Algorithmes, Types, Preuves

Martin Strecker

Année 2006/2007

Plan du semestre

- Session 1 (8/11): Bases: Définitions inductives
- Session 2 (22/11): Sémantique opérationnelle
- Session 3 (29/11): WP-calcul: instructions simples
- Session 4 (6/12): Rappel de logique
- Session 5 (13/12): WP-calcul: fonctions
- Session 6 (20/12): Systèmes de transition; Résumé

Définitions inductives: Idée

Une définition inductive décrit comment un ensemble est généré.
Exemple: Ensemble des nombres pairs P

- *Définition non-inductive:* $P' = \{n \mid \exists k. n = 2 * k\}$
- *Définition inductive:*
  - *Cas de base:* 0 est un nombre pair: $0 \in P$
  - *Pas inductif:* Si n est un nombre pair, alors $n + 2$ l'est aussi: $n \in P \Rightarrow (n + 2) \in P$

Définitions inductives: Notation

Écriture de définitions inductives: en format de règle

$$\frac{\text{Hypothèse(s)}}{\text{Conclusion}}\ (\text{nom de la règle})$$

Exemple:
L'ensemble des nombres pairs P est généré par les règles

$$\frac{}{0 \in P}\ (Z) \qquad\qquad \frac{n \in P}{(n+2) \in P}\ (PD)$$

Une règle sans hypothèses est appelée axiome

Définitions inductives: Interprétation (1)

Première lecture:
$e \in S$ si et seulement si
$e \in S$ suit d'un nombre fini d'applications des règles (qui génèrent S)

Exemple: $4 \in P$, parce que:
$0 \in P$ (règle (Z)) $\rightsquigarrow$ $2 \in P$ (règle (PD)) $\rightsquigarrow$ $4 \in P$ (règle (PD))

Exemple: $5 \notin P$, parce qu'on ne peut pas atteindre 5 en applicant les règles (Z), (PD)
[argument un peu flou, voir la suite...]

Notions de base
Sémantique
WP-calcul
Définitions inductives
Chaînage en avant / en arrière
Preuves par induction
Logique: Syntaxe
Logique: Calcul

Définitions inductives: Interprétation (2)

Deuxième lecture:
Un ensemble S est *fermé par application d'une règle*

$$\frac{e \in S}{e' \in S}$$

si $e \in S$ implique $e' \in S$.
Un *ensemble inductif* est le plus petit ensemble fermé par application de ses règles.
$\Rightarrow$ définition qui est utilisée par la suite $\Leftarrow$

Notions de base
Sémantique
WP-calcul
Définitions inductives
Chaînage en avant / en arrière
Preuves par induction
Logique: Syntaxe
Logique: Calcul

Définitions inductives: Interprétation (3)

Exemples:

- $P_0 = \{0, 2, 4\}$ n'est pas fermé par application des règles (Z), (PD)
- $P_1 = NAT$ est fermé par application des règles (Z), (PD) ... mais n'est pas l'ensemble le plus petit
- $P_2 = \{n \mid \exists k. n = 2 * k\}$ est
  - fermé par application des règles (Z), (PD)
  - est le plus petit ensemble avec cette propriété

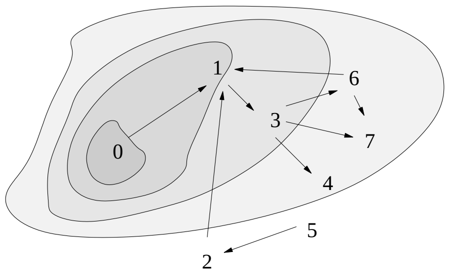
  (preuve: voir plus tard)

Notions de base
Sémantique
WP-calcul
Définitions inductives
Chaînage en avant / en arrière
Preuves par induction
Logique: Syntaxe
Logique: Calcul

Exemple: Graphes (1)

Donné: Graphe $G = (V, E)$ avec

- V un ensemble de noeuds (*vertices*)
- $E \subseteq V \times V$ un ensemble d'arcs (*edges*)



Ensemble de noeuds accessibles (à partir d'un *seul* noeud)
Définir l'ensemble A_e des noeuds accessibles à partir du noeud $e \in V$.

Exemple: Graphes (2)

Ensemble de noeuds accessibles (entre eux)
Définir l'ensemble A des pairs de noeuds (n_1, n_2) tels que n_2 est accessible de n_1 dans G
Solution (notation ensembliste)

$$\frac{}{(n_1, n_1) \in A} \qquad \frac{(n_1, n_2) \in A \quad (n_2, n_3) \in E}{(n_1, n_3) \in A}$$

Solution (notation relationnelle)

$$\frac{}{A(n_1, n_1)} \qquad \frac{A(n_1, n_2) \quad E(n_2, n_3)}{A(n_1, n_3)}$$

Exemples

Exemples de relations inductives:

- nombres pairs
- Relation de parentée
- Accessibilité dans un graphe
- Génération de types de données
- Déductibilité en logique
- Sémantique d'un langage de programmation

Appartenance à un ensemble inductif

Comment vérifier qu'un élément e appartient à un ensemble inductif S?
Deux méthodes:

- Chaînage en avant
- Chaînage en arrière

NB: Seulement applicable à des éléments e concrets.
$\Rightarrow$ voir preuve par induction

Chaînage en avant (1)

Pour vérifier $e \in S \dots$
Principe du chaînage en avant

- Appliquer des règles “de haut en bas”, en commençant par les axiomes
- Générer successivement les ensembles $S_1, S_2, \dots,$ correspondant à 1, 2, ... applications des règles
- Tester si $e \in S_i$ pour un i
- *Problèmes:*
  - les $S_1, S_2, \dots$ souvent très grands, voire infinis
  - Recherche peu ciblée
  - quand s'arrêter?

Chaînage en avant (2)

Exemples

- Vérifier que 6 est un nombre pair:
  $0 \in P \rightsquigarrow 2 \in P \rightsquigarrow 4 \in P \rightsquigarrow 6 \in P$
- Vérifier si 3 est un nombre pair:
  $0 \in P \rightsquigarrow 2 \in P \rightsquigarrow 4 \in P$ *arrêt*

Accessibilité

- Vérifier si $A(1,4)$ dans l'exemple du graphe
- Vérifier si $A(3,2)$

Chaînage en arrière (1)

Pour vérifier $e \in S \ldots$

Principe

- Chercher des règles applicables à e. Une règle est applicable si on peut la mettre dans la forme

$$\frac{e_1 \in S \quad \ldots \quad e_n \in S \quad \ldots}{e \in S}$$

- Appliquer les règles “de bas en haut”
- Vérifier récursivement les hypothèses (ou arrêter s'il y en a pas)
- *Problèmes:*
  - Déterminer si une règle est applicable
  - Boucles infinies

Chaînage en arrière (2)

Exemples

- Vérifier que 6 est un nombre pair:
  $6 \in P \rightsquigarrow 4 \in P \rightsquigarrow 2 \in P \rightsquigarrow 0 \in P \ \surd$
- Vérifier si 3 est un nombre pair:
  $3 \in P \rightsquigarrow 1 \in P$ *aucune règle applicable – échec*

Reprendre exemple de l'accessibilité dans un graphe

- Vérifier si $A(1,4)$ dans l'exemple du graphe
- Vérifier si $A(3,2)$

Chaînage en arrière (3)

Notation

- La propriété initiale à vérifier, $e \in S$, s'appelle *but*
- Les propriété intermédiaires, $e_i \in S$, s'appellent *sous-but*
- La démonstration entière de $e \in S$ s'appelle *dérivation*

Arbres de dérivation

Pour visualiser des dérivations:

- Racine de l'arbre: but
- Noeuds internes: sous-buts
- Feuilles: sous-buts vérifiés par des axiomes ou propriétés "évidentes"

$$\dfrac{\dfrac{\dfrac{}{A(0,0)} \quad E(0,1)}{A(0,1)} \qquad E(1,3)}{A(0,3)}$$

Règles et Prolog (1)

Prolog permet de définir des règles et démontrer des buts par chaînage en arrière.
Écriture des règles:

```
p(e1).
p(e2) :- p(e3), q(e4).
```

au lieu de

$$\dfrac{}{P(e1)} \qquad\qquad \dfrac{P(e3) \quad Q(e4)}{P(e2)}$$

À faire: Coder graphe et prédicat d'accessibilité en Prolog.

