

Théorie de la Programmation

Daniel Hirschhoff

Alexis Ghyselen, Enguerrand Prebet, Alexandre Talon

<http://perso.ens-lyon.fr/daniel.hirschhoff/ThPr>

De quoi on parle, comment on parle, ce qu'on dit

- ▶ ce dont il est question
langages de programmation
- ▶ **sémantique** des langages de programmation
méthode/techniques
- ▶ raisonnement : **établir des propriétés**

- ▶ d'un programme

ce programme calcule la racine cubique d'un nombre flottant

ce programme calcule le 4-coloriage d'un graphe planaire

si un programme n'a pas de bug, sa version compilée non plus

- ▶ d'un langage

dans ce langage, tout programme termine en un temps polynômial en la taille de son argument

domaine de recherche / enjeux industriels

Programmations

- ▶ **programmation impérative** C, C++, Java, ...
l'immense majorité des langages
- ▶ **programmation fonctionnelle** OCaml, Haskell, Scheme, ...
l'immense minorité des langages ?!
 - ▶ style différent par rapport à l'impératif
 - ▶ proximité avec les méthodes développées dans ce cours
- ▶ dépasser les clivages
 - ▶ Rust, Python : langages “multi-paradigmes”
 - ▶ les nouveaux langages sont souvent nourris
 - ▶ de besoins concrets (accès au bas-niveau (C), faire des stats (R), temps-réel (Lustre), *domain-specific languages*, ...)
 - ▶ d'avancées conceptuelles
- ▶ **pour ce cours**, on élague, pour présenter les idées fondamentales : le cœur d'un langage impératif, d'un langage fonctionnel, les différences

IMP, un petit langage impératif

- ▶ programmes

```
X := Y+3*Z
```

```
X := 52 ; Y := 32
```

```
if Z>3 then X := 32 else Y := 52
```

```
while C-1<Y do ( X := X+1; C := C * X )
```

- ▶ uniquement des **entiers**

- ▶ pas de flottants, de chaînes de caractères, de pointeurs, de tables de hachage, ...
- ▶ des entiers dans $\mathbb{Z}$ (voire $\mathbb{N}$)

Commençons par de la pratique

- ▶ point de départ : l'**induction**
(définitions inductives, raisonnements par induction)
 - ▶ une généralisation de ce que vous pratiquez depuis que vous avez appris à faire des raisonnements par récurrence
 - ▶ des raisonnements rigoureux en un sens “extrême” : faits sur machine l'**assistant de preuves Coq**
- ▶ considérations pratiques : **d'ici le premier TP** (lundi prochain)
 - ▶ soit vous prévoyez d'utiliser votre ordinateur portable, auquel cas installez-y
 1. le système de preuves **Coq**
 2. l'outil **coqide** pour développer des preuves
 - ▶ soit vous prévoyez d'utiliser les machines libre-service, auquel cas ouvrez une session et vérifiez que coqide se lance
en cas de difficulté : mail aux chargés de TP
- **TP** : répartissez-vous en 3 groupes
- **TD** : 2 groupes (1.5 + 1.5)

Premiers pas en Coq

RB-INSERT(T, z)

```
1   $y = T.nil$ 
2   $x = T.root$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11 elseif  $z.key < y.key$ 
12      $y.left = z$ 
13 else  $y.right = z$ 
14  $z.left = T.nil$ 
15  $z.right = T.nil$ 
16  $z.color = RED$ 
17 RB-INSERT-FIXUP( $T, z$ )
```

In this book, we shall typically describe algorithms as programs written in a pseudocode that is similar in many respects to C, C++, Java, Python, or Pascal. If you have been introduced to any of these languages, you should have little trouble reading our algorithms. What separates pseudocode from “real” code is that in pseudocode, we employ whatever expressive method is most clear and concise to specify a given algorithm. Sometimes, the clearest method is English, so do not be surprised if you come across an English phrase or sentence embedded within a section of “real” code.

“Informal proofs are algorithms; formal proofs are code.”

