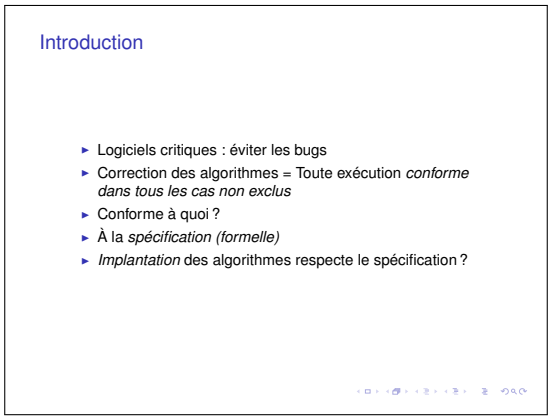
Outils de preuve et vérification

Pierre Courtieu

30 octobre 2008

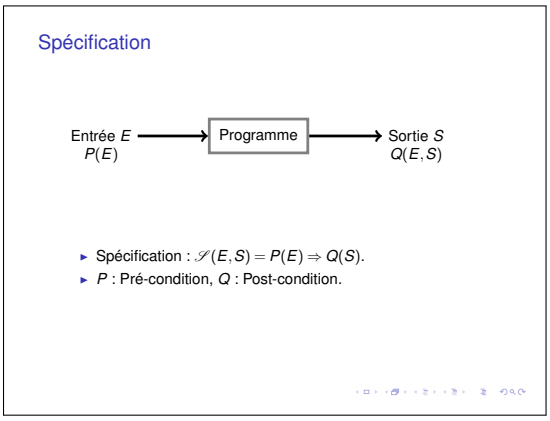
Table des matières

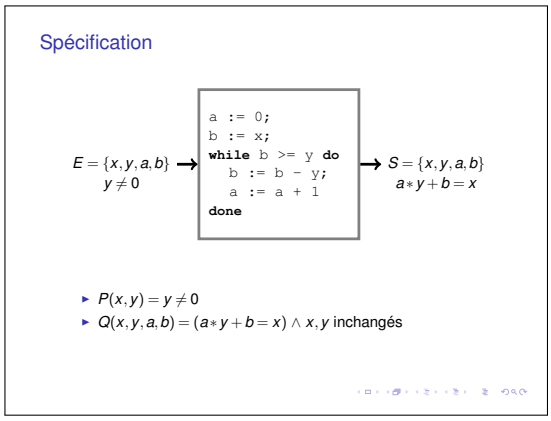
1 Introduction



Ce cours cherche à poser les bases de l'activité qui consiste à s'assurer formellement de la *correction* d'un algorithme ou d'un programme. Informellement, on dira qu'un programme est correct si il effectue sans se tromper la tâche qui lui est confiée *dans tous les cas permis possible*. Pour s'assurer de cela il est évidemment nécessaire de décrire précisément et sans ambiguïté la tâche en question et les cas permis. C'est ce qu'on appelle la *spécification* du programme. Si celle-ci est exprimée dans un langage mathématique non ambiguë, on qualifiera cette spécification de *formelle*. UML[aJRaIJ00] ne fait par exemple pas partie des langages acceptables (du moins pas dans sa globalité) pour une spécification formelle car il n'est pas pourvue de sémantique rigoureuse, il est parfois qualifié de *semi-formel*. La spécification décrit à la fois les cas pour lesquels le programme doit se comporter correctement, et le comportement correct lui-même.

Une spécification ne décrit pas nécessairement tous les détails de fonctionnement du programme, on parlerait alors plutôt d'une *implantation*[1]. On y décrit au contraire uniquement les propriétés que doit satisfaire le programme pour répondre au problème posé. En général on exprime ces propriétés comme une relation entre les *entrées* (ce que l'utilisateur tape au clavier, le contenu d'un fichier, la valeur des variables avant l'exécution, la valeur des arguments d'une fonction...) et les *sorties* du programme (ce qu'il y a à l'écran, la valeur des variable ou le contenu d'un fichier après l'exécution, la valeur de retour d'une fonction). Par exemple dans un programme calculant le quotient q et le reste r de la division entière de x et y (x, y, q et r étant des variables du programme), une des propriétés voulues à la fin du programme est la suivante : $q \times y + r = x$. La plupart du temps, les entrées seront les variables du programme au début du programme et/ou les valeurs des arguments d'une fonction et les sorties seront les variables du programme à la fin de l'exécution du programme et/ou la valeur de retour d'une fonction.





[1] On utilise à tord l'anglicisme *implémentation*.

Une fois la spécification formelle écrite et acceptée par une instance compétente (des spécialistes du domaine concerné, les clients du programmeur...), il s'agit de vérifier que le programme est correct *par rapport* à cette spécification. On utilise pour cela plusieurs techniques : la relecture, le test, la publication du code source etc. Nous nous intéressons à une méthode en particulier : la *démonstration*.

L'objet de ce cours est d'aborder des techniques permettant de démontrer mathématiquement qu'un programme respecte sa spécification. Les démonstrations de correction et de complétude de ces techniques (qui de toutes façons reposent sur la correction d'autres techniques et ainsi de suite) existent et se basent sur la *sémantique* du langage utilisé[2]. Nous verrons que ces techniques nécessitent de la part du vérificateur d'effectuer de vraies démonstrations mathématiques dans lesquelles on ne peut pas tolérer d'erreur. Un outil pour aider le vérificateur à faire des démonstration sans erreur devient alors nécessaire. C'est le domaine de l'*assistance à la preuve* et de la *preuve automatique*, que nous n'aborderons pas dans ce cours, même si lors des TP nous utiliserons `Coq` comme format de lecture des obligations de preuve.

Nous étudierons la technique la plus connue de vérification de programme : la logique de Hoare, puis nous expérimenterons ces notions à l'aide des outils de vérification programme Why et `Caduceus`.

Rappel de logique

Nous ferons appel à des notations logiques simples et usuelles, dont voici un récapitulatif :
- true est la propriété vraie
- false est la propriété fausse
- $\neg p$ signifie «non» p
- $p \wedge q$ signifie p «et» q
- $p \vee q$ signifie p «ou» q
- $p \Rightarrow q$ signifie p «implique» q, c'est-à-dire que si p est vraie alors q aussi. Notez que si p est fausse, alors $p \Rightarrow q$ est vraie quelque soit q.
- $\forall x, P$ signifie «pour tout» x, P est vrai. En général x apparaît dans P.
- $\exists x, P$ signifie «il existe au moins un» x tel que P est vraie. Idem.

