Algorithmes, Types, Preuves

M. Strecker

Année 2005/2006

Plan

Plan du semestre

- Session 1 (4/11): Bases: Définitions inductives
- Session 2 (9/11): Sémantique opérationnelle
- Session 3 (25/11): WP-calcul: instructions simples
- Session 4 (2/12): Rappel de logique
- Session 5 (9/12): WP-calcul: fonctions
- Session 6 (16/12): 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

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

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. . .]

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 $\Leftarrow$

Définitions inductives: Interprétation (3)

Exemples:

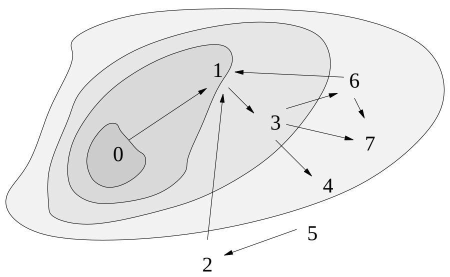
- $P_0 = \{0, 2, 4\}$ n'est pas fermé par application des règles (Z), (PD)
- $P_1 = NAT$ est fermé par application des règles (Z), (PD)
  . . . mais n'est pas l'ensemble le plus petit
- $P_2 = \{n \mid \exists k. n = 2 * k\}$ est
  - fermé par application des règles (Z), (PD)
  - est le plus petit ensemble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

  (preuve: voir plus tard)

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 $e \in S \ldots$

Principe du chaînage en avant

- Appliquer des règles “de haut en bas”, en commençant par les axiomes
- Générer successivement les ensembles $S_1, S_2, \ldots,$ correspondant à 1, 2, ... applications des règles
- Tester si $e \in S_i$ pour un i
- *Problèmes:*
  - les $S_1, S_2, \ldots$ 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

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

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:

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

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

Règles de Calcul (2)

... pour la logique propositionnelle

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

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

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

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

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

Règles de Calcul (3)

Règles supplémentaires pour la logique des prédicats

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

sous condition que x n'est pas libre dans Γ

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

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 $\mathcal{F} = A \vee (B \wedge C)$

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

Exemples de Dérivation: Logique des prédicats (1)

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

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

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 $\langle c, \sigma \rangle \rightarrow \sigma'$ 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 $\langle c, \sigma\rangle \to \sigma'$ par induction.

Affectation

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

Exemple:

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

Exécution de commandes

Skip

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

Séquence

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

Exemple:

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

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

donc:

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

Exécution de commandes

Sélection

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

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

Exemple:

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

Exécution de commandes

Boucle

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

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

Exemples:

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

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

Notions de base
Sémantique
WP-calcul
Introduction à B
Motivation
Langage de programmation
Sémantique opérationnelle
Équivalence de programmes

Exécution de commandes

Observations: L'exécution des commandes

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

Notions de base
Sémantique
WP-calcul
Introduction à B
Motivation
Langage de programmation
Sémantique opérationnelle
Équivalence de programmes

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.

Notions de base
Sémantique
WP-calcul
Introduction à B
Motivation
Langage de programmation
Sémantique opérationnelle
Équivalence de programmes

Extraction d'instructions communes (2)

Schéma général

original ...

