Programmation fonctionnelle - PG104 -

COURS 7

Myriam Desainte-Catherine et David Renault

February 6, 2020

Systèmes de types : motivation – correction de programmes

C : typage statique

```
int
contains0(int* l, int size)
{
   return belong(0, l, size);
}
```

Scheme : typage dynamique

```
(define (contains-zero? l)
  (member 0 l))
```

Systèmes de types : motivation – correction de programmes

C : typage statique

```
int
contains0(int* l, int size)
{
   return belongs(0, 1, size);
}
```

L'erreur est-elle détectée lors de la

- compilation
- ou lors de l'éxécution du programme?

Scheme : typage dynamique

```
(define (contains-zero? l)
  (member 0  1))
```

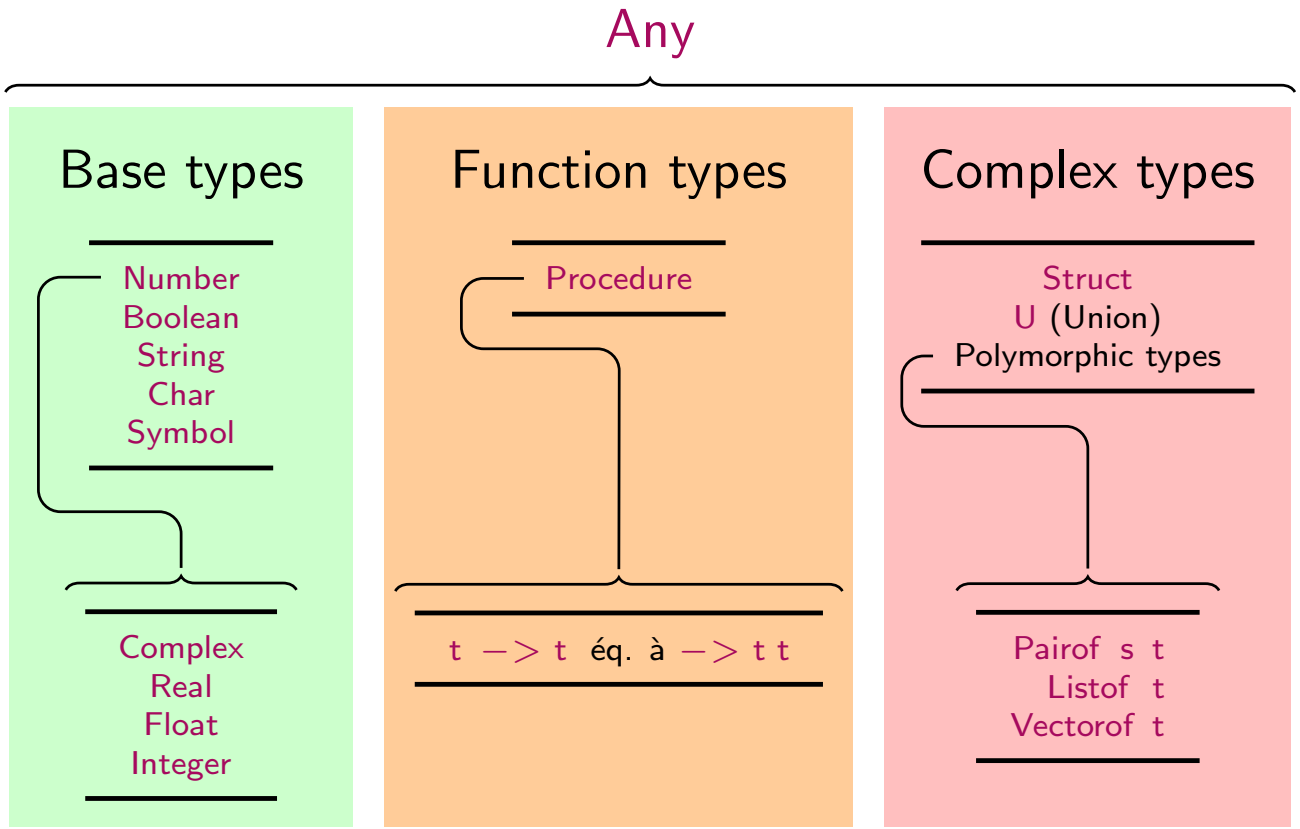
L'erreur est-elle détectée lors de la

- définition de la fonction (quand on clique sur exécuter dans Drracket)
- ou lors de son application ?

Typage dynamique : les seules erreurs détectées sont celles qui se produisent à l'éxécution. La correction dépend des tests effectués. Certaines erreurs peuvent rester cachées longtemps.

Système de type

Classification des valeurs en ensembles appelés types de manière à garantir la correction de certains programmes.



Styles de typage

Styles de typage

- le typage dynamique : déterminé pendant l'exécution par le runtime, il ne nécessite aucune intervention du programmeur;
- le typage statique : fixé avant l'exécution par le compilateur, il est soit inféré automatiquement, soit indiqué par des annotations dans le code.

