ECOLE SUPERIEURE D'INGENIEUR DE RENNES

Compilation

typage

Olivier Ridoux

UNIVERSITÉ DE RENNES 1

Plan

- Principes
  - système de typage
  - vérification des types
  - inférence des types
- Mise en œuvre en GA

Principes

Pragmatique du typage

- Ingénierie de base
  - classer/normaliser des interfaces
  - empêcher les montages **dangereux**
- Ingénierie informatique
  - **capacité de définition de nouvelles classes de composants**
  - **types et programmes sont tous les deux définis par le programmeur**

Théorie du typage

- Plusieurs paradigmes de typage
  - codage
  - ensemble
  - **propriété**
- Les ingrédients du typage
  - un langage de termes
  - un langage de types
  - un système de typage

Typage = codage (1)

- Un type dénote une certaine façon de coder/décoder l'information
  - **int 32 bits**
  - **float 64 bits**
  - **ASCII 7 bits**
  - **struct { int a ; float b }**
  - **char * [12]**
- Point de vue non avoué de beaucoup de langages anciens
  - **C, FORTRAN, ...**

Typage = codage (2)

- Point de vue antinomique avec le besoin d'abstraction
  - **type abstrait**
  - **interface publique vs. implémentation privée**

- Ne permet pas de formaliser la vérification de type, ni l'inférence

Type = ensemble (1)

- Un type dénote un ensemble de valeurs et les opérations qui s'y appliquent

  $([-2^{31}\ +2^{31}], +, -, *, /)$

- Extrêmement complexe si précis
  - **débordement**
  - **opérations non partout définies**

Type = ensemble (2)

- Ne traite pas bien les fonctions
  - rappel Cantor

    si E infini dénombrable

    alors E ⟶ E non dénombrable
  - mais les structures informatiques sont forcément dénombrables

- Ne permet pas de formaliser la vérification de type, ni l'inférence

Type = propriété

- Un type dénote une propriété
- Formalisation logique du typage
  → automatisation possible
- Formalisation aisée de la vérification et de l'inférence

Ingrédients du typage

- Un langage de **termes**
- Un langage de **types**
- Un **système de typage**
  - quel **type** a tel **terme** ?

Termes et types

- Les termes dénotent des valeurs
  - on les évalue pour obtenir leur dénotation
- Les termes ont des types
- Évaluer ne change pas le type d'un terme
  - donc pas de type dénotant la forme d'un terme
  - les types sont des **invariants**

    donc **vérification statique envisageable**

Langage de termes

- Les constructions du langage…

…ou leur **syntaxe abstraite**

- Représentation donnée naturellement par l'analyseur syntaxique

Ex : langage de termes (1)

- Les expressions élémentaires
  - constantes et variables

    "douze", "12", 12, 0x12, douze, XII

- Les expressions composées
  - opérations, appels, accès structures et tableaux

    XII+12, douze(12), douze.XII, douze[12]

Ex : langage de termes (2)

- Les instructions ou commandes

  douze := XII, douze(XII), douze.XII(12)

- Les fonctions, procédures ou classes

- Les programmes

Langage de types

- Pas forcément **explicite** dans le langage source

  si explicite, on peut ne garder que la syntaxe abstraite...

  ...sinon, en inventer une

- Situation intermédiaire possible

Langage de types

- Lisp, Scheme
  - complètement **implicite**
  - mais Racket, **explicite optionnel**
- ML, CAML
  - explicite optionnel, **tous les types ont une expression**
- C, Java, etc
  - **explicite obligatoire**, mais **certains types n'ont pas d'expression**

Langage de types

- Exemple avec des constructions classiques

  langage procédural/fonctionnel

Types atomiques

- bool, int, float, ...
- exec
  - le type des termes exécutables
- unit
  - le type des termes avec unique valeur possible
  - information nulle
- void
  - le type des termes sans valeur possible
- error

Type tableau (1)