Retour sur Coq

► tactiques

`reflexivity`, `induction n`, `simpl`, `rewrite ->` et `rewrite <-` (avec une hypothèse, avec un lemme), `intros`, `trivial`, `destruct`,
`apply`, `split`, `exists x`
`assert`, `replace` : introduire une étape de preuve
`Admitted` : tricher (n'est pas une tactique)

► la curryfication

savoir lire $A \rightarrow B \rightarrow C$

► listes : on fait une **induction sur la structure d'une liste**

- ce principe de preuve vient de la définition des listes
- pas une induction sur la taille de la liste
- au fond, cela revient au même / c'est une question de style

IMP : grammaire, sémantique dénotationnelle

► au tableau / sur papier

► en Coq

à l'image de

natlist
nat

on définit

comm
bexpr
aexpr
nat

► sémantique dénotationnelle de IMP

Coq, suite
négation, booléens

Sur les définitions inductives

- ▶ la récurrence sur les entiers naturels
 - on a dit que ça correspond à la définition des entiers de Peano
- ▶ on a généralisé, en disant que c'est un cas particulier d'une **définition inductive d'ensemble** avec des constructeurs
 - cf. les listes, les arbres, les booléens, les programmes IMP, ...
- ▶ on a généralisé, en disant que c'est un cas particulier d'une **définition inductive de relation**
 - ▶ ensemble = relation 0-aire
 - ▶ prédicat = relation unaire
 - ▶ on est habitué aux relations binaires
 - ▶ relations n -aires fréquemment utilisées en sémantique

Vendredi 27 septembre

13h30-15h30 : TD, en deux groupes

Règles d'inférence, terminologie

Les règles d'inférence sont une notation.

- **jugement** : manière de noter un tuple

$$c, \sigma \longrightarrow c', \sigma' \quad \underline{\text{Com}} \times M \times \underline{\text{Com}} \times M \quad n \geq k \quad \text{nat} \times \text{nat}$$

le jugement que l'on définit est en conclusion de toutes les règles d'inférence correspondantes

- **metavariabes**

$$C_{\text{seq}} \frac{c, \sigma \longrightarrow c', \sigma'}{c; c_2, \sigma \longrightarrow c'; c_2, \sigma'} \quad \text{pour tous } c \ \sigma \ c' \ \sigma' \ c_2 \dots$$

| Cseq : forall c sigma c' sigma', (pp c sigma c' sigma') ->
forall c2, (pp (seq c c2) sigma (seq c' c2) sigma')

- construire une **dérivation**

- “emboîter” les règles d'inférences en instanciant les metavariables
- arbre *fini* avec des axiomes aux feuilles

Règles d'inférence, anatomie

► prémisses uniquement des relations inductives

(**pré-existantes**, ou bien celle que l'on est en train de définir)

$$C_{aff} \frac{a, \sigma \longrightarrow a'}{X := a, \sigma \longrightarrow X := a', \sigma}$$

$$C_{seq} \frac{c, \sigma \longrightarrow c', \sigma'}{c; c_2, \sigma \longrightarrow c'; c_2, \sigma'}$$

- en particulier, pas de “non ($c_3, \sigma \longrightarrow c'_3, \sigma'$)”
- plusieurs prémisses : interprété comme une conjonction

p.ex.
$$\frac{a_1, \sigma \longrightarrow a'_1 \quad a_2, \sigma \longrightarrow a'_2}{a_1 + a_2, \sigma \longrightarrow a'_1 + a'_2}$$

► condition d'application (*side condition*)

$$A_p \frac{}{k_1 + k_2, \sigma \longrightarrow k} \quad \begin{array}{l} k \text{ est le résultat} \\ \text{de la somme} \\ \text{de } k_1 \text{ et } k_2 \end{array} \quad | \quad \begin{array}{l} A_p : \text{ forall } k_1 \ k_2 \ \text{sigma } k, \\ k=k_1+k_2 \rightarrow \quad (\text{ppa } (\text{add } (\text{cst } k_1) \\ (\text{cst } k_2)) \ \text{sigma } (\text{cst } k)) \end{array}$$

- ne contient pas de relation définie inductivement
- dit quelque chose “en maths” qui n’est pas formalisé
Coq est absolutiste de ce point de vue
- dans une dérivation, on vérifie “à côté” que les conditions d’application sont satisfaites