Règles et Prolog (2)

Quelle est la conséquence

- si on échange l'ordre des règles?
- si on échange l'ordre des prédicats dans le corps d'une règle?

Comment démontrer qu'un noeud n'est pas accessible à partir d'un autre?
Coder en Prolog le prédicat "est nombre pair".

Pourquoi des preuves?

Les méthodes de chaînage en avant/arrière ne permettent que de parler d'éléments concrets.
Soit $P' = \{n \mid \exists k.n = 2 * k\}$
et P l'ensemble des nombre pairs défini inductivement.
Comment vérifier la propriété $P = P'$?
Décomposition en:

1. $P' \subseteq P$
2. $P \subseteq P'$

Preuve par induction sur $\mathbb{N}$

1ère partie:
Démontrer $\{n \mid \exists k.n = 2 * k\} \subseteq P$
donc $\forall n.\exists k.n = 2 * k \Rightarrow n \in P$
donc $\forall k.2 * k \in P$
Faire la preuve par induction sur $\mathbb{N}$

- $2 * 0 \in P$
  simplifier et utiliser (Z)
- $2 * k \in P \Rightarrow 2 * (k + 1) \in P$
  simplifier et utiliser (PD)

Schéma d'induction (1)

2ème partie:
Démontrer $P \subseteq \{n \mid \exists k.n = 2 * k\}$
donc $\forall n.n \in P \Rightarrow \exists k.n = 2 * k$
Induction de règle: Montrer $\exists k.n = 2 * k$ pour tout $n \in P$.

- Propriété satisfait initialement (règle (Z)):
  $n = 0$, donc montrer
  $\exists k.0 = 2 * k$
- Propriété préservée par chaque application de règle (PD):
  Montrer: Si $n \in P$ et $\exists k.n = 2 * k$,
  alors $\exists k.n + 2 = 2 * k$

Schéma d'induction (2)

Soit S un ensemble défini inductivement.
À démontrer: $x \in S \Rightarrow I(x)$
Induction de règle: Pour chaque règle de la form

$$\frac{e_1 \in S \quad \ldots e_n \in S \quad C}{e \in S}$$

il faut montrer:
$e_1 \in S \wedge I(e_1) \wedge \ldots \wedge e_n \in S \wedge I(e_n) \wedge C \Rightarrow I(e)$
"Chaque application de la règle préserve l'invariant I"

Schéma d'induction (3)

Définir le schéma d'induction pour la relation d'accessibilité.

- 1ère règle:
  $I(n_1, n_1)$
- 2ème règle:
  $(n_1, n_2) \in A \wedge I(n_1, n_2) \wedge (n_2, n_3) \in E \Rightarrow I(n_1, n_3)$

Schéma d'induction (4)

Prouver: $\forall n_1 n_3 . A(n_1, n_3) \Rightarrow (\exists n_2 . A(n_1, n_2) \wedge A(n_2, n_3))$

1. Trouver le prédicat I: $I(n_1, n_3)$ est $\exists n_2 . A(n_1, n_2) \wedge A(n_2, n_3)$
2. Instance 1ère règle: $\exists n_2 . A(n_1, n_2) \wedge A(n_2, n_1)$
3. Instance 2ème règle: $A(n_1, n_2) \wedge (\exists n . A(n_1, n) \wedge A(n, n_2)) \wedge E(n_2, n_3)$ $\Rightarrow \exists n_2 . A(n_1, n_2) \wedge A(n_2, n_3)$

Logique des prédicats (1)

Syntaxe des formules:

- Constantes propositionnelles: $\perp$ (faux), $\top$ (vrai)
- Prédicats: $P(t_1, \ldots, t_n)$ (où $t_1, \ldots, t_n$ sont des termes)
- Négation: $\neg F$
- Conjonction ("et"): $F \wedge G$, disjonction ("ou"): $F \vee G$, implication: $F \Rightarrow G$
- Quantification universelle ("quelque soit"): $\forall x . P$
- Quantification existentielle ("il existe"): $\exists x . P$

Logique propositionnelle: Fragment sans quantificateurs

Logique des prédicats (2)

Dans la formule $\forall x . P$, la variable x est *liée* dans P.
Dans la formule $\exists x . P$, la variable x est *liée* dans P.
Toute variable qui n'est pas liée est *libre*.
Exemple: Dans $\forall x . P(x, y) \wedge \exists z . Q(x, z)$, les variables x et z sont liées, y est libre
Toute variable liée peut être *renommée*.
Exemples: Soit $F \equiv \forall x . P(x, y) \wedge \exists z . Q(x, z)$

- Renommer x par v dans F donne ... ???
- Renommer x par y dans F n'est pas possible. Pourquoi???
- Renommer x par z dans F n'est pas possible. Pourquoi???

Substitution

Expressions: Une substitution $e[x \leftarrow t]$ remplace toute occurrence de la variable x dans l'expression e par le terme t.
Exemples:

- $(x + 5 = x)[x \leftarrow x - 2]$ est $(x - 2) + 5 = x - 2$
- *Attention aux parenthèses!!* $(7 - v)[v \leftarrow y - 2]$ est $7 - (y - 2)$

Formules: $F[x \leftarrow t]$ ... pareil.
Attention: renommer des variables liées, si nécessaire !
Exemple: $(\forall x . P(x, y) \wedge \exists z . Q(x, z))[y \leftarrow x + 4]$

- Solution incorrecte: $(\forall x . P(x, x + 4) \wedge \exists z . Q(x, z))$
- Solution correcte ???

Logique: Calcul

Un calcul sert à dériver des propositions à partir d'autres propositions.
Le calcul s'écrit à l'aide de jugements de la forme

$$\Gamma \vdash F$$

où:

- Γ est un ensemble de formules (*hypothèses*)
- F est une formule (*conclusion*)

Exemple: $A, B \vdash A \vee B$
"Sous l'hypothèse que A et B sont vrais, $A \vee B$ est vrai"

Règles de Calcul (1)

Les règles sont définies inductivement, avec le cas de base:

$$\frac{}{\Gamma, A \vdash A}$$

Attention! On utilise *"conclusion"* et *"hypothèse"* pour les règles et pour les jugements!
Les autres règles sont partagées en

- *règles d'introduction* d'un connecteur:
  Connecteur dans la conclusion de la règle
  (*Ex.:* $\mathrm{I}\wedge$, $\mathrm{I}\vee_1, \dots$)
- *règles d'élimination* d'un connecteur:
  Connecteur parmi les hypothèses de la règle
  (*Ex.:* $\mathrm{E}\wedge_1$, $\mathrm{E}\vee, \dots$)

Règles de Calcul (2)

