

Rapport de projet à l'université d'Angers

Du 4 Avril au 4 Juin

Preuve de programme en logique de Hoare

Projet effectué et présenté par :

Pierre GRANIER--RICHARD

Thibaut ROPERCH

Projet encadré par :

Igor STEPHAN



Les auteurs du présent document vous autorisent à le partager, reproduire, distribuer et communiquer selon les conditions suivantes :

- Vous devez le citer en l'attribuant de la manière indiquée par l'auteur (mais pas d'une manière qui suggérerait qu'il approuve votre utilisation de l'œuvre).
- Vous n'avez pas le droit d'utiliser ce document à des fins commerciales.
- Vous n'avez pas le droit de le modifier, de le transformer ou de l'adapter.

Consulter la licence creative commons complète en français :

<http://creativecommons.org/licences/by-nc-nd/2.0/fr/>

Sommaire

[Sommaire](#)

[Remerciements](#)

[Introduction](#)

[Propriétés d'un programme](#)

[Logique de Hoare](#)

[Définition](#)

[Triplet de Hoare](#)

[Règles de Hoare](#)

[Résumé](#)

[Construction de preuve](#)

[Méthodologie](#)

[Finalité](#)

[Utilisation des règles](#)

[Exemple](#)

[Remarque](#)

[Vérification de preuve](#)

[Méthodologie](#)

[Finalité](#)

[Remarque](#)

[Travail effectué](#)

[Format de preuve](#)

[Exemple de preuve au format choisi](#)

[Outils d'analyse de preuve](#)

[Fonctionnement d'un analyseur lexical et syntaxique](#)

[Choix de l'analyseur syntaxique](#)

[Grammaire de l'analyse lexicale et syntaxique](#)

[Mise en place de l'analyse sémantique](#)

[Limites de notre vérificateur](#)

[Objectifs de notre interface graphique](#)

[Structure de notre interface graphique](#)

[Fonctionnalités de notre interface graphique](#)

[Jeu de tests](#)

[Planning de travail](#)

[Idées d'améliorations](#)

[Conclusion](#)

[Bibliographie](#)

[Engagement de non plagiat](#)

Remerciements

Nous tenons à remercier tout particulièrement M. Igor Stephan, enseignant chercheur à l'université d'Angers et encadrant de notre projet, pour sa disponibilité ainsi que son accompagnement dans notre étude.

Nous remercions aussi toutes les personnes qui ont contribué à l'aboutissement de notre stage et qui nous ont aidé lors la rédaction de notre rapport.

Introduction

Notre projet de deux mois à l'université d'Angers consiste à étudier la prouvabilité de la correction d'un programme en utilisant la logique de Hoare. Notre objectif est de concevoir un outil capable de vérifier une preuve construite avec la méthode de Hoare.

Propriétés d'un programme

De manière générale, un programme doit respecter les critères suivants :

- Terminaison Le programme s'arrête à un moment
- Correction Le programme donne la bonne réponse (implique la terminaison)
- Complexité Le temps de réponse est acceptable (implique la terminaison)

Chaque propriété peut être démontrée avec un raisonnement adéquat. En l'occurrence, nous devons démontrer la correction d'un programme en utilisant la logique de Hoare.

Logique de Hoare

Définition

La logique de Hoare est une méthode de preuve basée sur l'analyse de la syntaxe d'un programme dans le but de raisonner sur sa correction. Elle permet d'exprimer l'évolution d'un système lors de l'exécution d'un programme.

Afin d'étudier le code d'un programme, la méthode de Hoare donne des outils permettant d'analyser chacune de ses instructions. On appelle cet ensemble d'outils un formalisme logique, en l'occurrence fondé sur la syntaxe d'un programme.

Ce formalisme est composé d'un axiome pour l'instruction de base d'un langage de programmation (l'affectation) et d'un ensemble de règles d'inférence pour les blocs d'instructions d'un langage (condition et itération).

Cette méthode de preuve fonctionne avec des triplets de Hoare. Un triplet donne l'état du système avant et après l'exécution du programme.

Triplet de Hoare

Un triplet de Hoare $\{P\} \text{ prog } \{Q\}$ est composé du programme *prog*, de la pré-condition *P* et de la post-condition *Q*. Les termes *P* et *Q* sont des formules logiques représentant des propriétés sur les variables du programme.

Le prédicat *P* statue sur l'état initial du programme *prog*, avant son exécution, tandis que le prédicat *Q* statue l'état final du programme *prog*, après son exécution.

Un triplet est dit valide si, pour tout état initial vérifiant *P*, la sémantique de *prog* produit un état final de la mémoire dans lequel *Q* est vérifié. Autrement dit, si la pré-condition est vraie avant l'exécution du programme, ce dernier termine et la post-condition est vraie.

Nous pouvons interpr ter la validit  d'un triplet comme une formule logique en utilisant le connecteur de l'implication.

Afin de d montrer la validit  syntaxique d'un triplet, nous devons utiliser les r gles de Hoare.

P	Q	$\{P\} \text{ prog } \{Q\} : P \Rightarrow Q$
vrai	vrai	vrai
vrai	faux	faux
faux	vrai	vrai
faux	faux	vrai

R gles de Hoare

Une r gle de Hoare est g n ralement de la forme suivante :

$\text{triplet}_1 \dots \text{triplet}_n$	Pr�misse
triplet	Conclusion

Nous distinguons 0   n triplets en pr misse de la r gle et un unique triplet en conclusion.