(une manière d') Exécuter un programme IMP - petits pas

X:=Y+4; if $Y > 5$ then Y:=0 else X:=0

X	2
Y	3

→ X:=3+4; if $Y > 5$ then Y:=0 else X:=0

X	2
Y	3

→ X:=7; if $Y > 5$ then Y:=0 else X:=0

X	2
Y	3

→ skip; if $Y > 5$ then Y:=0 else X:=0

X	7
Y	3

→ if $Y > 5$ then Y:=0 else X:=0

X	7
Y	3

→ if $3 > 5$ then Y:=0 else X:=0

X	7
Y	3

→ if false then Y:=0 else X:=0

X	7
Y	3

→ X:=0

X	7
Y	3

→ skip

X	0
Y	3

Définitions inductives : ensembles, relations

Inductive `bexpr` : Set := ... on définit `bexpr`

Inductive `com` : Set := on définit `com`
.. | ite : `bexpr` -> `com` -> `com` -> `com` ..

on a défini $\longrightarrow$ ($a, \sigma \longrightarrow a'$), on définit $\longrightarrow$

$$\frac{a, \sigma \longrightarrow a'}{X := a, \sigma \longrightarrow X := a', \sigma}$$

$$\frac{c, \sigma \longrightarrow c', \sigma'}{c; c1, \sigma \longrightarrow c'; c1, \sigma'}$$

ou si l'on veut (à la notation près) :

$$\frac{(a, \sigma, a') \in \longrightarrow}{(X := a, \sigma, X := a', \sigma) \in \longrightarrow}$$

$$\frac{(c, \sigma, c', \sigma') \in \longrightarrow}{(c; c1, \sigma, c'; c1, \sigma') \in \longrightarrow}$$

Deux usages de Knaster-Tarski

constructeurs $\longleftrightarrow$ règles d'inférence

ensembles d'*objets*
(termes) $\longleftrightarrow$ ensembles de dérivations
pour un jugement

induction structurelle $\longleftrightarrow$ induction sur la dérivation

$\forall x. \mathcal{P}(x)$

$\forall x_1, \dots, x_n.$

$\rho(x_1, \dots, x_n) \implies \mathcal{P}(x_1, \dots, x_n)$

- “ $\rho(x_1, \dots, x_n) \implies \dots$ ”
se lit “pour toute dérivation de $\rho(x_1, \dots, x_n)$, ...”
- un cas par règle d'inférence
- dans chaque cas,
 - ▶ chaque prémisses de la forme $\rho(x_1, \dots, x_n)$ est vérifiée, et
 - ▶ la propriété $\mathcal{P}$ vaut pour les prémisses de la forme $\rho(\dots)$

exemple : $\forall c, \sigma, c', \sigma'. \underbrace{(c, \sigma \longrightarrow c', \sigma')}_{\rho(c, \sigma, c', \sigma')} \implies \mathcal{P}(c, \sigma, c', \sigma')$

Passages obligés

directives pour la rédaction

Passages obligés – définitions par induction

en rouge, ce qui est exigé dans une copie

- ▶ on se donne
des ensembles pré-existants, des relations pré-existantes
- ▶ on définit par induction
 - soit un ensemble, donné par les constructeurs suivants ...
(ou par la grammaire suivante ...)
 - soit un jugement,
noté ..., *[[précisions éventuelles sur les metavariables apparaissant dans le jugement]]*,
(p.ex. noté $\sigma, c \Downarrow \sigma'$, où c est une commande et σ, σ' sont des états mémoire)
donné par les règles d'inférence suivantes ...

Passages obligés – preuves par induction

en rouge, ce qui est exigé dans une copie

- ▶ on démontre l'énoncé suivant: *[[écrire l'énoncé]]* par induction
 - ▶ sur la structure de toto
 - lorsque toto est un élément d'un ensemble E défini inductivement
 - et l'énoncé est de la forme $\forall toto \in E, blabla$ ou bien
 - ▶ sur la dérivation de titi
 - lorsque titi est un jugement défini inductivement
 - et l'énoncé est de la forme $\forall.. \forall.., (titi) \Rightarrow blabla$
 - ▶ il y a k cas:
 - cas 1: *ici il faut être raisonnable: détailler les cas qui le méritent, regrouper les cas qui sont traités identiquement*
 - cas k: *on n'est pas des gallinacés*
- l'hypothèse d'induction dit *blabla*, ce qui permet de déduire... (énoncer l'hypothèse d'induction)

Passages obligés – définition de fonctions par induction

en rouge, ce qui est exigé dans une copie

lorsque E est (un ensemble/ un prédicat) défini par induction

on définit la fonction f par induction sur son argument $toto \in E$
il y a k cas

