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

Typage - c'est quoi?

On associe un type à des unités élémentaires:

- des constantes:
  `3` est de type `int`, `2.5` est de type `float`
- à des variables:
  `int n; float x;`
- à des fonctions
  `int fac (int n) { ... }`

... et peut ainsi déterminer le type d'expressions complexes:
`n + fac(3)` a le type `int`

Typage - pourquoi?

Pour des raisons de

- allocation de mémoire: le type d'une valeur détermine sa taille en mémoire
  Ex.: valeur de type `int`: 2 octets, de type `float`: 4 octets (varie selon l'architecture)
- prévention de fautes:
  - involontaires: addition d'un `int` et un pointeur
  - volontaires (brèches de sécurité)
- documentation
  - pour le programmeur
  - pour le compilateur (génération de code plus efficace)

Typage - comment?

Différentes combinaisons de ...

Dynamique vs. statique

- *Dynamique:* le type d'une expression n'est connu que lors de l'exécution
- *Statique:* le type est déjà connu au temps de la compilation

Fort vs. faible

- *Fort:* Une discipline stricte est imposée
- *Faible:* Des déviations de la discipline de typage sont tolérées

Unique vs. multiple

- *Unique:* une expression a un seul type
- *Multiple:* une expression peut avoir plusieurs types (possiblement hiérarchisés)

Typage - Lisp

Lisp (LISt Processor)

- langage fonctionnel
- développé ≈ 1960

Typage dynamique; typage multiple
typage fort (erreurs de typage détectés pendant exécution):

```
(defun foo (a)
  (cond ((<= a 3) (+ a 1))
        (t (+  (car a) 1))))
```

- Appel `(foo 3)` donne `4`
- Appel `(foo 4)`: erreur (`car`: tête de liste)
- Appel `(foo '(2 3))` donne `3`

Typage - Caml (1)

ML/Caml

- Famille de langages fonctionnels
- développés ≈ 1980-1990
- *Typage unique:* chaque terme de Caml a un seul type (plus précisément: un seul type "principal" (généricité!))

```
# 2 + 3;;
- : int = 5
# 2.0 +. 3.5 ;;
- : float = 5.5
# 2 + 3.5 ;;    pas de conversions automatiques!
Characters 4-7:
  2 + 3.5 ;;
      ^^^
This expression has type float
  but is here used with type int
# 2 + int_of_float 3.5 ;;
```


Typage - Caml (2)

- *Typage statique:*
  erreurs de typage détectées avant début de l'évaluation

```
# 3 / 0;;
Exception: Division_by_zero.
# (3 / 0) + [2] ;;
Characters 10-13:
  (3 / 0) + [2] ;;
            ^^^
This expression has type 'a list
  but is here used with type int
```

- *Typage fort*, mais ...
- pas de déclarations explicites: *inférence de types*

Préservation de typage:
Lors de l'évaluation, une expression "garde son type"
Conséquence: pas d'erreur de type lors de l'exécution

Typage - C (1)

C

- Langage impératif, développé $\approx$ 1970
- *Typage statique* (pendant la compilation)
- *Types multiples*:
    - Une expression peut adopter des types différents, selon contexte
    - Le compilateur insère des *conversions / casts*
      $\rightsquigarrow$ résultat souvent difficile à prédire

```
printf("%d", (2+4)/5); (* résultat: 1 *)
printf("%f", (2+4)/5); (* parfois: -0.045894 *)
printf("%f", (2.+4)/5); (* résultat: 1.2000 *)
printf("%f", (2+4)/5.); (* résultat: 1.2000 *)
```


Typage - C (3)

Langage au typage *faible* et *bizarre*

- pas de type booléen *Quelle est la valeur imprimée par:*

```
x = 0.5;
if (x = 2.5) printf ("true\n");
else  printf ("false\n");
```


Typage - C (4)

- confusion entre tableaux et types de pointeur
- conversions arbitraires entre caractères, entiers et pointeurs

```
int n;
int * p;
p = (int *) malloc (sizeof(int) * 2);
p[0] = 12345;
p[1] = 67899;
n = (int) p;
n = n + 4;
p = (int *) n;
printf ("%c\n", (char)*p);
```