Nous pouvons lire une r gle de la mani re suivante :

“Si les triplets pr misses sont vrais, alors le triplet conclusion aussi.”

Il existe une r gle de Hoare pour chaque instruction du langage, sauf pour la r gle de la cons quence qui est d finie ind pendamment.

Les r gles les plus utilis es en logique de Hoare sont les suivantes :

- L'affectation est l'instruction $x := E$, associant   la variable x la valeur de l'expression E .

$$\text{AFF} \quad \frac{}{\{P[E/x]\} x := E \{P\}}$$

- La s quence s'applique pour les programmes S et T s'ils sont ex cut s s quentiellement, o  S s'ex cute avant T . Le programme issu de cette ex cution est not  $S;T$

$$\text{SEQ} \quad \frac{\{P\} S \{Q\}, \{Q\} T \{R\}}{\{P\} S;T \{R\}}$$

- La cons quence signifie que pour  tablir le triplet $\{P\} S \{Q\}$, il suffit d' tablir   la place le triplet $\{P'\} S \{Q'\}$ o  P' est une cons quence de P et Q est une cons quence de Q' .

$$\text{CONSEQ} \quad \frac{P \rightarrow P' , \{P'\} S \{Q'\} , Q' \rightarrow Q}{\{P\} S \{Q\}}$$

- La règle de la conditionnelle s'applique aux instructions *Si... Alors... Sinon*.

$$\text{COND} \quad \frac{\{B \wedge P\} S \{Q\} , \{\neg B \wedge P\} T \{Q\}}{\{P\} \text{ si } B \text{ alors } S \text{ sinon } T \text{ fin si } \{Q\}}$$

- La règle de l'itération assure que la post-condition est vérifiée une fois la boucle terminée (si elle termine).

$$\text{WHILE} \quad \frac{\{I \wedge B\} S \{I\}}{\{I\} \text{ tant que } B \text{ faire } S \text{ fait } \{\neg B \wedge I\}}$$

- La règle de l'itération avec terminaison assure la terminaison de la boucle avec une variable appelée variant de boucle, ici notée V .

$$\text{WHILET} \quad \frac{\{I \wedge B \wedge V = z\} S \{I \wedge V < z\}}{\{I\} \text{ tant que } B \text{ faire } S \text{ fait } \{\neg B \wedge I\}}$$

La règle de l'itération avec terminaison introduit la notion de correction partielle et de correction totale. Dans un triplet $\{P\} \text{ prog } \{Q\}$, la correction partielle impose la validité de la post-condition Q pour tout état atteint lorsque le programme C termine. La correction totale implique en plus la terminaison du programme.

Résumé

Grâce à ses **règles**, la logique de Hoare permet de réécrire un programme afin de vérifier qu'il est correct **syntactiquement**. Les **triplets** de Hoare permettent de spécifier les états du système à chacune des instructions d'un programme, notamment la valeur des variables. Ils servent à prouver que le programme est correct **sémantiquement**. La méthode de Hoare offre l'avantage de prouver formellement des programmes sans avoir à les exécuter.

Si les états du système sont vérifiés et si la preuve est correctement construite avec les règles données, alors la correction du programme est prouvée.

Nous avons choisi de nous limiter aux principales règles de Hoare car nous avons préféré passer du temps à bien comprendre les bases de cette logique. De plus, nous nous sommes donné pour objectif de construire un assistant graphique de construction de preuve. C'est pourquoi les preuves de tableaux ne seront pas traitées ici.

Construction de preuve

M thodologie

La construction de la preuve d'un programme commence par l' criture de ce dernier sous la forme d'un triplet, d sign  comme  tant la racine de la preuve.

Nous appliquons   ce triplet une premi re r gle de Hoare afin de faire appara tre une pr missse, compos e d'un ou deux triplet(s) en fonction de la r gle appliqu e.

Chaque triplet obtenu nous permettra de d duire une nouvelle pr missse apr s l'application d'une r gle de Hoare, et ainsi de suite.

La construction d'une preuve est appel e arbre de d rivation (ou arbre de preuve) car l'application d'une r gle peut donner naissance   deux nouvelles sous-preuves en divisant la branche initiale.

La construction d'un arbre de preuve se fait de bas en haut (d'une conclusion nous voulons g n rer une pr missse) en appliquant une r gle   la fois.

Finalit 

L'arbre de preuve est termin  lorsque toutes les branches sont closes, la seule r gle permettant de cl turer une branche  tant l'axiome de l'affectation.

L'objectif de la construction d'une preuve est de d tailler toutes les instructions du programme jusqu'  l'affectation. L'arbre ainsi construit permettra de d terminer si le programme est correct ou non.

Utilisation des r gles

Dans la plupart des cas, les r gles seront appliqu es dans l'ordre des instructions que composent le programme du triplet.

Nous utiliserons la r gle de la cons quence lorsqu'aucune autre r gle ne pourra  tre appliqu e, par exemple entre une r gle de condition et une r gle d'affectation. Cette r gle est indispensable pour ajuster les triplets de Hoare   la sp cification syntaxique des autres r gles. Elle sert g n ralement de transition entre deux r gles.

Exemple

 tudions un programme simple permettant d'exploiter les r gles de Hoare :

1	// Pr�-condition : aucune
2	a := 1;
3	b := 2;
4	// Post-conditions : a = 1 et b = 2

Pour faire la preuve de ce programme, nous devons g n rer le triplet qui servira de racine   notre arbre de d rivation.