- array(Ind, Val)

  retourne un terme de type Val quand on **l'indexe** avec un terme de type Ind

  en C : **float t [12]** dénote **array(int, float)**

Type tableau (2)

on pense **rangée de cases mémoires consécutives**, mais ce n'est pas nécessaire

on pense **rangée de cases numérotées**, mais ce n'est pas nécessaire

mieux vaut penser
fonction définie en extension

Type fonction (1)

- fun(From, To)

  retourne un terme de type To quand on **l'appelle** avec un terme de type From

fun(float, int)

en C : **int f (float)**

en ML : **float -> int**

Type fonction (2)

on pense à une fonction **exécutable**,
mais ce n'est pas nécessaire

mieux vaut penser **dépendance fonctionnelle** :
les arguments déterminent le résultats

cas des fonctions à effet de bord

Type structure (1)

- struct(a_1:A_1, ..., a_i:A_i, ..., a_n:A_n)
  retourne un terme de type A_i
  quand on lui applique le
  sélecteur a_i

struct(i:int, f:float)

Type structure (2)

on pense **représentation consécutive** des champs a_i, mais ce n'est pas nécessaire

mieux vaut penser **composants** ou **facettes** d'un objet

Type union (1)

- union(a_1:A_1, ..., a_i:A_i, ..., a_n:A_n)

  retourne un terme de type A_i
  quand on lui applique le
  sélecteur a_i

union(i:int, f:float)

Type union (2)

- Différence avec structure ?
  - structure : tous les sélecteurs sont définis tous le temps
  - union : un seul sélecteur est défini à un moment donné

un seul sélecteur ; lequel ?

Type pointeur (1)

- ptr(A)

  retourne un terme de type A quand on le **déréférence**

ptr(int)

en C : **float*** dénote **ptr(float)**

Type pointeur (2)

on pense **adresse en mémoire,**
mais ce n'est pas nécessaire

mieux vaut penser
identifiant univoque d'un terme

Expressions de type

- En C

struct list {e car ; struct list * cdr }

dénote **list : struct(car:e, cdr:ptr(list))**

- ML

datatype list = Nil | Cons of e * list

dénote **union(Nil:list, Cons:struct(car:e, cdr:list))**

(A->B) -> (B->C) -> A->C

dénote **fun(fun(A,B), fun(fun(B,C), fun(A,C)))**

Curryfication

- fun(A, fun(B, C)) peut se lire fun(A, B, C)
- Il n'est pas nécessaire de prévoir un type de fonction n-aire
  - ex. **'+'** a le type **fun(int, fun(int, int))**

$$A \longrightarrow (B \longrightarrow C) \quad \equiv \quad (A, B) \longrightarrow C$$

- À rapprocher de

$$A \Rightarrow (B \Rightarrow C) \quad \equiv \quad (A \wedge B) \Rightarrow C$$

Biographie

- Haskell Curry (États-Unis, 1900 - 1982)
- Logicien et mathématicien
- Contributeur des bases théoriques de la programmation fonctionnelle (à la ML ou LISP)

λ-calcul

Système de typage

- Un système logique avec
  - des **jugements**
  - des **axiomes**
  - et des **règles de déduction**
- Notion formelle de preuve

Jugement de type

- $t : \tau$

le terme t a le type τ

- Peut être manifestement faux ou vrai, ou demande vérification

12.5 : int

12 : int

f(12) : int

Règles de déduction

$$\frac{\text{hypothèse}_1 \ldots \text{hypothèse}_n}{\text{conclusion}}$$

Si **toutes** les hypothèses sont satisfaites,
alors la conclusion l'est

Axiome : $\dfrac{}{\text{conclusion}}$

La conclusion est satisfaite tout le temps

Notion d'arbre de preuve

- Arbre de preuve
  - nœuds : **instances** de règles de déduction
  - racine : un jugement à prouver
- Analogie avec arbre de dérivation

Notion de preuve

- Un arbre de preuve est une preuve...

ssi

...toutes ses feuilles sont des instances d'axiomes

- **Un système de déduction est une grammaire de preuve**

Système de typage

- Des règles de déduction dont les jugements sont des jugements de types

Axiomes

$$\frac{}{\text{notation d'entier : int}}$$

$$\frac{}{\text{notation de booléen : bool}}$$

...