NB: $toto$ peut aussi être une dérivation

Récapitulons

Ensemble	nombre de constructeurs	
nat	2	entiers de Peano
<u>Arith</u>	3	expressions arithmétiques
<u>Bool</u>	4	expressions booléennes
<u>Com</u>	5	commandes
Relation	nombre de règles d'inférence	
$a, \sigma \longrightarrow a'$	4	petits pas pour <u>Arith</u>
$b, \sigma \longrightarrow b'$	6	petits pas pour <u>Bool</u>
$c, \sigma \longrightarrow c', \sigma'$	8	petits pas pour <u>Com</u>
$a, \sigma \Downarrow k$	3	grands pas pour <u>Arith</u>
$b, \sigma \Downarrow t$	4	grands pas pour <u>Bool</u>
$c, \sigma \Downarrow \sigma'$	7	grands pas pour <u>Com</u>

Partiel le mardi 22 ou mercredi 23 octobre

- *programme : tout jusqu'à la correction de la logique de Hoare incluse (Coq compris)*
- *documents autorisés : rien d'électronique (livres, notes de cours/TD : ok)*
- *le sujet de l'année dernière est disponible depuis la page [www du cours](#)*
- *vous y trouverez aussi un glossaire et des notes sur définitions inductives*

Ce qui a été vu jusqu'ici

Sémantiques

- . dénotationnelle
- . petits pas
- . grands pas

Ordres partiels et points fixes

- . justifier définitions inductives récursives et preuves par induction
- . calculer la sémantique dénotationnelle

- ▶ ce que sont les programmes et leurs exécutions
- ▶ on peut étendre cette approche à des langages plus riches
 - * / repeat for function(..) throw goto
 - certaines constructions sont plus difficiles à traiter que d'autres avec cette méthode
- ▶ on a raisonné sur *tous les programmes d'un langage*

Logique de Floyd-Hoare

Un exemple

si l'état initial vérifie

$$K = n \wedge K \geq 0$$

alors en exécutant le programme

```
R := 1;  
while (K>1) do  
  ( R:= R*K; K := K-1 )
```

on aboutit à un état qui vérifie

$$R = n!$$

Un autre exemple

si l'état initial vérifie

$$X1 \geq 1 \wedge X2 \geq 1$$

alors en exécutant le programme

```
Y1 := X1;
```

```
Y2 := X2;
```

```
while (Y1  $\neq$  Y2) do
```

```
  if Y1 > Y2 then Y1 := Y1 - Y2
```

```
    else Y2 := Y2 - Y1
```

on aboutit à un état qui vérifie

$$Y1 = \text{pgcd}(X1, X2)$$

$$(Y1 \text{ div } X1) \wedge (Y1 \text{ div } X2) \wedge (\forall i. (i \text{ div } X1) \wedge (i \text{ div } X2) \Rightarrow i \leq Y1)$$

Logique de Floyd-Hoare

- R. W. Floyd. *Assigning meanings to programs*.
In Mathematical Aspects of Computer Science, volume 19 of
Proceedings of Symposia in Applied Mathematics, pages
19–32, American Mathematical Society.
1967.
- C. A. R. Hoare. *An axiomatic basis for computer
programming*.
Communications of the ACM, 12(10):576–580 and 583.
1969

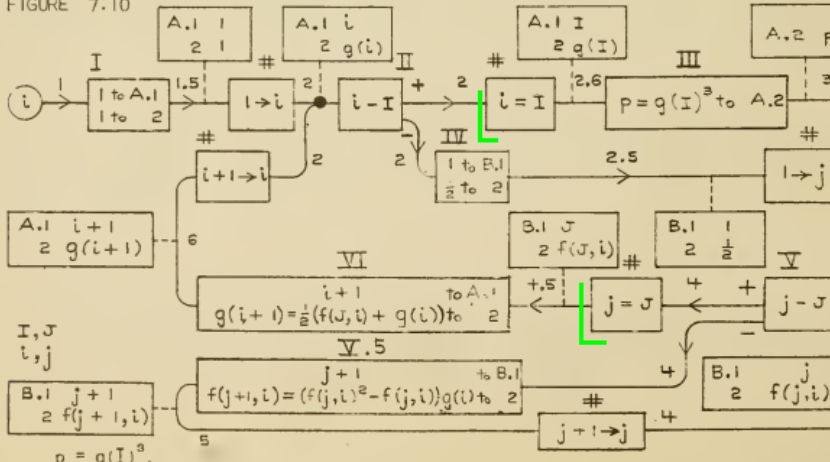
H. Goldstine and J. von Neumann, *Planning and coding of problems for an electronic computing instrument*