L' tat initial du syst me  tant inconnu, nous le consid rons comme valide.

Notre programme est compos  de deux instructions :

- La valeur 1 est affect e   la variable a
- La valeur 2 est affect e   la variable b

A la fin de l'ex cution de notre programme, la valeur des variables a et b doit  tre respectivement 1 et 2.

La racine de l'arbre de preuve est donc :

$$\{\text{vrai}\} \ a:=1; b:=2 \ \{a=1^{\wedge}b=2\}$$

Naturellement, nous voulons appliquer en premier la r gle de la s quence.  tant donn  qu'il n'y a que deux instructions dans le programme de la racine, nous pourrions envisager de clore chacune des deux branches g n r es par la r gle SEQ avec l'axiome de l'affectation.

Malheureusement, le triplet $\{\text{vrai}\} \ a:=1 \ \{a=1^{\wedge}2=2\}$ ne peut  tre clos directement avec l'axiome de l'affectation. Nous devons donc appliquer la seule r gle permettant de faire la transition,   savoir la r gle de la cons quence (en l'occurrence   gauche) :

$$\text{vrai} \Rightarrow 1=1^{\wedge}2=2 \qquad \{1=1^{\wedge}2=2\} \ a:=1 \ \{a=1^{\wedge}2=2\}$$

Nous obtenons l'arbre de preuve suivant :

$$\begin{array}{c}
 \text{AFF} \text{ ---} \\
 \{1=1^{\wedge}2=2\} \ a:=1 \\
 \{a=1^{\wedge}2=2\} \\
 \text{CONSEQ} \text{ ---} \qquad \text{AFF} \text{ ---} \\
 \{ \text{vrai} \} \ a:=1 \ \{a=1^{\wedge}2=2\} \qquad \{a=1^{\wedge}2=2\} \ b:=2 \ \{a=1^{\wedge}b=2\} \\
 \text{SEQ} \text{ ---} \\
 \{ \text{vrai} \} \ a:=1; b:=2 \ \{a=1^{\wedge}b=2\}
 \end{array}$$

Nous avons clos toutes les branches de l'arbre, la preuve est donc termin e. La syntaxe des r gles de Hoare ont  t  respect es, et les triplets de Hoare sont valides s mantiquement.

Nous avons donc r ussi   prouver que le programme est correct en utilisant la logique de Hoare.

Remarque

Nous noterons une différence de facilité d'utilisation entre les règles de Hoare : les règles telles que la conditionnelle, l'itération et l'affectation sont simples à utiliser étant donné qu'elles ne reposent que sur des mécanismes de manipulations purement syntaxiques.

Les règles de la séquence et de la conséquence quant à elles demandent de connaître les prédicats à générer, ce qui peut ajouter de la difficulté lors de la construction d'une preuve.

Vérification de preuve

Méthodologie

La vérification d'une preuve a pour rôle d'analyser syntaxiquement les règles et sémantiquement les triplets d'une preuve construite.

Analyser syntaxiquement les règles d'une preuve consiste à vérifier que les spécifications des règles de Hoare utilisées sont respectées.

Analyser sémantiquement les triplets d'une preuve consiste à vérifier que chaque triplet est valide.

Finalité

A l'issue de la vérification, nous pourrions déterminer si la preuve est conforme au formalisme de Hoare et ainsi raisonner sur la correction du programme analysé.

Remarque

La règle de la conséquence peut s'avérer complexe à analyser automatiquement car l'implication de deux formules logiques est sémantiquement difficile à vérifier.

Travail effectué

Notre travail consiste à créer un programme capable de faire automatiquement la vérification d'une preuve.

Après avoir étudié les mécanismes de la logique de Hoare, nous avons réfléchi à la façon dont un programme pourrait vérifier une preuve.

Format de preuve

Dans un premier temps, nous avons imaginé plusieurs formats que pourrait adopter une preuve contenue dans un fichier texte. Nous nous sommes particulièrement intéressés aux deux formats suivants :

- Une présentation semblable à une preuve construite à la main, c'est-à-dire sous forme d'arbre. Nous devons ainsi gérer le fait que plusieurs branches différentes

pouvaient se trouver sur une m me ligne de texte, ce qui impliquait de parcourir le fichier de haut en bas puis de gauche   droite.

```

1  AFF
2  {1=1^2=2} a:=1 {a=1^2=2}
3  CONSEQ AFF
4  {vrai} a:=1 {a=1^2=2} {a=1^2=2} b:=2 {a=1^b=2}
5  SEQ
6  {vrai} a:=1;b:=2 {a=1^b=2}
```

Exemple de fichier texte contenant une preuve arborescente

- Une description lin aire de la preuve ressemblant   une suite de fonctions imbriqu es, avec pour nom de fonctions les noms des r gles de Hoare et pour arguments un triplet et une fonction. La difficult  ici  tait de traduire une preuve sous forme d'arbre en une preuve lin aire.

```

1  SEQ {vrai} a:=1;b:=2 {a=1^b=2}
   CONSEQ {vrai} a:=1 {a=1^2=2} AFF {1=1^2=2} a:=1
   {a=1^2=2}
   AFF {a=1^2=2} b:=2 {a=1^b=2}
```

Exemple de fichier texte contenant une preuve lin aire

M me si le deuxi me format de preuve peut sembler peu intuitif, il nous semblait plus agr able   exploiter que le premier : l'analyse syntaxique se ferait de fa on structur e et r cursive, et la preuve s'analyserait plus facilement que si elle  tait  crite dans le premier format (car la lecture se ferait uniquement de gauche   droite).

