Théorie de la Programmation

Daniel Hirschkoff
Alexis Ghyselen, Enguerrand Prebet, Alexandre Talon

http://perso.ens-lyon.fr/daniel.hirschkoff/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 emps 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 ≠ 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 -> B -> 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

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

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

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

```
| 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)

$$C_{aff}\ \frac{a, \sigma \longrightarrow a'}{X := a, \sigma \longrightarrow X := a', \sigma} \qquad C_{seq}\ \frac{c, \sigma \longrightarrow c', \sigma'}{c; c_2, \sigma \longrightarrow c'; c_2, \sigma'}$$

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

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

- **condition d'application** (*side condition*)

$$A_p\ \frac{}{k_1 + k_2, \sigma \longrightarrow k}$$

k est le résultat de la somme de k_1 et k_2

```
| Ap :  forall k1 k2 sigma k,
    k=k1+k2 ->   (ppa (add (cst k1)
                           (cst k2)) sigma (cst k))
```

  - 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
$\longrightarrow$	`X:=3+4; if` $Y > 5$ `then Y:=0 else X:=0`	X 2 / Y 3
$\longrightarrow$	`X:=7; if` $Y > 5$ `then Y:=0 else X:=0`	X 2 / Y 3
$\longrightarrow$	`skip; if` $Y > 5$ `then Y:=0 else X:=0`	X 7 / Y 3
$\longrightarrow$	`if` $Y > 5$ `then Y:=0 else X:=0`	X 7 / Y 3
$\longrightarrow$	`if` $3 > 5$ `then Y:=0 else X:=0`	X 7 / Y 3
$\longrightarrow$	`if false then Y:=0 else X:=0`	X 7 / Y 3
$\longrightarrow$	`X:=0`	X 7 / Y 3
$\longrightarrow$	`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 $\longrightarrow$

$$\frac{\mathtt{a}, \sigma \longrightarrow \mathtt{a'}}{\mathtt{X := a}, \sigma \longrightarrow \mathtt{X := a'}, \sigma} \qquad \frac{\mathtt{c}, \sigma \longrightarrow \mathtt{c'}, \sigma'}{\mathtt{c;c1}, \sigma \longrightarrow \mathtt{c';c1}, \sigma'}$$

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

$$\frac{(\mathtt{a}, \sigma, \mathtt{a'}) \in \longrightarrow}{(\mathtt{X := a}, \sigma, \mathtt{X := a'}, \sigma) \in \longrightarrow} \qquad \frac{(\mathtt{c}, \sigma, \mathtt{c'}, \sigma') \in \longrightarrow}{(\mathtt{c;c1}, \sigma, \mathtt{c';c1}, \sigma') \in \longrightarrow}$$

Deux usages de Knaster-Tarski

constructeurs	$\longleftrightarrow$	règles d'inférence
ensembles d'*objets* (termes)	$\longleftrightarrow$	ensembles de dérivations pour un jugement
induction structurelle $\forall\, x\,.\, \mathcal{P}(x)$	$\longleftrightarrow$	induction sur la dérivation $\forall x_1, \ldots, x_n.$ $\rho(x_1, \ldots, x_n) \implies \mathcal{P}(x_1, \ldots, x_n)$

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

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 — on n'est pas des gallinacés*
  cas k: ...
  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
Arith	3	expressions arithmétiques
Bool	4	expressions booléennes
Com	5	commandes

Relation	nombre de règles d'inférence	
$a, \sigma \longrightarrow a'$	4	petits pas pour Arith
$b, \sigma \longrightarrow b'$	6	petits pas pour Bool
$c, \sigma \longrightarrow c', \sigma'$	8	petits pas pour Com
$a, \sigma \Downarrow k$	3	grands pas pour Arith
$b, \sigma \Downarrow t$	4	grands pas pour Bool
$c, \sigma \Downarrow \sigma'$	7	grands pas pour Com

Ce qui a été vu jusqu'ici

Sémantiques	Ordres partiels et points fixes
. dénotationnelle	. justifier définitions inductives récursives et preuves par induction
. petits pas	
. grands pas	. calculer la sémantique dénotationnelle

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 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

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

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 ≠ Y2) do
 if Y1 > Y2  then Y1 := Y1 - Y2
             else Y2 := Y2 - Y1
```

on aboutit à un état qui vérifie

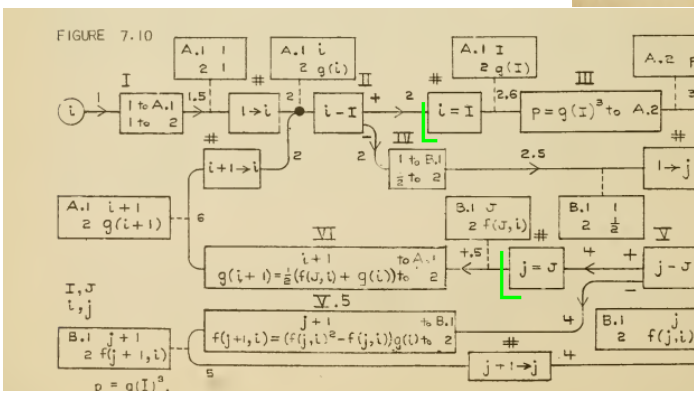
$$Y1 = pgcd(X1, X2)$$

$$(Y1 \ div \ X1) \wedge (Y1 \ div \ X2) \wedge (\forall i.\, (i \ div \ X1) \wedge (i \ div \ X2) \Rightarrow i \leq Y1)$$

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



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

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