Sémantiques mécanisées, quatrième cours

Des logiques pour raisonner sur les programmes

Xavier Leroy

2020-01-09

Collège de France, chaire de sciences du logiciel

Raisonner sur un programme

Vérifier (avec la rigueur d'une démonstration mathématique) que le programme ou fragment de programme ≪se comporte bien≫ :

- Correction totale : le programme termine et produit le résultat attendu.
- Correction partielle : si le programme termine, il produit le résultat attendu.
- Robustesse : le programme ne fait pas d'erreurs à l'exécution (pas de ≪plantages≫), ne fait pas fuiter d'informations confidentielles, etc.

Raisonner sur un programme

En principe : il suffit d'avoir une sémantique formelle du langage de programmation utilisé ; on raisonne alors directement sur les exécutions possibles du programme.

(Exemples dans le fichier Coq `HoareLogic.v`)

Raisonner sur un programme

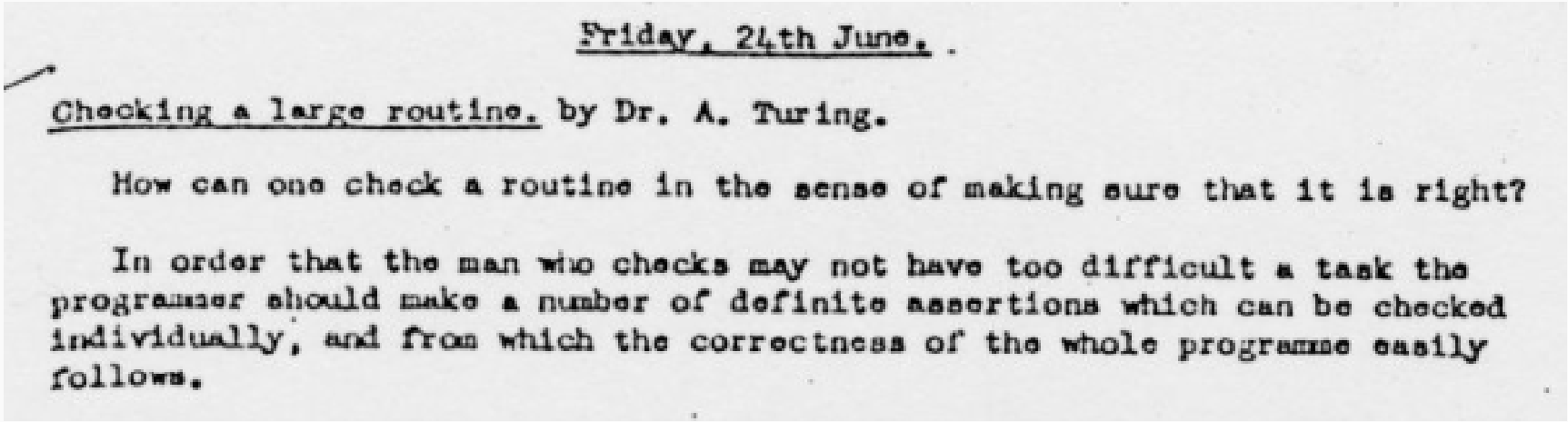
En principe : il suffit d'avoir une sémantique formelle du langage de programmation utilisé ; on raisonne alors directement sur les exécutions possibles du programme.

(Exemples dans le fichier Coq `HoareLogic.v`)

En pratique : il vaut mieux utiliser des principes de raisonnement de plus haut niveau, à savoir une logique de programmes.

Une idée aussi ancienne que l'informatique

Alan Turing, *Checking a large routine*, **1949**.



Friday, 24th June.

Checking a large routine. by Dr. A. Turing.

How can one check a routine in the sense of making sure that it is right?

In order that the man who checks may not have too difficult a task the programmer should make a number of definite assertions which can be checked individually, and from which the correctness of the whole programme easily follows.

Exposé à la *inaugural conference of the EDSAC computer*, Cambridge University, juin 1949. Le tapuscrit a été corrigé, commenté et republié par F.L. Morris et C.B. Jones dans *Annals of the History of Computing*, 6, 1984.

Le ≪gros programme≫ de Turing

Calculer $n!$ en n'utilisant que des additions.

Deux boucles imbriquées.

```
int fac (int n)
{
  int s, r, u, v;
  u = 1;
  for (r = 1; r < n; r++) {
    v = u; s = 1;
    do {
      u = u + v;
    } while (s++ < r);
  }
  return u;
}
```

Le ≪gros programme≫ de Turing

Pas de programmation structurée en 1949 ; juste des organigrammes.