Racket définit en fait plusieurs langages :

- En Racket classique, le typage est dynamique;
- En Typed/Racket, le typage est dynamique mais autorise des annotations statiques vérifiées par le compilateur.

Comment typer ?

Afin d'annoter une valeur $\langle val \rangle$ par un type $\langle typ \rangle$, il suffit d'écrire *avant la définition* de $\langle val \rangle$:

- $(: \langle val \rangle \langle typ \rangle)$ Ex : (: int_exm Integer)

Les types primitifs contiennent en particulier :
Number, Integer, Float,
Char, String, ...
Les types des fonctions sont écrits à l'aide d'une des syntaxes suivantes :

- $(\langle arg_1 \rangle \langle arg_2 \rangle \ldots \langle arg_n \rangle -> \langle res \rangle)$ Ex : (Number Number -> Boolean)
- $(-> \langle arg_1 \rangle \langle arg_2 \rangle \ldots \langle arg_n \rangle \langle res \rangle)$ Ex : (-> Number Number Boolean)

Exemple

```
#lang typed/racket                    ;; Choice of the language
(: num-fun (Number -> Number))        ;; Type annotation of num-fun
(define (num-fun n) (add1 n))         ;; Definition of num-fun
(:print-type num-fun)                 ;; (-> Number Number)
```

Intérêts du typage

- Détection d'erreurs de type : passer une valeur de type String à une fonction Int -> Int est incohérent.

Exemple

```
(: num-fun (Number -> Number))
(define (num-fun n) (add1 n))
(num-fun "abc")     ;; -> Type Checker error : type mismatch
```

- Compatibilités de type : passer une valeur de type Integer à une fonction Number -> Number est cohérent car un Integer est aussi un Number.

Exemple

```
(: intg Integer)  ;; Type annotation of intg
(define intg 3)
(num-fun intg)    ;; -> 4
```

- Optimisations : le compilateur peut écrire du code dédié à des types particuliers (exemple : nombre flottants et instructions FPU).

Exemples

Exemple

```
(: greater-than (Number Number -> Boolean))
(define (greater-than x y)
  (> x y))          ;; Type error : Number cannot be compared
```

En effet, un Number peut aussi être un Complex, donc non comparable.

Exemple

```
(: plus (Number Number -> Number))
(define (plus x y) (+ x y))

(: greater-than (Real Real -> Boolean))
(define (greater-than x y) (> x y))

(greater-than (plus 3 4) 5)
;; Type error : Number cannot be compared
```

En effet, plus renvoie un Number, qui potentiellement n'est pas un Real.

Remarque : en réalité, le type de l'opérateur + est générique.

Contrôler les types

Soit la fonction typée suivante

```
(: racine-carree (Positive-Float -> Positive-Float))
(define (racine-carree n)
  (sqrt n))
```

Quels sont les résultats des expressions suivantes?

> (racine-carree 1.5) Type Checker error 1.22474487139 > (racine-carree 1.5) 1.22474487139

> (racine-carree -1.4) Type Checker error 0+1.18321595i > (racine-carree -1.4) Type Checker error

La fonction suivante est-elle correcte? Oui ou Non

```
(: valeur-absolue (Number->Number))
(define (valeur-absolue x)
  (if (negative? x)
      (- x)
      x))
```

Définir ses propres types

Le code suivant définit un Struct représentant des points du plan :

```
(struct: point ([x : Real] [y : Real]))

(: distance (point point -> Real))
(define (distance p1 p2)
  (sqrt (+ (sqr (- (point-x p2) (point-x p1)))
           (sqr (- (point-y p2) (point-y p1))))))
```

Cette construction définit en même temps les fonctions suivantes :

- Un constructeur point permettant de construire des instances comme par exemple (point 3 4).
- Deux accesseurs point-x et point-y permettant d'accéder aux champs de la structure.

Types inductifs : les listes

Définition récursive des listes en Scheme (Rappel)

- Soit la liste vide : **null** ou '()
- Soit une paire (car , cdr) où car est un élément de la liste et cdr est une liste.

La définition de type pour les listes en Racket :

```
;; A List is either a Pair or Empty
(define-type (List a) (U (Pair a) Empty))
;; A Pair is a struct with car and cdr, and Empty is empty
(struct: (a) Pair ([car : a] [cdr : (List a)]))
(struct: Empty ())
```

Remarque : le type pour les listes est un type polymorphe.

```
(: a_list (List Integer))
(define a_list (Pair 1 (Pair 2 (Pair 3 (Empty)))))
```

Reconnaissance de motif

La forme **match**

```
(match t
   [⟨pat_1⟩ res_1]
   [⟨pat_2⟩ res_2]
   ...
   [⟨pat_n⟩ res_n]
   [_       default]
)
```

La reconnaissance de motif ou pattern-matching :

- compare l'expression t à chacun des motifs $\langle pat_k \rangle$
- et renvoie le résultat associé au premier indice pour lequel t correspond.