Cette liste respecte l'ordre standard de *précédence*, du symbol le plus fort ($\neg$) au symbole le plus faible. Par conséquent l'expression suivante :

$$\forall x \in \mathbb{N}, x > 3 \wedge x < 4 \vee \neg x > 12 \Rightarrow x = 3$$

se parenthèse implicitement comme ceci :

$$\forall x \in \mathbb{N}, \overbrace{(\overbrace{(\overbrace{(x > 3 \wedge x \leq 6)} \vee \overbrace{(\neg(x > 12))})} \Rightarrow \overbrace{x = 3})}$$

Par ailleurs nous utiliserons la notation suivante :

$$p[x \leftarrow v]$$

pour signifier p «dans lequel on remplace» x par v.

2 La logique de Hoare

Dans la logique de Hoare [Hoa69], un programme est considéré comme un *transformateur d'états*. Un état représente ici les valeurs de toutes les variables d'un programme[3]. L'exécution d'un programme a pour effet, si elle se termine, de transformer un état initial en un état final. La spécification d'un programme s'effectuera en formulant des propriétés sur ces deux états.

[2] La sémantique des langage n'est pas abordée dans ce polycopié, mais fait l'objet de la deuxième partie du cours.

[3] En toute rigueur ceci n'est pas suffisant : il faut aussi considérer la place mémoire disponible, le taux de remplissage des unités de stockage, la qualité du compilateur utilisé... Il n'est pas question ici de gérer tous ces paramètres.

2.1 Les triplets de Hoare

Un triplet de Hoare $\{P\}$ `prog` $\{Q\}$ est composé du programme `prog`, de la pré-condition P et de la post-condition Q. P et Q sont des *formule logique* représentant des *propriétés* sur les variables du programme. En général il s'agit de formules de la *logique des prédicats*[4].

Soit le programme `divide` suivant :

```
a := 0;
b := x;
while (b >= y) do
  b := b - y;
  a := a + 1
done
```

Voici un premier exemple de triplet de Hoare :

$$\{\ y \neq 0\ \}\ \texttt{divide}\ \{a \times y + b = x \wedge b \leq y\ \} \tag{1}$$

Il faut lire ce triplet comme suit :

1. Si la propriété $y \neq 0$ est vraie avant l'exécution de `divide` (c'est-à-dire que l'état initial vérifie $y \neq 0$)
2. *Et l'exécution de* `divide` *se termine*,
3. Alors la propriété $a \times y + b = x \wedge b \leq y$ est vraie après l'exécution de `divide`.

Il faut comprendre qu'un triplet peut être vrai ou faux, exactement comme une formule logique[5]. On donne une définition précise de la notion de triplet vrai à la section 2.2. Par exemple le triplet ci-dessous n'est pas vrai car il existe des états initiaux pour lesquels l'état final ne sera pas tel que $b = 0$. :

$$\{\ y \neq 0\ \}\ \texttt{divide}\ \{a * y + b = x \wedge b = 0\} \tag{2}$$

En revanche le triplet suivant semble vrai :

$$\{x \leq 0 \wedge y \leq 0\}\ \texttt{divide}\ \{a \times y + b = x \wedge 0 \leq b \leq y\} \tag{3}$$

2.2 Correction d'un triplet de Hoare

Le premier triplet ci-dessus (1) est problématique car, bien que vrai selon notre description intuitive ci-dessus, il implique des cas ou le programme `divide` ne se termine pas. Par exemple si `x` est négatif et `y` positif. On est donc amené à considérer deux formes de *correction* pour les programme.

Définition 2.2.1 (correction partielle)**.** *Le triplet de Hoare suivant* $\{P\}$ `prog` $\{Q\}$ *est vrai si pour tout état initial vérifiant P, si l'exécution de* `prog` *se termine, alors Q est vraie après l'exécution de* `prog`*. On dit que le programme* `prog` *est* partiellement correct *par rapport à* `P` *et* `Q`*.*

La correction totale s'écrit avec des $\langle\rangle$ (parfois avec des $[\,]$) :

Définition 2.2.2 (correction totale)**.** *Le triplet de Hoare suivant* $\langle P\rangle$ `prog` $\langle Q\rangle$ *est vrai si pour tout état initial de* `prog` *vérifiant P,* `prog` *se termine et Q est vraie après l'exécution de* `prog`*. On dit que le programme* `prog` *est* totalement correct *par rapport à* `P` *et* `Q`*.*

[4] Voir le cours `NFP108`.
[5] Voir le cours `NFP108`.

Nous verrons plus loin qu'il existe des règles pour prouver qu'un triplets est vrai pour chacune de ces deux définitions. Avant cela nous allons préciser le langage de programmation que nous considérons et le langage des prémisses.

2.3 Les instructions

Il n'est pas impossible – c'est un domaine de recherche actuel – de vérifier des programmes utilisant des fonctionnalités comme l'arithmétique sur les pointeurs, le partage de structures, le parallélisme etc, mais il n'en est pas question ici. On se fixe l'ensemble d'instructions autorisées suivant :

$$
\begin{array}{rl}
I ::= & I\texttt{;}I \\
\mid & \texttt{begin } I \texttt{ end} \\
\mid & \texttt{x := } E \\
\mid & \texttt{if } B \texttt{ then } I \texttt{ else } I \\
\mid & \texttt{while } B \texttt{ do } I \texttt{ end} \\
E ::= & \texttt{x} \mid \texttt{y} \mid \ldots \mid E\texttt{+}E \mid E\texttt{*}E \mid E\texttt{-}E \mid \ldots \mid f(E,E\ldots) \mid B \\
B ::= & B \texttt{ and } B \mid B \texttt{ or } B \mid \texttt{not } B \ldots \mid E\texttt{<}E \mid E\texttt{<=}E \mid E\texttt{=}E \ldots
\end{array}
$$

Ceci est une version simplifiée de ce que `Why` autorise. E correspond aux expressions, I aux programmes. *on ne considérera que les programmes bien typés*. Notons que malgré le faible nombre d'instructions on peut exprimer de nombreux programmes.