... pour la logique propositionnelle

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \mathrm{I}\wedge \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \mathrm{E}\wedge_1 \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \mathrm{E}\wedge_2$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \mathrm{I}\vee_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \mathrm{I}\vee_2$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \mathrm{E}\vee$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \mathrm{I} \Rightarrow \qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \mathrm{E} \Rightarrow$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \mathrm{I}\neg \qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A}\ \mathrm{E}\neg \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash \neg A}{\Gamma \vdash \bot}\ \mathrm{I}\bot$$

Règles de Calcul (3)

Règles supplémentaires pour la logique des prédicats

$$\frac{\Gamma \vdash P}{\Gamma \vdash \forall x.P}\ \mathrm{I}\forall \qquad \frac{\Gamma \vdash \exists x.P \quad \Gamma, P \vdash C}{\Gamma \vdash C}\ \mathrm{E}\exists$$

sous condition que x n'est pas libre dans Γ

$$\frac{\Gamma \vdash \forall x.P}{\Gamma \vdash P[x \leftarrow t]}\ \mathrm{E}\forall \qquad \frac{\Gamma \vdash P[x \leftarrow t]}{\Gamma \vdash \exists x.P}\ \mathrm{I}\exists$$

Attention: renommage de variables, si nécessaire

Exemples de Dérivation: Logique propositionnelle

À prouver: $A \wedge B \vdash A \vee B$

$$\dfrac{\dfrac{A \wedge B \vdash A \wedge B}{A \wedge B \vdash A}\ \mathrm{E}\wedge_1}{A \wedge B \vdash A \vee B}\ \mathrm{I}\vee_1$$

À prouver: $A \vee (B \wedge C) \vdash (A \vee B)$ Soit $\mathcal{F} = A \vee (B \wedge C)$

$$\dfrac{\mathcal{F} \vdash A \vee (B \wedge C) \qquad \dfrac{A \vee (B \wedge C), A \vdash A}{\mathcal{F}, A \vdash A \vee B}\ \mathrm{I}\vee_1 \qquad \dfrac{\dfrac{\mathcal{F}, B \wedge C \vdash B \wedge C}{\mathcal{F}, B \wedge C \vdash B}\ \mathrm{E}\wedge_1}{\mathcal{F}, B \wedge C \vdash A \vee B}\ \mathrm{I}\vee_2}{\mathcal{F} \vdash (A \vee B)}\ \mathrm{E}\vee$$

Exemples de Dérivation: Logique des prédicats (1)

À prouver: $\forall x.(P(x) \wedge Q(x)) \vdash (\forall x.P(x)) \wedge (\forall x.Q(x))$

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{P(x) \wedge Q(x) \vdash P(x)}}{\forall x.(P(x) \wedge Q(x)) \vdash P(x)}}{\forall x.(P(x) \wedge Q(x)) \vdash \forall x.P(x)} \qquad \dfrac{\vdots}{\forall x.(P(x) \wedge Q(x)) \vdash \forall x.Q(x)}}{\forall x.(P(x) \wedge Q(x)) \vdash (\forall x.P(x)) \wedge (\forall x.Q(x))}$$

à compléter!

Exemples de Dérivation: Logique des prédicats (2)

Lesquelles des implication suivantes sont vraies?

1. $(\forall x.P(x)) \Rightarrow (\exists x.P(x))$
2. $(\exists x.P(x)) \Rightarrow (\forall x.P(x))$

Attention aux conditions de variable
Soit la règle suivante (réflexivité de l'égalité):

$$\dfrac{}{t = t}\ \mathrm{Ref}$$

Prouver: $\forall x.\exists y.x = y$
Essayer de prouver: $\exists x.\forall y.x = y$

Sémantique: C'est quoi? (1)

La sémantique d'une langue

- décrit la *signification* des phrases de la langue
- ... contrairement à la *syntaxe*, qui décrit leur structure

Ces deux notions s'appliquent aux langues naturelles et artificielles (langages de programmation).

Sémantique: C'est quoi? (2)

Prérequis pour déterminer la sémantique d'une phrase: correction syntaxique.
Exemples:

- $(3 + x) - (/ * 8)$
  - syntaxiquement mal formé
- $(3 + x) - (y * 8)$
  - syntaxiquement bien formé
  - signification: si $x = 20$ et $y = 2$, alors le résultat est 7
  - Comment le définir mieux?

En langue naturelle, la situation est moins nette.
Exemple: Time flies like arrows

Sémantique: Pourquoi? (1)

Mille et une raisons, parmi lesquelles:

- *Désambiguïser des expressions:*
  Le résultat de `(x = 3) * (x + 2)` pour $x = 1$ est (??):
  - 15 (évaluation "gauche avant droite")
  - 9 (évaluation "droite avant gauche")
- *Clarification de cas extrèmes:*
  `MAXINT + 1` est (??):
  - `MININT`
  - une erreur

Important à savoir pour le programmeur

Sémantique: Pourquoi? (2)

- *Équivalence de programmes:*
  Est-il correct d'optimiser

```
while (x > 3) {
    a = f(y);
    ....}
```

en éliminant le calcul multiple de `a = f(y)`:

```
a = f(y);
while (x > 3) {
    ....}
```

Techniques utilisées dans les compilateurs modernes

Sémantique: Pourquoi? (3)

- *Correction de programmes:*
  Est-il possible que le programme

```
if (x * x - 1 == 0) {
     y = 5 / x; }
else {
     y = 5 / x - 1;}
```

provoque une division par 0? . . . pour être sûr que, cette fois, la fusée Ariane n'explose pas

Définition du langage: Plan

Le langage défini dans la suite sera

- *réduit* par rapport à un langage de programmation réel (comme C)
  Ex.: pas de boucle `for`; pas de pointeurs
- *idéalisé*
  Ex.: expressions sans effet de bord; valeurs d'expressions toujours définies

Structure globale du langage

Définition préliminaire: Les valeurs affectées aux variables d'un programme constituent son *état*.
Exemple: $\sigma = \{x = 3; y = 4\}$ est un état pour un programme avec les variables x, y.
Le langage est divisé en trois grandes catégories:

- *Expressions* calculent une *valeur* sans modifier l'état.
  Ex.: `(x + 2) * y` a la valeur 20 dans σ
- *Commandes* modifient l'état sans calculer de valeur.
  Ex.: `x = x + 1` change σ en $\sigma' = \{x = 4; y = 4\}$
- *Procédures* sont des abstractions de commandes.

Expressions

Deux classes d'expressions, définies de manière inductive:

- Arithmétiques a
  - Nombres n
  - Variables x
  - Opérations binaires: $a_1 + a_2$, $a_1 - a_2$, $a_1 * a_2$
- Boléennes b
  - Constantes `true`, `false`
  - Comparaisons: $a_1 = a_2$, $a_1 \leq a_2$
  - Négation: $\neg b$
  - Conjonction: $b_1 \wedge b_2$

À faire: Concevoir des types `aexpr, bexpr` en Caml

Commandes (1)

Commandes c (définition inductive):

- Affectation: $x := a$
- Séquence: c_1 `;` c_2
- Programme vide: `Skip`
- Sélection: `if (`b`) {`c_1`} else {`c_2`}`
- Boucle: `while (`b`) {`c`}`

Fonctions, procédures, entrées / sorties: plus tard
À faire: Concevoir un type `com` en Caml

Commandes (2)

Traitement de

- sélection sans `else`:
  Traduire en `if – then – else`:
  `if` (b) $\{c_1\}$ devient
  `if` (b) $\{c_1\}$ `else {Skip}`
- boucle `for`:
  Traduire en initialisation; boucle `while`
- boucle `do .. while`:
  Traduire en séquence; boucle `while`

Syntaxe concrète / abstraite

La syntaxe concrète est la représentation externe d'une unité syntaxique. Elle

- est parfois redondante. *Ex.:* Accolades superflues dans:
  `if (x > 0) {x = x + 1;}`
- s'appuie sur des conventions (règles de précédence . . .)
  Ex.: $2 + 3 * 4$ et $2 + (3 * 4)$ sont équivalents

La syntaxe abstraite est la représentation interne. Elle est unique pour éléments syntaxiquement équivalents.
Donner l'arbre syntaxique pour les expressions en haut!