Report on the Mathematical and Logical aspects of an Electronic Computing Instrument, Institute for Advanced Study, Princeton, 1947

mélange de ce qu'on fait et de ce qu'on dit



FIGURE 7.10



Logique de Hoare, validité, dérivabilité

programmeurs

X, Y, Z

$X = 3$

$X := 3$



logiciens

i, j

$X = 3$



$\vdash \{A\} c \{A'\}$

énoncés, affirmations
(en logique de Hoare)

$\models \{A\} c \{A'\}$

ce qui "est vrai"
(ce qui se passe)

► correction: $\vdash \Rightarrow \models$

tout ce que je dis est vrai

► complétude: $\models \Rightarrow \vdash$

tout ce qui est vrai, je sais le dire

Dans l'autre sens

- ▶ scénario: étant donnés A , A' , c , déterminer (*aussi*) automatiquement (*que possible*) si $\vdash \{A\} c \{A'\}$ est dérivable
- ▶ deux manières de répondre
 1. réponse pragmatique
 2. réponse théorique

Réponse pragmatique: conditions de vérification

- ▶ où faut-il “deviner” en logique de Hoare?
 - ▶ règle de la séquence: l'assertion intermédiaire
 - ▶ règle du `while`: l'invariant
 - ▶ règle de conséquence
 - nota: en pratique, rien de très méchant ne se cache habituellement dans les usages de la conséquence
- ▶ **outils** pour la preuve de programmes impératifs:
 - ▶ l'utilisateur insère des jalons, qui sont des annotations
 - ▶ $c_0; \{A\}c_1$ (lorsque c_1 n'est pas une affectation)
 - ▶ `while b do {A}c` A est ici l'invariant
 - ▶ le système engendre des obligations de preuve (des $\models A \Rightarrow A'$), qui sont traitées de manière plus ou moins automatisée
 - ▶ à la fin, une dérivation est construite en insérant des usages de la règle de conséquence aux bons endroits

Vérification de programmes : l'outil Why3 (Univ. Orsay)

```
module AmortizedQueue

use import int.Int
use import option.Option
use import list.ListRich

type queue 'a = { front: list 'a; lenf: int;
                  rear : list 'a; lenr: int; }
invariant { length front = lenf >= length rear = lenr }
by { front = Nil; lenf = 0; rear = Nil; lenr = 0 }

function sequence (q: queue 'a) : list 'a =
  q.front ++ reverse q.rear

let empty () : queue 'a
  ensures { sequence result = Nil }
= { front = Nil; lenf = 0; rear = Nil; lenr = 0 }

let head (q: queue 'a) : 'a
  requires { sequence q <> Nil }
  ensures { hd (sequence q) = Some result }
= let Cons x _ = q.front in x

let create (f: list 'a) (lf: int) (r: list 'a) (lr: int) : queue 'a
  requires { lf = length f /\ lr = length r }
  ensures { sequence result = f ++ reverse r }
= if lf >= lr then
  { front = f; lenf = lf; rear = r; lenr = lr }
else
  let f = f ++ reverse r in
  { front = f; lenf = lf + lr; rear = Nil; lenr = 0 }

let tail (q: queue 'a) : queue 'a
  requires { sequence q <> Nil }
  ensures { tl (sequence q) = Some (sequence result) }
= let Cons _ r = q.front in
  create r (q.lenf - 1) q.rear q.lenr

let enqueue (x: 'a) (q: queue 'a) : queue 'a
  ensures { sequence result = sequence q ++ Cons x Nil }
= create q.front q.lenf (Cons x q.rear) (q.lenr + 1)

end
```

- programme décoré

- *verification conditions*

- systèmes de preuve
(automatiques ou pas)

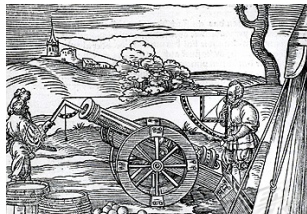
Partiel : mardi 22/10 de 10h15 à 12h15, amphi B

