Programmation Dirigée par la Syntaxe (PDS)

CM4 - Typage & Vérification de type

ISTIC, Université de Rennes 1
Sebastien.Ferre@irisa.fr

PDS, M1 info

Plan

Plan

Typer c'est quoi ?

Typer c'est :

- classer et normaliser les objets élémentaires
  - ex : entiers, booléens, flottants, caractères
- afin de savoir les composer, les assembler
  - ex : tableaux, structures
- en se protégeant contre des erreurs courantes
  - ex : `tab.champ`, `entier[i]`, `entier + booléen`

Langage de types :

1. types élémentaires : ex, entier, booléen
2. types composés : ex, tableau, structure, fonction

→ *fait partie ou non du langage source*

Typer c'est quoi ?

Typer c'est :

- classer et normaliser les objets élémentaires
  - ex : entiers, booléens, flottants, caractères
- afin de savoir les composer, les assembler
  - ex : tableaux, structures
- en se protégeant contre des erreurs courantes
  - ex : `tab.champ`, `entier[i]`, `entier + booléen`

Langage de types :

1. types élémentaires : ex, entier, booléen
2. types composés : ex, tableau, structure, fonction

→ *fait partie ou non du langage source*

Pourquoi typer ?

- rôle d'abstraction
  - les expressions `1`, `1+2`, `(3*x)+1` sont toutes de type entier *utilisables partout où un entier est attendu*
  - analogie : n'importe quelle ampoule à vis peut être utilisée dans n'importe quelle douille à vis
- prévenir les erreurs de type
  - ex : entier utilisé là où un booléen est attendu
  - analogie : une ampoule à vis dans une douille à baionnette
  - analogie : une prise male USB dans une prise femelle 220V !

Remarque

Le processeur ne voit que des emplacement mémoires et des octets !

Histoire des langages de programmation

- apparition pragmatique en C et Fortran [1960s]
  - conditionne allocation mémoire + détection erreurs
- formalisation très complète en ML [1970s]
  - type = propriété abstraite des expressions
    expressions entières
  - règles de propagation des types
    - la somme de deux entiers est un entier
    - si T est un tableau de chaines et i est un entier, alors $T[i]$ est une chaine
  - le tout forme une théorie logique avec axiomes et règles

Distinctions entre langages (1/3)

Vérification de type vs Inférence de type

- vérification (C, Java)
  déclarations du type des variables
  vérification des utilisations des variables
  - ex : `int x;` $\Longrightarrow$ `x + 1` est correct
- inférence (ML)
  déduction des types d'après l'utilisation des variables
  - ex : `x + 1` $\Longrightarrow$ `x : int`

Distinctions entre langages (2/3)

- typage statique (ML) vs dynamique (Lisp, Smalltalk) vs mixte (C, Java)
  - statique : tout est vérifié à la compilation
    avantage : les erreurs sont détectées plus tôt
  - dynamique : tout est vérifié à l'éxécution
    avantage : offre plus de flexibilité
- typage fort (ML, Lisp, Java) vs faible (C)
  - typage fort : aucune opération mal typée n'est permise
  - certaines vérifications sont possibles à la compilation
    ex. : `tab["toto"]`
  - d'autres doivent être faites à l'éxécution
    ex. : dépassement des bornes d'un tableau

ATTENTION : les deux aspects sont orthogonaux

Distinctions entre langages (2/3)

- typage statique (ML) vs dynamique (Lisp, Smalltalk) vs mixte (C, Java)
  - statique : tout est vérifié à la compilation
    avantage : les erreurs sont détectées plus tôt
  - dynamique : tout est vérifié à l'éxécution
    avantage : offre plus de flexibilité
- typage fort (ML, Lisp, Java) vs faible (C)
  - typage fort : aucune opération mal typée n'est permise
  - certaines vérifications sont possibles à la compilation
    ex. : `tab["toto"]`
  - d'autres doivent être faites à l'éxécution
    ex. : dépassement des bornes d'un tableau