Nous avons donc fait le choix d'exploiter des fichiers de preuves  crites sur une seule ligne de texte.

Exemple de preuve au format choisi

Reprenons l'exemple de l'arbre preuve pr c dent :

```

AFF _____
      {1=1^2=2} a:=1
      {a=1^2=2}
CONSEQ _____  AFF _____
      {vrai} a:=1 {a=1^2=2}      {a=1^2=2} b:=2 {a=1^b=2}
SEQ _____
      {vrai} a:=1;b:=2 {a=1^b=2}
```

Nous voulons obtenir la représentation de cette preuve au format linéaire. Pour se faire, nous devons parcourir l'arbre de bas en haut (jusqu'à un axiome) puis de gauche à droite.

Nous lisons ainsi en premier la sous-preuve SEQ qui possède un triplet en conclusion et deux triplets en prémisses. Or, les triplets de la prémisse sont eux aussi des sous-preuves : le premier triplet est une sous-preuve CONSEQ et le second une sous-preuve AFF.

De même pour la prémisse de la sous-preuve CONSEQ : on remarque qu'elle est une sous-preuve AFF.

Indice	Règle	Conclusion	Prémisse	
0	SEQ	{vrai} a:=1;b:=2 {a=1^b=2}	1	2
1	CONSEQ	{vrai} a:=1 {a=1^2=2}	1'	
1'	AFF	{1=1^2=2} a:=1 {a=1^2=2}	Aucune	
2	AFF	{a=1^2=2} b:=2 {a=1^b=2}	Aucune	

Ordre de lecture des sous-preuves de l'arbre de preuve

Nous remarquons que l'ordre de lecture de l'arbre se fait bien de bas en haut, puis de gauche à droite. Nous devons finir de lire une branche (par défaut en commençant toujours par celle de gauche s'il y a un choix à faire) avant d'en commencer une autre.

Nous obtenons la preuve suivante (écrite sur une seule ligne de texte), à donner à l'analyseur de preuve :

```

SEQ {vrai} a:=1;b:=2 {a=1^b=2}
CONSEQ {vrai} a:=1 {a=1^2=2}
AFF {1=1^2=2} a:=1 {a=1^2=2}
AFF {a=1^2=2} b:=2 {a=1^b=2}

```

Outils d'analyse de preuve

Dans un second temps, nous nous sommes intéressés à la façon dont notre programme analyserait une preuve. Nous savions qu'il procéderait à une analyse lexicale en manipulant des chaînes de caractères, et que certaines parties de la preuve contiendraient des formules logiques (avec des entiers et des opérateurs mathématiques).

Si nous avions voulu coder notre propre analyseur syntaxique, nous aurions dû impérativement implanter un analyseur sémantique.

Étant donné qu'il existe des outils de génération d'analyseurs syntaxiques, nous avons étudié les mécanismes de ces langages de programmation.

Fonctionnement d'un analyseur lexical et syntaxique

Un analyseur syntaxique (ou *parser*) met en évidence la structure d'un texte en fonction d'une grammaire formelle.

Cette grammaire sert de définition lors de l'analyse d'un texte : il sera découpé en fonction des lexèmes et règles de construction définis.

Notre travail consistait donc à repérer et spécifier les lexèmes (ou *tokens*) et les règles en fonction de ce que nous voulions vérifier afin de construire la grammaire formelle de l'analyseur.

Reprenons l'exemple de preuve linéaire précédent :

```
SEQ {vrai} a:=1;b:=2 {a=1^b=2}
CONSEQ {vrai} a:=1 {a=1^2=2}
AFF {1=1^2=2} a:=1 {a=1^2=2}
AFF {a=1^2=2} b:=2 {a=1^b=2}
```

L'objectif de notre travail est d'arriver à vérifier, grâce à l'analyseur, que la preuve est bien construite et conforme sémantiquement aux spécifications de la logique de Hoare.

Dans un premier temps, l'analyseur vérifie que la chaîne de caractères lue correspond à un lexème défini. Sinon, il renvoie une erreur de syntaxe. Nous avons donc à définir les différents tokens présents dans une preuve.

Dans notre exemple il faut définir un lexème par nom de règle de Hoare rencontré et par caractère (les accolades, le signe d'affectation, le "et" logique).

Ensuite, l'analyseur cherche une correspondance entre la suite de tokens lus et une règle définie dans la grammaire. Il est possible d'appeler récursivement une règle en la réutilisant dans sa définition.

Pour notre exemple, nous avons besoin de construire la règle *RègleDeHoare*, représentant la forme des règles de Hoare pouvant être rencontrées dans une preuve. Ses définitions commenceront par un token associé à un nom de règle de Hoare, suivi par zéro, un ou deux triplets. Nous devons lui donner plusieurs définitions car, en fonction de la règle de Hoare utilisée, il y a un certain nombre de triplets en prémisses. Nous aurons à définir la règle triplet, qui est composée de deux prédicats et un programme. Ces règles aussi seront à spécifier.

Ainsi, la définition de la règle *RègleDeHoare* sera de cette forme :

```
OU   SEQ Triplet RègleDeHoare RègleDeHoare
      CONSEQ Triplet RègleDeHoare
OU   AFF Triplet
```

A chaque définition de la règle, nous devons vérifier que les spécifications syntaxiques des règles de Hoare sont respectées (en utilisant le C++).