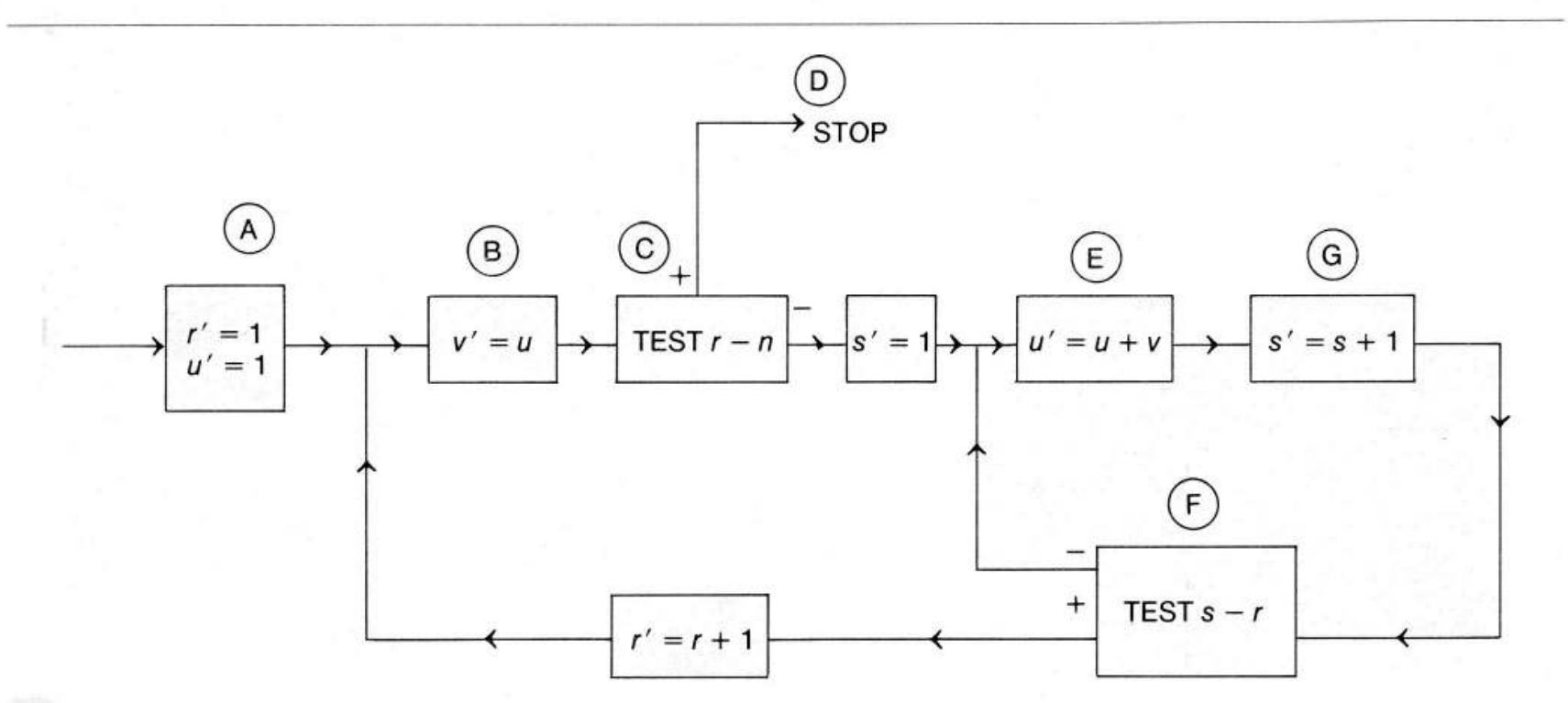


Figure 1 (Redrawn from Turing's original)

L'idée géniale de Turing

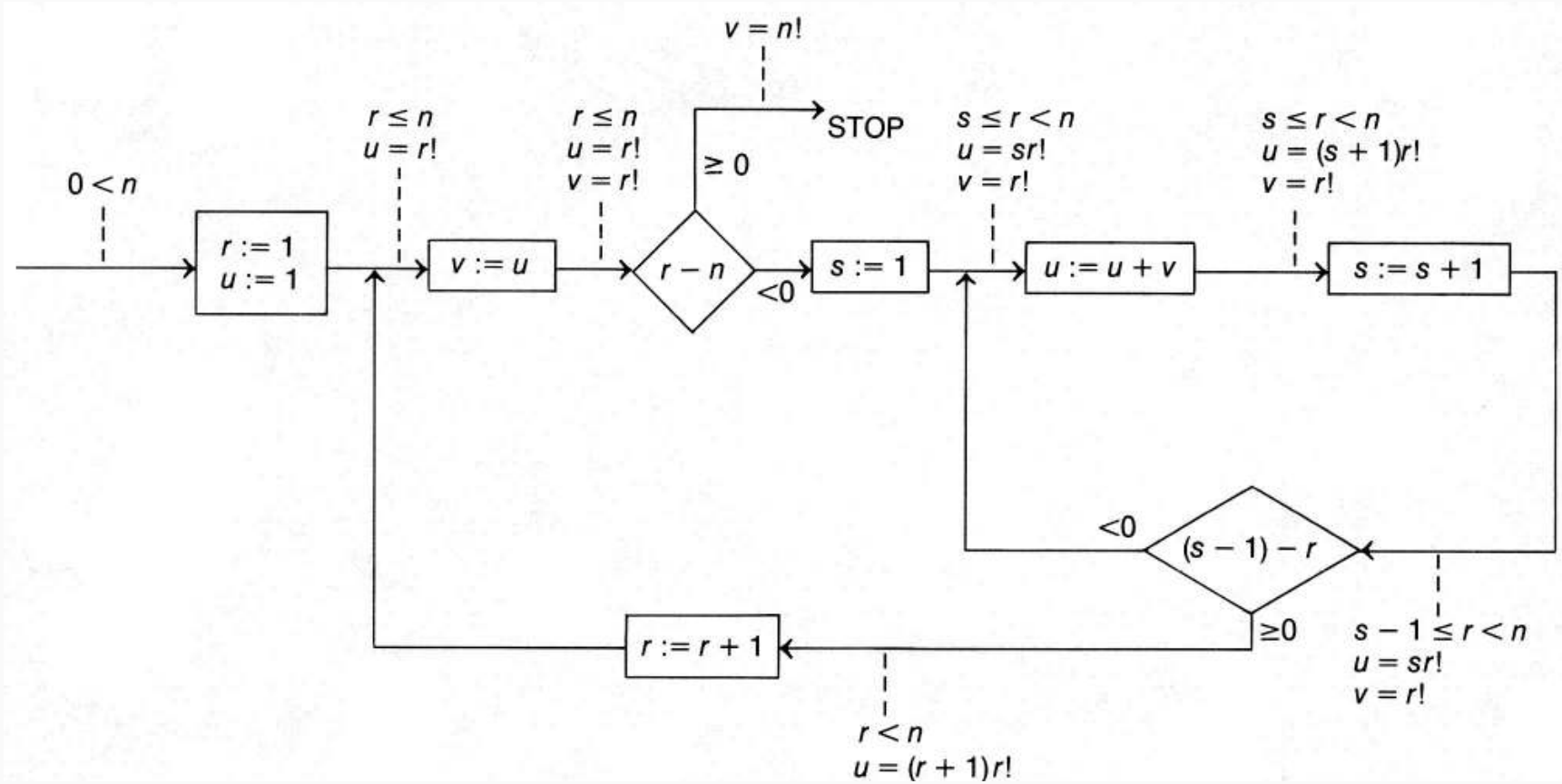
Associer à chaque point de programme un invariant logique : une relation entre les valeurs des variables qui est vraie dans toute exécution.

