UNIVERSITÉ
CÔTE D'AZUR

Paradigmes et Interprétation

Typage et classes

Julien Provillard
julien.provillard@univ-cotedazur.fr

UNIVERSITÉ CÔTE D'AZUR

Classes

- Nous savons typer une expression dans un langage qui ne contient que des expressions.
- Nous savons interpréter un langage orienté objets avec classes.
- Comment concilier les deux ?

```
{class Posn extends Object
  {x y}
  [dist {+ {get this x} {get this y}}]
  [addDist {* {send this dist 0} {send arg dist 0}}]}

{class Posn3D extends Posn
  {z}
  [dist {+ {get this z} {super dist arg}}]}

{send {new Posn3D 1 2 3} addDist {new Posn 4 5}}
```

UNIVERSITÉ CÔTE D'AZUR

Qu'attend-on du système de typage ?

- Un programme bien formé ne devrait jamais produire des erreurs que l'on pourrait détecter statiquement.

Erreur	Cause	Exemple
`"not a number"`	Un opérande n'est pas un nombre	`{+ 1 {new Posn 1 2}}`
`"not a object"`	Accès à un champ d'une valeur non-objet	`{get 1 x}`
`"not a object"`	Envoi d'un message à une valeur non-objet	`{send 1 dist 2}`
`"wrong field count"`	Initialisation avec un nombre d'arguments incorrect	`{new Posn 1 2 3}`
`"not found"`	Classe introuvable	`{new Posn4D 1 2 3 4}`
`"not found"`	Champ introuvable	`{get {new Posn 1 2} z}`
`"not found"`	Méthode introuvable	`{send {new Posn 1 2} f 0}`

`"not found"`	Méthode introuvable	`{class C extends Object`
		`  {}`
		`  [f {super f arg}]}`

UNIVERSITÉ CÔTE D'AZUR

Grammaire

```
<Class>  ::= {class <Symbol> extends <Symbol>
               {<Field>*}
               <Method>*}
<Field>  ::= [<Symbol> : <Type>]
<Method> ::= [<Symbol> {[arg : <Type>]} : <Type> <Exp>]
<Type>   ::= num
           | <Symbol>
<Exp>    ::= arg
           | this
           | <Number>
           | {+ <Exp> <Exp>}
           | {* <Exp> <Exp>}
           | {new <Symbol> <Exp>*}
           | {get <Exp> <Symbol>}
           | {send <Exp> <Symbol> <Exp>}
           | {super <Symbol> <Exp>}
```

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {send {get this x} dist 0} {send {get this y} dist 0}}]}

42
```

Non, les champs `x` et `y` ne sont pas des objets.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {get this z}}]}

42
```

Non, la classe `Posn` n'a pas de champ `z`.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {send this get-y}}]}

42
```

Non, la classe `Posn` n'a pas de méthode `get-y`.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : Posn
    {+ {get this x} {get this y}}]}

42
```

Non, le type de retour de la méthode `dist` ne correspond pas au type du corps.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {get this y}}]}

42
```

Oui.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {get this y}}]}

{new Posn 12}
```

Non, le nombre d'arguments ne correspond pas au nombre de champs.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {get this y}}]}

{new Posn 12 {new Posn 3 4}}
```

Non, le type du deuxième argument ne correspond pas au type déclaré.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {get this y}}]
  [clone {[arg : num]} : Posn
    {new Posn {get this x} {get this y}}]}

{send {new Posn 1 2} clone 0}
```

Oui.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {get this y}}]
  [clone {[arg : num]} : Posn
    {new Posn {get this x} {get this y}}]}

{class Posn3D extends Posn
  {[z : num]}
  [dist {[arg : num]} : num
    {+ {get this z} {super dist arg}}]}

{new Posn3D 1 2 3}
```

Oui.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {get this y}}]
  [clone {[arg : num]} : Posn
    {new Posn {get this x} {get this y}}]}

{class Posn3D extends Posn
  {[z : num]}
  [dist {[arg : num]} : Posn
    {+ {get this z} {super dist arg}}]}

{new Posn3D 1 2 3}
```

Non, la redéfinition de la méthode `dist` change le type de retour.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {get this y}}]
  [clone {[arg : num]} : Posn
    {new Posn {get this x} {get this y}}]}

{class Posn3D extends Posn
  {[z : num]}
  [dist {[arg : num]} : num
    {+ {get this z} {super dist arg}}]
  [clone {[arg : num]} : num
    {new Posn3D {get this x} {get this y} {get this z}}]}

{new Posn3D 1 2 3}
```

Non, la redéfinition de la méthode `clone` change le type de retour.

UNIVERSITÉ CÔTE D'AZUR

Typage et classes

- Le programme suivant est-il bien formé ?