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

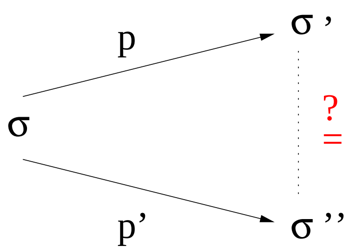
transformé ...

```
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:
$\forall \sigma, \sigma', \sigma''. \langle p, \sigma \rangle \to \sigma' \Rightarrow \langle p', \sigma \rangle \to \sigma'' \Rightarrow \sigma' = \sigma''$



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:

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

Preuves d'équivalence de programmes (3)

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

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

Premier cas: Simplifier:
$\langle \mathtt{if}\ (b)\ \{c_1; c_2;\}\ \mathtt{else}\ \{c_1; c_3\}, \sigma \rangle \to \sigma_1$
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
- $\langle (c_1; \mathtt{if}\ (b)\ \{c_2;\}\ \mathtt{else}\ \{c_3\}), \sigma \rangle \to \sigma_2$ 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?
$\rightsquigarrow$ 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
$évaluable(e_1 + e_2) = évaluable(e_1) \wedge évaluable(e_2)$
$évaluable(e_1 / e_2) = évaluable(e_1) \wedge évaluable(e_2) \wedge e_2 \neq 0$
$évaluable(a[i]) = évaluable(a) \wedge évaluable(i) \wedge 0 \leq i \leq taille(a)$

Exemple: $évaluable(a[k + 5]) = 0 \leq k + 5 \leq taille(a)$

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:

- $pfp(\texttt{x = x - 1}, x > 0) \equiv$
  $x > 0\ [\,\mathrm{x} \leftarrow \mathrm{x} - 1] \wedge \textit{évaluable}(x - 1) \equiv$
  $x - 1 > 0 \wedge \top$
- 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:
  $$pfp(\texttt{\{\}}, Q) = Q$$

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:

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

Exemple:

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

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

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

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

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

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

$$pfp(\texttt{while (b) /* INV: I */ c}, \quad Q \wedge \neg b) = I$$

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: $(I \wedge b) \Rightarrow pfp(\texttt{c}, I)$
3. Sortie de la boucle: $(I \wedge \neg b) \Rightarrow Q$

Correction d'une boucle (2)

Exemple:

$\{\top\}$
`x = 0; y = 0;`
$\{x = 0 \wedge y = 0\}$
`while (x < 3)        /* INV: {` $y = 2 * x \wedge x \leq 3$ `} */`
`  { x = x + 1;    y = y + 2;   }`
$\{x = 3 \wedge y = 6\}$

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

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

$\{z = A^B\}$

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

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.

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?

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 "symmétrique", 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":*

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

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?

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

Vérification:

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

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` $\{Q\}$

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}
}
```


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);`
$\{r \leq e1 \wedge r \leq e2\}$

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);`
$\{r \le e1 \wedge r \le e2\}$
$\{r \le e1\}$ `/* 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 $C \equiv e1 \le n$)

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

Spécifications de Procédures

Est-il possible de dériver:

$\{e1 \le e2\}$
`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*

Interfaces (1)

Une interface décrit le mode d'interaction avec un composant / une fonction / une procédure

But:

- Documenter le fonctionnement
- Éviter certaines erreur par analyse statique du code

Interfaces (2)

Exemple: Typage de fonctions

```
bool divisible (int n, int d) {
  return (n mod d == 0); }
```

- Impossible d'appeler `divisible(3.0, 2.0)`
  $\rightsquigarrow$ erreur de typage
- En C: possible d'écrire `5 + divisible(3, 2)`
  $\rightsquigarrow$ langage mal conçu
- ... et en Java?

Interfaces (3)

La fonction `divisible` ne doit pas être appelée avec un diviseur = 0.

```
bool divisible (int n, int d) {
  /* Precondition: d /= 0 */
  return (n mod d == 0); }
```

- Les langages de programmation courants ne prennent pas en compte les "contraintes sémantiques"
- Méthode B: Conçue pour exprimer des pré- / post-conditions sur des programmes

Spécification vs. Implantation (1)

Spécification:

- Description *abstraite* d'une fonction / d'une procédure
- N'impose aucune implantation particulière
- En général plus court qu'une implantation

Exemple: "Le plus grand diviseur de *n*":

$$pgd(n) = d \Leftrightarrow n \mod d = 0 \wedge (\forall d'.n \mod d' = 0 \Rightarrow d' \leq d)$$

pgd(14) = 7 (et non 2) ???
Comment améliorer la spec?

Spécification vs. Implantation (2)

Implantation:

- Description *algorithmique* d'une fonction / d'une procédure
- Influencée par des considérations de complexité / efficacité
- Une spec peut être réalisée par plusieurs implantations

Exemple: Pour calculer $pgd(n)$:
Algorithme 1

1. Initialiser $d := n - 1$
2. Tant que $n \mod d \neq 0$, décrémenter d
3. Renvoyer d

Spécification vs. Implantation (3)

Algorithme 2

1. Décomposer n en k facteurs premiers
2. Calculer d comme le produit des $k - 1$ plus grands facteurs

Les deux algorithmes sont-ils corrects?
Lequel est plus efficace?