2.4 Les assertions

Dans les triplets ci-dessus (2) $\{\texttt{y} \neq \texttt{0}\}$, $\{\texttt{x} \leq \texttt{0} \wedge \texttt{y} \leq \texttt{0}\}$ et $\{\texttt{a} \times \texttt{y+b=x} \wedge \texttt{0} \leq \texttt{b} \leq \texttt{y}\}$ sont des *assertions*. Plus précisément dans un triplet de la forme $\{P\}\texttt{prog}\{Q\}$ les assertions P et Q sont en général des formules de la logiques de prédicats où les prédicats portent sur les variables du programme (notées en caractères d'imprimerie `x`, `y`,...) et sur des variables supplémentaires (notées x, y,...).

Notez que les expression booléennes (B ci-dessus) du programme, par exemple `x <= 0 and y <= 0` ont un équivalent immédiat dans les assertions, ici $\texttt{x} \leq 0 \wedge \texttt{y} \leq 0$, où $\leq$ est un prédicat binaire. Les premières sont des expressions du programme, ayant à l'exécution la valeur `vrai` ou `faux`, alors que les deuxièmes sont des formules logiques décrivant des propriétés sur les variables du programme.

Elles servent à écrire :

- Des pré- et post-conditions,
- Des assertions spéciales pour les boucles :
  - Les *invariants de boucle* sont des formules logiques destinées à établir qu'une propriété est vraie à chaque itération d'une boucle. Voir la section 3.5.
  - Les *variants de boucle* ne sont pas des formules, mais des expressions (E ci-dessus), dont la valeur doit diminuer à chaque itération, permettant ainsi de démontrer que l'itération s'arrêtera toujours. Voir la section 3.6.

2.5 Les règles d'inférence

Un système logique est un couple $(\mathscr{J},\mathscr{R})$, où $\mathscr{J}$ est un ensemble de *jugements*, et $\mathscr{R}$ un ensemble de *règles de déduction* ou *règles d'inférence* permettant de définir le sous-ensemble $\mathscr{R}(\mathscr{J}) \subset \mathscr{J}$ des jugements dits *valides*.

La forme des jugements est très variable, dans certains systèmes logiques il s'agit des formules, dans d'autres de *séquents* de la forme $\Gamma \vdash F$ où Γ est un ensemble de formules et F une formule[6]. Dans la logique de Hoare *il s'agira de triplets de Hoare ou de formules logiques*.

[6] Voir le cours `NFP108`.

Une *règle de déduction*, ou *règle d'inférence*, est une fraction de la forme :

$$\frac{J_1 \ \ldots \ J_n}{\mathrm{J}}$$

où J_i et J sont des *jugements*. Les J_i sont les *prémisses* et J est la *conclusion*. Il existe des règles particulières –appelées *axiomes*– qui n'ont aucune prémisse :

$$\frac{}{\mathrm{J}}$$

Bien entendu la signification de ce type de règle est "le jugement J est toujours vrai".

Voici un exemple de système de règles de déduction dont les jugement sont de la forme $i < j$ où $i, j \in \mathbb{N}$ composé des deux règles nommées INF0 et INF+ :

$$\textsc{Inf0}\ \frac{}{0 \geq 0} \qquad\qquad \frac{x \geq y}{x+1 \geq y}\ \textsc{Inf+}$$

Si, comme dans la règle INF+, les jugements contiennent des variables, alors on peut *appliquer* la règle à toute valeur des variables. Par exemple les règles suivantes sont des *instances* de INF+ :

$$\textsc{Inf+}\ \frac{3 \geq 2}{4 \geq 2} \qquad\qquad \frac{5 \geq 1}{6 \geq 1}\ \textsc{Inf+}$$

Dans la première x vaut 3 et y vaut 2, dans la deuxième x vaut 5 et y vaut 1.

2.6 Arbre de déduction

Un arbre de déduction dans un système logique $(\mathscr{J}, \mathscr{R})$ est une combinaison finie de règles de la forme :

$$\frac{\dfrac{\dfrac{\vdots}{J_{11}} \cdots \dfrac{\vdots}{J_{1k_1}}}{J_1} \cdots \dfrac{\dfrac{\vdots}{J_{n1}} \cdots \dfrac{\vdots}{J_{nk_n}}}{J_n}}{\mathrm{J}}$$

où chaque règle appliquée est une instance d'une règle. J est la *racine* de l'arbre, les jugements n'ayant pas de prémisses sont les *feuilles*. Plus précisément :

Définition 2.6.1 (Arbre de déduction)**.** *L'ensemble des arbres de déduction d'un système $\mathscr{R}$ est défini récursivement comme suit :*

- *Si r est une instance d'une règle de $\mathscr{R}$, alors r est un arbre de déduction ;*
- *Si $t_1 \ldots t_2$ sont des arbres de déduction dont les racines sont $J_1 \ldots J_n$, et*

$$\frac{J_1 \ \ldots \ J_n}{J}$$

est une instance de règle de $\mathscr{R}$ alors l'arbre suivant est un arbre de déduction :

$$\frac{t_1 \ \ldots \ t_n}{J}$$

Définition 2.6.2 (Arbre de déduction complet)**.** *Un arbre de déduction* complet *est un arbre dans laquelle toutes les feuilles sont des (instances d') axiomes.*

Le principe des systèmes de déduction est que les règles définissent l'ensemble des jugement *prouvables* comme suit :

Définition 2.6.3 (Jugement prouvable)**.** *Soit* $(\mathscr{J},\mathscr{R})$ *un système logique, un jugement J est* prouvable *dans* $(\mathscr{J},\mathscr{R})$ *si il existe un arbre de déduction complet avec J à la racine. On parle alors d'*arbre de preuve *pur J.*

Donc pour prouver qu'un jugement J est prouvable dans un système de déduction $(\mathscr{J},\mathscr{R})$, il faut et il suffit d'exhiber un arbre de preuve pour J dans $\mathscr{R}$. Notez que ceci est la *définition* de "être prouvable" dans $(\mathscr{J},\mathscr{R})$. Le fait que l'ensemble des jugements prouvables est «intéressant» ou «utile» est un autre problème. Dans le cas de la logique de Hoare, où les jugement sont des triplets de Hoare, il existe un théorème (voir plus bas le théorème 1) qui établit que si un triplet est prouvable, alors il est vrai.