```
{class Posn extends Object
  {[x : num] [y : num]}
  [dist {[arg : num]} : num
    {+ {get this x} {get this y}}]
  [clone {[arg : num]} : Posn
    {new Posn {get this x} {get this y}}]}

{class Posn3D extends Posn
  {[z : num]}
  [dist {[arg : num]} : num
    {+ {get this z} {super dist arg}}]
  [clone {[arg : num]} : Posn
    {new Posn3D {get this x} {get this y} {get this z}}]}

{new Posn3D 1 2 3}
```

Oui, ce qui signifie qu'une notion de sous-typage est nécessaire.

UNIVERSITÉ CÔTE D'AZUR

Typage : besoins

- Utiliser le nom des classes comme type.
- Vérifier l'existence des classes, champs et méthodes.
- Vérifier le type des corps de méthodes.
- Vérifier le type des arguments de méthodes.
- Vérifier les types des arguments lors de l'initialisation des champs.
- Vérifier la cohérence des redéfinitions de méthodes.
- Mettre en place un sous-typage entre classes mère et classes filles.

UNIVERSITÉ CÔTE D'AZUR

Représentation

```
(define-type ClassT
  [classT (super-name : Symbol)
          (fields : (Listof (Symbol * Type)))
          (methods : (Listof (Symbol * MethodT)))])

(define-type MethodT
  [methodT (arg-type : Type)
           (res-type : Type)
           (body : ExpS)])

(define-type Type
  [numT]
  [objT (class-name : Symbol)])
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (typecheck [expr : ExpS]
                   [classes : (Listof (Symbol * ClassT))]) : Type
  (begin
    (map (lambda (class)
           (typecheck-class class classes))
         classes)
    (typecheck-expr expr classes (objT 'Object) (numT))))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une classe

```
(define (typecheck-class
         [class : (Symbol * ClassT)]
         [classes : (Listof (Symbol * ClassT))]) : Void
  (type-case ClassT (snd class)
    [(classT super-name fds mtds)
     (begin
       (map (lambda (mtd)
              (begin
                (typecheck-method (snd mtd) (objT (fst class)) classes)
                (check-override mtd class classes)))
            mtds)
       (void))]))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une méthode

```
(define (typecheck-method
         [mtd : MethodT]
         [this-t : Type]
         [classes : (Listof (Symbol * ClassT))]) : Void
  (type-case MethodT mtd
    [(methodT arg-t res-t body)
     (let ([body-t (typecheck-expr body classes this-t arg-t)])
       (if (is-subtype? body-t res-t classes)
           (void)
           (type-error body res-t body-t)))]))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une redéfinition

```
(define (check-override
         [mtd : (Symbol * MethodT)]
         [class : (Symbol * ClassT)]
         [classes : (Listof (Symbol * ClassT))]) : Void
  (let ([super-name (classT-super-name (snd class))])
    (type-case (Optionof MethodT)
               (find-method-in-ancestors (fst mtd) super-name classes)
      [(none) (void)]
      [(some super-mtd)
       (if (and (equal? (methodT-arg-type (snd mtd)) (methodT-arg-type super-mtd))
                (equal? (methodT-res-type (snd mtd)) (methodT-res-type super-mtd)))
           (void)
           (error 'typecheck (cat (list "bad override of method "
                                        (to-string (fst mtd))
                                        " in class "
                                        (to-string (fst class))))))])))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une expression

```
(define (typecheck-expr
         [expr : ExpS]
         [classes : (Listof (Symbol * ClassT))]
         [this-t : Type]
         [arg-t : Type]) : Type
  (let* ([typecheck-expr-r
          (lambda (expr)
            (typecheck-expr expr classes this-t arg-t))])
    (type-case ExpS expr
      ...
      )))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une expression

```
(define (typecheck-expr expr classes type-t arg-t)
  (let* ([typecheck-expr-r ...])
    (type-case ExpS expr
      ...
      [(thisS) this-t]
      [(argS) arg-t]
      [(numS n) (numT)]
      ...
      )))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une expression

```
(define (typecheck-expr expr classes type-t arg-t)
  (let* ([typecheck-expr-r ...]
         [typecheck-op (lambda (l r)
                         (let ([t1 (typecheck-expr-r l)]
                               [t2 (typecheck-expr-r r)])
                           (if (numT? t1)
                               (if (numT? t2) (numT) (type-error r (numT) t2))
                               (type-error l (numT) t1)))))])
    (type-case ExpS expr
      ...
      [(plusS l r) (typecheck-op l r)]
      [(multS l r) (typecheck-op l r)]
      ...
      )))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une expression

```
(define (typecheck-expr expr classes type-t arg-t)
  (let* ([typecheck-expr-r ...])
    (type-case ExpS expr
      ...
      [(newS class-name args)
       (let ([fds-types (map snd (extract-fields class-name classes))])
         (if (= (length args)
                (length fds-types))
             (begin
               (map2 check-type args fds-types)
               (objT class-name))
             (error 'typecheck "wrong fields count")))]
      ...
      )))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une expression

