueur 1, le résultat *r* est égal à *rmin*.

Dans le cas du calcul par incrémentation, la stratégie de calcul est de prendre une approximation inférieure la plus grande possible et de l'incrémenter jusqu'à ce qu'on obtienne la valeur. Le prédicat qui définit une approximation par valeur inférieure, sachant que $n \geq 0$ et que *r* a été initialisé à zéro est : $r^2 \leq n$. Ce prédicat doit rester vrai pendant tout le calcul. Le prédicat qui permet de savoir que cette approximation est le résultat est : $n<(r+1)^2$. Donc l'itération s'exécute tant que cette condition n'est pas atteinte, d'où le programme de la Figure 9. ♦

Exemple 32 ***(découverte de l'invariant de factorielle)***

Figure 12, l'invariant du programme de calcul de factorielle est $I \stackrel{def}{=} f = \prod_{j=1}^{i} j$. On le trouve en cherchant I tel que $I \wedge i=n \Rightarrow f = \prod_{j=1}^{n} j$. La solution consiste à paramétrer la post-condition $f = \prod_{j=1}^{n} j$ par *i* car la condition d'arrêt est $i=n$. On obtient I en remplaçant *n* par *i* dans la post-condition comme le permet la condition d'arrêt $i=n$. Cet invariant signifie qu'à l'étape *i* on a calculé $f \stackrel{def}{=} 1*2* \ldots *i$. En particulier après l'initialisation *i := 0 ; f := 1*, l'invariant est vrai car $\prod_{j=1}^{i} j$ est égal à 1, l'élément neutre du produit. ♦

Figure 16, l'invariant du programme du tri à bulle est $I \stackrel{def}{=} \forall k.\forall u.((k \in [1..i] \wedge u \in [1..i] \wedge k<u) \Rightarrow t[k] \leq t[u])$. On le trouve en cherchant I tel que $I \wedge i>n-1 \Rightarrow post$. La solution consiste à paramétrer $post$ par *i* en remplaçant *n* par *i* car $i>n-1$ signifie en fait $i=n$ puisque *i* est incrémenté de 1 en 1 à partir de zéro, donc la première valeur de *i* supérieure à *n-1* est *i=n.*

Figure 14, l'invariant du programme de recherche dichotomique est $I \stackrel{def}{=} \forall k.(k \in [1..min] \Rightarrow t[k] \leq x) \wedge \forall k.(k \in [max+1..n] \Rightarrow t[k]>x)$. On le trouve en deux étapes. Comme l'itération est suivie de l'affectation *posx := min*, pour obtenir le prédicat p qui précède cette instruction, on applique l'axiome d'affectation à partir de $post$. On obtient $p \stackrel{def}{=} [posx := min]post \stackrel{def}{=} \forall k.(k \in [1..min] \Rightarrow t[k] \leq x) \wedge \forall k.(k \in [min+1..n] \Rightarrow t[k]>x)$. Puis dans une seconde étape on trouve I en cherchant I tel que $I \wedge min=max \Rightarrow p$. La solution consiste à substituer *min* par *max* dans un seul terme de p. Il ne faut substituer que l'occurrence de *min* dans l'expression *min+1* afin que les deux intervalles *1..min* et *max+1..n* soit séparés par un intervalle *min+1..max* d'éléments parmi lesquels la recherche n'a pas encore été effectuée. C'est la compréhension de la méthode qui nous fait faire ce choix. On n'est donc pas totalement syntaxique dans le raisonnement de cet exemple qui est plus difficile que les trois précédents.

7. Comment justifier la validité d'un prédicat p⇒q ?

Pour effectuer des preuves en logique de HOARE, les prédicats dont nous aurons à justifier la validité sont toujours de la forme $p \Rightarrow q$. Nous avons vu que ces prédicats apparaissent pour appliquer les règles *Pré* ou *Post* de cette logique. On propose cinq stratégies simples pour justifier de la validité de ces formules. Pour les quatre premières, il faut justifier les réécritures suivantes de la formule $p \Rightarrow q$:

- $p \Rightarrow q \equiv> faux \Rightarrow q$, donc justifier $p \equiv> faux$,
- $p \Rightarrow q \equiv> p \Rightarrow vrai$, donc justifier $q \equiv> vrai$,
- $p \Rightarrow q \equiv> p \Rightarrow p$, donc justifier $q \equiv> p$,
- $p \Rightarrow q \equiv> p' \wedge q \Rightarrow q$, donc justifier $p \equiv> p' \wedge q$.

En effet, les formules $faux \Rightarrow q$ et $p \Rightarrow vrai$ sont toujours vraies par définition de l'opérateur implique (voir Figure 17). C'est également le cas des formules de la forme $p \Rightarrow p$ et $p' \wedge q \Rightarrow q$ (voir Figure 17).

La cinquième consiste à faire une analyse par cas en justifiant que q est vrai si p l'est. Autrement dit on montre que le seul cas où $p \Rightarrow q$ pourrait être faux, c'est-à-dire où p est vrai et q est faux est impossible. Donnons un exemple de chacun des cinq cas.

Exemple 33 ***(Justification de validité de prédicats de la forme p ⇒ q par récriture en faux ⇒ q)***

Pour l'exemple de la recherche dichotomique (voir Figure 14), pour justifier que la formule suivante

$(n \geq 0 \wedge n \leq MAX \wedge \forall i.(i \in [1..n-1] \Rightarrow t[i] \leq t[i+1])) \Rightarrow$

$(\forall i.(i \in [1..0] \Rightarrow t[i] \leq x) \wedge \forall i.(i \in [n+1..n] \Rightarrow x<t[i]))$ est valide,

on peut justifier que les deux prédicats à droite de l'implication sont équivalents à vrai, c'est-à-dire valide : $\forall i.(i \in [1..0] \Rightarrow t[i] \leq x)$ et $\forall i.(i \in [n+1..n] \Rightarrow x<t[i])$. Ces prédicats sont des conjonctions pour tout *i* des prédicats suivants : $i \in [1..0] \Rightarrow t[i] \leq x$ et $i \in [n+1..n] \Rightarrow x<t[i]$. Pour que la conjonction soit vraie, il faut que chacun de ses membres soit vrai. Pour cela, on justifie que ces 2 prédicats se

mettent respectivement sous la forme $faux \Rightarrow t[i] \leq x$ et $faux \Rightarrow x < t[i]$. C'est le cas car pour tout *i*, le prédicat $i \in [1..0]$ est faux car l'intervalle $[1..0]$ est vide. On applique la même justification pour le second cas où le prédicat $i \in [n+1..n]$ est faux. ♦

Exemple 34 ***(Justification de validité de prédicats de la forme $p \Rightarrow q$ par récriture en $p \Rightarrow vrai$)***

Pour l'exemple de factorielle (voir Figure 12), pour justifier que $n \geq 0 \Rightarrow 1 = \prod_{j=1}^{0} j$, il suffit de constater que $1 = \prod_{j=1}^{0} j \equiv> vrai$, car l'élément neutre du produit est un, pour obtenir un prédicat de la forme $p \Rightarrow vrai$.♦

Exemple 35 ***(Justification de validité de prédicats de la forme $p \Rightarrow q$ par récriture en $p \Rightarrow p$)***

Dans l'exemple de racine carrée par incrémentation (voir Figure 9), pour justifier que $n \geq 0 \Rightarrow 0^2 \leq n$, il suffit de constater que $0^2 \leq n \equiv> 0 \leq n$ pour obtenir un prédicat de la forme $p \Rightarrow p$. ♦

Exemple 36 ***(Justification de validité de prédicats de la forme $p \Rightarrow q$ par récriture en $p' \wedge q \Rightarrow q$)***

Pour l'exemple de factorielle (voir Figure 12), pour justifier que $i \neq n \wedge f = \prod_{j=1}^{i} j \Rightarrow f*(i+1) = \prod_{j=1}^{i+1} j$, il suffit d'appliquer la stratégie en constatant que $f*(i+1) = \prod_{j=1}^{i+1} j \equiv> f = \prod_{j=1}^{i} j$ en divisant chaque membre de l'égalité par *i+1*.♦

Exemple 37 ***(Justification de validité de prédicats de la forme $p \Rightarrow q$ par analyse par cas)***

Par exemple pour justifier que $n \geq 0 \Rightarrow 0^2 \leq n \wedge n < (n+1)^2$ est un prédicat valide, on examine le cas où $n \geq 0$ est vrai. Dans ce cas le prédicat $0^2 \leq n \wedge n < (n+1)^2$ est vrai car pour tout *n* positif ou nul, $(n+1)^2$ est supérieur à *n.* ♦

8. Correction partielle et totale

Le système de vérification que nous avons présenté réalise des preuves de *correction partielle.* Ceci car la règle *Tantque* ne démontre pas la terminaison de l'itération. Pour elle, cette terminaison est une hypothèse.