Langage: Résumé

Rappel: Distinction entre:

- Expressions (arithmétique, booléen)
- Commandes

Définitions en sémantique basées sur syntaxe abstraite.
Limitations du langage peuvent

- en partie être levées de manière syntaxique (*ex.:* codage de `for` par `while`)
- nécessitent parfois un cadre plus complexe (*ex.:* pointeurs)

Sémantique: Principes

On définit la sémantique comme suit:

- pour des *expressions*
  - *arithmétiques* a: une fonction $\mathcal{A}(a, \sigma)$, qui calcule la valeur de a dans l'état σ
  - *booléennes* b: une fonction $\mathcal{B}(b, \sigma)$, qui calcule la valeur de b dans l'état σ
- pour des *commmandes*: une relation $\langle c, \sigma\rangle \to \sigma'$ qui transforme un état σ en état σ' par exécution de la commande c

Sémantique: Manipulation de l'état

Un état σ définit les valeurs de chaque variable du programme.
État comme type abstrait, avec les opérations suivantes:

- État initial σ_0, renvoie 0 pour toute variable
- Mise à jour de la valeur d'une variable x d'un état σ avec une valeur v, écrit: $\sigma.x \leftarrow v$
- Sélection de la valeur d'une variable x d'un état σ, écrit: $\sigma.x$

Quelques équivalences:

- $(\sigma.x \leftarrow v).x = v$
- $(\sigma.x \leftarrow v).y = \sigma.y$ (pour $x \neq y$)

Sémantique: Représentation de l'état

Représentation en Caml comme *record*:

- type d'état
  ```
  type state = { x : int; y : int }
  ```
- état initial
  ```
  let sig0  = { x = 0; y = 0 }
  ```
- mise à jour, sélection ... c'est juste la notation de Caml

Les définitions suivantes sont génériques!!
(ne dépendent pas de la définition spécifique d'état)
Représentation alternative: comme *fonction* Comment?

Évaluation d'expressions (1)

Définition par récursion sur la structure des expressions:

$$
\begin{array}{lcl}
\mathcal{A}(n, \sigma) & = & n \\
\mathcal{A}(x, \sigma) & = & \sigma.x \\
\mathcal{A}(e_1 + e_2, \sigma) & = & \mathcal{A}(e_1, \sigma) + \mathcal{A}(e_2, \sigma) \\
\ldots & & \\
\mathcal{B}(\mathtt{true}, \sigma) & = & \mathtt{true} \\
\ldots & &
\end{array}
$$

complétez – et écrivez les fonctions Caml correspondantes:

```
aeval : aexpr -> state -> int
beval : bexpr -> state -> bool
```


Évaluation d'expressions (2)

Calculer

- $\mathcal{A}(x + 4, \{x = 3; y = 2\})$
- $\mathcal{A}(x * 0, \sigma)$

Observations: L'évaluation des expressions est

- une fonction *totale* (toujours définie)
- déterministe

Discussion: Comment traiter une expression a_1 / a_2 ?

Exécution de commandes

Définition de la relation $\langle c, \sigma\rangle \rightarrow \sigma'$ par induction.

Affectation

$$\frac{}{\langle x := a, \sigma\rangle \rightarrow (\sigma.x \leftarrow \mathcal{A}(a, \sigma))}$$

Exemple:

$\langle x := x + y, \{x = 5; y = 4\}\rangle \rightarrow \{x = 9; y = 4\}$

Exécution de commandes

Skip

$$\frac{}{\langle \texttt{Skip}, \sigma\rangle \rightarrow \sigma}$$

Séquence

$$\frac{\langle c_1, \sigma\rangle \rightarrow \sigma' \quad \langle c_2, \sigma'\rangle \rightarrow \sigma''}{\langle (c_1; c_2), \sigma\rangle \rightarrow \sigma''}$$

Exemple:

$\langle x := x + y, \{x = 5; y = 4\}\rangle \rightarrow \{x = 9; y = 4\}$

$\langle y := 1, \{x = 9; y = 4\}\rangle \rightarrow \{x = 9; y = 1\}$

donc:

$\langle x := x + y; y := 1, \{x = 5; y = 4\}\rangle \rightarrow \{x = 9; y = 1\}$

Exécution de commandes

Sélection

$$\frac{\mathcal{B}(b, \sigma) = \mathit{true} \quad \langle c_1, \sigma\rangle \rightarrow \sigma'}{\langle \texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}}, \sigma\rangle \rightarrow \sigma'}$$

$$\frac{\mathcal{B}(b, \sigma) = \mathit{false} \quad \langle c_2, \sigma\rangle \rightarrow \sigma'}{\langle \texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}}, \sigma\rangle \rightarrow \sigma'}$$

Exemple:

$\langle \texttt{if (x < 7) \{x:=x + y\} else \{y:=1\}}, \{x = 5; y = 4\}\rangle \rightarrow ???$

Exécution de commandes

Boucle

$$\frac{\mathcal{B}(b, \sigma) = \mathit{true} \quad \langle c, \sigma\rangle \rightarrow \sigma' \quad \langle \texttt{while (}b\texttt{) \{}c\texttt{\}}, \sigma'\rangle \rightarrow \sigma''}{\langle \texttt{while (}b\texttt{) \{}c\texttt{\}}, \sigma\rangle \rightarrow \sigma''}$$

$$\frac{\mathcal{B}(b, \sigma) = \mathit{false}}{\langle \texttt{while (}b\texttt{) \{}c\texttt{\}}, \sigma\rangle \rightarrow \sigma}$$

Exemples:

$\langle \texttt{while (x<3) \{x:= x+1; y:=x+y\}}, \{x = 1; y = 4\}\rangle \rightarrow ???$

$\langle \texttt{while (x<3) \{x:= x-1; y:=x+y\}}, \{x = 1; y = 4\}\rangle \rightarrow ???$

Exécution de commandes

Observations: L'exécution des commandes

- est une relation déterministe: $\langle c, \sigma \rangle \rightarrow \sigma'$ et $\langle c, \sigma \rangle \rightarrow \sigma''$ impliquent $\sigma' = \sigma''$
- ... mais pas totale:
  il existe c, σ pour lesquels il n'y a pas de σ' avec $\langle c, \sigma \rangle \rightarrow \sigma'$
  Lesquels?

Extraction d'instructions communes (1)

... d'une sélection (*Exemple*):

original ...

```
if (x < 5) {
  y = x + 1;
  x = x + 1;
  }
else {
  y = x + 1;
  x = x - 1;
  }
```

transformé ...

```
y = x + 1;
if (x < 5) {
  x = x + 1;
  }
else {
  x = x - 1;
  }
```

Les deux programmes sont équivalents (pourquoi?), mais le deuxième est plus court / lisible.