Valeur imprimée: ;

Java

- Langage orienté objet (classes, interfaces)
- développé $\approx$ 2000
- Typage *statique fort*
  ($\rightsquigarrow$ pas d'erreur de type lors de l'exécution)
- Sous-typage
    - classes $\leftrightarrow$ classes
    - interfaces $\leftrightarrow$ interfaces

  . . . et réalisation
    - classes $\leftrightarrow$ interfaces

  $\rightsquigarrow$ typage multiple (assez complexe . . .)

$\Longrightarrow$ Syntaxiquement pareil à C, Java a un typage considérablement plus “sûr” que C $\Longleftarrow$

Présentation des langages

Dans la suite, on étudiera le système de typage d'un langage impératif.
On introduit des langages de complexité croissante:

1. langage avec expressions et instructions simples: $\mathcal{L}_e$
2. langage avec fonctions: $\mathcal{L}_f$
3. langage avec sous-typage: $\mathcal{L}_s$

Expressions et instructions de $\mathcal{L}_e$ (1)

Expressions
Syntaxiquement, on ne distingue plus entre expressions arithmétiques et booléennes:

$$
\begin{array}{lcll}
E & ::= & n & \text{(Constantes entières } n \in \mathbb{Z}) \\
 & | & b & \text{(Constantes booléennes } b \in \mathit{Bool}) \\
 & | & v & \text{(Variables } v \in \mathcal{V}) \\
 & | & (E + E) \mid (E - E) \mid \ldots & \\
 & | & (E == E) \mid (E < E) \mid \ldots & \\
 & | & !\, E \mid\mid (E \,\&\&\, E) \mid \ldots &
\end{array}
$$

Instructions presque comme avant ...

$$
\begin{array}{lcl}
C & ::= & v = E \\
 & | & C \,;\, C \\
 & | & \texttt{if } E \texttt{ then } C \texttt{ else } C \\
 & | & \texttt{while } E \texttt{ do } C
\end{array}
$$

Expressions et instructions de $\mathcal{L}_e$ (2)

On peut maintenant écrire des expressions

- *bien typées:*
  - $3 + (5 - 2)$
  - $((4 * 2) < 42) \&\& ((4 + 2) == 5)$
- *mal typées:*
  - $((4 * 2) < \mathit{true}) + 7$

Est-ce que $((v1 * 2) < 42) || v2$ *est bien typée?* Ceci dépend des déclarations:

- Oui, si `v1:int` et `v2:bool`
- Autrement: Non

Types et déclarations

Types Nous utilisons les types suivants:

- `bool` pour les valeurs de vérité
- `int` pour les entiers
- `void` pour des instructions bien typées

Déclarations

- Une déclaration associe un type (`bool` ou `int`) à une variable
- Il n'est pas possible de déclarer une variable de type `void`

Environnements

Un environnement est une liste (*ordre important!*) qui

- associe un type à une variable
- représente les déclarations en vigueur à une position du programme.

Les environnements peuvent varier, selon la position:

```
int n;
int f1(int b) {
  int n;
  //(1)
  return (b + n); }
```

```
int f2(int a) {
   bool b;
   //(2)
   n = a;
   return 0; }
```

- Env1 = [(int n), (int b), (int n)]
- Env2 = [(int n), (int a), (bool b)]

Règles de typage (1)

Format des règles: $Env \vdash e : T$, où

- Env est un environnement
- e une expression (resp. une instruction)
- T un type

Définition par induction sur la structure des expressions/instructions:

Constantes:

$$\frac{n \in \mathbb{Z}}{Env \vdash n : int} \qquad \frac{b \in \{true, false\}}{Env \vdash b : bool}$$

Règles de typage (2)

Variables:

$$\frac{tp(v, Env) = T}{Env \vdash v : T}$$

où $tp(v, Env) = T$ si $(T\ v)$ est la décl. la plus à droite dans Env

Opérations unaires et binaires:

$$\frac{Env \vdash e_1 : int \quad Env \vdash e_2 : int}{Env \vdash e_1 + e_2 : int}$$

$$\frac{Env \vdash e : bool}{Env \vdash !e : bool}$$

Compléter!