Par exemple, pour la définition commençant par le token `AFF`, nous devons implémenter la vérification suivante : la pré-condition du triplet doit correspondre à la post-condition dans laquelle la variable du programme a été remplacée par sa valeur.

Nous pouvons anticiper le type des règles à spécifier : par défaut les tokens et les règles sont des chaînes de caractères, la règle `RègleDeHoare` aura pour type une structure `triplet` composée de deux prédicats et d'un programme afin d'y accéder dans les définitions de la règle.

En résumé, il nous est indispensable de repérer et définir rigoureusement les lexèmes et les règles rencontrés dans une preuve construite avec la logique de Hoare. Cette tâche nous permettra de travailler plus efficacement sur l'analyse.

Choix de l'analyseur syntaxique

Nous avons opté pour les outils Flex et GNU Bison (équivalents de Lex et Yacc).

Flex est un outil de génération d'analyseurs lexicaux, et GNU Bison est un outil de génération d'analyseurs syntaxiques (en C ou C++). Ce sont les deux outils sur lesquels nous avons trouvé le plus de documentation.

Grammaire de l'analyse lexicale et syntaxique

Enfin, nous avons à implanté notre grammaire dans l'analyseur.

Nom	Valeur	Informations
<code>AFF</code> <code>SEQ</code> <code>COND</code> <code>CONSEQ</code> <code>WHILE</code> <code>WHILET</code>	<code>"AFF"</code> <code>"SEQ"</code> <code>"COND"</code> <code>"CONSEQ"</code> <code>"WHILE"</code> <code>"WHILET"</code>	Ces tokens ont pour valeur exactement le nom donné aux règles de Hoare.
<code>ACCOLADE_G</code> <code>ACCOLADE_D</code>	<code>"{"</code> <code>"}"</code>	Ces caractères sont utilisés en début et fin de prédicat de triplet.
<code>ET</code>	<code>"^"</code>	Ce caractère est le "et" logique. Il est utilisé dans un prédicat entre deux formules logiques.
<code>COMP</code>	<code>"<"</code> , <code>">"</code> , <code>"<="</code> , <code>">="</code>	Ce token représente l'ensemble des opérateurs de comparaison utilisés dans un prédicat.
<code>PT_VIRGULE</code>	<code>";"</code>	Ce caractère est utilisé pour séparer deux instructions d'un programme.

AFFECTATION	“:=”	Ces deux caractères représentent l'affectation dans le langage de programmation que nous utilisons et sont situés entre une variable et une valeur.
MOT	[a-zA-Z]+	Un mot est un ensemble de lettres (minuscules et/ou majuscules). Dans un programme, une variable est un mot.
ENTIER	[0-9]+	Un entier est un ensemble de chiffres.
OPERATEUR	“+”, “-”, “*”, “/”	Ce token représente l'ensemble des opérateurs utilisés dans une expression.

Principaux lexèmes (ou tokens) de la grammaire de l'analyseur lexical

Définition	Informations
<u>Regle :</u> AFF Triplet SEQ Triplet Regle COND Triplet Regle CONSEQ Triplet Regle WHILE Triplet Regle WHILET Triplet Regle	Cette règle est composée d'un token correspondant au nom d'une règle de Hoare, d'un Triplet et d'une Règle. Note : le token AFF n'est suivi que par un Triplet car la règle AFF est un axiome.
<u>Triplet :</u> Predicat Programme Predicat	Un Triplet est défini par un Predicat suivi d'un Programme et d'un autre Predicat.
<u>Predicat :</u> ACCOLADE_G Conditions ACCOLADE_D	Un Predicat est composé de la règle Conditions entre les tokens associés aux accolades.
<u>Conditions :</u> Condition ET Conditions Condition	La règle Conditions est constituée d'un ensemble de règles Condition séparées par le token associé au “et” logique.
<u>Condition :</u> Expression COMP Condition Expression	Une Condition est composée d'une Expression, d'un token associé à un symbole de comparaison et d'une autre Expression.
<u>Programme :</u> Instruction PT_VIRGULE Programme Instruction	Un Programme est un ensemble de règles Instruction séparées par le token associé au point virgule.
<u>Instruction :</u> Mot AFFECTATION Expression	Une Instruction est définie par un Mot suivi d'un token représentant le symbole d'affectation puis par une Expression.
<u>Expression :</u> ExpressionMot ExpressionEntier	Une Expression est une ExpressionMot ou une ExpressionEntier.

<u>ExpressionEntier</u> : ENTIER OPERATEUR ExpressionEntier ENTIER	Une ExpressionEntier est composée d'une suite d'opérations entre des Entier.
<u>ExpressionMot</u> : MOT OPERATEUR Expression ENTIER OPERATEUR ExpressionMot MOT	Une ExpressionMot est composée d'une suite d'opérations entre des Entier et/ou des Mot.

Principales règles syntaxiques de la grammaire de l'analyseur syntaxique

Mise en place de l'analyse sémantique

Afin de vérifier qu'un prédicat est sémantiquement vrai ou faux, il faut que les formules logiques qui le composent soient calculables (qu'elles aient une valeur booléenne). Or, jusqu'ici, elles sont représentées seulement par des chaînes de caractères car nous vérifions que les prédicats sont syntaxiquement corrects selon la méthode de Hoare.