```
(define (typecheck-expr expr classes type-t arg-t)
  (let* ([typecheck-expr-r ...]
         [check-type (lambda (expr t)
                       (let ([expr-t (typecheck-expr-r expr)])
                         (if (is-subtype? expr-t t classes)
                             (void)
                             (type-error expr t expr-t))))])
    (type-case ExpS expr
      ...
      [(newS class-name args)
       ... (map2 check-type args fds-types) ... ]
      ...
      )))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une expression

```
(define (typecheck-expr expr classes type-t arg-t)
  (let* ([typecheck-expr-r ...])
    (type-case ExpS expr
      ...
      [(getS obj fd-name)
       (let ([obj-t (typecheck-expr-r obj)])
         (type-case Type obj-t
           [(objT class-name)
            (find fd-name (extract-fields class-name classes))]
           [else (type-error-object obj obj-t)]))]
      ...
      )))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une expression

```
(define (typecheck-expr expr classes type-t arg-t)
  (let* ([typecheck-expr-r ...])
    (type-case ExpS expr
      ...
      [(sendS obj mtd-name arg)
       (let ([obj-t (typecheck-expr-r obj)]
             [new-arg-t (typecheck-expr-r arg)])
         (type-case Type obj-t
           [(objT class-name)
            (typecheck-send class-name mtd-name arg new-arg-t classes)]
           [else (type-error-object obj obj-t)]))]
      ...
      )))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation : vérifier une expression

```
(define (typecheck-expr expr classes type-t arg-t)
  (let* ([typecheck-expr-r ...])
    (type-case ExpS expr
      ...
      [(superS mtd-name arg)
       (let ([this-class (find (objT-class-name this-t) classes)]
             [new-arg-t (typecheck-expr-r arg)])
         (typecheck-send (classT-super-name this-class)
                         mtd-name arg new-arg-t classes))]
      ...
      )))
```

UNIVERSITÉ CÔTE D'AZUR

Fonctions utilitaires : appel de méthode

```
(define (typecheck-send
         [class-name : Symbol] [mtd-name : Symbol]
         [arg : ExpS] [arg-t : Type]
         [classes : (Listof (Symbol * ClassT))]) : Type
  (type-case (Optionof MethodT) (find-method-in-ancestors
                                 mtd-name
                                 class-name
                                 classes)
    [(none) (error 'typecheck "not found")]
    [(some mtd)
     (type-case MethodT mtd
       [(methodT arg-t-expected res-t body)
        (if (is-subtype? arg-t arg-t-expected classes)
            res-t
            (type-error arg arg-t-expected arg-t))])]))
```

UNIVERSITÉ CÔTE D'AZUR

Fonctions utilitaires : sous-typage

- Le sous-typage doit être réflexif : tout type est sous-type de lui-même.
- Une classe est sous-type d'une autre classe si c'est une sous-classe.

```
(define (is-subtype?
         [t1 : Type]
         [t2 : Type]
         [classes : (Listof (Symbol * ClassT))]) : Boolean
  (type-case Type t1
    [(objT name1)
     (type-case Type t2
       [(objT name2)
        (is-subclass? name1 name2 classes)]
       [else #f])]
    [else (equal? t1 t2)]))
```

UNIVERSITÉ CÔTE D'AZUR

Fonctions utilitaires : sous-typage

- Quand est-ce qu'une classe `C1` est sous-classe d'une classe `C2` ?
  - Quand `C1 = C2`,
  - Ou quand la classe mère de `C1` est une sous-classe de `C2`,
  - Si `C1` admet une classe mère (et n'est donc pas `Object`).

```
(define (is-subclass? [name1 : Symbol] [name2 : Symbol]
                      [classes : (Listof (Symbol * ClassT))]) : Boolean
  (cond
    [(equal? name1 name2) #t]
    [(equal? name1 'Object) #f]
    [else (is-subclass?
           (classT-super-name (find name1 classes))
           name2
           classes)]))
```

UNIVERSITÉ CÔTE D'AZUR

Comment interpréter de bout en bout ?

- Les fonctions `parse` et `parse-class` réalisent l'analyse syntaxique et renvoie des `classT` et `ExpS`.
- On effectue la vérification des types.
- On transforme les `classT` en `classS` en ignorant les types pour retrouver la représentation de `inherit.rkt`.
- On compile les classes obtenues et on peut réaliser l'interprétation.

- Le typage intervient donc avant la compilation.
- S'il réussit, on repasse à une version non-typée du langage.

UNIVERSITÉ CÔTE D'AZUR

Comment interpréter de bout en bout ?

```
(define (strip-types [typed-raw : ClassT]) : ClassS
  (type-case ClassT typed-raw
    [(classT super-name fds mtds)
     (classS
      super-name
      (map fst fds)
      (map (lambda (mtd)
             (pair
              (fst mtd)
              (methodT-body (snd mtd))))
           mtds))]))
```

UNIVERSITÉ CÔTE D'AZUR

Comment interpréter de bout en bout ?

```
(define (interp-expr [s : S-Exp] [classes : (Listof S-Exp)]) : Value
  (let ([expr (parse s #f)]
        [typed-raw (map parse-class classes)])
    (begin
      (typecheck expr typed-raw)
      (let* ([untyped-raw (map (lambda (class)
                                 (pair (fst class)
                                       (strip-types (snd class))))
                               typed-raw)]
             [compiled-classes (compile-classes untyped-raw)])
        (interp (exp-s->e expr 'Object)
                compiled-classes
                (objV 'Object empty)
                (numV 0))))))
```