3 Les règles de Hoare

Une règle de Hoare est une règle de déduction, c'est-à-dire une fraction de la forme :

$$\frac{\text{triplet}_1 \dots \text{triplet}_n}{\text{triplet}}$$

Les prémisses et la conséquence sont des triplets de Hoare. Une telle fraction se lit de la manière suivante : "Si les triplets prémisses sont vrais, alors le triplet conclusion aussi". Dans un premier temps on n'étudiera que la correction partielle des triplets, puis on verra comment s'assurer de la terminaison des programme en modifiant certaines règles.

Toute la technique de la logique de Hoare repose sur le théorème suivant, que nous ne démontrons pas ici mais qui est démontré *par rapport* à la sémantique du langage présenté en section 2.3.

Theorem 1 (Correction des règles de Hoare)**.** *Étant donné un triplet de Hoare donné t, si il existe un arbre de déduction de Hoare complet ayant t à sa racine, alors le triplet est vrai au sens de la section 2.2.*

Nous allons définir et illustrer ce qu'est un arbre de déduction de Hoare dans la suite en détaillant chaque règle. Démontrer cette propriété nécessite de définir la sémantique mathématique de chaque instruction, de vérifier cette sémantique dans le compilateur utilisé pour compiler ce programme et enfin d'en déduire la correction des règles de Hoare une par une. Nous ne chercherons pas à le faire, nous admettrons simplement que *si il est possible de construire un arbre de dérivation complet* (c'est-à-dire dont toutes les branches se terminent par un axiome) *dont la racine est un triplet T, alors T est vrai.*

3.1 Affectation

L'exemple suivant illustre la règle d'affectation, dont la première formulation ci-dessous est peu intuitive. Soit le triplet, manifestement vrai, `{x-z≥0} x:=z-y {x≥0}`. La règle de déduction de l'affectation doit permettre de prouver ce triplet donc la règle de l'affectation doit pouvoir s'appliquer comme suit :

$$\frac{}{\texttt{\{z-y≥0\}\ x:=z-y\ \{x≥0\}}}$$

On voit que `x ≥ 0` est vrai après l'exécution du programme seulement si la même propriété est vraie *pour* `z-y` *au lieu de* `x`. La règle de l'affectation exprime ceci de la manière suivante :

$$\textsc{Aff1}\ \frac{}{\{P[x \leftarrow E]\}\ x := E\ \{P\}}$$

où la notation $P[x \leftarrow E]$ signifie «P dans laquelle x a été remplacé par E». Si on lit cette règle de la manière suivante elle devient claire : Pour que P soit vraie pour x après le programme $x := E$, il fallait qu'elle soit vraie pour E avant le programme. On peut trouver une forme plus intuitive pour cette règle en essayant d'avoir la pré-condition P et quelque-chose en fonction de P dans la post-condition. Soit une règle de cette forme :

$$\frac{}{\{P\}\quad x := E\quad \{??\}}$$

Malheureusement ce n'est pas simple car l'affectation entraîne les deux choses suivantes :

- Si x apparaît dans E, il s'agit de l'ancienne valeur de x. Cette ancienne valeur de x, qu'on notera x_0[7] est donc telle que après l'affectation on a : $x = E[x \leftarrow x_0]$.
- Si x apparaît dans P, alors P n'est plus vraie car x a changé, en revanche $P[x \leftarrow x_0]$ est vraie.

On exprime ces deux propriétés dans la règle de hoare de l'affectation (deuxième version) :

$$\textsc{Aff}\ \frac{}{\{P\}\quad x := E\quad \{P[x \leftarrow x_0]\ \wedge\ x = E[x \leftarrow x_0]\}}$$

On gagnera en compréhension en voyant que le x_0 est en fait une variable existentielle :

$$\textsc{Aff}\ \frac{}{\{P\}\quad x := E\quad \{\exists x_0,\ P[x \leftarrow x_0]\ \wedge\ x = E[x \leftarrow x_0]\}}$$

Dans l'exemple, on applique la règle comme suit :

$$\textsc{Aff}\ \frac{}{\{x - y \geq 0\}\quad x := x - y\quad \{x_0 - y \geq 0\ \wedge\ x = x_0 - y\}}$$

Ce qui permet de déduire que $x \geq 0$ après l'exécution de cette instruction.

Exercice 1. *Écrivez un arbre de dérivation ayant à sa racine un triplet de la forme :* $\{ x-y \geq 0 \}\ y := y+x\ \{??\}$.

Solution:

$$\textsc{Aff}\ \frac{}{\{ x - y \geq 0 \}\quad y := y+x\quad \{ x - y_0 \geq 0\ \wedge\ y = y_0 + x\}}$$

□

3.2 Séquence

Pour la «séquence» d'instructions, la règle est relativement simple à construire :

$$\textsc{Seq}\ \frac{\{P\}\,C_1\,\{Q\} \qquad \{Q\}\,C_2\,\{R\}}{\{P\}\,C_1\ ;\ C_2\,\{R\}}$$

[7] le nom x_0 de cette variable importe peu, il suffit qu'il soit «frais» c'est-à-dire qu'aucune autre variable ne possède déjà ce nom.

On applique cette règle sur un exemple (notez que les deux prémisses sont prouvées avec la règle AFF1) :

$$\textsc{Seq}\ \frac{\textsc{Aff1}\ \dfrac{}{\{0+x\geq 0\}\ \ \texttt{a:=0}\ \ \{a+x\geq 0\}} \qquad \textsc{Aff1}\ \dfrac{}{\{a+x\geq 0\}\ \ b:=x\ \ \{a+b\geq 0\}}}{\{0+x\geq 0\}\ \ \texttt{a:=0;b:=x}\ \ \{a+b\geq 0\}}$$

Exercice 2. *Prouvez le triplet suivant à l'aide des règle* AFF *et* SEQ *(mais pas* AFF1*) :*

$$\{0+x\geq 0\}\ \texttt{a:=0; b:=x}\ \{a+b\geq 0\}$$

Solution:

$$\textsc{Cons}\ \frac{\textsc{Seq}\ \dfrac{\textsc{Aff}\ \dfrac{}{\{0+x\geq 0\}\ a:=0;\ \{a=0\wedge 0+x\geq 0\}} \qquad \textsc{Aff}\ \dfrac{}{\{a=0\wedge 0+x\geq 0\}\ b:=x\ \{b=x\wedge a=0\wedge 0+x\geq 0\}}}{\{0+x\geq 0\}\ a:=0;\ b:=x\ \{b=x\wedge a=0\wedge 0+x\geq 0\}}}{\{0+x\geq 0\}\ a:=0;\ b:=x\ \{a+b\geq 0\}}$$