Pour préserver les deux formats, nous avons créé une structure (en C) comportant deux attributs : la valeur sous forme de chaîne de caractères et la valeur sous forme de booléen.

Ainsi, en fonction de ce que nous voulons vérifier, nous pouvons facilement accéder au format du prédicat qui nous intéresse.

De la même façon, nous avons créé une structure pour les programmes composés (Si... Alors... Sinon) ainsi que pour les expressions contenant des entiers et des opérateurs.

Limites de notre vérificateur

Comme nous l'avions anticipé, la vérification de la règle de la conséquence s'est avérée difficile à implémenter.

$$\text{CONSEQ} \quad \frac{P \rightarrow P' , \{P'\} S \{Q'\} , Q' \rightarrow Q}{\{P\} S \{Q\}}$$

En effet, prouver l'implication de deux formules est difficile dans certains cas.

Prenons l'exemple de preuve suivant :

$$\text{CONSEQ} \quad \frac{\{\text{faux}\} x:=0 \{\text{faux}\}}{\{a>0 \wedge a<0\} x:=0 \{\text{faux}\}}$$

Nous savons que $a > 0 \wedge a < 0 \rightarrow \text{faux}$, mais l'analyseur sémantique ne pourra pas le vérifier. En l'état, il calculera indépendamment les termes de la formule logique et en déduira que le prédicat est vrai. Il faut donc spécifier dans sa grammaire le cas particulier suivant :

“Si un prédicat contient à la fois une formule et sa négation, alors il est faux.”

C'est un cas parmi d'autres. Afin de simplifier le travail de l'analyseur sémantique, nous avons fait le choix d'implémenter seulement la règle suivante (dans la règle de la séquence) :

“Si la valeur booléenne des termes de l'implication est différente, alors la preuve est fausse.”

Nous savons que cette règle n'est pas toujours juste, c'est pourquoi elle ne renvoie pas un message d'erreur mais un message d'avertissement. Dans cette situation, seul l'utilisateur pourra décider de la vérification de sa preuve.

Nous avons appliqué le même raisonnement concernant l'analyse sémantique de la validité d'un triplet :

“Si la pré-condition est vraie et la post-condition est fausse, alors la preuve est fausse.”

Nous nous sommes donc limités dans la vérification de la règle de la conséquence et de la validité d'un triplet à une comparaison logique des formules, en émettant une réserve à l'intention de l'utilisateur.

Objectifs de notre interface graphique

La création d'une interface graphique nous permettrait d'améliorer l'affichage assez brut du terminal, pour le rendre plus agréable et plus facile à comprendre. Aussi, nous aimerions proposer un assistant de construction de preuve facile d'utilisation.

Nous avons donc opté pour une représentation de la preuve en arbre, comme dans les cours que nous avons trouvés sur la logique de Hoare. L'affichage est alors nettement plus intuitif que lorsque nous écrivons la preuve de façon linéaire.

Nous avons par ailleurs supprimé quelques mots de certaines règles que nous ne trouvions pas indispensables. Ainsi, les combinaisons de mots SI... ALORS... SINON... et TANTQUE... FAIRE... FAIT, présents dans des preuves normales, provoqueront des erreurs de syntaxe si nous les écrivons dans l'interface.

De plus, nous avons supprimé l'écriture obligatoire des implications dans la règle de la conséquence. Nous pensons que devoir indiquer $P \Rightarrow P'$ avant le triplet prémisses n'est pas nécessaire puisque nous avons P en conclusion et P' en prémisses.

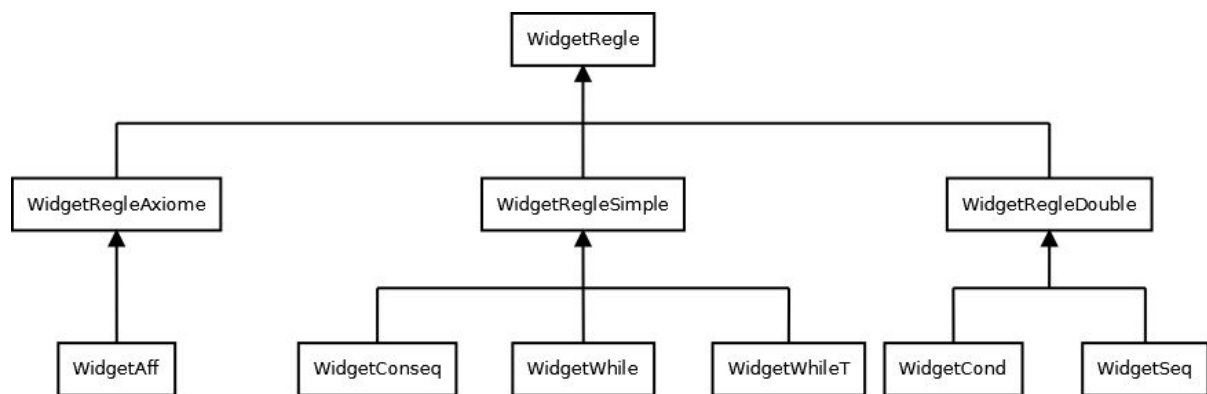
Structure de notre interface graphique

Nous avons choisi de créer l'interface en utilisant les composants graphiques proposés par la librairie Qt5, étudiée au cours de cette année universitaire.

Cette bibliothèque nous a permis de construire en premier lieu l'interface principale composée d'une fenêtre, d'un menu et de deux zones de travail : une zone de construction et une zone de vérification de preuve.