STORAGE LOCATION	(INITIAL) (A) $k = 6$	(B) $k = 5$	(C) $k = 4$	(STOP) (D) $k = 0$	(E) $k = 3$	(F) $k = 1$	(G) $k = 2$
27					s	$s + 1$	s
28		r	r		r	r	r
29	n	n	n	n	n	n	n
30		$\lfloor r$	$\lfloor r$		$s \lfloor r$	$(s + 1) \lfloor r$	$(s + 1) \lfloor r$
31			$\lfloor r$	$\lfloor n$	$\lfloor r$	$\lfloor r$	$\lfloor r$
	TO (B) WITH $r' = 1$ $u' = 1$	TO (C)	TO (D) IF $r = n$ TO (E) IF $r < n$		TO (G)	TO (B) WITH $r' = r + 1$ IF $s \geq r$ TO (E) WITH $s' = s + 1$ IF $s < r$	TO (F)

Figure 2 (Redrawn from Turing's original)

L'idée géniale de Turing

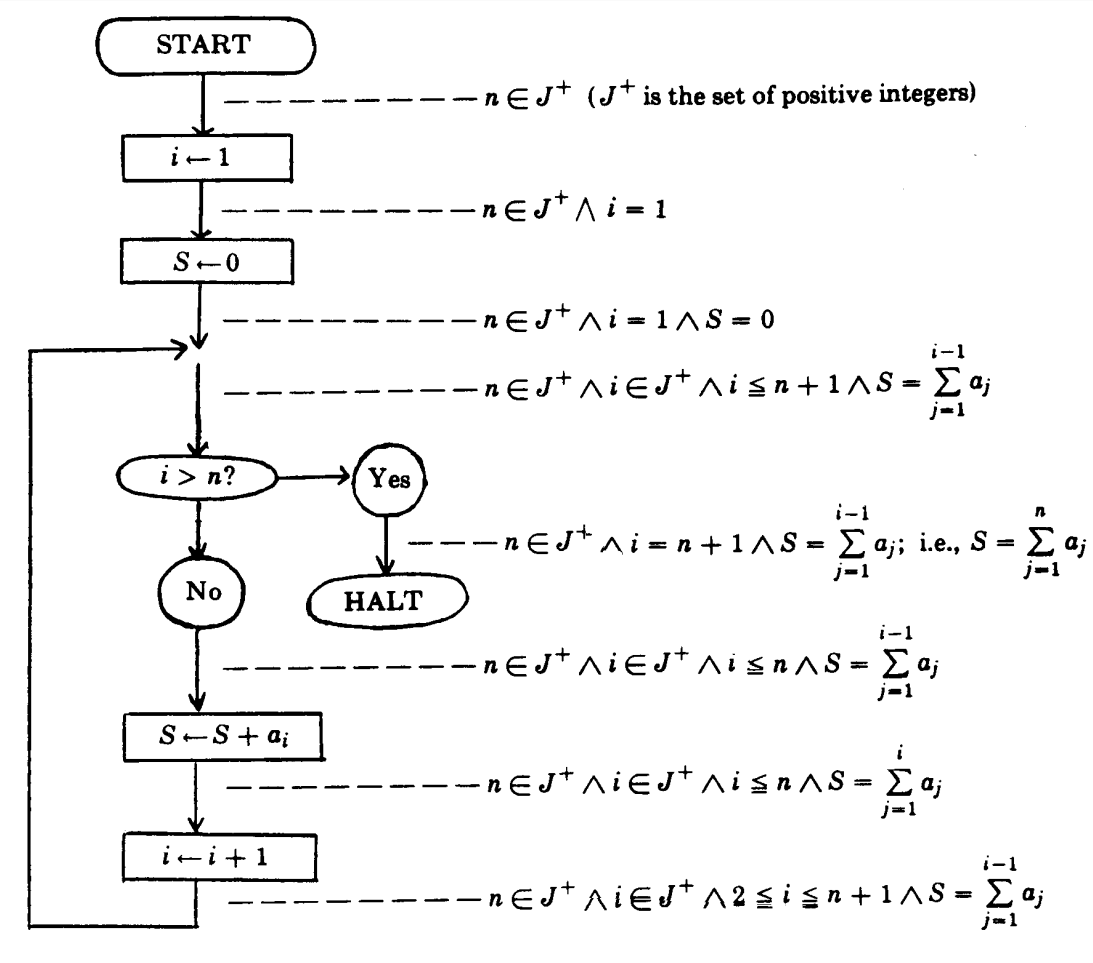
En notation plus moderne :



Pour vérifier le programme, il suffit de vérifier que chaque assertion implique (logiquement) les assertions des successeurs.

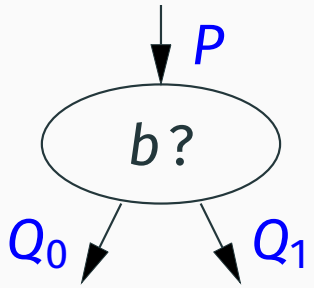
Robert Floyd, *Assigning meanings to programs*, 1967

18 ans après, Floyd réinvente et généralise l'idée de Turing.