ATTENTION : les deux aspects sont orthogonaux

Distinctions entre langages (3/3)

notation littérale des valeurs d'un type

- seulement pour les types élémentaires (C)
  - entier : `1`, `127`
  - chaine : `"Hello"`
  - arbre (type composé) :
    - déf. type : `struct tree { int val; struct tree* left, right; }`
    - création valeur : code nécessaire ...
- pour la plupart des types (ML, Prolog)
  - entier, chaine : idem C
  - arbre :
    - déf. type : `type tree = Leaf of int | Node of tree * int * tree`
    - notation valeur : `Node(Leaf(2), 1, Leaf(3))`

Typage

3 ingrédients

1. constructions du langages source (syntaxe abstraite)
   - *ASD du langage source*
2. langage de types (élémentaires et complexes)
   - *ASD du langage de type*
3. système de type (association entre les deux)
   - *ASD attribuée du langage source*
   - *calculant le type des constructions sources*
     - décoration des AST source par des AST de types
     - fonction : ASD-source $\rightarrow$ ASD-type

Plan

Langage de types

- directement sous forme de syntaxe abstraite
  $\Rightarrow$ pour être indépendant de la syntaxe concrète de tel ou tel langage
- on retrouve peu ou prou les mêmes expressions de types d'un langage à l'autre
  - expressions élémentaires
  - expressions complexes

Expressions de types élémentaires

- Boolean : valeurs vrai ou faux
- Integer : signés ou non, courts ou longs, etc.
- Float : à precision simple ou double
- String : ASCII ou UTF
- Void : information vide, 1 seule valeur
  - `unit` en ML, `void` en C
- SideEffect : type des instructions, des procédures et effets de bord
  - ex. affichage, lecture, communication
  - cas particulier du type vide

Expressions de type complexes

- Table(A,B) : indices de type A et valeurs de type B

a_1	a_2	a_3	...
b_1	b_2	b_3	...

  - en général, A = entier
  - sinon, tables de hachage
  - en C : B`[]`
  - en ML : B `array`, `(`A`,` B`)` `Hashtbl.t`

Remarque

Table est un constructeur de type, et les paramètres A et B peuvent être des expressions de type quelconques.

Expressions de type complexes

- Function(A,B) : fonctions à un argument de type A et un résultat de type B
  - en C : $B\,(A)$
  - en ML : $A \rightarrow B$

Expressions de type complexes

- Struct($a_1 : A_1, \ldots, a_n : A_n$) : structure, enregistrement (*record*), type produit
  - les a_i sont des noms de champs et les A_i leurs types
  - les valeurs définissent tous les champs (type produit)
  - en C : `struct { A1 a1; ...; An an }`
  - en ML : `{ a1 : A1; ...; an : An }`

Expressions de type complexes

- Union($a_1 : A_1, \ldots, a_n : A_n$) : union, type somme
  - les a_i sont des noms de champs ou constructeurs
  - les valeurs définissent un seul champ (type somme)
  - en C : `union { A1 a1; ...; An an }`
  - en ML : a_1 `of` A_1 `|` `...|` a_n `of` A_n

Expressions de type complexes

- Pointer(A) : pointeur ou référence sur un A
  - les valeurs sont des adresses d'emplacements mémoires de type A
  - en C : $A*$
  - en ML : A `ref`

Expressions de type

Ce sont les types les plus courants.
On peut ajouter :

- les types des langages OO (ex., classes)
  - classe $\approx$ structure dont les champs sont des fonctions (méthodes)
  - objet $\approx$ structure(id, classe, attributs)
- les définitions de type
  - ex : type dictionnaire = Table(chaine,chaine)