Notions de base de B

Le *raffinement* est une relation entre une spécification et une implantation.
En B:

- La spécification est donnée par une *machine abstraite*
- qui peut être raffinée (successivement) par plusieurs implantations (*implementation*)

Les spécifications en B sont *typées* . . . par des *ensembles*.
Les raffinements et la vérification de types peuvent engendrer des *obligations de preuve*.

Méthode B et Atelier B (1)

Méthode B:
Précurseurs:

- Preuves de programmes promues par R. Floyd, CAR Hoare, E. Dijkstra, D. Gries (années 1965-1985)
- Méthodes de spécification: VDM, Z (plus répandu dans le monde anglo-saxon)
- Méthode B développée par J.R. Abrial (années 1990-. . .)

Méthode B et Atelier B (2)

Atelier B: Outil pour aider à la

- écriture des machines B (gestion de projets, éditeurs . . .)
- vérification des types
- gestion des obligations de preuves
- preuves automatiques et assistées par l'utilisateur

Disponibilité:

- Atelier B (utilisé en TP), commercialisé par ClearSy System Engineering
- Version *open source*: `http://www.B4free.com/`
- Plus de références: `http://vl.fmnet.info/b/`

Machine: Syntaxe

Structure de base (dans le fichier `compte.mch`):

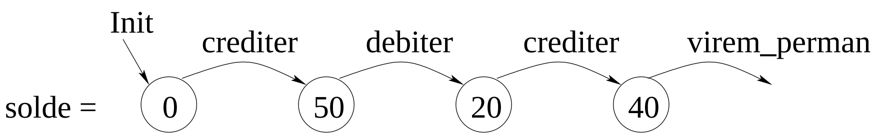
```
MACHINE compte       /* Machine avec le nom 'compte' */

VARIABLES solde

INITIALISATION solde := 0

INVARIANT      /* Propriété à préserver */
   solde : INT & solde > - 100

OPERATIONS     /* Manipulent l'état de la machine*/
/* débiter, créditer ... */
END
```

On peut omettre ou rajouter certaines clauses . . .

Machine: Sémantique

Système de transition:

- Espace d'état: donné par les variables
- Transitions: Comme définies par les opérations
- Invariant: Propriété à préserver après chaque opération



Machine: autres clauses

- `CONSTANTS`: Comme variables, mais non modifiables
- `SETS`
  - Ensembles abstraits: `SETS Personne`
  - Ensembles énumérés: `SETS Sexe = {femme, homme}`
- `PROPERTIES`: . . . des constantes / ensembles

```
CONSTANTS
   decouv_autor    /* decouvert autorisé */

PROPERTIES        /* les valeurs possibles */
   decouv_autor : INT &
   -300 <= decouv_autor  & decouv_autor <= -100

INVARIANT      /* Propriété à préserver */
   solde : INT & solde > decouv_autor
```

Opérations (1)

Deux genres d'opérations:

- *Substitutions généralisées*
  - Utilisées dans les machines abstraites
  - En général: Non pas exécutables
- *Opérations B0*
  - Utilisées dans les implantations
  - B0: sous-langage exécutable de B
  - directement traduisible vers des langages impératifs (comme C)
  - ⇝ voir plus tard

Opérations (2)

Une forme courante de substitution généralise:
précondition - postcondition

```
OPERATIONS
crediter (somme) =
   PRE
     somme : NAT & somme > 0
   THEN
     solde := solde + somme
   END
```

Définir: opération `débiter`

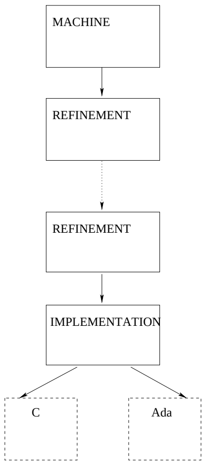
Opérations (3)

Spécifications trop algorithmiques?
Scénario: Retrait d'une somme d'un distributeur de billets.
Comportement du distributeur: non-déterministe, peut rendre moins que souhaité

```
retrait_aut (somme) =
   PRE
     somme : NAT & somme > 0
   & solde - somme > decouv_autor
   THEN
     solde : (solde$0 - somme <= solde
              & solde < solde$0)
   END
```

Notation: `solde$0` est l'ancienne valeur de `solde`.