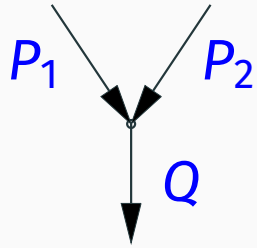


Robert Floyd, *Assigning meanings to programs*, 1967

Formalise les règles logiques qui relient les préconditions P et les postconditions Q des noeuds d'un organigramme.



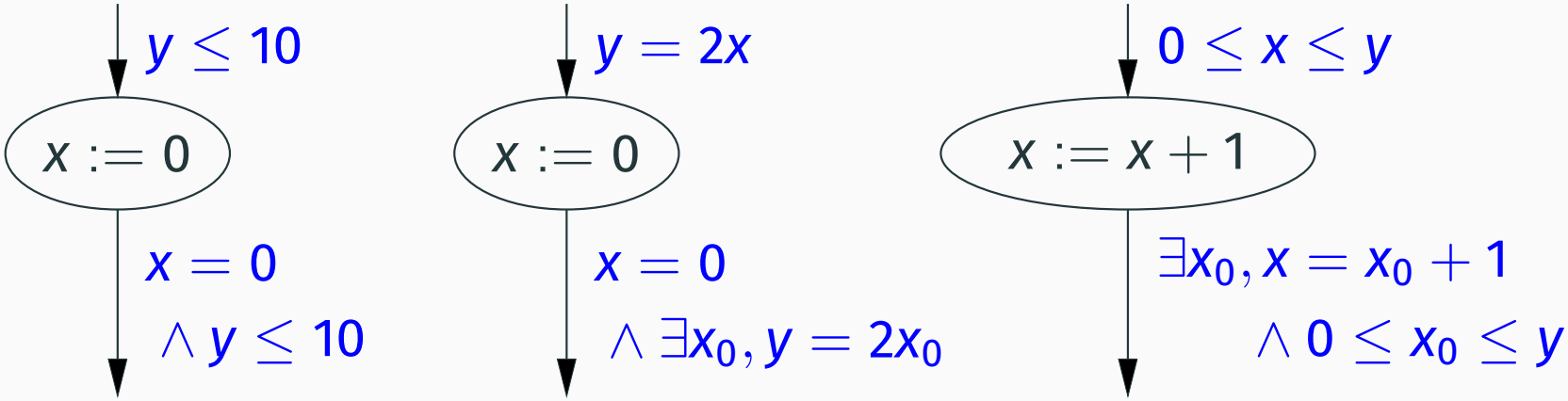
$$P \wedge \neg b \Rightarrow Q_0$$
$$P \wedge b \Rightarrow Q_1$$



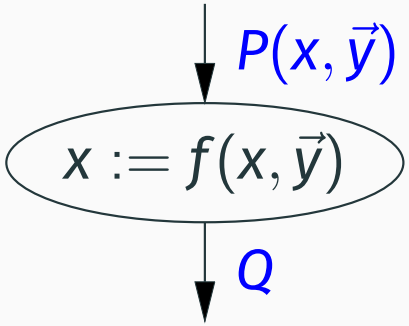
$$P_1 \vee P_2 \Rightarrow Q$$

La règle de Floyd pour l'affectation

Exemples :



Le cas général :



$$(\exists x_0, x = f(x_0, \vec{y}) \wedge P(x_0, \vec{y})) \Rightarrow Q$$

Robert Floyd, *Assigning meanings to programs*, 1967

Formalise les règles logiques pour annoter un organigramme.

Observe que ces règles définissent une sémantique du langage. (Naissance de la sémantique axiomatique).

Montre que ces règles sont correctes vis-à-vis de la sémantique opérationnelle intuitive.

Montre que ces règles sont complètes.

Esquisse un critère supplémentaire pour garantir la terminaison.

Esquisse une extension au langage Algol.

C. A. R. Hoare, *An axiomatic basis for computer programming*, 1969

Reformulation de l'approche de Floyd en termes de contrôle structuré (`if`/`then`/`else`, boucles, ...) au lieu d'organigrammes.

Présentation par axiomes et règles d'inférence
$\rightarrow$ une logique pour raisonner sur les programmes
(de même que la géométrie euclidienne est une logique pour raisonner sur les figures).

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

(Communications of the ACM, 12(10), 1969)

Les logiques de Hoare

Les triplets de Hoare

Les énoncés de la logique de Hoare :

$$[\,P\,]\ c\ [\,Q\,] \qquad\qquad \{\,P\,\}\ c\ \{\,Q\,\}$$

c : commande d'un langage impératif structuré (IMP, Algol, …)

P, Q : assertions logiques portant sur les variables du programme.

P : précondition, supposée vraie ≪avant≫ l'exécution de c

Q : postcondition, garantie vraie ≪après≫ l'exécution de c

Les triplets de Hoare

Logique de Hoare ≪forte≫ : (correction totale)

$[\,P\,]\ c\ [\,Q\,]$ si P est vraie ≪avant≫,
alors c termine et Q est vraie ≪après≫

Logique de Hoare ≪faible≫ : (correction partielle)

$\{\,P\,\}\ c\ \{\,Q\,\}$ si P est vraie ≪avant≫ et si c termine,
alors Q est vraie ≪après≫

Les règles de la logique de Hoare faible

Contrôle structuré :

$$\frac{\{\, P \,\}\ c_1\ \{\, Q \,\} \qquad \{\, Q \,\}\ c_2\ \{\, R \,\}}{\{\, P \,\}\ c_1; c_2\ \{\, R \,\}}$$

$$\frac{\{\, P \wedge b \,\}\ c_1\ \{\, Q \,\} \qquad \{\, P \wedge \neg b \,\}\ c_2\ \{\, Q \,\}}{\{\, P \,\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{\, Q \,\}}$$

$$\frac{\{\, P \wedge b \,\}\ c\ \{\, P \,\}}{\{\, P \,\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{\, P \wedge \neg b \,\}}$$

Les règles de la logique de Hoare

Commande vide :

$$\{ P \}\ \texttt{SKIP}\ \{ P \}$$

Affectation :

$$\{ Q[x \leftarrow a] \}\ x := a\ \{ Q \}$$

Notons le style «en arrière» : la postcondition Q détermine la précondition.

