UNIVERSITÉ
CÔTE D'AZUR

Paradigmes et Interprétation

Types

Julien Provillard
julien.provillard@univ-cotedazur.fr

UNIVERSITÉ
CÔTE D'AZUR

TYPES

UNIVERSITÉ CÔTE D'AZUR

Validité des expressions

- **Question :** Quelle est la valeur de l'expression suivante ?

```
{+ 1 2}
```

- **Réponse :** 3

- **Question :** Quelle est la valeur de l'expression suivante ?

```
{+ lambda 1 2}
```

- **Réponse erronée :** L'évaluation produit une erreur.
- **Réponse :** `{+ lambda 1 2}` n'est pas une expression.

UNIVERSITÉ CÔTE D'AZUR

Grammaire

```
<Exp> ::= true
        | false
        | <Number>
        | <Symbol>
        | {+ <Exp> <Exp>}
        | {* <Exp> <Exp>}
        | {= <Exp> <Exp>}
        | {lambda {<Symbol>*} <Exp>}
        | {<Exp> <Exp>*}
        | {if <Exp> <Exp> <Exp>}
```

UNIVERSITÉ CÔTE D'AZUR

Validité des expressions

- **Question :** Le code suivant est-il une expression ?

```
{{lambda {} 1} 2}
```

- **Réponse erronée : Non**
- **Réponse : Oui**, le code est bien conforme à la grammaire.

- **Question :** Quelle est la valeur de l'expression ?
- **Réponse : 1**, avec une tolérance qui n'existe que dans de rares langages.
- **Meilleur réponse :** L'expression devrait être rejetée.

UNIVERSITÉ CÔTE D'AZUR

Validité des expressions

- Appelons ce genre d'expressions : expression mal formée.
  - `{{lambda {} 1} 2}` est mal formée car la fonction `{lambda {} 1}` n'attend pas d'argument.
- Partons du principe que l'interpréteur rejette les expressions mal formées.

- **Question :** Quelle est la valeur de l'expression suivante ?

```
{{lambda {} 1} 2}
```

- **Réponse : Aucune**, l'expression est mal formée.

UNIVERSITÉ CÔTE D'AZUR

Validité des expressions

- **Question :** L'expression suivante est-elle bien formée ?

```
{+ {lambda {} 1} 2}
```

- **Réponse : Oui,** il n'y a pas d'application de fonction.

- **Question :** Quelle est la valeur de l'expression ?
- **Réponse :** Aucune, une erreur se produit.

```
interp : not a number
```

- Une addition ou une multiplication contenant une lambda-expression sera également considérée comme mal formée.

UNIVERSITÉ CÔTE D'AZUR

Validité des expressions

- **Question :** L'expression suivante est-elle bien formée ?

```
{+ {lambda {} 1} 2}
```

- **Réponse : Non**

- **Question :** L'expression suivante est-elle bien formée ?

```
{+ {{lambda {x} x} 1} 2}
```

- **Réponse :** Cela dépend de ce que l'on entend par ne pas contenir de lambda-expression.
  - Aucune sous-expression n'est une lambda-expression -> **Non**
  - Les arguments ne sont pas des lambda-expressions -> **Oui**
- Le résultat de l'expression étant 3, c'est bien entendu la deuxième interprétation qui convient.

UNIVERSITÉ CÔTE D'AZUR

Validité des expressions

- **Question :** L'expression suivante est-elle bien formée ?

```
{+ {{lambda {x} x} {lambda {} 1}} 2}
```

- **Réponse : Oui**
- Mais on souhaiterait que ce ne soit pas le cas.

UNIVERSITÉ CÔTE D'AZUR

Validité des expressions

- **Question :** Est-il possible de définir un procédé **algorithmique** qui rejette toutes les expressions mal formées ?
- **Réponse : Oui**
  - Il suffit de rejeter toutes les expressions.

UNIVERSITÉ CÔTE D'AZUR

Validité des expressions

- **Question :** Est-il possible de définir un procédé **algorithmique** qui ne rejette **que** les expressions mal formées ?
- **Réponse : Non**

```
{+ 1 {if expression 2 {lambda {} 3}}}
```

  - Si `expression` est l'expression `true` alors l'expression est bien formée.
  - Si `expression` est l'expression `false` alors l'expression est mal formée.
  - Dans le cas général, il faudrait connaître la valeur de `expression` et donc en particulier savoir si l'évaluation de l'expression termine. Ça impliquerait de résoudre le problème de la halte !