Modules



Il y a trois catégories de modules en B:

- Machine abstraite (`MACHINE`): Spécification du module
- Raffinement (`REFINEMENT`): un ou plusieurs modules intermédiaires
- Implantation (`IMPLEMENTATION`): Programme exécutable ⇝ permet génération de code (en C, C++, Ada)

Implantation: Syntaxe

Structure de base (dans le fichier `compte_imp.imp`):

```
IMPLEMENTATION compte_imp
REFINES compte   /* Raffinement de la spéc. */

CONCRETE_VARIABLES solde

INITIALISATION
   solde := 0

VALUES
   decouv_autor = -200

OPERATIONS
/* créditer ... */

END
```


Implantation: Sémantique

Une implantation fournit des opérations exécutables qui satisfont les contraintes imposées par la machine abstraite.
Par exemple:

- Raffinement de constantes:
  - Machine abstraite: Déscription par une propriété

    ```
    PROPERTIES
     decouv_autor : INT &
     -300 <= decouv_autor & decouv_autor <= -100
    ```
  - Implantation: Valeur spécifique

    ```
    VALUES
       decouv_autor = -200
    ```
- Opérations . . .

Implantation d'une opération (1)

Une opération simple:

```
OPERATIONS
crediter (somme) =
  BEGIN
     solde := solde + somme
  END
```

- Presque aucune différence par rapport à la machine abstraite
- Pas de typage explicite
  $\leadsto$ `somme : NAT` hérité de la machine abstraite

Implantation d'une opération (2)

L'implantation "naturelle" de `retrait_aut`

```
retrait_aut (somme) =
  BEGIN
     solde := solde - somme
  END
```

Une implantation alternative (distributeur de billets méchant):

```
retrait_aut (somme) =
  BEGIN
     solde := solde - somme / 2
  END
```

Les deux implantations sont-elles correctes?

Obligations de preuve (1)

Des obligations de preuve (*proof obligations, PO*) sont générées pour garantir

- la consistance interne d'un composant:
  - Vérification de types
  - Respect des invariants
- la correction d'un raffinement:
  - Valeurs de constantes . . .
  - Opérations: Correction par rapport à pre-/ postcondition

Obligations de preuve (2)

Vérification de types:

```
solde: INTEGER & somme: INTEGER & ....
 => solde+somme: INTEGER
```

dérivé des déclarations de type de `solde` et `somme`
NB: Strictement parlant, des propriétés de typage sont des propositions . . .

Obligations de preuve (3)

Respect des invariants:
Par exemple: Invariant `solde > decouv_autor`
et opération `crediter`
avec précondition `somme > 0`
et substitution `solde := solde + somme`
donne lieu à la PO:

```
solde > decouv_autor & somme > 0 & ...
   => solde+somme > decouv_autor
```

réécrit par B:

```
decouv_autor+1<=solde & 1<=somme & ...
   => decouv_autor+1<=solde+somme
```


Obligations de preuve (4)

Correction pre-/ postcondition: . . . de `retrait_aut`:
Précondition `somme : NAT & somme > 0`
substitution `solde := solde - somme`
et postcondition
`solde$0 - somme <= solde & solde < solde$0`
donnent lieu à la PO:

```
solde$0 - somme <= solde$0 - somme &
solde$0 - somme < solde$0
```

Faire la même chose pour `retrait_aut` (version méchante)

Utilisation de B

Vérification de composants:

- *Type Check:* Vérification de types
- *P0 generate:* Génération d'obligations de preuve
- *Prove:* Lance le prouveur en mode
  - *Automatique:* application automatique de "tactiques" *force* 0 . . . 4 (rapide, + incomplet . . . lent, - incomplet)
  - *Interactif:* preuves guidées par l'utilisateur
- *B0 Check:* Vérifie que module est "implantable"
- *Translator:* Génération de code (C, C++, Ada) d'une implantation

Analyse et documentation:

- *Analysing: Show/Print PO:* Affiche les POs du module
- *Document:* Documents en format LaTeX, Word

Substitutions Généralisées (1)

Idée: Notion générale d'un "programme"

- Substitutions abstraites: n'imposent pas d'implantation, par ex.:
  - Substitution "devient tel que": `x : (P)` choisit un `x` (sans préciser lequel) qui satisfait `P`
    `n : (n : NAT & n mod 2 = 0)`
  - Substitution parallèle, n'impose pas d'ordre: `x1, x2 := a1, a2`
- Substitutions implantables, par ex.:
  - Affectation: `x := 3`
  - Séquence: `x := 3; y := 4`
  - Sélection, boucle, . . .