La *correction totale* d'un programme nécessite donc de prouver séparément la terminaison. Nous ne développons pas de technique de vérification de terminaison dans ce cours. L'auteur intéressé par le sujet se reportera à [Berlioux]. Pour prouver la terminaison, il faut trouver une expression arithmétique entière dont on peut démontrer qu'elle décroît strictement à chaque fois que s'exécute un pas d'itération. Cette expression est appelée *variant* dans les langages d'annotations tels que ACSL, JML et SPEC#. Quelques fois, il faut trouver également un invariant d'itération (différent de l'invariant qui permet de démontrer la correction partielle) qui est nécessaire pour démontrer la décroissance de l'expression entière.

Par exemple, le programme de dichotomie défini dans le premier énoncé d'exercices ne termine pas toujours (voir la solution d'un des exercices de TD pour avoir un exemple de données pour lesquelles ce programme ne termine pas). Le variant de ce programme est *d-g.* Pour qu'il diminue strictement à chaque pas d'itération, il faut, soit que *d* diminue, soit que *g* augmente. *g* prend la valeur de *m.* Pour que *g* augmente, il faut que *m* soit toujours strictement plus grand que *g.* Or *m* étant égal à *(g+d) div 2*, quand *d=g+1*, *m=g* car la division entière arrondi à la valeur inférieure. Dans ce cas le programme ne termine pas. Pour qu'il termine, il suffit de faire le calcul de *m* ainsi : *m := (g+d+1) div 2.*

9. Bilan

Quelques Idées à retenir à l'issue du cours sur la Logique de HOARE

1- *Spécifier, construire, vérifier* et *évaluer* sont les quatre piliers de de la programmation ; dans cette partie de cours, nous n'avons pas abordé la question de l'évaluation qui consiste à calculer la complexité des programmes en nombre d'opérations élémentaires et en place mémoire utilisée afin de les comparer. Par exemple, le programme de calcul de la racine carrée par dichotomie effectue un nombre d'opérations de l'ordre de $\log_2$ de *n* alors que celui par incrémentation de l'ordre de *n*.

2- *Vérifier* est une étape à part entière du processus de développement d'un programme.

3- *Vérifier* c'est comparer deux énoncés décrivant la solution du même problème à deux niveaux :

- la spécification qui décrit QUOI FAIRE ?
- le programme qui décrit COMMENT LE FAIRE ?

4- *Prouver* est une vérification formelle basée sur une théorie des programmes appelée logique de HOARE. Des outils d'aide sont nécessaires.

5- De nombreux prouveurs existent, citons l'atelier B, les greffons wp et Jessie de Frama-C, PVS, HOL, Coq, …Ces outils demandent encore une bonne expertise pour guider les preuves.

6- Un autre sorte d'outils totalement automatique existe, ils s'appellent " vérificateurs de modèles " (Model checker en anglais) comme SPIN (ATT), SMV, MEC, VEDA, ... La technique appliquée revient à effectuer toutes les exécutions possibles du système. Ces outils ne s'appliquent pas dans le cas général et s'appliquent uniquement sur des programmes qui ont un nombre d'états mémoire fini, donc faisant peu de calcul, mais faisant du contrôle comme les protocoles de communication, les logiciels de commande de systèmes automatiques,… En résumé, on a le choix entre des outils automatiques, mais applicables à une classe très restreinte de programmes ou des outils plus généraux, mais beaucoup moins automatiques.

7- La *vérification rigoureuse* est faisable " à la main" en annotant les programmes par des annotations informelles, mais claires, rigoureuses et synthétiques. Elle est utile pour aider à construire des programmes corrects. Pour cela, les invariants sont nécessaires car ils expriment le principe fondamental du calcul par approximations successives :

Exemple : programme de tri, soit T[1..max] un tableau non trié, Dans le programme suivant, les assertions permettent de vérifier rigoureusement, mais non formellement, la cohérence de ce programme de tri par sélection :

i := 0; `{T[1..i] trié}`

tantque i<max **faire**

`{T[1..i] trié et i<max}`

i := i+1 ;

Recherche du plus petit élément dans T[i..max] ;

Echange de T[i] et du plus petit élément ;

`{T[1..i] trié}`

fait

`{T[1..max] trié et i=max}}`

8- Spécifier permet d'abstraire un programme pour permettre une vérification par parties. Une procédure étant vérifiée, pour l'utiliser, on peut raisonner uniquement à partir de sa spécification et on peut oublier le programme. Cette méthode s'appelle également *programmation par contrats.* Ceci est un autre avantage de savoir spécifier. Par analogie, on sait utiliser une voiture à partir des spécifications des actionneurs mis à disposition de l'utilisateur. Par exemple, pour conduire, il n'est pas nécessaire de savoir comment le volant fait tourner les roues (à partir d'engrenages ou d'une commande électronique de moteurs pas à pas), mais il suffit de savoir simplement qu'il fait tourner les roues.

Epilogue : Exemple de découverte d'erreur par preuve

Nous présentons ci-dessous un exemple de spécification et de programme qui ne sont pas cohérents. L'expérience montre que l'erreur n'a pas été découverte à la conception du programme. Cette incohérence a été commise dans les conditions suivantes. La spécification a été rédigée informellement pour un énoncé d'examen. La spécification formelle était demandée dans les questions. Un programme à compléter a été conçu à partir de la spécification informelle pour une autre question du sujet d'examen. On est donc dans une situation courante où le programme et la spécification sont conçus séparément par deux personnes différentes. Cette situation est celle pratiquée dans la méthode de test à partir de modèles où le testeur fait un modèle de la solution et ignore totalement l'implémentation développée par une autre équipe.

L'échec de la preuve met en évidence l'existence d'un problème. Il peut être de trois types différents : soit une mauvaise stratégie de preuve (dans un cas où la preuve est possible, mais n'a pas été trouvée), soit une erreur de spécification (mauvaise interprétation de l'énoncé informel), soit une erreur du programme. Nous verrons que, sur l'exemple, ce n'est pas une difficulté d'effectuer la preuve, mais un problème de cohérence entre spécification et programme qui peut être corrigé soit en modifiant la spécification, soit en modifiant le programme. La modification de la spécification consiste à ajouter des conditions supplémentaires aux données. Elle consiste donc à restreindre le problème. La solution de modification du programme prend en compte le problème initial dans toute sa généralité. L'expérience a montré que seulement 2 étudiants sur 23 ont remarqué qu'il était nécessaire d'ajouter une hypothèse sur les données pour rendre le programme cohérent avec l'énoncé.

1. Enoncé du problème et spécification : Somme de deux polynômes.

Décrire la spécification de la somme de 2 polynômes P_1 et P_2 représentés par des tableaux TP_1 et TP_2 et des entiers NP_1 et NP_2 définis ainsi :

Un polynôme P est représenté par un tableau TP de NP éléments (NP=degré (exposant) maximum apparaissant dans P) de telle sorte que TP[i] contient la valeur du coefficient du monôme d'exposant *i*. Par exemple, le polynôme $P \stackrel{def}{=} 1+5x + x^2 + 2x^3 + 3x^5$ est représenté par le tableau suivant où NP=5 (Par exemple, TP[3]=2 est le coefficient du monôme de degré 3, $2x^3$) :

$NP \stackrel{def}{=} 5$

$TP \stackrel{def}{=}$

1	5	1	2	0	3
0	1	2	3	4	5

Le résultat de la somme est un polynôme P représenté par un tableau TP et un entier NP de la même manière que les données. La somme consiste à ajouter les coefficients des termes de même exposant. Par exemple la somme de :

- $P_1 = 1+5x+x^2+2x^3+3x^5$
- et de $P_2 = 5+2x+6x^2+15x^4$
- est égale à $P=6+7x+7x^2+2x^3+15x^4+3x^5$.

La spécification proposée pour ce problème est décrite dans la Figure 42.

Constante EMAX =100 /* exposant maximum */
Données TP_1, TP_2 **: tableau** [0..EMAX] **d'entier** **/*** tableaux des coefficients des polynômes */
NP_1, NP_2 : 0..EMAX ; /*valeur des exposants maximum de P_1 et P_2 */
Résultat TP : **Tableau** [0.. EMAX] **d'entier** /*tableau des coefficients du polynôme P somme de P_1 et P_2 */
NP : 0.. EMAX ; /* valeur de l'exposant maximum de P */
Pré-condition $Pre \stackrel{def}{=} 0 \leqslant NP_1 \wedge NP_1 \leqslant EMAX \wedge 0 \leqslant NP_2 \wedge NP_2 \leqslant EMAX$
Post-condition $Post \stackrel{def}{=} NP=Max(NP_1, NP_2) \wedge$
$(NP=NP_1 \Rightarrow \forall j.(j \in 0..NP_2 \Rightarrow TP[j]=TP_1[j]+TP_2[j]) \wedge$
$\forall j.(j \in NP_2+1..NP_1 \Rightarrow TP[j]=TP_1[j]))$
$(NP=NP_2 \Rightarrow \forall j.(j \in ..NP_1 \Rightarrow TP[j]=TP_1[j]+TP_2[j]) \wedge$
$\forall j.(j \in NP_1+1..NP_2 \Rightarrow TP[j]=TP_2[j])).$