Extraction d'instructions communes (2)

Schéma général

original ...

```
if (b) { c1; c2; }
else { c1; c3; }
```

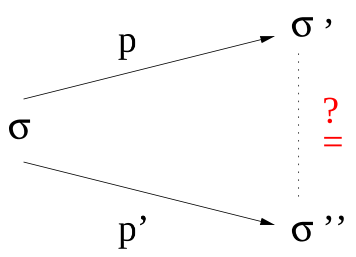
transformé ...

```
c1;
if (b) { c2; }
else { c3; }
```

Cette transformation est-elle correcte? On verra ...
Soit p le programme original, p' le programme transformé.

Preuves d'équivalence de programmes (1)

Équivalence de programmes: p et p' sont *équivalents* s'ils induisent le même changement d'état:
$\forall \sigma, \sigma', \sigma''. \langle p, \sigma \rangle \rightarrow \sigma' \Rightarrow \langle p', \sigma \rangle \rightarrow \sigma'' \Rightarrow \sigma' = \sigma''$



Preuves d'équivalence de programmes (2)

Principe de preuve "standard": Inductions de règle imbriquées:
Si $\langle p, \sigma \rangle \to \sigma'$ est de la forme:

- $\langle \texttt{Skip}, \sigma \rangle \to \sigma$: alors, si $\langle p', \sigma \rangle \to \sigma''$ est de la forme:
  - $\langle \texttt{Skip}, \sigma \rangle \to \sigma$, montrer $\sigma = \sigma$
  - $\langle (c_1; c_2), \sigma \rangle \to \sigma_2$, montrer $\sigma = \sigma_2$
  - etc.
- $\langle (c_1; c_2), \sigma \rangle \to \sigma_2$: alors, si ...
- $\langle \texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}}, \sigma \rangle \to \sigma_2$, avec $\mathcal{B}(b, \sigma) = true$
- $\langle \texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}}, \sigma \rangle \to \sigma_2$, avec $\mathcal{B}(b, \sigma) = false$
- etc.

Preuves d'équivalence de programmes (3)

Appliqué à l'extraction d'instructions communes:
Seuls cas applicables:

- $\langle \texttt{if (}b\texttt{) \{}c_1\texttt{;}c_2\texttt{;\} else \{}c_1\texttt{;}c_3\texttt{\}}, \sigma \rangle \to \sigma_1$, avec $\mathcal{B}(b, \sigma) = true$
  - $\langle (c_1\texttt{;if (}b\texttt{) \{}c_2\texttt{;\} else \{}c_3\texttt{\}}), \sigma \rangle \to \sigma_2$, montrer $\sigma_1 = \sigma_2$
- pareil, avec $\mathcal{B}(b, \sigma) = false$

Premier cas: Simplifier:
$\langle \texttt{if (}b\texttt{) \{}c_1\texttt{;}c_2\texttt{;\} else \{}c_1\texttt{;}c_3\texttt{\}}, \sigma \rangle \to \sigma_1$
devient
$\langle c_1; c_2;, \sigma \rangle \to \sigma_1$

Preuves d'équivalence de programmes (4)

Il reste à montrer: Si

- $\langle c_1; c_2;, \sigma \rangle \to \sigma_1$ et
- $\langle (c_1\texttt{;if (}b\texttt{) \{}c_2\texttt{;\} else \{}c_3\texttt{\}}), \sigma \rangle \to \sigma_2$ et
- $\mathcal{B}(b, \sigma) = true$

alors $\sigma_1 = \sigma_2$
Est-ce vrai?? Pour compléter la preuve:

- Définir une fonction *vars*(b)
  (ensemble des variables contenues dans l'expression b)
- Définir une fonction *modifs*(c)
  (ensemble des variables modifiées dans l'instruction c)
- Trouver une précondition pour l'équivalance des programmes. Laquelle?

Quelques exercices

- Dériver de nouvelles règles, par exemple pour `do .. while (b)`
- montrer que `do {c1; c2 } while (b)` peut être transformé en `c1; do {c2 } while (b)` si `c2` et `b` ne "dépendent pas" de `c1`.

Preuves de programmes: Pourquoi? (1)

Exemple: Pour un nombre *n*, trouver le plus grand diviseur *d* de *n* (sauf *n* . . .)
Première solution:

```
int n, d;

printf("Entrer n: ");
scanf("%d", &n);
d = n - 1;

while (n % d != 0) {
  d = d - 1;
}

printf("d est: %d\n", d);
```


Preuves de programmes: Pourquoi? (2)

La solution est-elle correcte? Quelques tests:

```
> div
Entrer n: 12345
d est: 4115
```

4115 est diviseur de 12345, mais le plus grand??

```
> div
Entrer n: 6
d est: 3
> div
Entrer n: 5
d est: 1
```

C'est correct. Mais et-ce qu'on a traité tous les cas importants??

Comparaison: Test vs. Preuves de programmes (1)

Test:
Avantages:

- Intuitives
- Souvent faciles à mettre en oeuvre

Désavantages: Difficile à savoir si

- on a traité tous les cas critiques
- le résultat fourni par le programme est correct:
  - Test d'un programme qui traite des matrices de taille $10^3 \times 10^3$??
  - Test d'un compilateur ??

Comparaison: Test vs. Preuves de programmes (2)

Preuves:
Avantages:

- Vérification exhaustive
- Demande une programmation structurée et disciplinée

Désavantages:

- difficile ou impossible à automatiser (indécidabilité des logiques “intéressantes”)
- donc: requiert souvent une intervention manuelle
- donc: un travail exigeant . . .

Comment raisonner sur des programmes ? (1)

Exemple 1: Affectation simple

```
(1)  {x = 4}
(2)  x = x - 1;
(3)  {x = 3}
```

Raisonnement possible:

1. $\{x = 4\}$, donc $\{x - 1 = 3\}$
2. Affectation: `x - 1` "devient" `x`
3. Après affectation: $\{x = 3\}$

Comment raisonner sur des programmes ? (2)

Exemple 2: Affectation moins simple

```
(1)  {y = 5 * x + 15}
(2)  x = 2 * x + 6;
(3)  {2 * y = 5 * x}
```

Raisonnement possible:

1. $\{y = 5 * x + 15\}$, donc
   $\{2 * y = 10 * x + 30 = 5 * (2 * x + 6)\}$
2. Affectation: `2 * x + 6` "devient" `x`
3. Après affectation: $\{2 * y = 5 * x\}$

Problème: Calculation peu ciblée.

Comment raisonner sur des programmes ? (3)

Comment faire mieux? Au lieu de raisonner en avant
. . . raisonner en arrière !

```
(1)  {x = 4}
(2)  x = x - 1;
(3)  {x = 3}
```

1. A démontrer: $\{x = 3\}$ après affectation
2. Substituer: $\{(x = 3)[x \leftarrow x - 1]\}$
3. Calculer: $\{(x = 3)[x \leftarrow x - 1]\} \equiv \{x - 1 = 3\} \equiv \{x = 4\}$

Exercice: Faire l'Exemple 2 !!

Comment raisonner sur des programmes ? (4)

Est-ce qu'il y a un problème avec le raisonnement suivant?

```
(1)  {x = 4}
(2)  x = x + 3;
(3)  {x > 5}
```

Non !