$\rightsquigarrow$ *Instructions* d'un langage de programmation

Substitutions Généralisées (2)

Utilisation: Une substitution peut être utilisée dans le corps d'une opération.
Restrictions:

- Certaines instructions ne sont pas utilisables dans une `MACHINE`
- Substitutions abstraites interdites dans une `IMPLEMENTATION` $\rightsquigarrow$ B0-Check

Substitutions Généralisées (3)

Sémantique:

- Une substitution S appliquée à un prédicat Q
  - décrit un ensemble d'états
  - . . . qui valident le prédicat Q après application de S

Notation: $[S]Q$
Exemple: $[x := 3](x + y > 5)$
"Ensemble d'états qui valident $(x + y > 5)$ si on calcule $x := 3$"
Solution: États qui valident $(3 + y > 5)$, donc $y > 2$
Analogie avec WP-calcul!

Substitutions Généralisées (3)

Raffinement:

```
MACHINE foo
OPERATIONS
res <-- foop (xx, yy) =
 PRE xx : NAT & yy : NAT & xx <= yy
 THEN
  res : (res: NAT & xx <= res & res <= yy)
END

IMPLEMENTATION foo_imp
REFINES foo
OPERATIONS
res <-- foop (xx, yy) =
 res := (xx + yy) / 2
END
```


Analyses de code

La vérification de la consistance d'un composant (machine, implantation) se fait en plusieurs phases:

1. *Analyse lexicale* de mots-clés, identificateurs, ...
   Ex.: `PROPERTIES` n'est pas écrit `PROPRIETES`
   Ex.: Tout identificateur a au moins deux caractères
2. *Analyse syntaxique* de la structure du code
   Ex.: `x + * 3` est refusé
3. *Analyse sémantique:* Le code "fait du sens"

Les phases 1. et 2. se trouvent dans chaque compilateur
la phase 3. en partie ...

Analyse sémantique

Quelques aspects de l'analyse sémantique:

1. *Expressions bien typées:* $3 + \{x | x > 5\}$ est mal typé
2. *Porté et visibilité de variables:* Dans
   `!xx. (#yy. (yy:INT & xx > yy)) or xx < yy`
   - `xx` n'est pas typé
   - `yy` est hors portée dans `xx < yy`
3. *Règles d'anticollision ...* pour éviter des malentendus:
   Ex: nom de la machine $\neq$ nom de ses opérations
4. *Propriétés à prouver* (invariants, raffinements)

1. et 2. se font habituellement dans des compilateurs
3. se fait dans certains analyseurs (`lint`)
4. est spécifique à B

Types (1)

Petit bestiaire des langages et leurs types:
C (simple et confus):

- *Types de base:* types numériques (`char, int, float...`)
- *Constructeurs de type:* Pointeur (`*`), tableau, `struct`, `union`

Java (complexe, mais propre):

- *Types de base:* types numériques, `bool`; variables de type
- *Constructeurs de type:* classes, interfaces

Types (2)

ML (simple et propre):

- *Types de base:* types numériques; variables de type (`'a`)
- *Constructeurs de type:* Produit (`*`), fonction (`->`), types inductifs (`bool`, `list`, arbres, ...)