Règles de typage (3)

Toutes les instructions c sont typées avec *void*

Donc, $Env \vdash c : void$ signifie: c est bien typée

Affectation:

$$\frac{tp(v, Env) = T \quad Env \vdash e : T}{Env \vdash (v = e) : void}$$

Boucle:

$$\frac{Env \vdash e : bool \quad Env \vdash c : void}{Env \vdash \texttt{while}\ e\ \texttt{do}\ c : void}$$

Compléter!

Exemples de typage

- Montrer: L'expression `!((3 + 4) < 34)` est bien typée
- L'expression `(x == 2) || (y == z) || b` est bien typée dans quels environnements?
- Montrer: `while (1) do x = x + 1;` n'est pas bien typé

Syntaxe de $\mathcal{L}_f$ (1)

Déclarations

D ::= $T v$ ($v \in \mathcal{V}$, $T \in$ *types*)

Définition de fonctions

F ::= T fn (D*) { D*; C; `return` E; } composée de:

- type de résultat T
- nom de la fonction *fn* (un identificateur)
- une liste $D*$ de déclarations de paramètre (séparées par ',')
- une liste $D*$ de déclarations de variables locales (sép. ';')
- une instruction C
- `return` avec une expression E

Syntaxe de $\mathcal{L}_f$ (2)

Définition de programmes

P ::= D* F*

composée de:

- une liste D^* de déclarations de variables globales
- une liste F^* de définitions de fonctions

Instructions: on ajoute des appels de fonction avec affectation:

C ::= ...

| v = *fn* (E*) $v \in \mathcal{V}$, *fn* nom de fct.

Environnements pour $\mathcal{L}_f$ (2)

Un environnement *Env* a maintenant deux composants:

- *Env.vars* (comme avant): liste qui associe des types à des variables
- *Env.fns* (nouveau): liste composée des profils des fonctions du programme

Un profil de fonction est un en-tête de programme, sans les noms de variable.

Ex.: le profil de la fonction

```
int f (int n, bool b) { ... }
```

est

```
int f (int, bool)
```

Vérification de types (1)

1. Analyse syntaxique du programme P
2. Construction d'un environnement initial Env_{in}, avec
   - $Env_{in}.vars$: déclar. des var. globales de P
   - $Env_{in}.fns$: profil des fonctions de P
3. Vérification des corps des fonctions.
   Pour chaque f définie dans P, faire:
   1. Construire environnement Env_f, avec
      - $Env_f.vars$:
        $Env_{in}.vars$, plus décl. des paramètres et des var. locales de f
      - $Env_f.fns$: le même que $Env_{in}.fns$
   2. Vérifier le corps de f sous l'environnement Env_f
      $\rightsquigarrow$ légère adaptation des règles de typage

Vérification de types (2)

Adaptation des règles

- Modifier les règles pour prendre en compte l'environnement plus complexe
- Ajouter une règle pour l'appel des fonctions
- Ajouter une règle pour `return` E

Points délicats

- Pourquoi n'est-il pas possible de faire la vérification de types en même temps que l'analyse syntaxique?
- Comment sont traitées différentes occurrences de variables du même nom?

Convertibilité et sous-types

Un élément e_1 d'un type T_1 est convertible vers un type T_2 si e_1 peut être utilisé dans un contexte où des éléments de T_2 sont attendus.