Exemple

$$\{ 0 = 0 \wedge y \leq 10 \}\ x := 0\ \{ x = 0 \wedge y \leq 10 \}$$

$$\{ 1 \leq x + 1 \leq 11 \}\ x := x + 1\ \{ 1 \leq x \leq 11 \}$$

La règle de conséquence

Permet d'oublier des préconditions et des postconditions inutilisées, et de les remettre en forme.

$$\frac{P \Rightarrow P' \qquad \{\, P' \,\}\ c\ \{\, Q' \,\} \qquad Q' \Rightarrow Q}{\{\, P \,\}\ c\ \{\, Q \,\}}$$

Exemple

$$\frac{\begin{array}{c} 0 \leq x \leq 10 \Rightarrow 1 \leq x + 1 \leq 11 \\ \{\, 1 \leq x + 1 \leq 11 \,\}\ x := x + 1\ \{\, 1 \leq x \leq 11 \,\} \\ 1 \leq x \leq 11 \Rightarrow 1 \leq x \leq 11 \end{array}}{\{\, 0 \leq x \leq 10 \,\}\ x := x + 1\ \{\, 1 \leq x \leq 11 \,\}}$$

La logique de Hoare forte

Mêmes règles que la logique faible, sauf pour les boucles.

$$[P]\ \texttt{SKIP}\ [P] \qquad [Q[x \leftarrow a]]\ x := a\ [Q]$$

$$\frac{[P]\ c_1\ [Q] \qquad [Q]\ c_2\ [R]}{[P]\ c_1; c_2\ [R]} \qquad \frac{[P \wedge b]\ c_1\ [Q] \qquad [P \wedge \neg b]\ c_2\ [Q]}{[P]\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ [Q]}$$

$$\frac{P \Rightarrow P' \qquad [P']\ c\ [Q'] \qquad Q' \Rightarrow Q}{[P]\ c\ [Q]}$$

Vérifier la terminaison des boucles

Pas de règle générale, mais une règle qui suffit souvent :
une expression V (la ≪variante≫) à valeurs entières positives décroît strictement à chaque tour de boucle.

$$\frac{\forall n \in \mathbb{Z},\ [P \wedge b \wedge V = n]\ c\ [P \wedge 0 \leq V < n]}{[P]\ \texttt{while}\ b\ \texttt{do}\ c\ [P \wedge \neg b]}$$

Mécanisation de la logique de Hoare

Comment représenter les assertions logiques ?

Par un langage dédié avec sa propre syntaxe et sémantique.

(plongement profond)

Termes : $t ::= x \mid 0 \mid 1 \mid t_1 + t_2 \mid \ldots$

Assertions : $P, Q ::= t_1 = t_2 \mid P \wedge Q \mid \forall x.P \mid \ldots$

Par un prédicat de la logique de Coq. *(plongement superficiel)*

$$P, Q : \texttt{store} \to \texttt{Prop}$$

Exemple : $\ll 0 \leq x < y \gg$ devient `fun s -> 0 <= s "x" < s "y"`.

(Voir le fichier Coq `HoareLogic`, sections 4.2 et 4.3)

Triplets syntaxiques, triplets sémantiques

Approche syntaxique : $\{ P \}\ c\ \{ Q \}$ est dérivable à partir des axiomes et des règles de la logique de Hoare.

Approche sémantique : $\{ P \}\ c\ \{ Q \}$ est vrai ssi
$\forall s, s',\ P\ s \wedge c/s \Downarrow s' \Rightarrow Q\ s'$.

Les deux notions sont équivalentes !

- Correction : si $\{ P \}\ c\ \{ Q \}$ est dérivable par les règles de Hoare, il est sémantiquement vrai.
- Complétude relative : si $\{ P \}\ c\ \{ Q \}$ est sémantiquement vrai, il est dérivable par les règles de Hoare.

(Voir le fichier Coq `HoareLogic`, sections 4.4 et 4.5)

Automatiser la logique de Hoare

En général, on ne peut pas décider automatiquement la validité d'un triplet de Hoare.
(Le triplet $\{ \mathit{True} \}\ c\ \{ \mathit{False} \}$ est valide ssi c ne termine pas.)

Cependant, de nombreuses étapes de déduction en logique de Hoare sont guidées par la syntaxe de la commande. P.ex :

$$\{ P \}\ x_1 := a_1; \ldots; x_n := a_n\ \{ Q \}$$

On peut appliquer la règle d'affectation n fois puis la conséquence à gauche, obtenant la condition de vérification

$$P \Rightarrow Q[x_n \leftarrow a_n][\cdots][x_1 \leftarrow a_1]$$

C'est une formule en logique du premier ordre qui se prête à la démonstration automatique ou assistée.

Génération des conditions de vérification

Soit c une commande où les boucles `while` sont manuellement annotées par un invariant de boucle *Inv*.

On peut produire automatiquement une formule de logique usuelle `vcgen` $P\ c\ Q$ qui est vraie si et seulement si le triplet $\{\, P \,\}\ c\ \{\, Q \,\}$ est valide en logique de Hoare.

C'est l'approche suivie par les outils de vérification déductive tels que ESC/Java, Frama-C, KeY, Why3, …

(Voir le fichier Coq `HoareLogics`, section 4.8.)