ASD des expressions de types

$$
\begin{array}{lll}
\textit{type} & ::= & \textbf{Boolean} \\
& | & \textbf{Integer} \mid \textbf{Float} \\
& | & \textbf{String} \\
& | & \textbf{Void} \mid \textbf{SideEffect} \\
& | & \textbf{Table}(\textit{type}, \textit{type}) \\
& | & \textbf{Function}(\textit{type}, \textit{type}) \\
& | & \textbf{Struct}(\textit{field}+) \\
& | & \textbf{Union}(\textit{field}+) \\
& | & \textbf{Pointer}(\textit{type}) \\
& | & \textbf{Defined}(\textit{string}) \\
\textit{field} & ::= & \textbf{Field}(\textit{string}, \textit{type}) \equiv \textit{string} : \textit{type} \\
\textit{typedef} & ::= & \textbf{Define}(\textit{string}, \textit{type})
\end{array}
$$

Exemples : AST d'expressions de types

(1)……

Discussion sur les tableaux

- dans Table(A,B)
  - A n'est pas un intervalle de positions
  - ne pas confondre le "10" avec A dans B[10] (positions 0..9)
- on pourrait avoir Table(A,Inf,Sup,B)
  - avec *Inf* et *Sup* les bornes du tableau
  - mais on ne sait pas vérifier le respect de ces bornes de façon statique (à la compilation)
  - ces bornes serviront à produire le code d'adressage (et de vérification dynamique) lors de la traduction du code source en code intermédiaire/cible

Discussion sur les fonctions

- Les fonctions peuvent être combinées récursivement de façon arbitraire
  - fonction en argument ou en résultat
  - ex : Function(Function(A,B), Function(Function(B,C), Function(A,C)))
    en ML : $(A \to B) \to (B \to C) \to (A \to C)$
  - c'est le type de la composition de fonction
- Cela ne gêne pas la vérification de type, ni ne la complexifie !
- Par contre, cela implique un schéma d'exécution particulier, pour manipuler les fonctions passées en paramètre
  - valeurs d'ordre supérieur

Discussion sur les fonctions

- Les fonctions peuvent être combinées récursivement de façon arbitraire
  - fonction en argument ou en résultat
  - ex : Function(Function(A,B), Function(Function(B,C), Function(A,C)))
    en ML : $(A \to B) \to (B \to C) \to (A \to C)$
  - c'est le type de la composition de fonction
- Cela ne gêne pas la vérification de type, ni ne la complexifie !
- Par contre, cela implique un schéma d'exécution particulier, pour manipuler les fonctions passées en paramètre
  - valeurs d'ordre supérieur

Ordre d'un type

Definition (Ordre d'un type)

- L'ordre d'un type peut être défini de façon récursive comme suit :
  - $Ordre(\textbf{Function}(A, B)) = Max(Ordre(A) + 1, Ordre(B))$
  - $Ordre(A) = 0$, pour tout type non fonctionnel
- Un type A est dit d'ordre supérieur si $Ordre(A) > 0$ (fonctions)
- Une fonction de type T est dite d'ordre supérieur si $Ordre(T) > 1$ (fonctions de fonctions)

Fonctions d'ordre supérieur ?

- Function(String, Function(Integer, Function(Integer, String))) ?
  (2).....
- Function(Function(String,SideEffect), Function(Table(Integer,String), SideEffect)) ?
  (2).....
- Function(Integer, Function(Function(Integer,Integer), Function(Integer,Integer))) ?
  (2).....

Remarque

Les fonctions d'ordre 2 sont communes dans les langages fonctionnels (ex., map, fold). Les fonctions d'ordre 3 ou plus sont rarissimes en pratique.

Fonctions d'ordre supérieur ?

- Function(String, Function(Integer, Function(Integer, String))) ?
  (2).....
- Function(Function(String,SideEffect), Function(Table(Integer,String), SideEffect)) ?
  (2).....
- Function(Integer, Function(Function(Integer,Integer), Function(Integer,Integer))) ?
  (2).....

Remarque

Les fonctions d'ordre 2 sont communes dans les langages fonctionnels (ex., map, fold). Les fonctions d'ordre 3 ou plus sont rarissimes en pratique.

Curryfication des fonctions

Definition (Curryfication)

La curryfication est le codage d'une fonction n-aire en une imbrication de fonctions unaires.