□

Exercice 3. *Écrivez un arbre de dérivation ayant à sa racine un triplet de la forme :*
$\{\ x-y\geq 0\ \}\texttt{x:=x-y;y:=y+x}\ \{??\ \}$

Solution:

$$\textsc{Seq}\ \frac{\textsc{Aff}\ \dfrac{}{\{\ x-y\geq 0\ \}\ \ x:=x-y}\qquad \dfrac{}{\begin{array}{c}\{\ x_0-y\geq 0\wedge x=x_0-y\ \}\\ y:=y+x\end{array}}\ \textsc{Aff}}{\{\ x_0-y\geq 0\wedge x=x_0-y\}\ \ \{\ x_0-y_0\geq 0\wedge\ x=x_0-y_0\wedge y=y_0+x\}}$$
$$\{\ x-y\geq 0\ \}\quad x:=x-y;y:=y+x\quad \{\ x_0-y\geq 0\wedge\ x=x_0-y_0\wedge y=y_0+x\ \}$$

□

Exercice 4. *Démontrez la correction des triplets suivants :*

1. $\{x=0\}\ \texttt{x:=x+1; x:=x+1}\ \{x=2\}$
2. $\{x>0\}\ \texttt{x:=x+1; x:=x+1}\ \{x>2\}$
3. $\{x=0\}\ \texttt{x:=x+1; x:=x+1}\ \{x\geq 2\}$
4. $\{x>0\}\ \texttt{x:=x+1; x:=x+1}\ \{x\geq 2\}$
5. $\{P\}\ \texttt{swap}\ \{Q\}$ *où* `swap` *est le programme qui intervertit deux variables et P et Q la spécification naturelle pour* `swap`*.*

□

Exercice 5. *Démontrez la règle suivante :*

$$\textsc{Seq3}\ \frac{\{P\}\,C_1\,\{Q\} \qquad \{Q\}\,C_1\,\{R\} \qquad \{R\}\,C_2\,\{S\}}{\{P\}\,C_1\ ;\ C_2\,\{S\}}$$

Solution:
Pour cela on exhibe un arbre de déduction dont la racine est la racine voulue et les feuille les prémisses voulues :

$$\textsc{Seq}\ \dfrac{\textsc{Aff}\ \dfrac{\{P\}\ i_1\ \{Q\} \quad \{Q\}\ i_2\ \{R\}}{\{P\}\ i_2;\ i_2\ \{R\}} \qquad \{R\}\ i_3\ \{S\}}{\{P\}\ i_2;\ i_2;\ i_3\ \{S\}}$$

□

3.3 Conséquence

Les propriétés que l'on peut prouver directement à partir des règles vues jusqu'ici sont d'une forme très contrainte, or il faut pouvoir déduire de ces propriétés celles qui nous intéressent. Ceci se fait par simple *déduction logique* à partir des propriétés prouvées par les règles précédentes. Pour cela on ajoute la règle suivante (notez que les deux implications ne sont pas dans le même sens) :

$$\textsc{Conseq}\ \dfrac{P \Rightarrow P' \quad \{P'\}C\{Q'\} \quad Q' \Rightarrow Q}{\{P\}C\{Q\}}$$

Ce qui permet de déduire dans l'exercice ci-dessus que le triplet de Hoare suivant est valide (car $0+x=x$) :

$$\textsc{Conseq}\ \dfrac{\mathtt{x}\geq 0 \Rightarrow 0+\mathtt{x}\geq 0 \qquad \textsc{Seq}\ \dfrac{\textsc{Aff1}\ \dfrac{}{\{0+\mathtt{x}\geq 0\}\mathtt{a:=0}\{\mathtt{a+x}\geq 0\}} \qquad \textsc{Aff1}\ \dfrac{}{\{\mathtt{a+x}\geq 0\}\mathtt{b:=x}\{\mathtt{a+b}\geq 0\}}}{\{0+\mathtt{x}\geq 0\}\mathtt{a:=0;b:=x}\{\mathtt{a+b}\geq 0\}} \qquad \mathtt{a+b}\geq 0 \Rightarrow \mathtt{a}\geq -\mathtt{b}}{\{\mathtt{x}\geq 0\}\mathtt{a:=0;b:=x}\{\mathtt{a}\geq -\mathtt{b}\}}$$

Exercice 6. *Complétez la dérivation de l'exercice 3 pour en déduire une post-condition plus compacte :*

Solution:

$$\textsc{Conseq}\ \dfrac{\textsc{Seq}\ \dfrac{\vdots}{\{x-y\geq 0\}x:=x-y;y:=y+x\{x_0-y\geq 0\wedge x=x_0-y_0\wedge y=y_0+x\}} \qquad y=y_0+x_0-y_0 \Rightarrow y=x_0}{\{x-y\geq 0\}x:=x-y;y:=y+x\{y=x_0\}}$$

□

Les preuves des implications sont à prouver aussi, On peut ajouter des règles de déduction logique pour les vérifier en même temps que les triplets de Hoare. Dans le cadre de `Why` les preuves logiques sont laissées à l'utilisateur sous la forme d'obligations de preuves dans le format d'un assistant de preuve (`Coq`, PVS, Simplify etc). Ces implications ne sont en général pas prouvables automatiquement. En particulier, il peut arriver que la correction d'un algorithme repose sur des résultats mathématiques difficiles, dont la démonstration est hors de portée des outils de preuve.

3.4 Conditionnelle

On ajoute ensuite la règle de la conditionnelle, qui exprime bien que la post-condition doit être vérifiée dans les deux branches du **if** *chacune sous la condition que le résultat du test est conforme* à la branche. En particulier si une des branches est impossible, alors la prémisse correspondante est trivialement juste (règle CONSEQ).

$$\textsc{Cond}\ \frac{\{\{P\wedge B\}\quad I_1\quad \{Q\}\qquad \{P\wedge\neg B\}\quad I_2\quad \{Q\}}{\{P\}\quad \texttt{if } B \texttt{ then } I_1 \texttt{ else } I_2\quad \{Q\}}$$

Exemple :

$$\textsc{Cond}\ \frac{\textsc{Aff}\ \dfrac{}{\{true\wedge \texttt{y=0}\}\ \texttt{x:=y}\ \{\texttt{x=0}\}} \qquad \textsc{Conseq}\ \dfrac{(true\wedge \texttt{y}\neq 0)\Rightarrow(0=0)\quad \{\texttt{0=0}\}\ \texttt{x:=0}\ \{\texttt{x=0}\}}{\{true\wedge \texttt{y}\neq 0\ \}\ \texttt{x:=0}\ \{\texttt{x=0}\}}}{\{true\}\ \texttt{if y=0 then x:=y else x:=0}\ \{\texttt{x=0}\}}$$

Exercice 7 (dur)**.** *Prouvez que le triplet suivant est vrai.* $\{x\geq 0\}$ ***if*** $x >= 0$ ***then*** $y := 8$ ***else*** $y := 9$ $\{y = 8\}$.