$$\frac{}{\text{ident(X) : } \tau} \quad \text{si decl X } \tau$$

Tableau

$$\frac{t : \text{array}(\tau_1,\tau_2) \qquad i : \tau_1}{t[i] : \tau_2}$$

Fonction

$$\frac{f : fun(\tau_1,\tau_2) \qquad x : \tau_1}{f(x) : \tau_2}$$

- **LA règle de déduction de type !**

Structure et union

$$\frac{X : \text{struct}(a_1:\tau_1, \ldots, a_i:\tau_i, \ldots, a_n:\tau_n)}{X.\text{ident}(a_i) : \tau_i}$$

$$\frac{X : \text{union}(a_1:\tau_1, \ldots, a_i:\tau_i, \ldots, a_n:\tau_n)}{X.\text{ident}(a_i) : \tau_i}$$

Pointeur

$$\frac{X : ptr(\tau)}{*X : \tau}$$

$$\frac{X : \tau}{\&X : ptr(\tau)}$$

si X désigne une mémoire

Aparté – bonnes et mauvaises règles (1)

- Pourquoi pas la règle suivante ?

$$\frac{*X : \tau}{X : ptr(\tau)}$$

Aparté – bonnes et mauvaises règles (2)

- Les règles non-axiomes décomposent le terme du jugement de conclusion

$$\frac{\text{sous-terme}_1 : \tau_1 \ \ldots \ \text{sous-terme}_n : \tau_n}{\text{terme} : \tau}$$

L'arbre de preuve suit l'arbre de syntaxe abstraite du terme

Aparté – bonnes et mauvaises règles (3)

- En fait

$$\frac{\text{sous-terme}_1 : \tau_1 \ldots \text{sous-terme}_n : \tau_n}{\text{terme} : \tau}$$

...est une forme attribuée de

$$\text{terme} \longrightarrow \text{sous-terme}_1 \ldots \text{sous-terme}_n$$

...où les τ_i sont les **attributs**

Aparté – bonnes et mauvaises règles (4)

- En utilisant des règles…

$$\frac{\text{sous-terme}_1 : \tau_1 \ \ldots \ \text{sous-terme}_n : \tau_n}{\text{terme} : \tau}$$

…l'arbre de preuve du typage s'inscrit dans l'arbre de syntaxe abstraite du terme

⟶ une plus grande facilité de mise en œuvre

Quelques concepts (1)

- Surcharge
  - symboles de types ≠ mais de même nom
    - '+' : fun(int, fun(int, int))
    - '+' : fun(float, fun(float, float))

    +, -, *, /, =, == sont très surchargés
- Polymorphisme
  - terme unique qui a plusieurs types
    - length : fun(list(X), int)
    - window : Window ⟶ Component ⟶ Object

    ML et POO

Quelques concepts (2)

- Programme bien typé
  - un programme **p** est **bien typé** si un jugement **p : τ** peut être prouvé
  - **vérifier le bon typage** d'un programme consiste à rechercher cette preuve

Quelques concepts (3)

- Vérification statique / dynamique
  - la vérification des types est **statique** si elle se fait sans exécuter le programme
  - elle est **dynamique** sinon
    - liaison dynamique
    - Lisp, Scheme
    - C : printf(s, i, j)

Quelques concepts (4)

- Propriété du typage sain
  - un système de typage est **sain** (*sound*) si un programme bien typé statiquement ne peut pas causer d'erreur de type dynamique

propriété recherchée, mais rare

CamL, ML

Mise en œuvre en GA

Vérification des types en GA

- 2 attributs synthétisés : ok et type

Expr ⟶ Expr' '(' Expr'' ')'

$$\frac{f : fun(\tau_1,\tau_2) \qquad x : \tau_1}{f(x) : \tau_2}$$

{Expr.ok = Expr'.ok ⋀ Expr''.ok
if Expr.ok ⋀ isFun(Expr'.type)
⋀ sameType(fromFun(Expr'.type), Expr''.type)
then Expr.type = toFun(Expr'.type)
else Expr.ok = false ; Expr.type = error
fi}

$$\frac{ident(X) : \tau}{si\ decl\ X\ \tau}$$

$$\frac{}{notation\ d'entier : int}$$

Expr ⟶ ident { Expr.ok = true ;
Expr.type = TS(ident).type }

Expr ⟶ cst { Expr.ok = true ; Expr.type = cst.type }

Fonctions d'accès pour vérification des types

- isFun, fromFun, toFun
- isArray, indArray, valArray
- isStruct, isSelStruct, selStruct, fldStruct
- isPtr, toPtr

...