Figure 42 : Spécification de la somme de 2 polynômes

2. Programme solution du problème

Le programme proposé est décrit dans la Figure 43.

```
Début
   si NP1>NP2 alors NP := NP1 sinon NP := NP2 finsi ;
   i := 0 ;
   tantque i≠NP+1 faire
         TP[i] := TP1[i]+TP2[i] ;
         i := i+1
   fait
fin
```

Figure 43 : programme de calcul de la somme de deux polynômes

3. Preuve du programme

3.1. Annotation du programme

Pour faire la preuve, le programme est annoté (voir Figure 44) ainsi pour que les règles *Seq*, *Tantque* s'appliquent :

```
Début
   {Pre}
   si NP1>NP2 alors NP := NP1 sinon NP := NP2 finsi ;
   {NP=Max(NP1, NP2)}
   i := 0 ;
   {I}
   tantque i≠NP+1 faire
         {I ∧ i≠NP+1}
         TP[i] := TP1[i]+TP2[i] ;
         i := i+1
         {I}
   fait
{I ∧ i=NP+1}          /* cette condition doit impliquer Post */
fin
```

Figure 44 : Programme annoté calculant la somme de 2 polynômes

3.2. Conception de l'invariant d'itération

Pour faire la preuve, on propose d'utiliser l'invariant I défini ainsi :

$I \stackrel{def}{=} \forall j.(j \in 0..i\text{-}1 \Rightarrow TP[j]=TP_1[j]+TP_2[j]) \wedge NP=Max(NP_1, NP_2).$

Cet invariant spécifie que la somme est réalisée pour tous les monômes de degré 0 à *i-1* et que le nombre final de monômes est le maximum des nombres de monômes des données. Il est obtenu en paramétrant le prédicat $\forall j.(j \in 0..NP \Rightarrow TP[j]=TP_1[j]+TP_2[j])$ par *i*. Comme l'itération termine avec NP+1=i, on remplace NP par i-1 pour obtenir l'invariant d'itération.

3.3. Décomposition de la preuve

Les annotations posées décomposent la preuve en quatre formules à prouver :

- {Pre} **si** $NP_1>NP_2$ **alors** NP := NP_1 **sinon** NP := NP_2 **finsi** {NP=Max(NP_1, NP_2)},
- {NP=Max(NP_1, NP_2)} i := 0 {I},
- {I ∧ i≠NP+1} TP[i] := TP_1[i]+TP_2[i] ; i := i+1 {I},
- (I ∧ i=NP+1) ⇒ Post.

La preuve des trois premières formules ne pose pas de difficulté.

Exercice 8 ***(Preuves en logique de Hoare)***

Faire la preuve des trois premières formules de la liste ci-dessus. ♦

3.4. Echec de preuve

Par contre la preuve que la quatrième formule est valide n'est pas faisable. Examinons cette formule :

La formule (I ∧ i=NP+1) ⇒ Post est équivalente à :

$$(NP=Max(NP_1, NP_2) \wedge \forall j.(j \in 0..i\text{-}1 \Rightarrow TP[j]=TP_1[j]+TP_2[j]) \wedge i=NP+1) \Rightarrow$$
$$(NP=Max(NP_1, NP_2) \wedge$$
$$(NP=NP_1 \Rightarrow \forall j.(j \in 0..NP_2 \Rightarrow TP[j]=TP_1[j]+TP_2[j]) \wedge \forall j.(j \in NP_2+1..NP_1 \Rightarrow TP[j]=TP_1[j]))$$
$$(NP=NP_2 \Rightarrow \forall j.(j \in 0..NP_1 \Rightarrow TP[j]=TP_1[j]+TP_2[j]) \wedge \forall j.(j \in NP_1+1..NP_2 \Rightarrow TP[j]=TP_2[j]))).$$

Le membre gauche de l'implication se réécrit ainsi en substituant *i* par NP+1 :

$$(NP=Max(NP_1, NP_2) \wedge \forall j.(j \in 0..NP+1\text{-}1 \Rightarrow TP[j]=TP_1[j]+TP_2[j])).$$

La formule (I ∧ i=NP+1) ⇒ Post est alors équivalente à :

$$(NP=Max(NP_1, NP_2) \wedge \forall j.(j \in 0..NP \Rightarrow TP[j]=TP_1[j]+TP_2[j])) \Rightarrow$$
$$(NP=Max(NP_1, NP_2) \wedge$$
$$(NP=NP_1 \Rightarrow \forall j.(j \in 0..NP_2 \Rightarrow TP[j]=TP_1[j]+TP_2[j]) \wedge \forall j.(j \in NP_2+1..NP_1 \Rightarrow TP[j]=TP_1[j]))$$
$$(NP=NP_2 \Rightarrow \forall j.(j \in 0..NP_1 \Rightarrow TP[j]=TP_1[j]+TP_2[j]) \wedge \forall j.(j \in NP_1+1..NP_2 \Rightarrow TP[j]=TP_2[j]))).$$

Par équivalence du membre droit de l'implication, elle est équivalente à :

$$(NP=Max(NP_1, NP_2) \wedge \forall j.(j \in 0..NP \Rightarrow TP[j]=TP_1[j]+TP_2[j])) \Rightarrow$$
$$(NP=Max(NP_1, NP_2) \wedge$$
$$\forall j.(j \in 0..Min(NP_1, NP_2) \Rightarrow TP[j]=TP_1[j]+TP_2[j]) \wedge$$
$$\forall j.((j \in Min(NP_1, NP_2)+1..NP \wedge NP=NP_1) \Rightarrow TP[j]=TP_1[j]) \wedge$$
$$\forall j.((j \in Min(NP_1, NP_2)+1..NP \wedge NP=NP_2) \Rightarrow TP[j]=TP_2[j])).$$

Cette formule n'est pas valide car il est impossible de prouver que pour $j \in Min(NP_1, NP_2)+1..NP$, soit $TP[j]=TP_1[j]$ et $TP[j]=TP_1[j]+TP_2[j]$, soit $TP[j]=TP_2[j]$ et $TP[j]=TP_1[j]+TP_2[j]$. Pour y parvenir, il faudrait respectivement pour hypothèse que dans le premier cas $TP_2[j]=0$ et que dans le second cas $TP_1[j]=0$. Ce problème de preuve révèle que dans la représentation choisie, les coefficients de TP_i au-delà de l'indice NP_i ont une valeur non significative qui n'a pas obligatoirement été initialisée à zéro.

4. Correction du programme

Il y a donc deux solutions pour corriger ce problème :

- soit restreindre le problème traité en ajoutant, en pré-condition, l'hypothèse que les coefficients au-delà du plus grand sont initialisés à zéro, hypothèse formalisée par la formule H suivante : $H \stackrel{def}{=} \forall j.(j \in NP_1+1..EMAX \Rightarrow TP_1[j]=0) \wedge \forall j.(j \in NP_2+1..EMAX \Rightarrow TP_2[j]=0)$,
- soit modifier le programme A comme dans la Figure 45 pour ne pas faire la somme des coefficients de degré plus grand que NP_1 ou NP_2 :

Début
i := 0 ;
tantque $i<NP_1+1 \wedge i<NP_2+1$ **faire** $TP[i] := TP_1[i]+TP_2[i]$; i := i+1 **fait ;**
tantque $i<NP_1+1$ **faire** $TP[i] := TP_1[i]$; i := i+1 **fait ;**
tantque $i<NP_2+1$ **faire** $TP[i] := TP_2[i]$; i := i+1 **fait ;**
fin

Figure 45 : Programme effectuant la somme de 2 polynômes corrigé

Notons que l'erreur ci-dessus, n'est pas facile à détecter par le test car certains compilateurs peuvent initialiser les variables entières à zéro et d'autres pas. Si le compilateur initialise les variables à zéro, l'erreur ne sera sans doute pas découverte en effectuant du test unitaire du module somme. Par contre, lors du test d'intégration, si ce module est appelé avec des données fournies par un autre module qui ne remet pas à zéro des éléments supprimés après le dernier (par exemple un module qui dérive et ne remet pas à zéro le coefficient du monôme de plus grand degré qui est supprimé), alors une erreur va apparaître. Elle sera sans doute difficile à trouver car le succès des tests unitaires du module *somme* aura tendance à disculper ce module. Le développeur risque de chercher l'erreur en priorité à l'extérieur du module.

Leçon 8 : Etude de cas - racine carrée entière par division

Cet exemple illustre le cours de Sémantique et preuve où j'ai présenté la méthode de preuve de programmes séquentiels appelée Logique de Hoare. Cet exemple est suffisamment complexe pour que nous prenions quelques libertés avec la démarche présentée en cours. En particulier, dans la section 3, après avoir présenté le problème en section 2, nous présentons une solution algorithmique sous la forme d'un système de suites récurrentes avant de présenter le programme qui calcule ces suites en section 4. Puis dans la section 5, nous présentons la preuve mathématique de correction du résultat directement au niveau de cette solution. Enfin, dans la section 6, nous présentons la preuve de programmes en appliquant la Logique de Hoare. Nous décomposons la preuve pour que les règles *Seq* et *Tantque* s'appliquent. On peut noter que si la méthode de passage du système de suites récurrentes au programme de calcul est rigoureuse, la preuve au niveau des suites suffit. Mais ici, nous voulions illustrer la preuve de programmes. C'est l'occasion de constater que la preuve est plus légère au niveau de l'énoncé mathématique des suites.