Solution:

$$\textsc{Cond}\ \frac{\textsc{Aff}\ \dfrac{}{\{y\geq 0\wedge y\geq 0\}\ x:=8\ \{x=8\}} \qquad \textsc{Conseq}\ \dfrac{x\geq 0\wedge x<0\Rightarrow false\quad \{false\}\ x:=9\ \{false\}\quad false\Rightarrow x=8}{\{x\geq 0\wedge x<0\}x:=0\{x=8\}}}{\{x\geq 0\}\ \texttt{if } x >= 0 \texttt{ then } y:=8 \texttt{ else } y:=9\ \{y=8\}}$$

□

Exercice 8. *Prouvez le triplet suivant* $\{x \geq 0\}$ **if** $x<>0$ **then** $x:=(-4)$ **else** $x:=x+1$ $\{x>0\}$ □

3.5 While

$$\textsc{While}\ \frac{\{P\wedge B\}\quad C\quad \{P\}}{\{P\}\quad \texttt{while } B \texttt{ do } C \texttt{ done}\quad \{P\wedge\neg B\}}$$

Par exemple :

$$\textsc{While}\ \frac{\textsc{Conseq}\ \dfrac{(\texttt{x}\geq 0\wedge \texttt{x}<\texttt{b})\Rightarrow \texttt{x}+1\geq 0 \qquad \textsc{Aff}\ \dfrac{}{\{\texttt{x}+1\geq 0\}\texttt{x:=x+1}\{\texttt{x}\geq 0\}}}{\{\texttt{x}\geq 0\wedge \texttt{x}<\texttt{b}\}\texttt{x:=x+1}\{\texttt{x}\geq 0\}}}{\{\texttt{x}\geq 0\}\quad \texttt{while x<b do x:=x+1 done}\quad \{\texttt{x}\geq 0\wedge\neg(\texttt{x}<\texttt{b})\}}$$

Exercice 9. $\{\}$ **while** $x>0$ **do** $x:=x-1$ **done** $\{x=0\}$

Solution:

$$\textsc{While}\ \dfrac{\textsc{Conseq}\ \dfrac{\textsc{Aff}\ \dfrac{}{\{\}\ \texttt{x:=x-1}\ \{x=x_0-1\}\quad x=x_0-1 \Rightarrow true}}{\{\}\ \texttt{x:=x-1}\ \{true\}}}{\textsc{Conseq}\ \dfrac{\{\}\ \texttt{while x<>0 do x:=x-1 done}\ \{x=0 \wedge true\}\quad x=0 \wedge true \Rightarrow x=0}{\{\}\ \texttt{while x<>0 do x:=x-1 done}\ \{x=0\}}}$$

□

3.6 While avec variant (correction totale)

Pour prouver la correction totale d'un programme, il faut d'une part prouver sa correction partielle, d'autre part s'assurer que les boucles (et les fonctions récursives, ce que nous ne considérons pas dans ce cours) s'arrêtent nécessairement lorsque les pré-conditions sont vérifiées. On modifie donc la règle du while afin d'exhiber une expression, qu'on appellera le *variant*, qui doit :

- Être une fonction des variables du programme ;
- Être positive lorsqu'on entre dans la boucle (c'est-à-dire lorsque la condition de la boucle est vérifiée *et* lorsque l'invariant de la boucle est vérifié ;
- Décroître strictement à chaque itération de la boucle.

Voici la règle modifiée dans laquelle le variant est noté E :

$$\textsc{WhileT}\ \frac{\langle P \wedge B \wedge (E=n)\rangle\ C\ \langle\ P \wedge E < n \wedge E \geq 0\ \rangle}{\langle P\rangle\quad \texttt{while}\ B\ \texttt{do}\ C\ \texttt{done}\quad \langle P \wedge \neg B\rangle}$$

Exercice 10. *Que faut-il faire pour prouver la correction totale des triplets de l'exercice 4 ?*

Solution:

Rien car il n'y a pas de while ni de fonction récursive.

□

Exercice 11. $\langle x \geq 0\rangle$ `while x>0 do x:=x-1 end` $\langle x=0\rangle$

Solution:

On prend $E = x$

$$\textsc{While}\ \dfrac{\textsc{Conseq}\ \dfrac{\textsc{Aff}\ \dfrac{}{\langle x \geq 0 \wedge x > 0 \wedge x = n\rangle\ x := x-1\ \langle x_0 \geq 0 \wedge x_0 > 0 \wedge x_0 = n \wedge x = x_0 - 1\rangle}}{\langle x \geq 0 \wedge x > 0 \wedge x = n\rangle\ x := x-1\ \langle x \geq 0 \wedge x < n\rangle}\quad x \geq 0 \wedge x > 0 \Rightarrow x \geq 0}{\langle x \geq 0\rangle\ \textbf{\textit{while}}\ x > 0\ \textbf{\textit{do}}\ x := x-1\ \textbf{\textit{end}}\ \langle x = 0\rangle}$$

□

Exercice 12. $\langle$ `y > 0` $\rangle$**while** `y <= r` **do** `r:=r-y; q=q+1` **done** $\langle$ `?` $\rangle$

Solution:
On prend :
- $P = y > 0$
- $B = y \leq r$
- $E = r$
- $C =$ `r:=r-y; q=q+1`

□

3.7 Les tableaux

La preuve de programmes utilisant des tableaux nécessite la preuve qu'aucun accès aux tableaux ne se fait en dehors de ceux-ci. Ceci nécessite de vérifier à chaque accès `t[i]` que `i` est compris entre 0 et taille de `t` -1. Afin d'adapter la logique de Hoare à ce type de vérification on peut commencer par modifier la règle AFF :

Cette règle amène plusieurs remarques. Premièrement l'expression `t[i]` à l'intérieur des pré/post-conditions doit être défini. Dans le cas des tableaux de taille prédéfinie et en l'absence d'*aliasing* (superposition de tableaux, partage par plusieurs entité d'un même espace mémoire), il suffit de considérer qu'un tableau de taille n est équivalent à n nouvelles variables. Le problème de savoir traiter correctement les tableaux superposés et de taille non statique est difficile et la recherche commence à peine à fournir des solutions réellement utilisables.