1. A démontrer: $\{x > 5\}$ après affectation
2. Substituer: $\{(x > 5)[x \leftarrow x + 3]\}$
3. Calculer: $\{(x > 5)[x \leftarrow x + 3]\} \equiv \{x + 3 > 5\} \equiv \{x > 2\}$
4. *Implication:* $(x = 4) \Rightarrow (x > 2)$

Donc, $\{x = 4\}$ est une bonne précondition . . . *mais pas la plus faible !*

Comment raisonner sur des programmes ? – Schéma général

But: Démontrer la correction du programme `prog` par rapport a sa spécification:

$$\{P\}\ \texttt{prog}\ \{Q\}$$

Calculer la plus faible précondition *pfp*
(en anglais: weakest precondition, *wp*):

$$pfp(\texttt{prog}, Q)$$

Démontrer implication:

$$P \Rightarrow pfp(\texttt{prog}, Q)$$

Comment ça fonctionne pour des programmes plus complexes?
$\leadsto$ voir la suite

Traitement d'erreurs (1)

Nous considérons les cas d'erreurs suivants:

- Arithmétique: Division par zéro: `x / 0`; modulo zéro: `x % 0`
- Dépassement de bornes lors d'un accès à un tableau `a[i]`: $i < 0$ ou $i > n - 1$

Nous ne prenons pas en compte:

- Dépassement de bornes lors d'opérations arithmétiques (ex.: `a + b > INT_MAX`)
  supposition idéalisante: arithmétique sans limites . . .

Traitement d'erreurs (2)

Evaluable: Le prédicat *évaluable*(e) fournit une condition

- sous laquelle l'expression e peut être évaluée
- sans produire d'erreur

Définition (incomplète):

$évaluable(v) = \top$ pour v variable
$évaluable(e_1 + e_2) = évaluable(e_1) \wedge évaluable(e_2)$
$évaluable(e_1 / e_2) = évaluable(e_1) \wedge évaluable(e_2) \wedge e_2 \neq 0$
$évaluable(a[i]) = évaluable(a) \wedge évaluable(i) \wedge 0 \leq i < taille(a)$
Exemple: $évaluable(a[k + 5]) = 0 \leq k + 5 < taille(a)$

Règles d'inférence: Plan

L'ensemble des commandes a été défini de manière *inductive*.
Nous allons définir une fonction *pfp*(`c`, Q) par *récursion* sur les commandes `c`
Donc: une règle par constructeur

pfp: Affectation

Règle:

$$pfp(\texttt{x = e}, Q) = Q[x \leftarrow e] \wedge \textit{évaluable}(e)$$

Exemples:

- $pfp(\texttt{x = x - 1}, x > 0) \equiv$
  $x > 0\,[\,x \leftarrow x - 1] \wedge \textit{évaluable}(x - 1) \equiv$
  $x - 1 > 0 \wedge \top$
- Démontrer: $\{x > 0\}$ `x = x - 1` $\{x > 0\}$
  1. Calculer $pfp(\texttt{x = x - 1}, x > 0) \equiv (x - 1 > 0)$
  2. Implication $(x > 0) \Rightarrow (x - 1 > 0) \ \equiv \perp$

pfp: Séquence (1)

Règles:

- Séquence non vide:
  $$pfp(\texttt{c1; c2}, Q) = pfp(\texttt{c1}, pfp(\texttt{c2}, Q))$$
- Séquence vide:
  $$pfp(\{\,\}, Q) = Q$$

Exemple:

$pfp(\texttt{x = x + 5; y = x - y}, (x = 7 \wedge y = 10)) \equiv$
$pfp(\texttt{x = x + 5}, (x = 7 \wedge x - y = 10)) \equiv$
$(x + 5 = 7 \wedge (x + 5) - y = 10) \equiv$
$x = 2 \wedge y = -3$

pfp: Séquence (2)

Exemple: Échange de deux variables
Démontrer:

$\{x = A \wedge y = B\}$
```
x = x + y;
y = x - y;
x = x - y
```
$\{x = B \wedge y = A\}$

pfp: Séquence (3)

Solution:

$pfp(\texttt{x = x + y; y = x - y; x = x - y}, (x = B \wedge y = A))$
$\equiv$
$pfp(\texttt{x = x + y; y = x - y}, (x - y = B \wedge y = A)) \equiv$
$pfp(\texttt{x = x + y}, (x - (x - y) = B \wedge x - y = A)) \equiv$
$((x + y) - ((x + y) - y) = B \wedge (x + y) - y = A) \equiv$
$y = B \wedge x = A$

pfp: Sélection (1)

Règle:

$pfp($`if (b) c1 else c2`$, Q) =$
$b \Rightarrow pfp($`c1`$, Q) \wedge$
$\neg b \Rightarrow pfp($`c2`$, Q)$

Exemple:

$pfp($`if (x <= y) m = x else m = y`$, (x > m)) \equiv$
$(x \leq y \Rightarrow pfp($`m = x`$, (x > m))) \wedge$
$(x > y \Rightarrow pfp($`m = y`$, (x > m))) \quad \equiv$

$(x \leq y \Rightarrow x > x) \wedge (x > y \Rightarrow x > y) \quad \equiv$
$(x > y \vee x > x) \wedge \top \quad \equiv x > y$

pfp: Sélection (2)

Démontrer

$\{x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z}\}$

```
if (y % 2 == 0) {
  x = x * x;
  y = y / 2;
  }
else {
  z = z * x;
  y = y - 1;
  }
```

$\{x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z}\}$

pfp: Sélection (3)

Solution: Soit $F \equiv x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z}$
$pfp($`if(y % 2 == 0) {..} else {..}`$, F) \equiv$

$y\%2 = 0 \Rightarrow pfp($ `x = x * x; y = y / 2`$, F) \wedge$
$y\%2 \neq 0 \Rightarrow pfp($ `z = z *x; y = y - 1`$, F) \quad \equiv$

$y\%2 = 0 \Rightarrow (x * x)^{(y/2)} * z = A^B \wedge (x * x) \in \mathbb{Z} \wedge (y/2) \in \mathbb{Z} \wedge$
$y\%2 \neq 0 \Rightarrow x^{y-1} * z * x = A^B \wedge x \in \mathbb{Z} \wedge (y-1) \in \mathbb{Z} \quad \equiv$

$y\%2 = 0 \Rightarrow x^y * z = A^B \wedge (x * x) \in \mathbb{Z} \wedge (y/2) \in \mathbb{Z} \wedge$
$y\%2 \neq 0 \Rightarrow x^y * z = A^B \wedge x \in \mathbb{Z} \wedge (y-1) \in \mathbb{Z}$
ce qui est impliqué par: $x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z}$

pfp: Sélection (4)

Exemple: Règle pour `if (b) c1`
Développer cette règle, en utilisant

- La traduction en `if - then - else`
- La règle pour la sélection
- La règle pour la séquence vide

pfp: Boucle (1)

Exemple: Soit `prog` le programme:

```
while (x < 3) {
  x = x + 1;
  y = y + 2;
  }
```
$\{x = 3 \wedge y = 6\}$

Quelle est *pfp*(`prog`, $x = 3 \wedge y = 6$),
pour 3 parcours de la boucle?

- Lors du dernier passage de la boucle: $x = 2 \wedge y = 4$
- ... avant-dernier ...: $x = 1 \wedge y = 2$
- ... premier ...: $x = 0 \wedge y = 0$

Donc: *pfp*(`prog`, $x = 3 \wedge y = 6$) = ($x = 0 \wedge y = 0$)

pfp: Boucle (2)

Problèmes:

- En général: nombre d'itérations inconnu *a priori*
- La *pfp* est différente à chaque itération

Observation: Une propriété reste inchangée: $y = 2 * x \wedge x \leq 3$
$\rightsquigarrow$ *invariant* de la boucle

pfp: Boucle (2)

Notation pour des boucles avec invariant `I`

```
while (b)    /* INV: { I } */
  c
```

Exemple:

```
while (x < 3)    /* INV: { y = 2 * x } */
 { x = x + 1;
    y = y + 2;
  }
```


pfp: Boucle (3)

Règles pour une boucle de la forme

```
while (b)    /* INV: { I } */
  c
```

Calcul de la pfp:

pfp(`while (b) /* INV: I */ c`, $Q \wedge \neg b$) = I

Correction de l'invariant:

$$(I \wedge b) \Rightarrow pfp(\texttt{c}, I)$$

Correction d'une boucle (1)

Comment vérifier?