1. Enoncé du problème

Décrire un programme -efficace pour de grands entiers- qui calcule la racine carrée entière d'un nombre *n* positif ou nul.

2. Spécification

Soit *n* la donnée, soit *r* le résultat, soit *racine* la fonction solution de profil $\texttt{entier} \rightarrow \texttt{entier}$. Sa spécification est la suivante : $\forall r.\forall n.(n \geq 0 \Rightarrow r^2 \leq n \wedge n < (r+1)^2)$.

Cette spécification peut aussi s'exprimer ainsi : $\forall r.\forall y.\forall n.(n \geq 0 \Rightarrow r^2+y=n \wedge y<2r+1)$. Dans cette spécification y représente $n-r^2$, c'est-à-dire le reste de n après extraction. Comme la différence entre $(r+1)^2$ et r^2 est égal à $2r+1$, si celle-ci est supérieure à y, alors r^2 est la plus grande approximation inférieure de la racine car et $(r+1)^2$ est supérieur à n.

3. Solution

3.1. Documents de présentation de la solution

Ces documents ont été trouvés sur le Web à l'adresse suivante : http://sectionsperso-orange.fr/therese.eveilleau/sections/truc_mat/textes/r_carree_anc.htm#zerobis.

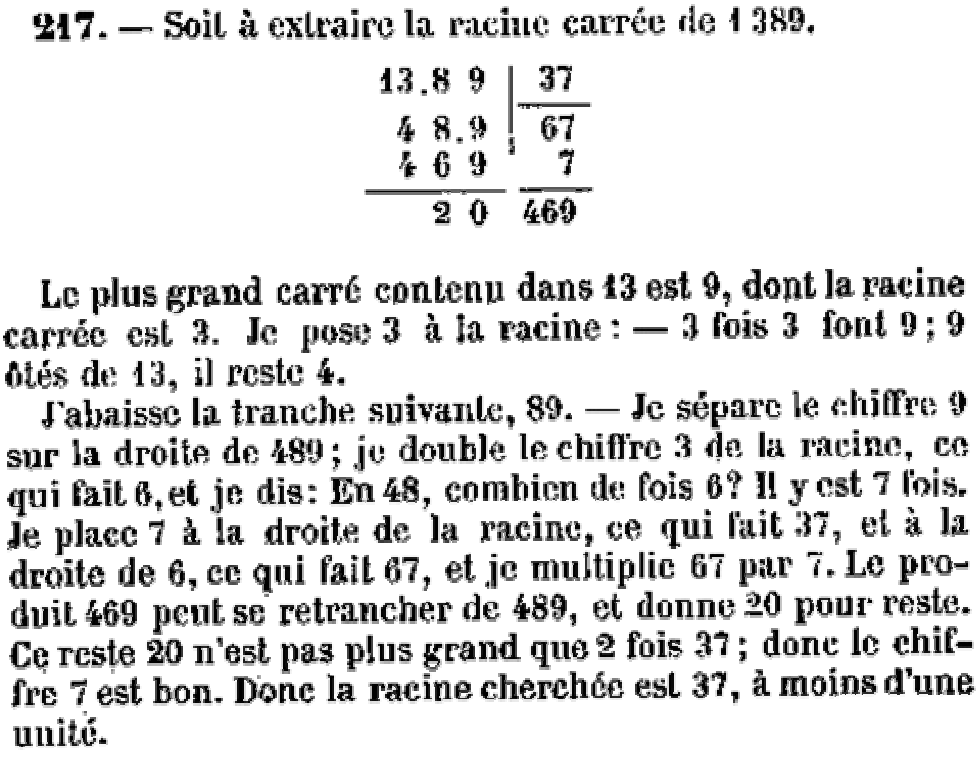
Cet programme était présenté dans le cours supérieur de certificat d'études des 11-13 ans en 1910. La Figure 46 présente une photocopie des documents du cours supérieur qui illustrait la méthode par un exemple, racine de 1389.

Voici maintenant ce même procédé, exprimé sous forme de règles pratiques, extrait d'un manuel *de Terminale C et T de V.Lespinard et R.Pernet 1968*

REGLE PRATIQUE

1. Ecrire le nombre dont on veut extraire la racine comme le dividende d'une division.
2. Séparer en tranches de deux chiffres à partir de la droite ; la dernière tranche à gauche peut n'avoir qu'un chiffre.
3. Extraire la racine de la première tranche à gauche ; on obtient ainsi le premier chiffre de la racine cherchée qu'on écrit à la place du diviseur habituel.
4. Retrancher le carré de ce nombre d'un chiffre de la première tranche à gauche.
5. Abaisser à droite du résultat de la soustraction précédente (premier reste partiel), la tranche suivante.

6. Séparer dans le nombre obtenu le dernier chiffre à droite et diviser le nombre restant par le double du nombre d'un chiffre écrit à la place du diviseur ; on écrit le double de ce nombre à la place du quotient.
7. Si le quotient est inférieur à 10 l'essayer, sinon commencer par essayer 9 ; l'essai se fait en écrivant ce quotient à droite du double de la racine de la première tranche et en multipliant le nombre obtenu par le quotient considéré. Si le produit peut être retranché du nombre formé au 5, le quotient convient, sinon on essaie un nombre inférieur jusqu'à ce que la soustraction soit possible.
8. Le résultat de la soustraction est le deuxième reste partiel. Ecrire le nombre essayé à droite du premier chiffre écrit à la place du diviseur.
9. Recommencer avec le deuxième reste partiel comme avec le premier et ainsi de suite, jusqu'à ce que l'on ait utilisé toutes les tranches. Le dernier reste partiel est le reste de la racine carrée.



217. — Soit à extraire la racine carrée de 1 389.

13.8 9	37
4 8.9	67
4 6 9	7
2 0	469

Le plus grand carré contenu dans 13 est 9, dont la racine carrée est 3. Je pose 3 à la racine : — 3 fois 3 font 9 ; 9 ôtés de 13, il reste 4.

J'abaisse la tranche suivante, 89. — Je sépare le chiffre 9 sur la droite de 489 ; je double le chiffre 3 de la racine, ce qui fait 6, et je dis : En 48, combien de fois 6 ? Il y est 7 fois. Je place 7 à la droite de la racine, ce qui fait 37, et à la droite de 6, ce qui fait 67, et je multiplie 67 par 7. Le produit 469 peut se retrancher de 489, et donne 20 pour reste. Ce reste 20 n'est pas plus grand que 2 fois 37 ; donc le chiffre 7 est bon. Donc la racine cherchée est 37, à moins d'une unité.

Figure 46 : Illustration du programme d'extraction de la racine carrée issue du cours supérieur du certificat d'études de 1910.

3.2. Quelques éléments historiques

Le texte suivant, disponible sur http://newton.mat.ulaval.ca/amq/bulletins/mai06/HodgsonMai06.pdf (adresse Web), issu du livre intitulé « Coups d'œil à saveur historique sur l'extraction de racine carrée de Bernard R. Hodgson de l'Université de Laval montre la richesse des travaux et donne quelques éléments sur leur origine historique et géographique :

« Les méthodes que nous présentons ont été développées en divers moments et lieux de l'histoire des mathématiques et illustrent bien, nous semble-t-il, la richesse et l'ingéniosité des points de vue que l'on a su adopter d'une époque à l'autre. Notre périple nous amènera tout d'abord en Mésopotamie, où on observera des valeurs approchées de racines carrées pouvant se justifier par un argument géométrique ; puis en Grèce, avec les calculs par approximations successives résultant de la célèbre méthode de Héron ; cet programme est lui-même un cas particulier de la méthode de Newton-Raphson, dans laquelle intervient la dérivée d'une fonction bien choisie ; puis on verra comment une valeur de $\sqrt{2}$ présente dans la tradition mathématique indienne peut s'expliquer en faisant appel à une dissection astucieuse de deux carrés ; on empruntera ensuite à la tradition chinoise une approche géométrique menant à le programme de type « chiffre à chiffre » encore enseigné dans nos écoles primaires il a quelques décennies, avant l'avènement de la calculatrice ; enfin on terminera par une technique qui peut être rattachée à l'équation de Pell-Fermat. »

3.3. Présentation de la solution algorithmique

L'exemple de la Figure 47 présente intuitivement le programme d'extraction par division, autrefois enseigné au collège. Ce programme a été conçu par des mathématiciens (algorithmicien) chinois (voir le livre de Hodgson). Dans cette leçon, nous retreignons sa présentation au cas du calcul de la racine carrée entière. Il est généralisable pour calculer la racine réelle et également la racine cubique (voir http://sectionsperso-orange.fr/therese.eveilleau/sections/truc_mat/textes/r_carree_anc.htm#zerobis). La Figure 47 applique le programme au nombre 22391825 et trouve pour racine 4732.

Sur cette figure, les chiffres en gras représentent la suite des chiffres de la solution.

Cet programme définit *r* comme le dernier terme d'une suite de valeurs ayant autant de valeurs que la décomposition de la donnée *n* en nombres compris entre 0 et 99.