- type C : $R(A_1,...,A_n)$
  $\rightarrow$ Function(A_1, ..., Function(A_n,R))
- appel C : $f(x_1,...,x_n)$
  $\rightarrow f(x_1)(x_2)...(x_n) = (...((f(x_1))(x_2))...(x_n))$

Centralité du type fonction

Rôle central des fonctions :

- **Table**(A,B) peut être assimilé à une fonction (de A vers B) définie en extension (au cas par cas)
- dans Struct($a_1 : A_1, \ldots, a_n : A_n$), les champs a_i peuvent être assimilés à des fonctions
  - a_i : **Function**(**Struct**($a_1 : A_1, \ldots, a_n : A_n$), A_i)
- dans **Union**($a_1 : A_1, \ldots, a_n : A_n$), les constructeurs a_i peuvent être assimilés à des fonctions
  - a_i : **Function**(A_i, **Union**($a_1 : A_1, \ldots, a_n : A_n$))

Remarque

Le λ-calcul est un formalisme Turing-complet qui ne connaît que les fonctions unaires. Même les entiers et les booléens y sont codés par des fonctions !

Centralité du type fonction

Rôle central des fonctions :

- **Table**(A,B) peut être assimilé à une fonction (de A vers B) définie en extension (au cas par cas)
- dans Struct($a_1 : A_1, \ldots, a_n : A_n$), les champs a_i peuvent être assimilés à des fonctions
  - a_i : **Function**(**Struct**($a_1 : A_1, \ldots, a_n : A_n$), A_i)
- dans **Union**($a_1 : A_1, \ldots, a_n : A_n$), les constructeurs a_i peuvent être assimilés à des fonctions
  - a_i : **Function**(A_i, **Union**($a_1 : A_1, \ldots, a_n : A_n$))

Remarque

Le λ-calcul est un formalisme Turing-complet qui ne connaît que les fonctions unaires. Même les entiers et les booléens y sont codés par des fonctions !

Exemples d'expressions de types en C et ML

(3).....

Soyons réflexifs !

La définition du type “type” des types !

```
Define("type", Union(
  "Boolean" : Void,
  "Integer" : Void,
  ...
  "Table" : Struct("index" : Defined("type"),
                   "val" : Defined("type")),
  "Function" : Struct("arg" : Defined("type"),
                      "res" : Defined("type")),
  "Struct" : Table(String, Defined("type")),
  "Union" : Table(String, Defined("type")),
  "Pointer" : Defined("type"),
  "Defined" : string))
```

C’est une valeur de l’ASD des types qui dénote le type dont les valeurs sont des expressions de type.

Plan

ASD d'un petit langage de blocs

Bloc = séquence de déclarations et d'instructions, terminée par un return

bloc	::=	**Decl**(*decl*, *bloc*)
	|	**Statement**(*expr*, *bloc*)
	|	**Return**(*expr*)
decl	::=	**Var**(*ident*, *type*)
expr(*E*)	::=	**Id**(*ident*)
	|	**Const**(*int*)
	|	**Apply**(*expr*, *expr*)
ident	::=	*string*

On veut faire la vérification de types de ce langage.

Vérification de types

Exemple courant de vérification contextuelle non exprimable dans l'ASD :

⇒ analyse sémantique

⇒ ASD attribuée

attribut	description	mode	type (du compilateur)
BT	"bien typé"	s	*bool*
TS	table des symboles	h	$set(ident \times type)$
type	expression de type	s	*type*

- ATTENTION : bien distinguer
  - les types utilisés dans l'écriture du compilateur
    ex., types de Java
  - et les types du langage source (ASD des types)
- ident est le type des identificateurs des programmes sources
- type est le type défini par l'ASD des types
  - les valeurs sont les AST d'expressions de types !

Exemple : déclarations et expressions (1/2)

ASD attribuée pour la vérification de type du petit langage de blocs (déclarations et d'expressions)
(4)……

Exemple : déclarations et expressions (2/2)

(4). . . . . .