- isInt, isString, ...
- **sameType**

Inférence de type (1)

- Déclarations de type **absentes**
  en partie ou totalement
- Déclarations à **reconstituer**
  en analysant le programme
- Archétype ML, ...

Inférence de type (2)

- Point de vue logique
  - polymorphisme

    ex. length, $\forall \tau$. length:fun(list(τ), int)
  - inférence

    ex. f dans f(x), $\exists \tau$. f:fun(τ_x, τ)
- Point de vue opérationnel
  - des variables de type

    fun(int, var(A))

Inférence des types en GA

- 2 attributs synthétisés : ok et type

Expr ⟶ Expr' '(' Expr'' ')'

{Expr.ok = Expr'.ok ∧ Expr''.ok

if Expr.ok ∧ unifType(Expr'.type,
mkFun(Expr''.type, new varType))

then Expr.type = varType

else Expr.type = error

fi}

Procédure d'unification (1)

- unifType(fun(int, int), fun(int, var(A)))

  ok, var(A) ⟵ int

- unifType(fun(int, int), fun(var(V), var(A)))

  ok, var(V) ⟵ int, var(A) ⟵ int

- unifType(fun(int, int), fun(float, var(A)))

  ¬ ok

- unifType(fun(var(X), var(X)), fun(int, var(A)))

  ok, var(X) ⟵ int, var(A) ⟵ int

Procédure d'unification (2)

- unifType(fun(var(X), var(X)), fun(var(V), var(A)))

  ok, var(X) ⟵ var(V), var(A) ⟵ var(V)

- unifType(fun(var(X), var(Y)), fun(var(V), var(A)))

  ok, var(X) ⟵ var(V), var(A) ⟵ var(Y)

- unifType(fun(var(X), var(X)),

  fun(fun(var(A), var(A)), var(A)))

  ¬ ok

Procédure d'unification (3)

unifType(τ_1, τ_2)

{ if sameType(τ_1, τ_2) then true

elif isFun(τ_1) $\wedge$ isFun(τ_2)

then unifType(fromFun(τ_1), fromFun(τ_2))

$\wedge$ unifType(toFun(τ_1), toFun(τ_2))

elif isVar(τ_1) and not occurIn(τ_1, τ_2) then substVar(τ_1, τ_2)

elif isVar(τ_2) and not occurIn(τ_2, τ_1) then substVar(τ_2, τ_1)

else false

fi }

Procédure d'unification (4)

{ ... ; Z = X(Y) ;

unifType(typeOfX, fun(typeOfY, var(A)))

typeOfX ⟵ fun(typeOfY, var(A))

... ; X = Y /* ((=(X))(Y) */ ; ...

}

unifType(typeOfEq, fun(TypeOfX, var(B)))
≈ unifType(fun(var(V), fun(var(V), bool)),
fun(typeOfX, var(B))

var(V) ⟵ typeOfX

var(B) ⟵ fun(typeOfX, bool)

unifType(fun(typeOfX, bool), fun(typeOfY, var(C)))
≈ unifType(fun(**fun(typeOfY, var(A))**, bool) ,
fun(**typeOfY**, var(C)))

Fonctions d'accès pour inférence de type

- ... +
- mkFun, mkArray, mkStruct, mkPtr, mkInt, mkString, ...
- isVar, substVar
- OccurIn

Conclusion

- Type comme propriété
- Typage bien modélisé par déduction logique
- Vérification = recherche de preuve...

...facile à réaliser en GA

- Inférence = recherche de « témoin » de preuve (∃)...

...plus difficile, mais faisable