22	39	18	25	4 7 3 2
22				(0*20+**4**)***4** ≤ 22 < (0*20+5)*5
-16				
6	39			(4*20+**7**)***7** ≤ 639< (4*20+8)*8
- 6	09			
	30	18		(47*20+**3**)***3** ≤ 3018 < (47*20+4)*4
	-28	29		
	1	89	25	(473*20+**2**)***2** ≤ 18925 < (473*20+3)*3
	- 1	89	24	

Figure 47 : exemple d'extraction pour un entier long

Soit *t2* une procédure qui résout le sous-problème suivant : décomposition d'un entier donné *n* en une suite de nombres compris entre 0 et 99 (de deux chiffres). Cette procédure est spécifiée dans la Figure 48.

```
{n≥0}
procédure t2(donnée n : entier; résultats nt2 : 0..MAX; t : tableau[1..MAX] de 0..99);
{n = Σ(k=1..nt2) t[k]*100^(k-1) ; ex : n=22391825, nt2=4, t[1]=25, t[2]=18, t[3]=39, t[4]=22}
```

Figure 48 : spécification de la procédure de décomposition d'un nombre en tranches de 2 chiffres

La Figure 50 fait apparaître une solution r_{nt2} qui est un terme du système de quatre suites récurrentes suivant :

- la suite c_i (en gras Figure 50) des chiffres de la valeur de la racine; ex : 4, 7, 3, 2,
- la suite x_i des parties de dividende qui servent au calcul des c_i ; ex : 22, 639, 3018, 18 925,
- la suite y_i des restes des parties de dividende après extraction du chiffre de la racine ; ex : 0, 6, 30, 189, 1,
- la suite r_i des valeurs de la racine ; ex : 0, 4, 47, 473, 4732.

Ces suites sont définies dans la Figure 49 à partir de *t* et de *nt2* pour $i \in [1..nt2]$.

Notons que nous ne définissons pas les termes x_0 et c_0 qui sont inutiles pour définir les termes de rang i=1. Par contre, nous définissons les termes y_0 pour définir x_1 et r_0 pour définir y_1.

$$
\begin{array}{l}
\cdot\ \ x_i = y_{i-1}*100+t[nt2-i+1], \\
\cdot\ \ \begin{cases} y_0 = 0 \\ y_i = x_i-(r_{i-1}*20+c_i)*c_i \end{cases} \\
\cdot\ \ c_i \text{ tel que } (r_{i-1}*20+c_i)*c_i \le x_i < (r_{i-1}*20+c_i+1)*(c_i+1), \\
\cdot\ \ \begin{cases} r_0 = 0 \\ r_i = r_{i-1}*10+c_i \end{cases}
\end{array}
$$

Figure 49 : Suites récurrentes définissant la solution racine

4. Programme

Soit la relation de dépendance "x précède y" représentée par : "x → y" dans la Figure 50 et définie ainsi dans [Grégoire 86]:

. *x* précède *y* si y_i est défini en fonction de x_i,

. *x* précède *y* si x_i est défini en fonction de y_{i-1}.

Les dépendances entre les suites de la Figure 49 sont représentées par le graphe représenté dans la Figure 50.

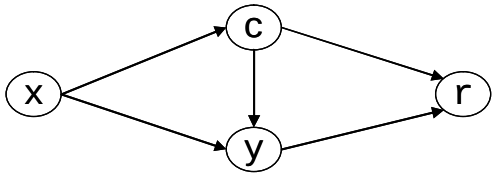


Figure 50 : graphe de dépendance du système de suites récurrentes

Les définitions étant, au plus, récurrentes d'ordre 1 et étant donné que le graphe de dépendances entre les variables représenté dans la Figure 50 est sans circuit, en représentant chacune des suites par une variable, il existe un ordre de calcul. *x, c, y, r*. C'est le seul ordre de calcul possible étant donné les dépendances.

La procédure définie dans la Figure 51 calcule le chiffre courant de la racine en fonction de la racine courante *r* et du dividende *x* :

```
{r≥0 ∧ x≥0 ∧ 200r+100>x}
procédure calcul_c (donnée r, x : entier ; résultat c : 0..9)
{Le résultat c est tel que (r*20+c)*c ≤x< (r*20+c+1)*(c+1)}
```

Figure 51 : Spécification pré-post du calcul d'un chiffre de la racine carrée

Le programme de calcul de la racine carrée d'un nombre entier positif ou nul, selon cette méthode, est défini dans la Figure 53. Notons que ce programme fait appel à la procédure *calcul_c* alors que l'appel de procédure ne fait pas partie du langage de programmation présenté dans la leçon 4. Cet appel permet de structurer le programme et de limiter la preuve à la procédure *racine* en faisant l'hypothèse que la procédure *calcul_c* est correcte. Ceci est possible dans ce cas car l'appel de la procédure *calcul_c* respecte des règles qui font qu'il ne provoque pas d'effet de bord par synonymie. En fait, l'appel pourrait être remplacé par le corps de la procédure dans lequel on aurait substitué les paramètres formels par les paramètres effectifs de l'instruction d'appel.

La procédure *calcul_c* peut être implémentée par le programme annoté de la Figure 52. Ce programme recherche *c* à partir de zéro en incrémentant de 1 à chaque fois. On peut facilement prouver qu'il est correct grâce aux annotations. L'invariant d'itération est $I \stackrel{def}{=} \{(r*20+c)*c \leq x$. Il est vrai après l'initialisation c := 0 puisqu'en remplaçant *c* par zéro dans {(r*20+c)*c, on obtient la partie x≥0 de la pré-condition. Il est maintenu par le pas d'itération puisque en substituant *c* par *c+1* dans I, on obtient la condition d'itération x≥(r*20+c+1)*(c+1). Enfin, la conjonction de I et de la condition de sortie de l'itération est exactement égale à la post-condition. Notons qu'un programme par dichotomie serait plus efficace bien que ce ne soit pas très spectaculaire dans un intervalle de dix valeurs.

```
{r≥0 ∧ x≥0 ∧ 200r+100>x}
procédure calcul_c (donnée r, x : entier ; résultat c : 0..9) ;
var
début
   c := 0 ; {(r*20+c)*c≤x}
   tantque x≥(r*20+c+1)*(c+1) faire
           {(r*20+c)*c≤x ∧ x≥(r*20+c+1)*(c+1)}
           c := c+1
           {(r*20+c)*c≤x}
   fait
   {(r*20+c)*c≤x ∧ x<(r*20+c+1)*(c+1)}
fin
{Le résultat c est tel que (r*20+c)*c ≤x< (r*20+c+1)*(c+1)}
```

Figure 52 : Programme de calcul d'un chiffre de la racine

```
{n≥0}
procédure racine( donnée n : entier ; résultat r : entier);
{Le résultat r est tel que : r²+y=n ∧ y<2r+1}
const MAX =…;
var          c : 0..9; nt2 : 0..MAX; t : tableau[1..MAX] de 0..99;
    i : entier; -- indice courant des suites
    x : entier;
    y : entier;
début
    t2(n, nt2, t);       -- décomposition en suite de nombres de 0 à 99
    y:=0; r:=0; i:=0;  -- calcul de y0 et r0
    tantque i < nt2 faire
              i:= i + 1;
              x:= y*100+t[nt2-i+1];          -- calcul de xi
              calcul_c(r, x, c);             -- calcul de ci
              y:= x-(r*20+c)*c;              -- calcul de yi
              r:= r*10+c                     -- calcul de ri
    fait
fin.
```

Figure 53 : Programme de calcul de la racine carrée par division

5. Preuve de solution

La preuve de solution consiste à montrer qu'une propriété invariante pour chaque occurrence des suites du système de suites solution implique la spécification pour le terme définissant le résultat. Sur l'exemple la propriété invariante choisie est la suivante :

$$[28]^2\ n=(r_i*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y_i*100^{nt2-i}.$$

Le résultat `r` est le terme r_{nt2} défini par le système de suites récurrentes de la Figure 49. Pour `i=nt2`, l'assertion [28] est équivalente à $n=(r_{nt2})^2+y_{nt2}$ car `nt2-i=0`, par conséquent $10^{nt2-i}=10^0=1$ et la somme de `k=1` à zéro est égale à zéro. Pour prouver que cette solution satisfait la spécification, il faudra montrer également que $y_{nt2}<2r_{nt2}+1$.

L'invariant [28] a donc été conçu à partir de l'objectif $n=(r_{nt2})^2+y_{nt2}$. C'est une généralisation de cette formule au cas `i`. Dans le cas `i`, `r` n'est pas la racine, mais les `i` premiers chiffres de la racine. C'est pourquoi on multiplie r_i par 10^{nt2-i}. Ce nombre est donc une approximation inférieure de la racine. C'est pourquoi on ajoute à $(r_i*10^{nt2-i})^2$ le reste du nombre pour lequel la racine carré n'est pas extraite : $\sum_{k=1}^{nt2-i} t[k]*100^{k-1}$. Il faut également ajouter le reste y_i de la partie extraite multiplié par 100^{nt2-i} Par exemple, à l'étape i=1, r_1=4, y_1=6, l'invariant est vérifié de la manière suivante : 22 39 18 25 = $(4*10^3)^2$+39 18 25+6*100^3.