Extension aux tableaux

Incorrect : $\{\, Q\{t[a_1] \leftarrow a_2\}\,\}\ t[a_1] := a_2\ \{\, Q\,\}$ ✘

(Il n'y a pas que $t[a_1]$ qui est modifié, mais aussi $t[a]$ pour tout a qui a la même valeur que a_1.)

Correct : $\{\, Q\{t \leftarrow t[a_1 \mapsto a_2]\}\,\}\ t[a_1] := a_2\ \{\, Q\,\}$ ✔

L'expression $t[a_1 \mapsto a_2]$ dénote un tableau égal à t sauf que l'indice a_1 a la valeur a_2.

On raisonne sur ces expressions de tableaux avec l'équation

$$(t[a_1 \mapsto a_2])\,[a] = \begin{cases} a_2 & \text{si } a = a_1 \\ t[a] & \text{si } a \neq a_1 \end{cases}$$

Extension aux pointeurs et à l'allocation dynamique

Exemple : listes simplement chaînées.


```
typedef struct cell * list;
struct cell { int head; list tail; };
```

Concaténation en place des listes `l1` et `l2` :

```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

Extension aux pointeurs et à l'allocation dynamique

Exemple : listes simplement chaînées.


```
typedef struct cell * list;
struct cell { int head; list tail; };
```

Concaténation en place des listes `l1` et `l2` :

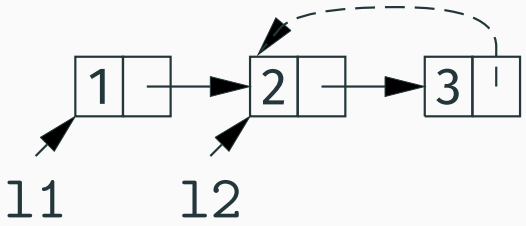
```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

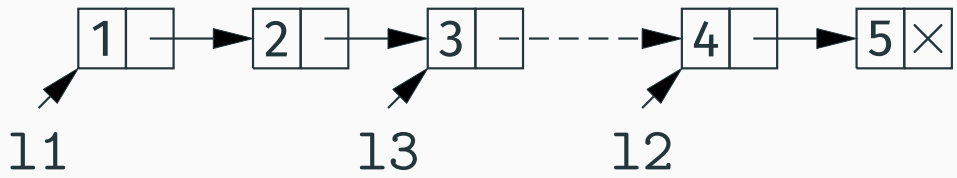
Pointeurs et structures chaînées

```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

Code difficile à vérifier… et même à spécifier !

- La liste `l1` doit être bien formée (pas circulaire) (sinon la boucle `while` ne termine pas).
- La liste `l2` ne doit partager aucune cellule avec `l1` (sinon la concaténation produit une liste circulaire).
- Toute liste `l3` qui partage avec `l1` est modifiée.





Logiques de Hoare pour les pointeurs

Folklore : un pointeur = un indice dans un grand tableau global (le ≪tas mémoire≫).

Burstall (1972), Morris (1981), Bornat (2000) :
un tableau global par champ de cellules mémoire.

$$p{\to}\texttt{tail} := a \quad \stackrel{def}{=} \quad \texttt{tail} := \texttt{tail}[p \mapsto a] \qquad (\texttt{head} \text{ est inchangé})$$

Bornat (2000), Mehta & Nipkow (2003), Hubert & Marché (2005) :
mécanisation de l'approche ≪un tableau par champ≫ ;
vérifications de l'algorithme de Schorr-Waite.

Les logiques de séparation

Appartion de la logique de séparation

Burstall (1972) : les *Distinct Nonrepeating List Systems*
($\approx$ structures simplement chaînées sans aucun partage)
+ règles de raisonnement ad-hoc.

Reynolds (1999), *Intuitionistic Reasoning about Shared Mutable Data Structures*. Introduit la notion de conjonction séparante.

O'Hearn et Pym (1999), *The Logic of Bunched Implications*.
Pour raisonner sur des ressources utilisées de manière linéaire.

O'Hearn, Reynolds, Yang (2001), *Local Reasoning about Programs that Alter Data Structures*. La présentation moderne de la logique de séparation.

Le raisonnement local

Un principe de bon sens :

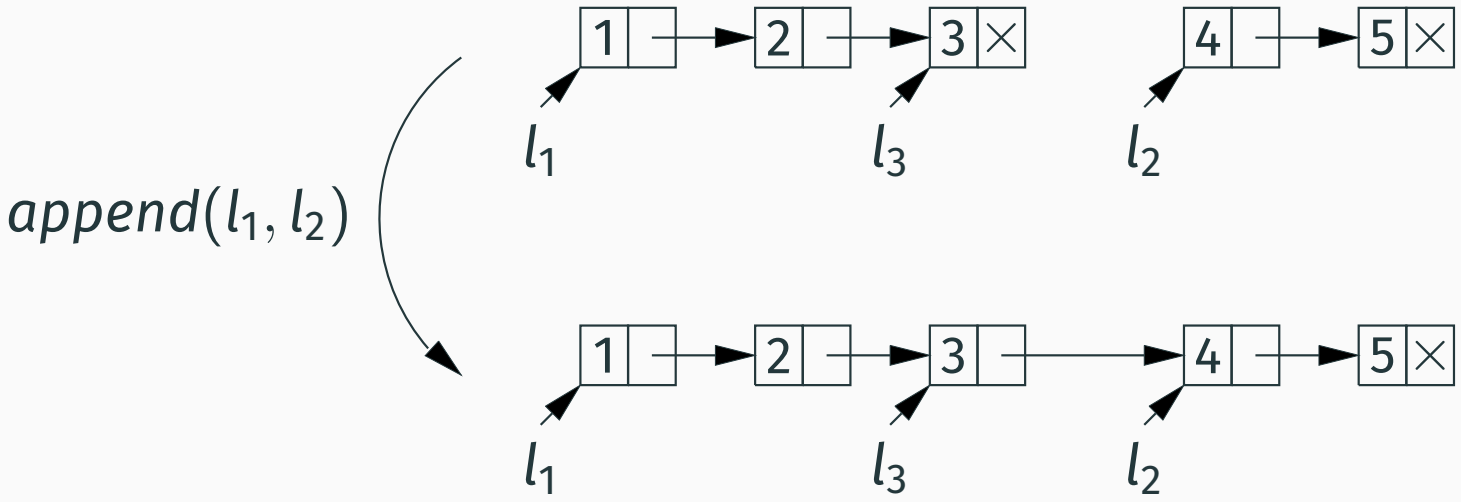
> *Tout ce qui n'est pas explicitement mentionné dans* $\{ P \}\ c\ \{ Q \}$ *est préservé par l'exécution de* c.

En logique de Hoare, ce principe se présente comme la règle d'encadrement (*frame rule*) suivante :

$$\frac{\{ P \}\ c\ \{ Q \} \qquad \text{aucune variable modifiée par } c \text{ n'apparaît dans } R}{\{ P \wedge R \}\ c\ \{ Q \wedge R \}}$$

Exemple : $\{ x = 0 \}\ x := x + 1\ \{ x = 1 \}$, et donc
$\{ x = 0 \wedge y = 8 \}\ x := x + 1\ \{ x = 1 \wedge y = 8 \}$.

Pointeurs et alias $\Rightarrow$ plus de raisonnement local



P = «l_1 représente la liste $[1, 2, 3]$ et l_2 représente la liste $[4, 5]$»

Q = «l_1 représente la liste $[1, 2, 3, 4, 5]$»

R = «l_3 représente la liste $[3]$».

$\{ P \}\ append(l_1, l_2)\ \{ Q \}$ est vrai, mais pas

$\{ P \wedge R \}\ append(l_1, l_2)\ \{ Q \wedge R \}$.

Empreintes mémoire et conjonction séparante

Chaque assertion logique P, Q a une empreinte mémoire : l'ensemble des adresses (pointeurs) dont elle décrit le contenu.

Exemple : l'assertion $p \mapsto 0$, «à l'adresse p il y a la valeur 0», a pour empreinte $\{p\}$.

La conjonction séparante $P * Q$ est vraie si et seulement si

- P est vraie (de l'état mémoire courant)
- Q est vraie (de l'état mémoire courant)
- P et Q ont des empreintes mémoires disjointes.

Exemple : $p \mapsto 0 * p \mapsto 0$ est toujours fausse.
$p \mapsto 0 * q \mapsto 0$ implique $p \neq q$.

Prédicats de représentation

À coups de conjonction séparante, on définit précisément le prédicat $list(p, L)$, ≪le pointeur p est la tête d'une liste chaînée bien formée qui représente la liste abstraite L≫ :

$$list(p, x :: L) = \exists q,\ p \mapsto \{\texttt{head} = x; \texttt{tail} = q\} * list(q, L)$$
$$list(p, nil) = p = \texttt{NULL}$$

Conjonctions séparantes $\Rightarrow$ pas de partage interne (toutes les cellules de la liste sont 2 à 2 disjointes).

L'empreinte mémoire de $\texttt{list}(p, L)$ est l'ensemble des cellules mémoire qui participent à la représentation de L.

Une spécification en logique de séparation

La concaténation de listes en place :

pour toutes listes abstraites L, L', si L est non vide,

$\{\ list(\texttt{l1}, L) * list(\texttt{l2}, L')\ \}$