Les motifs peuvent introduire des liaisons utilisées dans le résultat.

Exemple pour calculer la longueur d'une liste :

```
(define (list-length l)
  (match l
    [(Empty)      0]                    ;; Match Empty struct
    [(Pair x xs)                        ;; Match Pair struct
       (add1 (list-length xs))]         ;; and binds x and xs
    ))
```

Reconnaissance de motif

La forme match peut être utilisée en dehors de typed/racket

Somme des éléments d'une liste

Équations fonctionnelles :

- somme-liste('()) = 0
- somme-liste(l) = car(l) + somme-liste(cdr(l))
- somme-liste('()) = 0
- somme-liste(cons(n, r)) = n + somme-liste(r))

Exemple

```
(define (somme-liste l)
  (match l
    ('() 0)
    ((cons n r) (+ n (somme-liste r)))))
```

Fonctionnelles

Higher-order functions

- Fonctions anonymes : forme **lambda**
- Fonctionnelles de la bibliothèque : arguments fonctions
- Fonctionnelles de la bibliothèque : fonctions en retour

La forme **lambda** : rappel et utilisation

(**lambda** ($\langle p_1 \rangle$ $\langle p_2 \rangle$... $\langle p_n \rangle$) $\langle e \rangle$)

- **Nommage de λ-expressions** : `(define f (lambda (x y) (+ (* 10 x) y))`
  Équivalent à : `(define (f x y) (+ (* 10 x) y))`
- **Application de λ-expressions** : mise en position fonctionnelle, stratégie applicative (par valeur) – Ex. `((lambda (x) (sub1 x)) 1)`
- **Passage de λ-expressions en paramètres** : juxtaposition
- **λ-expressions en retour de fonction** : imbrication

Application de λ-abstractions : juxtaposition

Rappels sur les λ-termes

- une variable : x, y, ...
- une application : u v où u et v sont des λ-termes
- une λ-abstraction : $\lambda x.u$

On appelle rédex un terme de la forme $(\lambda x.u)v$. On définit alors la bêta-réduction

$$(\lambda x.u)v \longrightarrow u[x := v]$$

Applications d'une λ-abstraction à une λ-abstraction

- Soit le terme :$(\lambda x.xy)(\lambda x.ux)$, on a la suite de réductions suivante :
  $xy[x := \lambda x.ux] \longrightarrow (\lambda x.ux)y \longrightarrow ux[x := y] \longrightarrow uy$
- Soit $(\lambda f.f0)(\lambda x.(*2x))$, on a la suite de réductions suivante :
  $f0[f := \lambda x.(* \ 2 \ x)] \longrightarrow (\lambda x.(* \ 2 \ x))0 \longrightarrow (* \ 2 \ x)[x := 0] \longrightarrow (* \ 2 \ 0) \longrightarrow 0$
- En scheme : `((lambda(f) (f 0)) (lambda (x) (* 2 x)))`, on a la suite de réductions suivante (stratégie applicative) :

```
((lambda(x) (* 2 x)) 0) -> (* 2 0) -> 0
```

λ-expressions en retour de fonction : imbrication

Exemples en λ-calcul

- Soit $\lambda x.(\lambda y.xy)f$, on a $\lambda y.xy[x := f] \longrightarrow \lambda y.fy$
- Soit $(\lambda x.(\lambda y.xy)f)z$, on a
  $(\lambda y.xy[x := f])z \longrightarrow (\lambda y.fy)z \longrightarrow fy[y := z] \longrightarrow fz$

Exemples en scheme

```
> ((lambda(x) (lambda (y) (= (* 2 x) y))) 1)
(lambda (y) (= (* 2 1) y))


> (((lambda(x) (lambda (y) (= (* 2 x) y))) 1) 2)
-> ((lambda (y) (= (* 2 1) y) 2)
-> (= (* 2 1) 2)
#t
```

Méthode : pour effectuer une réduction, repérer la fonction qui est en première position, puis les arguments e_1, e_1, etc.. Évaluer les arguments puis appliquer la fonction.

Questions

```
> ((lambda (f)
     (f 1))
   (lambda (x)
     (add1 x)))
```

- #<procedure>
- 2
- erreur
- Réponse : 2

Questions

```
((lambda (x)
   (lambda (y)
     (cons x y)))
 '(1 2 3))
```

- #<procedure>
- '((1 2 3) y)
- erreur
- Réponse : #<procedure>

Questions

```
(((lambda (x)
    (lambda (y)
      (cons x y)))
  '(1 2 3))
 1)
```

- '(1 1 2 3)
- '((1 2 3) . 1)
- erreur
- Réponse : '((1 2 3) . 1)

Questions

```
((lambda (x)
    ((lambda (y)
       (cons x y))
     '(1 2 3)))
 1)
```

- '(1 1 2 3)
- '((1 2 3) . 1)
- erreur
- Réponse : '(1 1 2 3)