Finalement, sur cet exemple, la preuve s'effectue en deux étapes :

. preuve de l'invariance de la propriété [28],

. preuve de l'invariance de la propriété [29] suivante $y_i<2r_i+1$.

La preuve de terminaison ne pose aucun problème sur cet exemple puisque le nombre de termes des suites est fini et égal à *nt2*, le nombre de tranches de deux chiffres de *n*.

[2] Notons que c'est par hasard que la numérotation des formules débute à 28.

5.1. Preuve de l'invariant [28]

Cette preuve s'effectue par récurrence sur les suites x_i, c_i, y_i, r_i.

1. cas $i=0$

$$n=(r_i*10^{nt2-i})^2 + \sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y_i*100^{nt2-i}$$

$$n=(0*10^{nt2})^2 + \sum_{k=1}^{nt2} t[k]*100^{k-1}+0*100^{nt2} \qquad \text{car } r_0=0 \text{ et } y_0=0$$

$$n=\sum_{k=1}^{nt2} t[k]*100^{k-1}.$$

C'est vrai par définition de *t* et *nt2* qui sont résultats de l'application de la procédure *t2* (voir la post condition de la Figure 48 qui est égale à l'expression ci-dessus).

2. cas i quelconque

L'hypothèse de récurrence [HR] consiste à supposer que [28] est vraie pour $i-1$, c'est à dire :

$$[HR] \qquad n=(r_{i-1}*10^{nt2-i+1})^2 + \sum_{k=1}^{nt2-i+1} t[k]*100^{k-1}+y_{i-1}*100^{nt2-i+1}.$$

Des définitions des suites dans la Figure 49, nous déduisons les définitions suivantes de r_{i-1} et y_{i-1} en fonction des termes de rang i :

- $r_i=r_{i-1}*10+c_i$, donc $r_{i-1}=(r_i-c_i)/10$,
- $y_i=x_i-(r_{i-1}*20+c_j)*c_i$ donc $x_i=y_i+(r_{i-1}*20+c_i)*c_i$,
- $x_i=y_{i-1}*100+t[nt2-i+1]$, donc $y_{i-1}=(x_i-t[nt2-i+1])/100$. Par substitution de la définition précédente de x_i, on obtient :
- $y_{i-1}=(y_i+(r_{i-1}*20+c_i)*c_i-t[nt2-i+1])/100$. Par substitution de la définition de r_{i-1} :
- $y_{i-1}=(y_i+((r_i-c_i)*2+c_i)*c_i-t[nt2-i+1])/100$.

On substitue les définitions de r_{i-1} et y_{i-1} dans l'hypothèse [HR] pour démontrer la conclusion de la manière suivante :

$$n=(((r_i-c_i)/10)*10^{nt2-i+1})^2+ \sum_{k=1}^{nt2-i+1} t[k]*100^{k-1}+$$

$$((y_i+((r_i-c_i)*2+c_i)*c_i-t[nt2-i+1])/100)* \ 100^{nt2-i+1}).$$ On développe les produits et les divisions par 2, 10 et 100 :

$$n=(r_i*10^{nt2-i}-c_i*10^{nt2-i})^2+ \sum_{k=1}^{nt2-i+1} t[k]* \ 100^{k-1} +$$

$$y_i*100^{nt2-i}+(2r_i-c_i)*c_i*100^{nt2-i}-t[nt2-i+1]* \ 100^{nt2-i}).$$

On déplace et on développe le carré et la multiplication par c_i :

$$n=(r_i*10^{nt2-i})^2-2r_i*c_i*100^{nt2-i}+(c_i*10^{nt2-i})^2+ \sum_{k=1}^{nt2-i+1} t[k]*100^{k-1}$$

$$-t[nt2-i+1]*100^{nt2-i}+y_i*100^{nt2-i}+(2r_i*c_i)*100^{nt2-i}-c_i^2*100^{nt2-i}.$$

En appliquant les simplifications suivantes :

- $-2r_i*c_i*100^{nt2-i}+2r_i*c_i*100^{nt2-i}=0$,
- $\sum_{k=1}^{nt2-i+1} t[k]*100^{k-1}-t[nt2-i+1]*100^{nt2-i}= \sum_{k=1}^{nt2-i} t[k]*100^{k-1}$,
- $(c_i*10^{nt2-i})^2-c_i^2*100^{nt2-i}=0$,

on obtient la propriété invariante :

$$n=(r_i*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y_i*100^{nt2-i}.$$

5.2. Preuve que $y_i<2r_i+1$

Cette propriété se démontre à partir de la partie [30] de la post condition de la procédure *calcul_c* :
$[30]\ (r_{i-1}*20+c_i+1)*(c_i+1)>x_i$.

Ci-dessus nous avons établi la définition suivante pour x_i : $x_i=y_i+(r_{i-1}*20+c_i)*c_i$. On la substitue dans $[30]$: $(r_{i-1}*20+c_i+1)*(c_i+1)>y_i+(r_{i-1}*20+c_i)*c_i$.

On développe le produit de la partie gauche :
$(r_{i-1}*20+c_i)*c_i+c_i+(r_{i-1}*20+c_i+1)>y_i+(r_{i-1}*20+c_i)*c_i$. On simplifie en supprimant le terme $(r_{i-1}*20+c_i)*c_i$ qui apparaît de chaque côté de l'opérateur '>' :
$c_i+r_{i-1}*20+c_i+1>y_i$.

On substitue la définition de r_{i-1} : $r_{i-1}=(r_i-c_i)/10$, on obtient :
$c_i+((r_i-c_i)/10)*20+c_i+1>y_i$, c'est à dire : $c_i+2r_i-2c_i+c_i+1>y_i$, donc $2r_i+1>y_i$.

6. Preuve du programme

La preuve d'un programme consiste à prouver qu'il satisfait la spécification selon la méthode formelle établie à partir de la logique de Hoare. Sur cet exemple la preuve de terminaison ne pose pas de problème si on suppose que les procédures *t2* et *calcul_c* terminent puisque le nombre d'itérations est borné par *nt2*.

Soit I l'invariant suivant : $n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}$ obtenu en remplaçant les termes des suites par les variables qui les représentent dans [28] puisque le programme calcule les suites définies dans la Figure 49. La preuve de correction partielle peut s'effectuer en adoptant la stratégie suivante :

- Preuve que l'invariant I est vrai avant l'itération étant donné la pré condition $\{n\geq 0\}$,
- Preuve que l'itération conserve l'invariant,
- Preuve que la séquence de ces deux phases est correcte,
- Preuve que l'invariant et la condition d'arrêt permettent de démontrer la post condition de la spécification : $r^2+y=n$,
- Preuve que la propriété $y<2r+1$ est invariante dans l'itération. Alors en appliquant la règle [Et] entre les deux dernières propriétés on démontre la spécification $r^2+y=n \wedge y<2r+1$.

6.1. Preuve que l'invariant est vrai avant l'itération

Cette partie consiste à montrer que la partie initiale de le programme décrit dans la Figure 53 établit l'invariant I. Le but est donc de prouver la formule suivante :

$$\{n\geq 0\}\ t2(n, nt2, t);\ y:=0;\ r:=0;\ i:=0\ \{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}.$$

La preuve pas à pas est la suivante :

[31] $\{n\geq 0\}\ t2(n, nt2, t)\ \{n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$ -- par définition de t2

[32] $\{0=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}y:=0\ \{y=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$-- Axiome Aff

Notons que l'élimination de 0=0 dans la pré condition de la formule [32] se fait comme dans l'exemple précédent par application de la règle *Pré*, à partir de l'assertion [32]. On ne détaille pas cette étape.

[33] $\{n\geq 0\}$ t2(n, nt2, t); y:=0 $\{y=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$ -- Seq(31,32)

[34] $\{0=0 \wedge \{y=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$ r:=0 $\{r=0 \wedge y=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$-- Aff

[35] $\{n\geq 0\}$ t2(n, nt2, t); y:=0 ; r:=0 $\{r=0 \wedge y=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$ -- Seq(33,34)

[36] $\{0=0 \wedge r=0 \wedge y=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$ i:=0

$\{i=0 \wedge r=0 \wedge y=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$ -- Aff

[37] $\{n\geq 0\}$ t2(n, nt2, t); y:=0 ; r:=0 ; i:=0 $\{i=0 \wedge r=0 \wedge y=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$--Seq(35,36)

[38] $i=0 \wedge r=0 \wedge y=0 \wedge n=\sum_{k=1}^{nt2} t[k]*100^{k-1} \Rightarrow$

$n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}$ -- vrai (voir Justification ci-dessous)

[39] $\{n\geq 0\}$ t2(n, nt2, t); y:=0; r:=0; i:=0

$\{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$ -- Post(37,38)

Justification de [38] : Elle est de la forme $p \Rightarrow q$. Dans le cas où p est faux, la propriété est vrai. Dans le cas où p est vraie, i=0, r=0 et y=0 sont trois sous-formules vraies. En remplaçant i, r et y par 0 dans q, q est équivalent à $q' = n=\sum_{k=1}^{nt2} t[k]*100^{k-1}\}$. La formule [38] est donc de la forme $p' \wedge q' \Rightarrow q'$. C'est une tautologie.