Extension de l'exemple

Pour chaque nouvelle construction du langage

- accès tableau `tab[i]`
- accès champ structure `point.x`

il suffit d'ajouter un variant du type *expr* et de définir les attributs : *BT*, *TS* et *type*.

Notation formelle et concise

Remarque

On peut adopter une présentation plus formelle et plus concise pour la vérification de type.

- La notation $E : \tau$ équivaut aux équations
  - `E.BT := vrai`
  - `E.type :=` τ
  - la table des symboles est implicite
- La notation $\frac{H1 \; H2}{C}$ signifie que si $H1$ et $H2$ sont vérifiés, alors C l'est aussi

Notations formelles pour l'exemple étendu

(5)……

Système formel

On vient de construire un système formel permettant de prouver si une expression est bien typée et de calculer son type

- $\frac{}{C}$ est un axiome
- $\frac{H1 \quad H2}{C}$ est une règle d'inférence

Du système formel à l'ASD attribuée

Un tel système formel se traduit directement en ASD attribuée

- axiome : force la valeur de l'attribut "type"
  - ex : $\frac{}{\textbf{Const}(int) : Integer}$
    $expr(E) ::= \textbf{Const}(int)\ \{expr.type := Integer\}$
- règle : le type dépend des types des arguments
  - + contraintes de "bon typage"
    1. les différentes occurences d'une variable doivent être égales
    2. les types des arguments doivent "matcher" les constructeurs de type
    3. les conditions sur la TS doivent être vérifiées
  - ex :

$$\frac{E_1 \ : \ \textbf{Table}(\tau_1 = \textbf{Integer}, \tau_2) \quad E_2 \ : \ \tau_1 = \textbf{String}}{Lookup(E_1, E_2) \ : \ \textbf{Erreur}}$$

    - contrainte non vérifiée : 2 valeurs différents pour τ_1
    - échec vérification : BT = faux, type = erreur

Extension de la vérification aux instructions

La vérification de type peut être étendue aux instructions

- instruction = expression de type **SideEffect**
- affectation

$$\frac{\mathit{ident} : \tau \quad E : \tau}{\textbf{Assign}(\mathit{ident}, E) : \textbf{SideEffect}}$$

- conditionnelle

$$\frac{E_1 : \textbf{Boolean} \quad E_2 : \textbf{SideEffect} \quad E_3 : \textbf{SideEffect}}{\textbf{Cond}(E_1, E_2, E_3) : \textbf{SideEffect}}$$

- ...

Exemple complet de vérification de type

- expression à vérifier

  ```
  print((*objet).m1(x),10)
  ```

- avec comme table des symboles (TS)
  - `x : truc = Defined("truc")`
  - `objet : Pointer(Struct(v : Integer, m1 : Func(truc, String), m2 : Func(truc, Boolean)))`
  - `print : Func(String, Func(Integer, SideEffect))`

(6).....

Plan

Inférence de type

Principe

Laisser le compilateur inférer les types des variables plutôt que de demander au programmeur de les déclarer. Cela permet une plus grande concision des programmes.

- le type d'une variable est inféré à partir de ses contextes d'utilisation
  - ex : dans x + 1, x doit avoir le type entier
- la table des symbole est synthétisée par les expressions
- il faut vérifier qu'une même variable a toujours le même type !
  - ex : dans x + strlen(x), les 2 occurences de x ont des types incompatibles (entier vs chaine)
- on s'appuie sur le même système formel (axiomes et règles)

Exemple d'inférence

Typage de x + 1 = **Apply**(**Apply**(**Id**("plus"),**Id**("x")),**Const**(1))
où :

- plus : Function(Integer, Function(Integer, Integer))

(7).....

Exemple d'inférence avec erreur

Typage de x + strlen(x) où :

- strlen : Function(String, Integer)

(8)……

Autre exemple d'inférence

Typage de print((*objet).m1(x))(10) où :

- print : Function(String, Function(Integer, SideEffect))

(9).....

The End