Deuxièmement on ne peut pas traiter les instructions de la forme `t[i]:=3`.

Troisièmement même pour les autres instructions de la forme `t[e]:=`, on constate que la règle ne permet pas de traiter des instructions du genre `x := t[i] + 3` ni tout autre instruction concernant des tableaux qui ne soit pas exactement de la forme `x=t[i]`. On se convaincra néanmoins assez facilement que certains programmes peuvent être transformés en des programmes équivalents dans lesquels tous les accès aux tableaux sont de cette forme. Cette propriété permet d'entrevoir une technique très répandue dans la preuve de programme : la *transformation de programme*. Celle-ci consiste à effectuer des transformations *conservatives* sur un programme avant d'appliquer les règles de Hoare. Une transformation est conservative si elle préserve la sémantique du programme. Il en résulte que les triplets prouvés sur le programme transformé sont également vrais sur le programme original.

Finalement la règle de l'affectation doit être modifiée de la manière suivante :

$$\text{AFF'}\ \frac{\forall\, \mathtt{t}[e] \in e_i,\ P \Rightarrow 0 \leq e < |\mathtt{t}|}{\{P\}\quad e_1 := \mathtt{t}[e_2]\quad \{P[e_1 \leftarrow x_0]\ \wedge\ e_1 = \mathtt{t}[e_2[e_1 \leftarrow x_0]]\}}$$

où l'expression $\forall\, \mathtt{t}[e] \in e_i,\ P \Rightarrow 0 \leq e < |\mathtt{t}|$ signifie que pour tout expression `t[e]` apparaissant dans les expressions e_1 ou e_2, $P \Rightarrow 0 \leq e < |\mathtt{t}|$. $|\mathtt{t}|$ est une variable dédiée représentant la taille du tableau `t`. On rappelle que les règles de Hoare suppose implicitement que le programme est correctement typé, donc e_1 ne peut pas être n'importe quelle expression.

On peut modifier toutes les règles de Hoare précédentes afin de prendre en compte les accès aux tableaux :

Exercice 13. *Prouvez l'algorithme de la recherche dans un tableau d'entiers.*

Solution:
Première solution, sans tester les bornes (on considère juste les `t[i]` comme des variables).

$$\textsc{Seq}\ \dfrac{\textsc{Conseq}\ \dfrac{\textsc{While}\ \dfrac{\textsc{Aff}\ \dfrac{}{\{i<n \wedge n=|t| \wedge \forall 0\leq x<i, t[i]\neq m \wedge i\leq n\}\ \ i:=i+1\ \ \{i=i_0+1 \wedge i_0<n \wedge n=|t| \wedge \forall 0\leq x<i_0, t[x]\neq m \wedge i_0\leq n\}}}{\{n=|t| \wedge \forall 0\leq x<i, t[x]\neq m \wedge i\leq n\}\ \ \textbf{while}\ i<n\ \textbf{and}\ t[i]<>m\ \textbf{do}\ i:=i+1\ \textbf{done}\ \{(i\geq n \vee t[i]=m) \wedge i\leq n \wedge n=|t| \wedge \forall 0\leq x<i, t[x]\neq m \wedge i\leq n\}}}{\ldots\{n=|t| \wedge i=0\}\textbf{while}\ i<n\ \textbf{and}\ t[i]<>m\ \textbf{do}\ i:=i+1\ \textbf{done}\{t[i]=m \vee i=n \wedge \forall j,\ t[j]\neq m\}}}{\{n=|t|\}\ \ i:=0;\ \textbf{while}\ i<n\ \textbf{and}\ t[i]<>m\ \textbf{do}\ i:=i+1;\ \textbf{done}\ \{t[i]=m \vee i=n \wedge \forall j,\ t[j]\neq m\}}$$

□

Exercice 14. *Prouvez un algorithme qui inverse l'ordre des éléments d'un tableau d'entiers.* □

Exercice 15. *Prouvez un algorithme de tri sur un tableau d'entiers.* □

Exercice 16. *Prouvez un algorithme qui calcul la moyenne des éléments d'un tableau d'entiers.* □

4 Conclusion

4.1 Les difficultés

En général ce qui est difficile lors de la preuve d'un programme se situent à deux endroits : (a) L'écriture de assertions, en particulier les variants et invariants de boucles sont parfois durs à trouver, ils nécessitent une compréhension profonde le l'algorithme. (b) les preuves des étapes de déduction (règle CONSEQ) peuvent être arbitrairement difficiles (ex : théorème de Bertrand).

4.2 Programme annoté

Un programme annoté est un programme dans lequel des assertions sont insérées. On peut insérer des annotations avant et après chaque instruction. On peut même introduire plusieurs annotations entre deux instructions. On considère cependant en général qu'un programme est suffisamment (ce qui ne veut pas dire correctement) annoté si il y a des assertions aux endroits suivants :

- Avant et après le programme,
- avant chaque instruction qui n'est pas une affectation,
- après le mot-clé **do** dans les boucles **while**, c'est là qu'on insère l'invariant et le variant de la boucle.

Exemple :

```
{ x =n }
y:=1;
{ y = 1 /\x = n }
while x != 0 do { y * x! = n! } (* x! représente la factorielle au sens mathématique *)
  y:=y * x;
  x:=x-1
done
{ y = n! }
```

En théorie il n'y a pas besoin de mettre autant d'annotations, il suffit de préciser les variants et invariants de boucle. L'ajout d'annotation supplémentaire permet en fait de déclencher la règle de conséquence, afin de déduire des propriétés sur certains points intermédiaires du programme. Dans cet exemple, on voit que l'annotation $\{y = 1 \wedge x = n\}$ n'est en fait pas nécessaire car elle est exactement identique à ce que la règle AFF permet de déduire à partir du triplet incomplet suivant : `{x=n} x:=1 {?}`. L'ajout d'annotations en dehors des (in)variants de boucles est toutefois utile pour simplifier les propriétés en des points intermédiaires du programme, grâce à la règle de conséquence, afin de garder des formule de taille raisonnable.