Nous avons ensuite développé les composants (appelés *widgets*) permettant de construire une preuve, à raison d'un widget par règle de Hoare. La librairie Qt5 étant orientée objet et la structure des règles de Hoare différentes, nous avons établi une hiérarchie de classes.

Le widget principal de notre application, `WidgetRègle`, hérite directement de `QWidget`, un composant de la bibliothèque Qt5.



Hiérarchie des composants de notre interface graphique

Graphiquement, un `WidgetRègleAxiome` permet de fermer la sous-preuve en affichant un champ de texte servant de prémisse à la règle instanciée.

Un `WidgetRègleSimple` affiche un champ de texte et la possibilité d'invoquer une autre règle servant de sous-preuve.

Un `WidgetRègleDouble` possède lui aussi un champ de texte et permet d'ajouter deux branches à l'arbre de preuve.




Une fois l'interface graphique terminée, nous l'avons mise en lien avec l'analyseur que nous avons développé durant les semaines précédentes.

Fonctionnalités de notre interface graphique

Même si le rôle principal de notre interface est de vérifier une preuve de programme, il nous semblait important de proposer à l'utilisateur une application agréable à utiliser, notamment lors de la construction arborescente de la preuve.

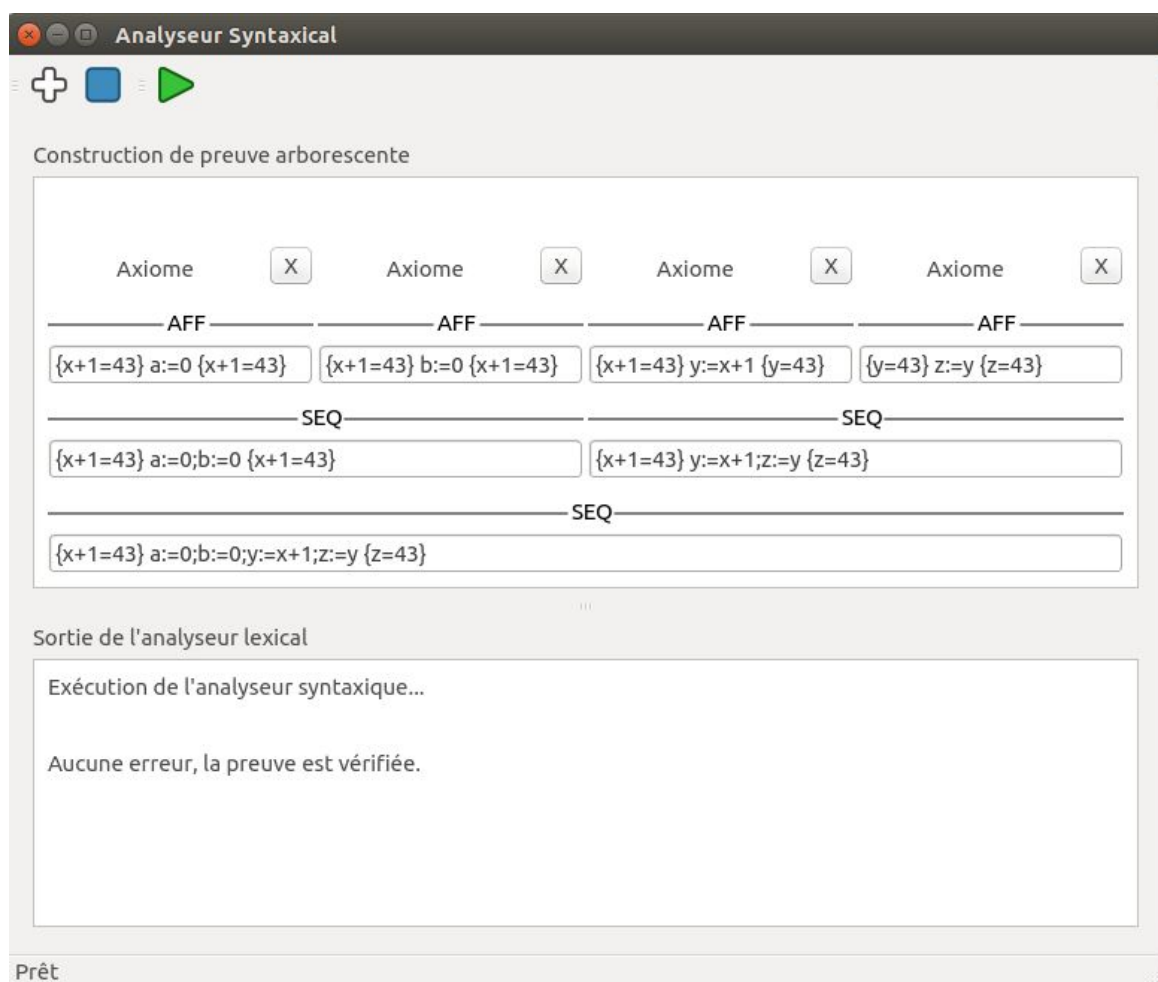
Ainsi, nous avons ajouté à côté du sélecteur de la règle une croix permettant de la supprimer en cas d'erreur.

Le menu principal de l'interface permet d'interagir avec les zones de construction de preuve et de sortie du vérificateur :

- Le bouton  permet de remettre à zéro la preuve entière.
- Le bouton  permet d'ouvrir la dernière preuve qui a été passée à l'analyseur.
- Le bouton  permet de vérifier la preuve. Si la preuve ne se termine pas par des axiomes, une popup nous en avertit.