Polymorphisme et sous-typage:

- En C: Toute expression a un seul type; mais possible: conversions: `3 + 2.5`
- En ML: Fonctions à type polymorphe: `map: ('a -> 'b) -> 'a list -> 'b list` Application à un élément spécifique: `map f [1;2;3]`
- En Java: Application d'une méthode à des instances de sous-classe

Types de B (1)

En B, les types sont assimilés à des *ensembles*.
Types de base:

- `INTEGER`, les entiers "mathématiques" (non bornés) notation alternative: $\mathbb{Z}$
- `BOOL`, avec des éléments `TRUE` et `FALSE`
- `STRING`: chaînes de caractères
- ensembles abstraits introduits dans la clause `SETS` *Ex.:* `PERSONNE`

Notation: On écrit `e:T` (ou $e \in T$) pour `e` a type `T`

Types de B (2)

Constructeurs de types:

- Ensemble des parties: Si `T` est un type, alors `POW(T)` est un type
  Éléments: `S : POW(T)` $\Leftrightarrow (\forall x. x \in S \Rightarrow x \in T)$
  Notation alternative: $\mathbb{P}(T)$
- Produit cartésien: Si `T1` et `T2` sont des types, alors `T1 * T2` est un type
  Éléments: Paires `(e1, e2)` avec `e1:T1` et `e2:T2`
  Notation alternative: $T_1 \times T_2$
- Structures: Si `T1 ... Tn` sont des types, alors `struct (id1:T1, ..., idn:Tn)` est un type
  Éléments: records `rec (id1:val1, ..., idn:valn)`
  Ex.: `rec(a: 5, b: TRUE)` a type `struct(a: INTEGER, b: BOOL)`

Construction d'ensembles

À partir des types, on peut construire des ensembles – essentiellement à l'aide de la compréhension ensembliste
Principe:

- Tout type est un ensemble.
- Si S est un ensemble, alors $\{x | x \in S \wedge P\}$ est un ensemble.

Appartenance: $e \in \{x | x \in S \wedge P\} \quad \Leftrightarrow \quad (e \in S \wedge [x := e]P)$
Exemple:
$F \in \{X | X \in \mathbb{P}(\mathbb{Z}) \wedge fin(X)\} \equiv$
$F \in \mathbb{P}(\mathbb{Z}) \wedge fin(F) \equiv$
$(\forall x. x \in F \Rightarrow x \in \mathbb{Z}) \wedge fin(F)$

Notions ensemblistes dérivées (1)

Les constructeurs et la compréhension permettent de définir:
Opérations ensemblistes:

- $S \cup T \equiv \{a | a \in U \wedge (a \in S \vee a \in T)\}$
- $S \cap T \equiv \{a | a \in U \wedge (a \in S \wedge a \in T)\}$
- $S - T, \emptyset, \ldots$ comment les définir?

U?? $\rightsquigarrow$ dépend du type, voir plus tard
Prédicats sur des ensembles:

- $S \subseteq T \equiv \ldots$
- $S \subset T \equiv \ldots$

Notions ensemblistes dérivées (2)

Relations:

- Relation: $S \leftrightarrow T \;\equiv\; \mathbb{P}(S \times T)$
- Inverse: $R^{-1} \;\equiv\; \{(b, a) \in U \times V \wedge (a, b) \in R\}$
- Domaine: $dom(R) \;\equiv\; \{a | a \in U \wedge \exists b.(b \in V \wedge (a, b) \in R)\}$
- Codomaine: $ran(R) \;\equiv\; dom(R^{-1})$
- Image: $R[S] \;\equiv\; \{t | t \in U \wedge \exists s.(s \in S \wedge (s, t) \in R)\}$
- Composition:
  $R_1 ; R_2 \;\equiv\; \{(a, c) | \exists b.(a, b) \in R_1 \wedge (b, c) \in R_2\}$
- Identité: $id(R) \;\equiv\; \{(a, b) | (a, b) \in U \times U \wedge a = b\}$

Notions ensemblistes dérivées (3)

Fonctions:

- Partielles: $S {+\!\!-\!\!>} T \;\equiv\; \{R | R \in S \leftrightarrow T \wedge (R^{-1} ; R) \subseteq id(T)\}$
- Totales: $S \to T \;\equiv\; \{f | f \in S {+\!\!-\!\!>} T \wedge dom(f) = S\}$
- Surjectives
- Injectives ...