4.3 exercices

Exercice 17. *Vérifiez le programme la correction partielle du programme annoté de la section 4.2. Quelle annotation faut–il ajouter pour la correction totale ? Vérifiez.* □

Exercice 18. *Écrivez et annotez le programme* `divide` *pour la correction partielle. Attention, Les annotations doivent assurer que le dividende et le diviseur n'ont pas changer. Recommencez pour la correction totale.* □

5 Caduceus

Exercice 19. *Traduisez en* `Caduceus` *le triplet suivant :*

```
{x = 1 \x =-1} if x<=0 then x:=x+1 else x:=x-1 end { x = 0 }
```

Les obligations de preuve générées sont-elles démontrables ?

Solution:
Facile.

□

5.1 max

Exercice 20. *Écrivez et spécifiez le programme* `max` *qui retourne le maximum de ces deux arguments.*

Solution:

```
/*@ ensures
  @   \result >= x && \result >= y &&
  @   \forall int z; z >= x && z >= y => z >= \result
  @*/
int max(int x, int y) {
  if (x > y) return x; else return y;
}
```

```
/*
*** Local Variables: ***
*** compile-command: "caduceus -coq-tactic \"intuition;subst\" max.c ; make -f max.makefile coq.depend; make -f ma
*** End: ***
*/
```

□

5.2 swap

Exercice 21. *Écrivez et spécifiez le programme* `swap` *qui échange les valeurs de deux variables globales. Les obligations de preuve générées sont-elles démontrables ?*

```
int q;
int r;

/*@ requires ...
    ensures ... */

void swap (...){
  ...
}
```

Solution:

□

5.3 divide

Exercice 22. *Écrivez et prouvez le programme* `divide`*. Les obligations de preuve générées sont-elles démontrables ? Le quotient q et le reste r seront stockés dans des variables globales.*

Solution:

```
int a;
int b;

/*@ requires y>0 && x >= 0
  @ ensures a*y+b==x && b >= 0 && b <= y @*/



int divide (int x, int y){
  a=0;
  b=x;

  /*@ invariant ( a ) * y + ( b ) == x && b >= 0
        variant b - y + 1 @*/
  while (b>=y)
        {
              b = b - y;
              a = a + 1;
```

```
        }
}


/*
*** Local Variables: ***
*** compile-command: "caduceus -coq-tactic \"intuition;subst\" divide.c ; make -f divide.makefile coq.depend; make -f
*** End: ***
*/
```

□

5.4 Autres exemples

Autres exemple à faire en TP :
La valeur absolue, deux version :

```
//@ ensures *p >= 0
void abs1(int *p) {
  if (*p < 0) *p = -*p;
}

/*@ requires \valid(p)
  @ ensures *p >= 0
  @*/
void abs2(int *p) {
  if (*p < 0) *p = -*p;
}



/*
*** Local Variables: ***
*** compile-command: "caduceus -coq-tactic \"intuition;subst\" abs.c ; make -f abs.makefile coq.depend; make -f abs.
*** End: ***
*/
```

La factorielle.
La recherche dans un tableau (il faut avoir vu les tableaux) :

```
/*@ requires \valid_range(t,0,n-1)
  @ ensures
  @   (0 <= \result < n => t[\result] == v) &&
  @   (\result == n => \forall int i; 0 <= i < n => t[i] != v)
  @*/
int index(int t[], int n, int v) {
  int i = 0;
  /*@ invariant 0 <= i && \forall int k; 0 <= k < i => t[k] != v
    @ variant n - i */
```

```
  while (i < n) {
    if (t[i] == v) break;
    i++;
  }
  return i;
}


/*
*** Local Variables: ***
*** compile-command: "caduceus -coq-tactic \"intuition;subst\" search.c ; make -f search.makefile coq.depend; make -
*** End: ***
*/
```

6 Why

On suivra les étapes suivantes :

- Écrire le programme avec juste les annotation de début et de fin. Lancer `Why` dessus jusqu'à ce qu'il ne râle plus.
- On n'a pas explicité les déclarations de variable. Dans la syntaxe de `Why` se sera fait de la manière suivante :

  ```
  let x = ref 0 in
  let y = x in
  I
  ```

  Pour utiliser la valeur d'une variable on utilisera l'opérateur '!'.

6.1 divide

Finalement le programme `divide` s'écrira de la manière suivante :

```
parameter x,y:int
parameter a,b:int ref

let divide =
begin
a := 0;
b := x;
while (!b >= y) do
  b := !b - y;
  a := !a + 1
done
end
```

Avec les annotations minimales, ça donne :

```
parameter x,y:int
parameter a,b:int ref

let divide =
```

```
  { y > 0 and x >= 0 }
begin
a := 0;
b := x;
while (!b >= y) do
  b := !b - y;
  a := !a + 1
done
end
{ a*y+b=x }
```

Une fois annoté complètement :

```
parameter moy,ne,nf:int
parameter a,b:int ref

let divide =
  { y > 0 and x >= 0 }
begin
a := 0;
b := x;

while (!b >= y) do
  {
        invariant a * y + b = x and b >=0
        variant b - y + 1 }
  b := !b - y;
  a := !a + 1
done
end
{ a*y+b=x and b >= 0 and b <= y}
```

6.2 mult

Autre exemple, la multiplication :

```
parameter x,y:int
parameter a,b:int ref
  (* Calcul de x*y *)

let mult =
  { x > 0 and y>=0}
begin
  a := x-1;
  b := y;

  while (!a > 0) do
    { invariant a>=0 and b=y*(x-a) variant a }
    b := !b + y;
    a := !a - 1
  done
end
{ b=y*x }
```

6.3 power

Autre exemple, la mise à la puissance :

```
logic Zpower: int,int -> int
parameter x,y:int
parameter res,m:int ref

let pow =
  { y >= 0 }
begin
res := 1;
m := y;
while !m > 0 do
  {
    invariant y-m>=0 and res = Zpower (x, (y - m)) and m>=0
    variant m }
  res:=!res*x;
  m:=!m-1
done
end
{ res = Zpower (x,y) }
```

Références

[aJRaIJ00] Grady Booch James Rumbaugh Ivar Jacobson. *Le guide de l'utilisateur UML*. 2000.

[Hoa69] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 :576–583, 1969.