$\{P\}$

```
while (b)     /* INV: { I } */
  c
```

$\{Q\}$

1. Initialisation correcte: $P \Rightarrow I$
2. Préservation de l'invariant: $(I \wedge b) \Rightarrow pfp(\mathtt{c}, I)$
3. Sortie de la boucle: $(I \wedge \neg b) \Rightarrow Q$

Correction d'une boucle (2)

Exemple:

```
{⊤}
x = 0; y = 0;
{x = 0 ∧ y = 0}
while (x < 3)        /* INV: { y = 2 * x ∧ x ≤ 3}  */
  { x = x + 1;     y = y + 2;   }
{x = 3 ∧ y = 6}
```

1. Initialisation correcte:
$(x = 0 \wedge y = 0) \Rightarrow (y = 2 * x) \wedge (x \leq 3)$
2. Préservation de l'invariant:
$(y = 2 * x) \wedge (x \leq 3) \wedge (x < 3) \Rightarrow (y + 2) = 2 * (x + 1) \wedge (x + 1) \leq 3$
3. Sortie de la boucle:
$y = 2 * x \wedge x \leq 3 \wedge \neg(x < 3) \Rightarrow x = 3 \wedge y = 6$

Correction d'une boucle (3)

Exemple: Calcul efficace de puissance

$\{A \in \mathbb{Z} \wedge B \in \mathbb{Z}\}$

```
x = A; y = B; z = 1;
while (y != 0)
      /* INV: {x^y * z = A^B ∧ x ∈ ℤ ∧ y ∈ ℤ}   */

   if (y % 2 == 0) {
      x = x * x; y = y / 2;
   }
   else {
      z = z * x; y = y - 1;
   }
```

$\{z = A^B\}$

Faire la vérification !

Correction d'une boucle (4)

Solution: Soit $A \in \mathbb{Z} \wedge B \in \mathbb{Z}$

1. Initialisation correcte:
$x = A \wedge y = B \wedge z = 1 \Rightarrow x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z}$
2. Préservation de l'invariant:
$x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z} \wedge (y \neq 0) \Rightarrow$
$x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z}$
(Voir exemple "sélection")
3. Sortie de la boucle:
$x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z} \wedge (y = 0) \Rightarrow z = A^B$

Motivation: Correction totale

Le programme "calcul de puissance" a été prouvé "correct"
– et pourtant il y a un problème.
Lequel?
Problème: Non-terminaison si `B` est négatif!

Correction partielle / totale: Définitions

Deux notions de correction:

- `prog` est *partiellement correct* par rapport à $\{P\}$ `prog` $\{Q\}$ si:
  - pourvu que P est satisfait avant exécution, et
  - pourvu que `prog` termine (... mais ce n'est pas assuré !)
  - alors Q est satisfait après
- `prog` est *totalement correct* par rapport à $\{P\}$ `prog` $\{Q\}$ si:
  - pourvu que P est satisfait avant exécution
  - alors `prog` termine
  - et Q est satisfait après exécution

Correction partielle / totale: Règles

Correction partielle: Les règles 1 ... 3 vues auparavant
Correction totale: En ajoutant deux règles:
Trouver une *variante* f telle que:

4. f strictement positif initialement: $I \wedge b \Rightarrow f > 0$
5. f strictement décroissante: $(T = f) \wedge I \wedge b \Rightarrow pfp(\mathtt{c}, T > f)$

Correction partielle / totale: Exemple (1)

Comment "réparer" le calcul de la puissance?

- Renforcer la précondition:
  au lieu de $\{A \in \mathbb{Z} \wedge B \in \mathbb{Z}\}$ prendre $\{A \in \mathbb{Z} \wedge B \in \mathbb{N}\}$
- Renforcer l'invariant:
  au lieu de `/* INV:` $\{x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{Z}\}$ `*/`
  prendre `/* INV:` $\{x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{N}\}$ `*/`
  et mettre à jour les preuves des étapes 1 .. 3 !!
- Choisir variante: f est y
- f strictement positif initialement:
  $x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{N} \wedge y \neq 0 \Rightarrow y > 0$

Correction partielle / totale: Exemple (2)

- f strictement décroissante:
  $(T = y) \wedge x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{N} \wedge y \neq 0 \Rightarrow$
  $(y\%2 = 0 \Rightarrow T > y/2) \wedge$
  $(y\%2 \neq 0 \Rightarrow T > y - 1)$
  ... après quelques simplifications:
  $x^y * z = A^B \wedge x \in \mathbb{Z} \wedge y \in \mathbb{N} \wedge y \neq 0 \Rightarrow$
  $(y\%2 = 0 \Rightarrow y > y/2) \wedge$
  $(y\%2 \neq 0 \Rightarrow y > y - 1)$

Notions de base
Sémantique
WP-calcul
Motivation
Conventions et notions de base
Règles d'inférence
Correction partielle vs. correction totale
Tableaux et Procédures

Correction totale: Exercice

Essayer de prouver la correction totale du programme qui calcule la factorielle de N:
$\{N \in \mathbb{Z}\}$

```
i = N;
r = 1;
while (i != 0) {
    r = r * i;
    i = i - 1;
}
```

$\{r = \Pi_{j=1}^{N} j\}$
Éventuellement, corriger le programme.

Notions de base
Sémantique
WP-calcul
Motivation
Conventions et notions de base
Règles d'inférence
Correction partielle vs. correction totale
Tableaux et Procédures

Résumé

- Vérification de programmes:
  - Tests: incomplets, mais (apparemment) faciles
  - Preuves: vérification exhaustive
- Schéma de preuve: Propagation d'assertions "d'arrière en avant"
- ... en calculant la *pfp*
- Deux notions de correction: partielle (sans terminaison) / totale (avec)

Regarder l'exemple initial de plus près. Est-il vraiment "correct"? Sous quelles conditions?

Notions de base
Sémantique
WP-calcul
Motivation
Conventions et notions de base
Règles d'inférence
Correction partielle vs. correction totale
Tableaux et Procédures

Tableaux: Notation

... sont traités (presque) comme des variables
... mais il faut utiliser des quantificateurs pour se référer à tous les éléments
Conventions:

- Pour un tableau A, on désigne sa taille par $taille(A)$
- Les éléments sont numérotés de $0 \ldots taille(A) - 1$
- Comme d'habitude: $A[i]$ est le i-ème élément de A

Exemple: *"Tous les éléments de A sont strictement positifs":*

$$\forall i. 0 \leq i \leq taille(A) - 1 \Rightarrow A[i] > 0$$

Tableaux: Spécification

Spécifier:

- Tous les éléments de A sont pairs
- Le tableau A est trié
- A est un palindrome
  (un mot “symétrique”, comme *abba* ou *12321*)

Fonctions auxiliaires
Dans les spécifications, on peut utiliser des fonctions auxiliaires définies par récursion.
Exemple: *“La somme des éléments de A est positif”:*

$$\sum_{i=0}^{taille(A)-1} A[i] \geq 0$$

Notions de base
Sémantique
WP-calcul
Motivation
Conventions et notions de base
Règles d'inférence
Correction partielle vs. correction totale
Tableaux et Procédures

Tableaux: Vérification (1)

Exemple: Calcul du minimum d'un tableau
Écrire algorithme; donner pré- et postcondition

$\{ taille(A) > 0 \}$

```
min = A[0];
j = 1;
while (j < taille(A)) {
   if (A[j] < min)
      min = A[j];
   j = j + 1;
  }
```

$\{\forall\ i.\ 0 \leq i \leq taille(A) - 1 \Rightarrow min \leq A[i]\}$

Invariant?