Completer!
À noter: En B, “$\rightarrow$” n’est pas un constructeur de types

Notions ensemblistes dérivées (4)

Nombres:

- $NATURAL \equiv \{n | n \in \mathbb{Z} \wedge n \geq 0\}$
- $INT \equiv \{i | i \in \mathbb{Z} \wedge representable(i)\}$
- $NAT \equiv \{n | n \in NATURAL \wedge representable(n)\}$
- $a..b \equiv \{i | i \in mathbbZ \wedge a \leq i \leq b\}$

où $representable(i)$ signifie: “représentable sur une machine”.
Dépend de l’architecture, par exemple: $-2^{32} \leq i \leq 2^{32} - 1$
Autres: On *peut* définir

- Types de données inductifs
- Relations inductives

... mais pas de manière élégante $\rightsquigarrow$ faiblesse de B

Typage: Idée (1)

But: Déterminer si une expression “fait du sens”

- Première étape
  (… en restant dans un fragment décidable)
- … avant d'aborder des preuves (difficiles, fragment indécidable)

Exemples:

- Soient:

```
b1: BOOL & x1: INTEGER &
f1: INTEGER --> INTEGER
```

  - `f1(x1) = 4` est bien typé
  - `(b1 = TRUE) & (f1(x1) = 4)` est bien typé
  - `f1(b1) = 4` et `f1(x1) = TRUE` ne sont pas bien typés

Typage: Idée (2)

Exemples (suite):

- Soient:

```
b1: BOOL & x1: INTEGER &
f1: 0 .. 10 --> INTEGER
```

  - `f1(x1) = 4` est bien typé
  - `(f1(3) = 4)` est bien typé
  - `(f1(13) = 4)` est bien typé
    (mais causera des difficultés de preuves)
    Raison: Tout i : 0..10 a le type $\mathbb{Z}$, et 13 a le type $\mathbb{Z}$
  - `f1(b1) = 4` n'est pas bien typé
    Raison: Tout i : 0..10 a le type $\mathbb{Z}$, et `b1` a le type `BOOL`.

Règles de typage: Préliminaires

La vérification de types se fait à l'aide:

- du prédicat *check(e)*: Vérifie que l'expression *e* est bien typée
- de la fonction *type(e)*: Détermine le type de l'expression *e*
  Ex.: $type(13) = \mathbb{Z}$
- de la fonction *super(S)*: Détermine l'ensemble le plus général contenant l'ensemble *S*
  Ex.: $super(\{x | x : NAT \& x > 10\}) = \mathbb{Z}$

La vérification se fait dans un environnement

- qui a la forme $[x_1 \in S_1, \ldots x_n \in S_n]$
- … associant un ensemble à une variable.

Jugements de vérification:
$E \vdash check(e)$ ou $E \vdash type(e) = T$ ou $E \vdash super(S) = T$

Règles de typage (1)

check Quelques règles, non exhaustives:
Opérateurs booléens (pareil pour $P \wedge Q, P \vee Q, \neg P \ldots$):

$$\frac{E \vdash check(P) \quad E \vdash check(Q)}{E \vdash check(P \Rightarrow Q)}$$

Quantificateurs (noter: restrictions de syntaxe):

$$\frac{x \notin free(E) \cup free(S) \quad E, x \in S \vdash check(P)}{E \vdash check(\forall x.(x \in S \Rightarrow P))}$$

$$\frac{x \notin free(E) \cup free(S) \quad E, x \in S \vdash check(P)}{E \vdash check(\exists x.(x \in S \wedge P))}$$

où *free(E)* / *free(S)* est l'ensemble des variables libres de *E* / *S*

Règles de typage (2)

check (suite)

$$\frac{E \vdash type(E) = type(F)}{E \vdash check(E = F)}$$

$$\frac{E \vdash type(e) = super(S)}{E \vdash check(e \in S)}$$

$$\frac{E \vdash super(S_1) = super(S_2)}{E \vdash check(S_1 \subseteq S_2)}$$

Règles de typage (3)

type

$$\frac{(x \in S)\ in\ E \quad E \vdash U = super(S)}{E \vdash type(x) = U}$$

$$\frac{E \vdash U = type(a) \times type(b)}{E \vdash type((a, b)) = U}$$

$$\frac{S\ Set \quad E \vdash U = \mathbb{P}(super(S))}{E \vdash type(S) = U}$$

Règles de typage (4)

super

$$\frac{\text{T type de base}}{E \vdash super(T) = T}$$

$$\frac{E \vdash check(\forall x.(x \in S \Rightarrow P)) \quad E \vdash super(S) = T}{E \vdash super(\{x | x \in S \wedge P\}) = T}$$