- . programme : tout jusqu'à demain inclus*
- . tous documents papier autorisés*

Des pré-assertions aux assertions

Complétude, réponse théorique: plus faible précondition

- ▶ affectation: Hoare raisonne “en remontant le temps”
- ▶ on généralise à une commande quelconque:
 - ▶ on peut voir une assertion A comme un *ensemble d'états mémoire*: $A \leftrightarrow \{\sigma \text{ t.q. } \sigma \models A\}$
 - ▶ un **triplet incomplet** $\{?\} c \{B\}$ aussi
on définit $wp(B, c) = \{\sigma \text{ t.q. } c, \sigma \Downarrow \sigma' \text{ implique } \sigma' \models B\}$
 - ▶ intuitivement, $\{?\} c \{B\}$
 - ▶ $wp(B, c)$ est un ensemble d'états mémoire, *pas une assertion*
- ▶ enjeu de la preuve de complétude (relative)
 - ▶ définir une assertion $W(B, c)$
exprimant $wp(B, c)$
 $\forall \sigma. \sigma \models W(B, c) \text{ ssi } \sigma \in wp(B, c)$
la logique est **expressive**



Expressivité de la logique de Hoare (esquisse)

juste les grandes idées: on définit $\mathcal{W}(B, c)$ par induction sur c

- ▶ c'est exactement ce que dit la règle pour l'affectation
- ▶ c'est aisé, par induction, pour les autres constructions ...
- ▶ ...sauf while

définir $\mathcal{W}(B, \text{while } b \text{ do } c)$ sachant qu'on sait,
par induction, définir $\mathcal{W}(B', c)$ pour tout B' ?

on s'appuie sur la caractérisation suivante:

Lemme: $\forall \sigma, \sigma', b, c, V. \quad \sigma \in wp(B, \text{while } b \text{ do } c) V$
si et seulement si

$\forall k, \sigma_0, \dots, \sigma_k.$

$$[\sigma_0 = \sigma \wedge \forall i < k. (b, \sigma_i \Downarrow \text{true} \wedge c, \sigma_i \Downarrow \sigma_{i+1})]$$
$$\implies (b \vee B), \sigma_k \Downarrow \text{true}$$

- ▶ on peut de fait définir des assertions pour
 - ▶ décrire un état mémoire $\sigma_i \quad (X_1^i = v_1 \wedge \dots \wedge X_n^i = v_n)$
 - ▶ exprimer les conditions $c, \sigma_i \Downarrow \sigma_{i+1}$ par induction
 - ▶ exprimer la quantification $\forall k, \sigma_0, \dots, \sigma_k$
- on téléphone à Kurt Gödel (voir livre de Winskel)

Fin de la preuve — complétude, complétude relative

- ▶ on dispose d'une assertion $W(B, c)$ pour tous B, c
 $W(B, c)$ caractérise $wp(B, c)$
- ▶ **Proposition:** on peut dériver $\vdash \{W(B, c)\} c \{B\}$
 - ▶ démonstration: aisée pour tous les cas (induction sur c). . .
 - ▶ ...sauf while
on s'appuie sur la caractérisation suivante:
Lemme: $\forall \sigma, \sigma', b, c. \quad \sigma, \text{while } b \text{ do } c \Downarrow \sigma'$
si et seulement si
 $\sigma, (\text{if } b \text{ then } c; \text{while } b \text{ do } c \text{ else skip}) \Downarrow \sigma'$
- ▶ **Théorème:** si $\models \{A\} c \{B\}$, alors $\vdash \{A\} c \{B\}$

$\underbrace{\text{modulo la décision effective de } \models A}_{\text{complétude relative}}$
- ▶ un résultat théorique
 - ▶ les plus faibles préconditions sont utiles pour raisonner sur la complétude de la logique de Hoare
 - ▶ elles sont trop illisibles pour être exploitées dans le cadre d'un outil informatique

A-t-on “mécanisé le raisonnement” ?