Notions de base
Sémantique
WP-calcul
Motivation
Conventions et notions de base
Règles d'inférence
Correction partielle vs. correction totale
Tableaux et Procédures

Tableaux: Vérification (2)

“Tous les éléments jusqu'à l'index `j` ont la propriété souhaitée”
Postcondition Q: $\{\forall i.0 \leq i \leq taille(A) - 1 \Rightarrow min \leq A[i]\}$
Invariant I: $\{(\forall i.0 \leq i \leq j - 1 \Rightarrow min \leq A[i]) \ \wedge \ (j \leq taille(A))\}$
Vérification:

- Presque immédiat: Sortie de la boucle:
  $j \geq taille(A) \wedge I \Rightarrow Q$
- Aussi facile: Entrée de la boucle:
  $taille(A) > 0 \wedge min = A[0] \wedge j = 1 \Rightarrow I$
- À faire: préservation de l'invariant

Notions de base
Sémantique
WP-calcul
Motivation
Conventions et notions de base
Règles d'inférence
Correction partielle vs. correction totale
Tableaux et Procédures

Tableaux: Exercices

- Montrer: Echanger deux éléments dans un tableau d'entiers n'affecte pas la somme des éléments.
- Écrire une fonction qui teste si le tableau contient un palindrome
- Vérifier un algorithme de tri à la bulle

Procédures (1)

Les procédures sont introduites en deux étapes:

- *Procédures non-récursives*
  - sont essentiellement des abréviations ("macros")
  - pas de souci de terminaison des appels récursifs
- *Procédures récursives*
  - Pas de traduction immédiate en code impératif
  - Terminaison n'est pas garantie

Pourtant, l'approche est essentiellement la même . . .

Procédures (2)

Nous nous limitons à un format spécifique:

- *Définitions de procédure* ont le format

  ```
  T f (T1 x1, ... Tn xn) { c }
  ```

  - Paramètres formels `x1..xn` (avec types `T1..Tn`)
  - Une seule valeur de résultat (de type `T`)
  - Passage de paramètres: seulement par valeur et non pas par référence
  - Interdit: Référence à des variables globales dans `c`
  - Réécrire `return e` par `res := e`, pour var. spéciale `res`
- *Appels de procédure* ont le format `x := f(a1,..,an)`
  - Pas d'appels imbriqués dans des expressions, comme dans `f(a) + g(b)`

Procédures (3)

Restrictions: *Pourquoi pas de passage par référence?*
Problèmes d'alias: deux variables référencent la même case de mémoire.
Quelle est la valeur de `x` après exécution de

```
f(&x, &x);
```

pour:

```
void f(int * a, int * b) {
  *a = 3;
  *b = 4; }
```

$\leadsto$ raisonnement complexe sur structure de la mémoire

Procédures (3)

Restrictions: *Pourquoi pas d'appels imbriqués?*
Quel est le résultat de

```
x = 1; y = 2;
z = f(x) + g();
```

avec les définitions

```
int f(int x) { y = y + 1; return x + 1; }
int g() { return y - 1; }
```

Solution: Introduire variables auxiliaires:

```
x = 1; y = 2;
vf = f(x); vg = g();
z = vf + vg;
```

Procédures (non-récursives) (1)

Spécification d'une procédure `f`

$\{P\}$ `T res := f (T1 x1, ... Tn xn)` $\{Q\}$

Implantation d'une procédure `f`

```
T f (T1 x1, ... Tn xn) { c }
```

Correction de l'implantation par rapport à la spécification:
Il faut démontrer que

$\{P\}$ `c` $\{Q\}$

Notions de base
Sémantique
WP-calcul
Motivation
Conventions et notions de base
Règles d'inférence
Correction partielle vs. correction totale
Tableaux et Procédures

Procédures (non-récursives) (2)

Exemple: Calcul du minimum
Spécification (version 1):

$\{\top\}$
`int res := min (int x1, int x2)`
$\{res \leq x1 \wedge res \leq x2\}$

Implantation et correction:

```
int min (int x1, int x2) {
  {⊤}
  if (x1 <= x2)
    res := x1;
  else
    res := x2;
  {res ≤ x1 ∧ res ≤ x2}
}
```


Notions de base
Sémantique
WP-calcul
Motivation
Conventions et notions de base
Règles d'inférence
Correction partielle vs. correction totale
Tableaux et Procédures

Procédures (non-récursives): Règles (1)

Appel de fonction: Première approximation:

$$pfp(\texttt{r := f(a1, .., an)}, Q') = P[\vec{x} \leftarrow \vec{a}]$$

si $Q[res \leftarrow r][\vec{x} \leftarrow \vec{a}] \equiv Q'$
Justification: Remplacer appel de procédure par le corps de la procédure
Exemple:

$\{\top\}$
`r := min(e1, e2);`
$\{r \leq e1 \wedge r \leq e2\}$

Notions de base
Sémantique
WP-calcul
Motivation
Conventions et notions de base
Règles d'inférence
Correction partielle vs. correction totale
Tableaux et Procédures

Procédures (non-récursives): Règles (2)

Souvent, la postcondition ne se trouve pas dans la forme requise. Alors, appliquer
Règle de conséquence:

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

Exemple:

$\{\top\}$
`r := min(e1, e2);`
$\{r \leq e1 \wedge r \leq e2\}$
$\{r \leq e1\}$ `/* par règle de conséquence */`

Procédures (non-récursives): Règles (3)

Problème: Perte d'information par application de la spécification “simple”
Mieux: prendre en compte le contexte:

$$pfp(\texttt{r := f(a1, .., an)}, Q' \wedge C) = P[\vec{x} \leftarrow \vec{a}] \wedge C$$

si $Q[res \leftarrow r][\vec{x} \leftarrow \vec{a}] \equiv Q'$
Exemple: (où $C \equiv e1 \leq n$)

$\{e1 \leq n\}$
`r := min(e1, e2);`
$\{r \leq e1 \wedge r \leq e2 \wedge e1 \leq n\}$
$\{r \leq n\}$ `/* par règle de conséquence */`

Spécifications de Procédures

Est-il possible de dériver:

$\{e1 \leq e2\}$
`r := min(e1, e2);`
$\{r = e1\}$

Non! La spécification de `min` est trop faible!
À noter:

- La notion de *pfp* est relative à une spécification
- Une spécification “mal faite” entraîne une perte d'information

À faire: Préciser la spécification de `min`

Procédures non-récursives: Résumé ...

Qu'est-ce qu'il faut prouver?

1. Correction de l'implantation par rapport à sa spécification
   - ne dit rien sur l'*utilisation correcte* de la procédure
2. Correction des appels
   - se fait vis-à-vis de la spécification
   - ne prend pas en compte l'implantation de la procédure
   - *Avantage:* Modularité: Implantation peut être échangée sans affecter les procédures appelantes.

Problème: Les deux aspects sont inséparables pour *procédures récursives*