- *Conversion explicite* avec une fonction:
  en Caml:
  ```
  # 2 + int_of_float 3.5 ;;
  - : int = 5
  ```
  en C: `2 + (int) 3.5`
- *Conversion implicite* (en C):
  le compilateur insère des conversions lui-même

Sous-typage: Forme plus stricte de convertibilité
Utilisé surtout en programmation orientée objet

Types arithmétiques en C

En C, on a les types pour représenter:

- des entiers:
  - `signed char, short int, int, long int, long long int`
  - des types `unsigned` correspondants
    ignorés dans la suite!
- des nombres réels: `float, double, long double`

Définir un rang $\preceq$ entre les types:
Fermeture transitive de: `signed char` $\preceq$ `short int` $\preceq \ldots$
$\preceq$ `double` $\preceq$ `long double`
Définir le rang maximal $max_{\preceq}$ de deux types:
Ex.: $max_{\preceq}(\texttt{int}, \texttt{long int}) = \texttt{long int}$
$max_{\preceq}(\texttt{int}, \texttt{double}) = \texttt{double}$

Vérification de types arithmétiques

Opérateurs arithmétiques sans sous-types:

$$\frac{Env \vdash e_1 : int \quad Env \vdash e_2 : int}{Env \vdash e_1 + e_2 : int}$$

... avec sous-types:

$$\frac{Env \vdash e_1 : T_1 \quad Env \vdash e_2 : T_2 \quad Tm = max_{\preceq}(T_1, T_2)}{Env \vdash e_1 + e_2 : Tm}$$

Ex.: Soit $x : \mathtt{float}$

$$\frac{Env \vdash x : \mathtt{float} \quad Env \vdash 3 : \mathtt{int} \quad \mathtt{float} = max_{\preceq}(\mathtt{float}, \mathtt{int})}{Env \vdash x + 3 : \mathtt{float}}$$

Conversions de types (1)

Il est parfois nécessaire d'utiliser des conversions explicites:

```
printf("%f\n", 2.3);      /* imprime 2.300000 */
printf("%d\n", 2.3);      /* imprime 1717986918 */
printf("%d\n", (int)2.3); /* imprime 2 */
```

Règle de typage

$$\frac{Env \vdash e : T' \quad convertible(T, T')}{Env \vdash (T)e \ : \ T}$$

Définition de *convertible(T,T')*: très complexe ...
Approximation: Si $T \preceq T'$ ou $T' \preceq T$, alors *convertible(T,T')*

Conversions de types (2)

Conversions implicites:

- Invisibles au niveau du langage source
- Insérées par le compilateur
- But (entre autres): Déterminer la “bonne” opération

Exemples:

- 5/2 division sur des entiers: $5 /_{int} 2$
- 5.0/2.0 division sur des flottants: $5.0 /_{float} 2.0$
- Comment traiter 5.0 / 2 ?
  - $5.0 /_{float} 2$ impossible
  - $5.0 /_{int} 2$ impossible

  Solution: conversion implicite: $5.0 /_{float} ((\mathtt{float})2)$

Conversions de types (3)

Principe:

- On ne se contente plus de déterminer le type
- mais on insère aussi des conversions, si nécessaire

Règle de typage et conversion
... pour des opérations arithmétiques:

$$\frac{Env \vdash e_1 \rightsquigarrow e_1' : T_1 \quad Env \vdash e_2 \rightsquigarrow e_2' : T_2 \quad T_2 \prec T_1}{Env \vdash e_1 + e_2 \ \rightsquigarrow e_1' + (T_1)e_2' : \ T_1}$$

Exemple

$$\frac{Env \vdash 5.0 \rightsquigarrow 5.0 : \mathtt{float} \quad Env \vdash 2 \rightsquigarrow 2 : \mathtt{int} \quad \mathtt{int} \preceq \mathtt{float}}{Env \vdash 5.0/2 \ \rightsquigarrow 5.0/(\mathtt{float})2 : \ \mathtt{float}}$$

Résumé

Nous avons étudié le système de typage d'un C simplifié.
Extensions pour décrire le langage complet:

- Pas de séparation entre expressions et instructions
- Confusion entre entiers et booléens
- Types arithmétiques `signed` vs. `unsigned`
- `struct, union, enum`
- Pointeurs, conversions pointeurs $\leftrightarrow$ entiers ...
- Pointeurs vs. tableaux, pointeurs de fonctions, ...