6.2. Preuve que l'itération conserve l'invariant

Cette partie consiste à prouver que chaque pas d'itération conserve l'invariant. Pour prouver ceci, il faut prouver que la formule [53] ci-dessous est un théorème de la Logique de HOARE. Effectuons la preuve pas à pas en établissant des axiomes sur les instructions d'affectation par la règle de substitution *Aff*, puis en les composant par la règle de la séquence d'instructions (règle *Seq*) et en appliquant la règle *Pré* quand c'est nécessaire pour arriver au but. C'est ce que nous présentons ci-dessous.

[40] $\{n=((r*10+c)*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$ r := r*10+c

$\{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$ -- Aff

[41] $\{n=((r*10+c)*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+(x-(r*20+c)*c)*100^{nt2-i}\}$

y := x-(r*20+c)*c $\{n=((r*10+c)*\ 10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$ -- Aff

[42] $\{n=((r*10+c)*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+x-(r*20+c)*c*100^{nt2-i}\}$ y := x-(r*20+c)*c;

$$r := r*10+c \ \{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\} \quad \text{-- Seq(41, 40)}$$

[43] $n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+x*100^{nt2-i} \Rightarrow$

$$n=((r*10+c)*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+(x-(r*20+c)*c)*100^{nt2-i} \quad \text{-- voir justification}$$

Justification de la formule [43] : On démontre que les deux membres de l'implication sont équivalents par développement et simplification de $((r*10+c)*10^{nt2-i})^2$ et de $(x-(r*20+c)*c*100^{nt2-i})$ dans le second membre de la formule [43].

[44] $\{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+x*100^{nt2-i}\}$ y := x-(r*20+c)*c ;

$$r:=r*10+c \ \{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\} \quad \text{-- Pré(42, 43)}$$

[45] $\{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+x*100^{nt2-i}\}$ calcul_c(r, x, c)

$$\{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+x*100^{nt2-i}\} \quad \text{-- Aff}$$

Notons que la pré condition et la post condition de la formule [44] est identique car la procédure *calcul_c* ne modifie aucune de ses variables.

[46] $\{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+(y*100+t[nt2-i+1])*100^{nt2-i}\}$

$$x := y*100+t[nt2-i+1] \ \{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+x*100^{nt2-i}\} \quad \text{-- Aff}$$

[47] $\{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-(i+1)} t[k]*100^{k-1}+(y*100+t[nt2-(i+1)+1])*100^{nt2-i}\}$ i := i+1

$$\{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+(y*100+t[nt2-i+1])*100^{nt2-i}\} \quad \text{-- Aff}$$

[48] $n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i} \Rightarrow$

$$n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-(i+1)} t[k]*100^{k-1}+(y*100+t[nt2-(i+1)+1])*100^{nt2-i} \quad \text{– voir justification}$$

Justification de la formule [48] : Le premier membre de l'implication est obtenu facilement par simplification du second membre.

[49] $\{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$ i := i+1

$$\{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+(y*100+t[nt2-i+1])*100^{nt2-i}\} \text{--Pré(48, 47)}$$

[50] $\{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$ i := i+1; x := y*100+t[nt2-i+1]

$\{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+x*100^{nt2-i}\}$ -- Seq(49, 46)

[51] $\{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$ i := i+1; x := y*100+t[nt2-i+1] ;

calcul_c(r, x, c) $\{n=(r*10^{nt2-i+1})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+x*100^{nt2-i}\}$ -- Seq(50, 45)

[52] $\{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$

i := i+1; x := y*100+t[nt2-i+1] ; calcul_c(r, x, c) ; y := x-(r*20+c)*c ; r := r*10+c

$\{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$ -- Seq(51, 44)

note : l'ajout de la conjonction avec `i<nt2` à la pré condition de la formule [52] ne pose pas de difficulté. Elle permet d'appliquer la règle [Pré] en justifiant que la formule d'implication est valide car elle est de la forme $p \wedge p' \Rightarrow p$. L'application de cette règle aboutit à la formule but [53] suivante :

[53] $\{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i} \wedge i<nt2\}$

i := i+1; x := y*100+t[nt2-i+1] ; calcul_c(r, x, c) ; y := x-(r*20+c)*c ; r := r*10+c

$\{n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i}\}$.

Notons que la formule [53] peut de simplifier ainsi abstrayant la formule `I` :

$\{I \wedge i<nt2\}$ i := i+1; x := y*100+t[nt2-i+1] ; calcul_c(r, x, c) ; y := x-(r*20+c)*c ; r := r*10+c $\{I\}$

6.3. Preuve que la séquence des deux phases est correcte

Le pas d'itération préservant l'invariant, l'invariant étant établit au point de contrôle qui précède l'itération, ces deux phases se composent en séquence et l'itération, quand elle termine, termine dans l'état défini par l'expression suivante :

[54] $n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i} \wedge i \geq nt2$.

6.4. Preuve de la post condition $r^2+y=n$

Le programme augmentant `i` de un à chaque pas, il termine avec `i=nt2` et jamais avec `i>nt2`. Il n'y pas de difficulté à montrer, en simplifiant le membre gauche de l'expression ci-dessous, que : $n=r^2+y$ dans le cas où `i=nt2`, c'est-à-dire où `nt2-i=0`. D'après la post condition (voir formule [54]) de l'itération et la remarque ci-dessus, la formule à démontrer est la suivante :

$$n=(r*10^{nt2-i})^2+\sum_{k=1}^{nt2-i} t[k]*100^{k-1}+y*100^{nt2-i} \wedge i=nt2 \Rightarrow n=r^2+y.$$

En remplaçant `nt2-i` par zéro, cette formule est équivalente à :

$n=(r*10^0)^2+\sum_{k=1}^{0} t[k]*100^{k-1}+y*100^0 \wedge vrai \Rightarrow n= r^2+y$. Cette formule est vraie car elle se réduit à : $n=r^2+y \Rightarrow n=r^2+y$.

Dans le cas où `i>nt2`, on sait par construction de le programme que cette condition est fausse quand le programme termine l'itération, donc la propriété n'est pas à considérer dans ce cas ce qui revient à dire qu'elle est vraie par définition de l'implication car `faux` $\Rightarrow$ `p` est vrai.

Notons que la démonstration précédente s'effectue comme si le calcul de `c` n'existait pas puisque la celui-ci ne modifie pas la propriété invariante. Par contre les propriétés du calcul de `c` servent à démontrer la deuxième partie de la spécification : `y<2r+1`.

6.5. Preuve que `y<2r+1` est invariante

Cette propriété assure que c'est bien le carré le plus proche par valeur inférieure qui est calculé et désigné par la variable `r`. On montre cette propriété en deux étapes :

. Preuve que l'assertion est vraie avant l'itération,
. Preuve qu'elle est conservée par l'itération.

6.5.1. Preuve que `y<2r+1` est vraie avant l'itération

On prouve la formule [63] qui établit que la séquence des quatre instructions d'initialisation conduit à un état vérifiant la condition `y<2r+1`. Dans la présentation ci-dessous, nous admettons que le prédicat `0=0` est le prédicat `vrai` et nous n'appliquons pas la règle *Pré*, ce que nous devrions faire en toute rigueur.

[55] `{vrai}` t2(n, nt2, t) `{vrai}` -- Aff
[56] `{0=0}` y:=0 `{y=0}` -- Aff
[57] `{vrai}` t2(n, nt2, t); y:=0 `{y=0}` -- Seq(55, 56)
[58] `{0=0 ∧ y=0}` r:=0 `{r=0 ∧ y=0}` -- Aff
[59] `{vrai}` t2(n, nt2, t); y:=0 ; r:=0 `{r=0 ∧ y=0}` -- Seq(57, 58)
[60] `{0=0 ∧ r=0 ∧ y=0}` i:=0 `{i=0 ∧ r=0 ∧ y=0}` - Aff
[61] `{vrai}` t2(n, nt2, t) ; y:=0 ; r:=0 ; i:=0 `{i=0 ∧ r=0 ∧ y=0}` -- Seq(59, 60)
[62] `i=0 ∧ r=0 ∧ y=0 ⇒ y<2r+1` -- voir justification
[63] `{vrai}` t2(n, nt2, t) ; y:=0 ; r:=0 ; i:=0 `{y<2r+1}` -- Post(61, 62)

Justification de la formule [62] : Dans le cas où la condition `i=0 ∧ r=0 ∧ y=0` est fausse, la formule est vraie par définition de l'implication. Dans le cas où cette condition est vraie, l'expression `y<2r+1` est équivalente à `0<2*0+1`. Cette expression est vraie dans les entiers naturels.

6.5.2. Preuve que l'assertion `y<2r+1` est invariante dans l'itération

Cette preuve s'effectue en deux étapes :

. Prouver que l'assertion `suivante r≥0 ∧ x≥0 ∧ 200r+100>x` est vraie avant l'appel de *calcul_c* si l'assertion `y<2r+1` est vraie en début d'itération,
. Prouver que les deux dernières actions conduisent à l'invariant `y<2r+1`.