```
p = l1;
while (p->tail != NULL) p = p->tail;
p->tail = l2;
```

$\{\ list(\texttt{l1}, L.L')\ \}$

Conjonction séparante dans la précondition

$\Rightarrow$ pas de partage externe entre `l1` et `l2`

(aucune cellule de liste en commun).

Rien sur `l2` dans la postcondition

$\Rightarrow$ `l2` n'est plus utilisable comme liste chaînée bien formée.

Conjonction séparante et encadrement

La *frame rule* de la logique de séparation :

$$\frac{\{\, P \,\}\ c\ \{\, Q \,\} \qquad \text{aucune variable modifiée par } c \text{ n'apparaît dans } R}{\{\, P * R \,\}\ c\ \{\, Q * R \,\}}$$

Idée de raisonnement local : P, Q décrivent les parties de la mémoire pertinentes pour l'exécution de c ; R décrit le reste.

Idée de ressources :
P décrit les ≪ressources mémoires≫ consommées par c ;
Q décrit les ressources produites ou rendues par c ;
R décrit des ressources inchangées.

Les ≪petites règles≫

Préconditions et postconditions ne parlent que du strict nécessaire à la commande.

$$
\begin{array}{rcl}
[\,a = n\,] & x := a & [\,x = n\,] \\
[\,(a = p) * (p \mapsto v)\,] & x := *a & [\,(x = v) * (p \mapsto v)\,] \\
[\,(a = p) * (a' = v) * (p \mapsto \_)\,] & *a := a' & [\,p \mapsto v\,] \\
[\,emp\,] & x := alloc(N) & [\,\exists p, (x = p) \\
 & & \quad * (p \mapsto \_) * \cdots \\
 & & \quad * (p + N - 1 \mapsto \_)\,] \\
[\,(a = p) * (p \mapsto \_)\,] & free(a) & [\,emp\,]
\end{array}
$$

$p \mapsto \_$ se lit ≪p est valide≫ et est défini comme $\exists v, p \mapsto v$.

Formalisation : un langage IMP avec pointeurs

Commandes :

$$
\begin{aligned}
c ::= {} & \texttt{SKIP} \mid x := a \mid c_1; c_2 \\
& \mid \texttt{if } b \texttt{ then } c_1 \texttt{ else } c_2 \\
& \mid \texttt{while } b \texttt{ do } c \\
& \mid x := \mathit{alloc}(N) && \text{allocation de } N \text{ mots} \\
& \mid x := *a && \text{lecture à l'adresse } a \\
& \mid *a_1 := a_2 && \text{écriture à l'adresse } a_1 \\
& \mid \mathit{free}(a) && \text{libération de l'adresse } a
\end{aligned}
$$

Les pointeurs sont des entiers $\Rightarrow$ arithmétique de pointeurs.

Exemple (Construction d'une liste)