- ▶ bon nombre de théorèmes d'arithmétique peuvent se formuler comme des assertions A

$$\models \{\text{true}\} \text{ skip } \{A\} ?$$

- ▶ peut-on alors établir ces théorèmes de manière automatique?
non Gödel (*1er thm. d'incomplétude*)
les $\models A$ sont établies via des “raisonnements mathématiques”
- ▶ il ne s'agit pas non plus d'avoir un algorithme qui répond à
étant donnés c, A, A' , est-ce que $\models \{A\} c \{A'\}$?
cf. $\models \{\text{true}\} c \{\text{false}\}$

Correction partielle vs correction totale

- ▶ on peut dans certains cas s'intéresser aussi à la *correction totale* des programmes, en écrivant des triplets de la forme

$$[A] \text{ c } [A'],$$

que l'on interprète “pour tous V, σ , si $\sigma \models^V A$ alors le **programme** c **termine** (il existe σ' t.q. $c, \sigma \Downarrow \sigma'$) et $\sigma', A' \Downarrow^V \text{true}$ ”

- ▶ les règles sont les mêmes, sauf pour **while**
 - ▶ on fait intervenir un *variant* v , ≥ 0 à chaque entrée dans le corps de la boucle, et qui y décroît strictement
 - ▶ difficulté: les assertions portent sur *un* état, et pas sur une relation entre des σ successifs dans le temps

$$\frac{\vdash [A \wedge b \wedge v = i \wedge i \geq 0] \text{ c } [A \wedge v < i]}{\vdash [A] \text{ while } b \text{ do } c [A \wedge \neg b]} \quad i \text{ frais}$$

Logique de Hoare dans le livre de G. Winskel

- ▶ chapitre 6: définition, correction
 - ▶ ne s'appuie pas sur $\sigma, c \Downarrow \sigma'$ (sémantique opérationnelle à grands pas), mais sur la sémantique dénotationnelle de IMP (hors programme pour nous, cf. M1)
 - ▶ valuation V se dit "interprétation I " dans Winskel
- ▶ chapitre 7: complétude



Des dérivations pour “décrire un phénomène”

$$\vdash \{A\} c \{A'\}$$

$$\models \{A\} c \{A'\}$$

$$c, \sigma \Downarrow \sigma'$$

$$c, \sigma \longrightarrow c', \sigma'$$



correction
complétude

modélisation

*“on rentre dans le corps d’une boucle while
si la condition booléenne est vérifiée ;
sinon, on passe à la suite”*

Trois approches pour la sémantique de IMP

opérationnelle

$$\text{while } b \text{ do } c \longrightarrow \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}$$

$$\frac{b, \sigma \Downarrow \text{true} \quad c, \sigma \Downarrow \sigma' \quad \text{while } b \text{ do } c, \sigma' \Downarrow \sigma''}{\text{while } b \text{ do } c, \sigma \Downarrow \sigma''}$$

axiomatique

$$\frac{\{A \wedge b\} c \{A\}}{\{A\} \text{while } b \text{ do } c \{A \wedge \neg b\}}$$

dénotationnelle

$$D(\text{while } b \text{ do } c) = \bigcup_{n \geq 0} F^n(\perp) \quad \text{avec } F = F_{D(b), D(c)}$$

compositionnalité pour l'**axiomatique** et la **dénotationnelle** :

la sémantique de `while b do c` est calculée à partir de la sémantique de `c`

Jusqu'ici — première partie

	déterminisme de $\longrightarrow$	déterminisme de $\Downarrow$ coïncidence $\longrightarrow / \Downarrow$	correction, complétude relative
grammaire de IMP	petits pas	grands pas	logique de Hoare
ensembles inductifs	induction sur la structure	relations inductives	induction sur la dérivation
treillis complet	théorème de Knaster-Tarski	domaine	théorème de point fixe

Deuxième partie — réécriture, termes

tout au tableau !

- ▶ SRA (système de réécriture abstraite)

terminaison : induction bien fondée, produit lexicographique, extension multiensemble

confluence, confluence locale, lemme de Newman

- ▶ SRM (système de réécriture de mots)

caractérisation de la **terminaison**, paires critiques (**confluence** locale), algorithme de Knuth Bendix

- ▶ SRT (système de réécriture de termes)

définitions : termes, substitutions, réécriture de termes, algorithme d'**unification** du premier ordre (correction et complétude)

NB : fin de la preuve pour l'algorithme d'unification disponible depuis la page [www du cours](#)

Troisième partie :

FUN, un petit langage fonctionnel