1. Preuve de la formule suivante : `{y<2r+1}` i:=i+1 ; x:=y*100+t[nt2-i+1] `{x<200r+100}`

[70] `y<2r+1 ⇒ y≤2r` -- pptés de < et ≤
[71] `{y<2r+1}` i:=i+1 `{y<2r+1}` -- Aff
[72] `{y<2r+1}` i:=i+1 `{y≤2r}` -- Post(71, 70)
[73] `{y≤2r}` x:=y*100+t[nt2-i+1] `{x/100-t[nt2-i+1]/100≤2r}` -- Aff
[74] `x/100-t[nt2-i+1]/100≤2r ⇒ x≤200r+t[nt2-i+1]` -- pptés sur les entiers
[75] `{y≤2r}` x:=y*100+t[nt2-i+1] `{x≤200r+t[nt2-i+1]}` -- Post(73, 74)
[76] `x≤200r+t[nt2-i+1] ⇒ x<200r+100` -- car t[nt2-i+1]<100 par déf. de t
[77] `{y≤2r}` x:=y*100+t[nt2-i+1] `{x<200r+100}` -- Post(75, 76)
[78] `{y<2r+1}` i:=i+1; x:=y*100+t[nt2-i+1] `{x<200r+100}` -- Seq(72, 77)

Notons que la preuve de l'assertion $r \geq 0 \wedge x \geq 0$ est évidente ; nous ne la démontrons pas.

2. Preuve de la formule suivante :

$\{r \geq 0 \wedge x \geq 0 \wedge x < 200r+100\}$ calcul_c(r, x, c) ; y:=x-(20*r+c)*c; r:=r*10+c $\{y < 2r+1\}$.

[79] $\{r \geq 0 \wedge x \geq 0 \wedge x < 200r+100\}$ calcul_c(r, x, c)
$\{r \geq 0 \wedge x \geq 0 \wedge ((20r+c)*c) \leq x \wedge x < (20r+c+1)*(c+1)\}$ -- par déf. de calcul_c

[80] $r \geq 0 \wedge x \geq 0 \wedge ((20r+c)*c) \leq x \wedge x < (20r+c+1)*(c+1) \Rightarrow$
$x < (20r+c+1)*(c+1)$ – formule de la forme $p' \wedge p \Rightarrow p$.

[81] $\{r \geq 0 \wedge x \geq 0 \wedge x < 200r+100\}$ calcul_c(r, x, c) $\{x < (20r+c+1)*(c+1)\}$--Post (79, 80)

[82] $\{x < (20r+c+1)*(c+1)\}$ y:=x-(20*r+c)*c $\{y+(20r+c)*c < (20r+c+1)*(c+1)\}$-- Aff

[83] $y+(20r+c)*c < (20r+c+1)*(c+1) \Rightarrow y < 20r+2c+1$ -- par simplification

[84] $\{x < (20r+c+1)*(c+1)\}$ y:=x-(20*r+c)*c $\{y < 20r+2c+1\}$ -- Post(82, 83)

[85] $\{y < 20r+2c+1\}$ r:=r*10+c $\{y < 20(r/10-c/10)+2c+1\}$ -- Aff

[86] $y < 20(r/10-c/10)+2c+1 \Rightarrow y < 2r+1$ -- 2 membres de l'implication équivalents

[87] $\{y < 20r+2c+1\}$ r:=r*10+c $\{y < 2r+1\}$ -- Post(85, 86)

[88] $\{r \geq 0 \wedge x \geq 0 \wedge x < 200r+100\}$ calcul_c(r, x, c); y:=x-(20*r+c)*c
$\{y < 20r+2c+1\}$ --Seq(81, 84)

[89] $\{r \geq 0 \wedge x \geq 0 \wedge x < 200r+100\}$ calcul_c(r, x, c); y:=x-(20*r+c)*c;
r:=r*10+c $\{y < 2r+1\}$ -- Seq(88, 87)

[90] $\{y < 2r+1\}$ i:=i+1; x:=y*100+t[nt2-i+1] ; calcul_c(r, x, c); y:=x-(20*r+c)*c;
r:=r*10+c $\{y < 2r+1\}$ -- Seq(78, 89)

Ayant démontré les deux sous buts suivants [91] $\{n \geq 0\}$ A $\{n = r^2+y\}$ et [92] $\{n \geq 0\}$ A $\{y < 2r+1\}$ où A est la totalité de la suite d'instructions entre *début* et *fin* dans la Figure 53. Nous démontrons le but final par la règle *Et* :

[93] $\{n \geq 0\}$ A $\{n = r^2+y \wedge y < 2r+1\}$ -- Et(91, 92).

Notons que la formule [91] est prouvée à partir des formules [63] et [90] par application des règles *Tantque* et *Seq*. Il en est de même pour la formule [92].

Références bibliographiques

[Abrial 96] ABRIAL J.R., The B Book. Cambridge University Press, - ISBN 0521-496195, 1996.

[Apt 81] APT K. R. - Ten years of Hoare's logic : a survey - part 1, ACM Trans. on programming languages & systems, vol 3, p. 431-483, 1981.

[Arsac 77] ARSAC J. – La construction des programmes structurés – Dunod, 1977.

[Berlioux 83] BERLIOUX P., BIZARD P. - Construction, preuve et évaluation de programmes – Dunod Informatique, 1983.

[Boussard 83] BOUSSARD J.-C. et MAHL R. – Programmation avancée – algorithmique et structures de données – Eyrolles, 1983.

[Burdy 05] BURDY L., CHEON Y., COK D., ERNST M., KINIRY J., LEAVENS G.T., RUSTAN K., LEINO M., and POLL E.. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer, vol. 7, Numéro 3, sections 212-232, June 2005. (The original publication is available at http://www.springerlink.com).

[Clarke 01] CLARKE E.M., GRUMBERG O. et PELED D.A., «Model Checking», MIT Press, Cambridge, Massachusetts, 2001.

[Dehornoy 00] DEHORNOY P. – Mathématiques de l'Informatique, cours et exercices corrigés, Dunod, 2000

[Dijkstra 76] DIJKSTRA E.W. - A discipline of programming - Prentice hall, 1976.

[Grégoire 86] GREGOIRE - Informatique, programmation, tome 1 – Masson, 1986.

[Hoare 69] HOARE C.A.R. - An axiomatic basis for computer programming. C.ACM vol 12, 10, p. 576-580, 1969.

[Hofstadter 93] HOFSTADTER D. – Gödel, Escher et Bach, les brins d'une guirlande éternelle – Inter éditions, 1993.

[Julliand 10] JULLIAND J. - Cours et exercices corrigés d'algorithmique, Vérifier, tester et concevoir des programmes en les modélisant, Vuibert, février 2010.

[Leavens 00] LEAVENS G.T. RUSTAN K., LEINO M., POL E., RUBY C., and JACOBS B. JML :notations and tools supporting detailed design in Java. In OOPSLA'00 Companion, Minneapolis, Minnesota, sections 105-106. Copyright ACM, 2000. Also Department of Computer Science, Iowa State University, TR #00-15, August 2000.

[Livercy 78] LIVERCY - Théorie des programmes - Dunod informatique, 1978.

[Mitchell 02] MITCHELL R., McKim J. - *Design by Contract: by example*, Addison-Wesley, 2002.

[Myers 79] MYERS G. J. - The Art of Software Testing. John Wiley and Sons. ISBN 0-471-04328-1, 1979.

[Schneider 01] SCHNEIDER S. – The B method – Palgrave, 2001.

[Utting 06] UTTING M., LEGEARD B.– Practical Model-Based Testing, a tool approach, Morgan Kaufmann, 2006.

[Wilf] WILF H. - Programme et complexité – Masson, ???.

[Wordworth 96] WORDWORTH J. B. - Software engineering with B – Addison Wesley, 1996.

Glossaire

CC : Condition Coverage – Critère de couverture de tests. Toutes les conditions (prédicats élémentaires des conditionnelles et des itérations) doivent être vraies et fausses.

DC : Decision Coverage – Critère de couverture de tests. Toutes les décisions (prédicats des conditionnelles et des itérations) doivent être vraies et fausses.

C/DC : Condition Decision Coverage – Critère de couverture de tests. Toutes les décisions et toutes les conditions doivent être couvertes.

IUT : Implémentation Sous Test

IHM : Interface Homme Machine

IIM : Implémentation Issue du Modèle

IST : Implémentation Sous Test

JML : Java Modeling Language

LH : Logique de Hoare

LP1 : Logique des Prédicats du 1er ordre

MBT : Model-Based Testing

PLTL : Propositional Linear Temporal Logic (Logique temporelle propositionnelle linéaire)

RAC : Runtime Assertion Checker

SC : Statement Coverage Critère de couverture de tests. Toutes les instructions du modèle doivent être exécutées par l'ensemble des tests.

SED : Système à Evénements Discrets

SK : Structure de Kripke

SKE : Structure de Kripke Etiquetée

ST : Système de Transition

STE : Système de Transition Etiquetée

SUT : Système Sous Test

UML : Unified Modeling Language

Index