```
l := alloc(2); *l := head; *(l + 1) := tail
```

Sémantique opérationnelle

Deux composantes de l'état mémoire :

- le *store* s : variable $\mapsto$ valeur (fonction totale)
- le *heap* h : adresse $\mapsto$ valeur (fonction partielle, finie)

Sémantique à réductions : $c/s/h \to c'/s'/h'$.

Quelques règles représentatives :

$$x := a/s/h \to \texttt{SKIP}/s[x \leftarrow [\![a]\!]\, s]/h$$

$$x := *a/s/h \to \texttt{SKIP}/s[x \leftarrow v]/h \qquad \text{si } h([\![a]\!]\, s) = v$$

$$*a := a'/s/h \to \texttt{SKIP}/s/h[[\![a]\!]\, s \leftarrow [\![a']\!]\, s] \qquad \text{si } [\![a]\!]\, s \in dom(h)$$

Une logique de séparation pour IMP

Assertions logiques = prédicats $\texttt{store} \to \texttt{heap} \to \texttt{Prop}$.

Triplets forts de base :

$$[\,P\,]\ c\ [\,Q\,] \stackrel{def}{=} \forall s, h,\ P\ s\ h \Rightarrow \exists s', h',\ c/s/h \stackrel{*}{\to} \texttt{SKIP}/s'/h' \wedge Q\ s'\ h'$$

Ne valident pas la règle d'encadrement (allocation dynamique).

Triplets forts :

$$[[\,P\,]]\ c\ [[\,Q\,]] \stackrel{def}{=} \forall R \text{ inchangé par } c, [\,P * R\,]\ c\ [\,Q * R\,]$$

Valident en plus la règle d'encadrement.

(Voir fichier Coq `SepLogic` et séminaire du 16 janvier.)

Pour aller plus loin : la logique de séparation concurrente

(O'Hearn, 2007, *Resources, Concurrency and Local Reasoning*.
Brookes, 2007, *A Semantics for Concurrent Separation Logic*.)

Contexte : parallélisme à mémoire partagée
(*threads*, multicoeurs, etc).

Règle de base : exécution parallèle sans interférences.

$$\frac{\{\, P_1 \,\}\ c_1\ \{\, Q_1 \,\} \qquad \{\, P_2 \,\}\ c_2\ \{\, Q_2 \,\}}{\{\, P_1 * P_2 \,\}\ c_1 \parallel c_2\ \{\, Q_1 * Q_2 \,\}}$$

Synchronisations et communications

Diverses règles pour rendre compte des mécanismes de synchronisation et de communication entre *threads* :

- Haut niveau : verrous, sémaphores, files de messages, …
- Bas niveau : barrières mémoire, *compare-and-swap*, *load-acquire*/*store-release*, …

Logique de séparation concurrente pour les verrous

Au verrou L on associe une assertion $INV(L)$:

- L'empreinte de $INV(L)$ décrit les cases mémoires protégées par le verrou.
- L'assertion $INV(L)$ décrit les invariants que les utilisateurs de ces cases mémoires doivent préserver.

Petites règles pour les verrous :

$$\{\, emp \,\} \quad \texttt{lock}(L) \quad \{\, INV(L) * Locked(L) \,\}$$
$$\{\, INV(L) * Locked(L) \,\} \quad \texttt{unlock}(L) \quad \{\, emp \,\}$$

Détenir le verrou = être le seul possesseur des cases protégées.
Rendre le verrou = s'obliger à rétablir l'invariant.

Point d’étape et perspectives

Point d'étape

Deux visions qui coexistent parfaitement :

- Vision axiomatique : une logique de programme définit la sémantique du langage.
- Vision opérationnelle : une logique de programme est un ensemble de théorèmes portant sur la sémantique opérationnelle du langage et qui facilitent le raisonnement sur les programmes.

Point d'étape

Des principes connus depuis longtemps,

longtemps restés purement théoriques,

mais maintenant utilisables grâce à des outils :

- Prouveurs de programmes + démonstrateurs automatiques : KeY, Frama-C WP, Infer, …
- Plongements dans Coq ou autres systèmes interactifs : CFML, IRIS, …

Perspectives

Un domaine de recherche très actif :

- Plus d'automatisation pour les preuves de programme.
  (P.ex. INFER et la ≪bi-abduction≫ ; les *shape analyses*.)
- Plus d'abstraction dans les logiques.
  (P.ex. IRIS : monoïde + invariants = logique concurrente)
- Raisonner sur d'autres sortes de ressources.
  (P.ex. systèmes de fichiers, temps d'exécution.)
- Raisonner sur le parallélisme de bas niveau.
  (Modèles mémoires faiblement cohérents.)

Bibliographie

Bibliographie

La logique de Hoare et sa mécanisation :

- Benjamin Pierce et al, *Software Foundations, volume 2 : Programming Languages Foundations*, chapitres *Hoare logic*.
- Tobias Nipkow et Gerwin Klein, *Concrete Semantics*, chap. 12.

Une introduction à la logique de séparation :

- Peter O'Hearn, *Separation Logic*, CACM 62(2), 2019.

Des cours sur la logique de séparation concurrente :

- Aleks Nanevski, *Separation Logic and Concurrency*, OPLSS 2016.
- Lars Birkedal, Ales Bizjak, *Lecture Notes on Iris : Higher-Order Concurrent Separation Logic*, 2018.