Voir: norme ISO, avec $>$ 500 pages Java: les mêmes principes, plus propre. Prendre en compte:

- Sous-types engendrés par sous-classes et interfaces
- Appel dynamique de méthodes

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`):

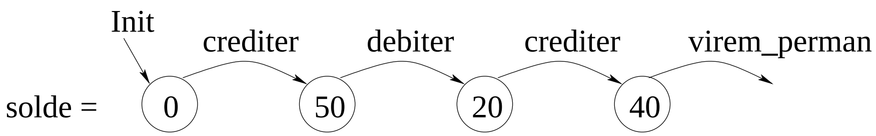
```
MACHINE compte     /* Machine avec le nom 'compte' */

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)
  - $\leadsto$ 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`

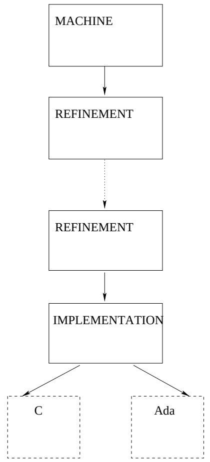
Opérations (3)

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 $\rightsquigarrow$ 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
  $\rightsquigarrow$ `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?

Notions de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

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

Notions de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

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 . . .

Notions de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

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* $0 \dots 4$ (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 de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

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 de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

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 de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

Notions ensemblistes dérivées (3)

Fonctions:

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

Notions de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

Règles de typage (2)

check (suite)

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

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

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

Notions de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

Règles de typage (3)

type

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

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

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

Notions de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

Règles de typage (4)

super

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

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

Idée

Analyses qui sont

- plus puissantes que la vérification de types
- ne demandent pas l'effort et l'expertise d'une preuve

Buts:

- Bonne intégration dans l'environnement de programmation
  $\leadsto$ pre- / postconditions dans commentaires spécifiques
- Rapidité de la vérification
  $\leadsto$ souvent incomplèt (quelques erreurs ne sont pas détectées)
  $\leadsto$ parfois incorrect (faux alarmes)

Notions de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

Splint

Historique:

- Analyseur `lint` disponible depuis $\approx$ 1980 comme analyseur de C
- Splint: Développé à l. Univ. de Virginia entre 2002-2004

Outils "comparables" (parfois techniquement très différents et beaucoup plus performants):

- Blast (base technique: model checking)
- Astrée (ciblée vers des applications embarquées)
- Absint (calcul de pire temps d'exécution)

Notions de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

Splint: Exemple d'utilisation

Programme:

```
int main () {
  int i;
  int tab[5];

  for (i = 0; i <= 5; i = i + 1)
    tab[i] = i; }
```

Appel:

```
> splint +bounds splint1.c
splint1.c: (in function main)
splint1.c:13:5: Possible out-of-bounds store:
    tab[i]
    Unable to resolve constraint:
    requires i @ splint1.c:6:9 <= 4
```


Notions de base
Sémantique
WP-calcul
Typage
Introduction à B
Spécification et Raffinement
Machines Abstraites
Raffinement et Implantation
Substitutions
Vérification de Types en B
Au delà de B: Analyses statiques

Classes d'erreurs détectées (1)

Expressions avec type inapproprié
Code:

```
if (i = 0) {...}
```

Message:

```
Test expr. for if not boolean, type int: i = 0
```

Classes d'erreurs détectées (2)

Bornes d'accès à des tableaux:

Code:

```
void f (int t [], int n)
/*@modifies t@*/
/*@requires (maxSet(t) + 1) >= (n); @*/
 {int i;
  for (i = 0; i <n; i = i + 1)
    t[i] = i;
  return;}
...
int tab[5];
f(tab, 6);
```

Message:

```
Unable to resolve constraint: requires 4 >= 5
 precondition: requires maxSet(tab ) >= 5
```


Classes d'erreurs détectées (3)

Modifications involontaires:

```
void g (int * x, int * y)
/*@modifies *x @*/ {
  *x = 3;
  *y = 4;}
```

Message:

```
Undocumented modification of *y: *y = 4
```

Note: Très faible (problème des alias):
Qu'est-ce qui se passe pour

```
void g (int * x, int * y)
/*@modifies *x @*/ {
  if (x == y) *x = 3;}
```