Jeu de tests

Voici un exemple d'utilisation de l'interface graphique (preuve correcte) :



Un exemple incorrect de preuve :

Analyseur Syntaxical

Construction de preuve arborescente

Axiome X Axiome X Axiome X Axiome X

AFF (2) AFF AFF AFF

{x+1=43} a:=0 {x=43} {x+1=43} b:=0 {x+1=43} {x+1=43} y:=x+1 {y=43} {y=43} z:=y {z=43}

SEQ SEQ

{x+1=43} a:=0;b:=0 {x+1=43} {x+1=43} y:=x+1;z:=y {z=43}

SEQ (1)

{x+1=43} a:=0;b:=0;y:=x;z:=y {z=43}

Sortie de l'analyseur lexical

Exécution de l'analyseur syntaxique...

[ERREUR][SYNTAXIQUE] La précondition de AFF x+1=43 est différente de x=43 2

[ERREUR][SYNTAXIQUE] La postcondition de (1) x=43 est différente de la précondition de la prémisse (2) x+1=43

[ERREUR][SYNTAXIQUE] Les programmes de SEQ sont incorrects : a:=0;b:=0;y:=x;z:=y est différent de a:=0;b:=0;y:=x+1;z:=y 1

Une ou plusieurs erreurs ont été trouvée(s).

Prêt

Planning de travail

Semaine	Activités
1	Initiation à la logique de Hoare : <ul style="list-style-type: none"> - Appréhension du formalisme mis en place par la méthode de Hoare - Exercices de construction de preuve
2	Réflexion sur la vérification d'une preuve : <ul style="list-style-type: none"> - Quel format de preuve adopter ? - Quel(s) outil(s) utiliser ? Début de la construction d'un vérificateur de preuve "à la main" (donc sans analyseur sémantique)
3	Étude des analyseurs syntaxiques existants Prise en main des outils Lex et Yacc Création des premiers lexèmes et des premières règles de base pour notre projet
4	Étude plus poussée du sujet : <ul style="list-style-type: none"> - Approfondissement des règles de la logique de Hoare

	- R�flexion sur la validit� d'un triplet (correction partielle et correction totale)
5	Finalisation de l'analyseur syntaxique �tude des objectifs et des fonctionnalit�s de l'interface graphique
6	D�but de la cr�ation de l'interface graphique Mise en lien avec l'analyseur syntaxique
7	Ajout de fonctionnalit�s suppl�mentaires : - Ouvrir une preuve - Supprimer une r�gle Finalisation de l'interface graphique
8	Finalisation du rapport Pr�paration � la soutenance Construction de la pr�sentation orale

Pour des raisons de r flexion et de compr hension, nous avons d cid  de travailler la premi re partie ensemble (l'analyseur syntaxique). Pour la deuxi me partie de notre travail, la construction de l'interface graphique, nous nous sommes r parti les t ches.

Nous avons r fl chi ensemble   la conceptualisation de l'interface ainsi qu'  l'h ritage des classes que nous devions impl menter, puis nous avons commenc    coder en nous ayant partag  les classes    crire.

Afin de collaborer efficacement et rapidement, nous avons h berg  notre projet sur la plate-forme GitHub. Le partage de nos fichiers est simplifi  avec le logiciel Visual Studio Code.

Id es d'am liorations

Bien s r, il est possible d'am liorer notre application.

Par exemple, nous pourrions int grer dans l'analyseur la gestion de la r gle des tableaux.

Dans l'interface graphique, il serait envisageable d'impl menter une fonctionnalit  d'ajout de r gle en dessous de la racine.

Conclusion

Tout d'abord, grâce à notre interface nous pouvons proposer un assistant complet de construction de preuve suivant la méthode de Hoare.

La structure des composants graphiques que nous avons mis en oeuvre permet de convertir facilement l'arbre de preuve construit vers le format que nous avons choisi.

Le langage utilisé pour notre interface nous donne la possibilité de lier efficacement l'analyseur syntaxique que nous avons développé à notre application.

Ensuite, ce projet nous a apporté des connaissances personnelles supplémentaires dans des domaines étudiés durant notre troisième année de licence.

Nous nous sommes initiés à la logique de Hoare et nous avons eu l'occasion de découvrir les outils d'analyse de langage tels que Flex et GNU Bison.

Nous avons eu l'opportunité d'appliquer nos connaissances acquises en C++ plus tôt dans l'année, ainsi qu'en développement d'interfaces graphiques.

Enfin, cette expérience nous a appris à travailler ensemble et à nous partager les tâches d'un même projet.

Bibliographie

Outils de preuve et vérification de Pierre Courtieu (30 octobre 2008)

Algorithmique et Analyse d'Algorithmes - Logique de Hoare de Benjamin Wack (2015 - 2016)

Logique de Hoare sur Wikipedia

Lex & Yacc Tutorial de Tom Niemann

Engagement de non plagiat

Nous, soussignés GRANIER--RICHARD PIERRE et ROPERCH THIBAUT déclarons être pleinement conscients que le plagiat de documents ou d'une partie d'un document publiée sur toutes formes de support, y compris internet, constitue une violation des droits d'auteur ainsi qu'une fraude caractérisée. En conséquence, nous nous engageons à citer toutes les sources que nous avons utilisées pour écrire ce rapport.