UNIVERSITÉ CÔTE D'AZUR

Validité des expressions : types

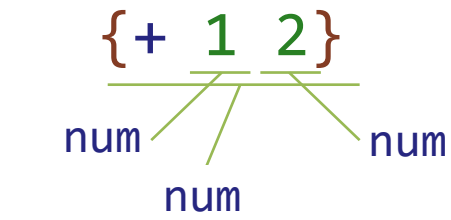
- Il n'est pas possible de ne rejeter que les expressions mal formées.
- Nous allons définir un procédé algorithmique qui rejette toutes les expressions mal formées. Il va donc nécessairement rejeter certaines expressions bien formées.
- `{+ 1 {if expression 2 {lambda {} 3}}}` sera rejeté quelque soit `expression` (y compris, si `expression` est `true`).
- Concrètement, cela consiste à :
  - Donner un type à chaque expression sans l'évaluer,
  - Calculer le type d'une expression complexe à partir du type de ses sous-expressions et
  - Rejeter les expressions que l'on ne peut pas typer.

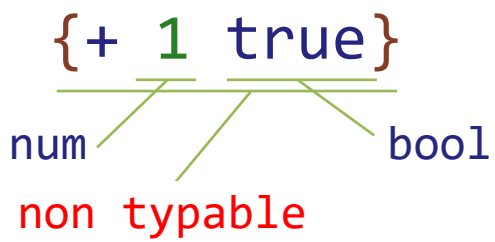
UNIVERSITÉ CÔTE D'AZUR

Exemples

```
1 : num
```

```
true : bool
```





UNIVERSITÉ CÔTE D'AZUR

Règles de typage

- On assigne des types aux expressions atomiques et on se donne des règles de déduction pour les expressions composées.

```
<Number> : num
true   : bool
false  : bool
```

$$\frac{\texttt{expr}_1 \texttt{ : num} \qquad \texttt{expr}_2 \texttt{ : num}}{\texttt{\{+ expr}_1 \texttt{ expr}_2\texttt{\} : num}}$$

$$\texttt{1 : num}$$

$$\texttt{true : bool}$$

$$\frac{\texttt{1 : num} \qquad \texttt{2 : num}}{\texttt{\{+ 1 2\} : num}}$$

$$\frac{\texttt{1 : num} \qquad \texttt{true : bool}}{\texttt{\{+ 1 true\} : non typable}}$$

UNIVERSITÉ CÔTE D'AZUR

Règles de typage

- On assigne des types aux expressions atomiques et on se donne des règles de déduction pour les expressions composées.

```
<Number> : num
true   : bool
false  : bool
```

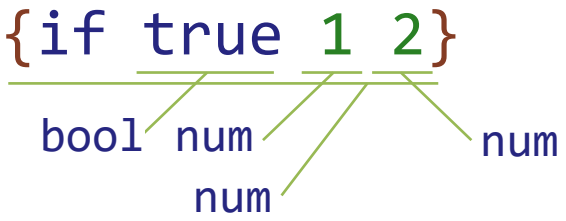
$$\frac{\texttt{expr}_1 \texttt{ : num} \qquad \texttt{expr}_2 \texttt{ : num}}{\texttt{\{+ expr}_1 \texttt{ expr}_2\texttt{\} : num}}$$

$$\frac{\dfrac{\texttt{1 : num} \qquad \texttt{2 : num}}{\texttt{\{+ 1 2\} : num}} \qquad \texttt{3 : num}}{\texttt{\{+ \{+ 1 2\} 3\} : num}}$$

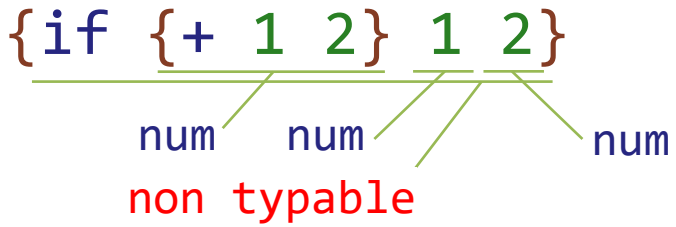
UNIVERSITÉ CÔTE D'AZUR

Règles de typage : branchement conditionnel

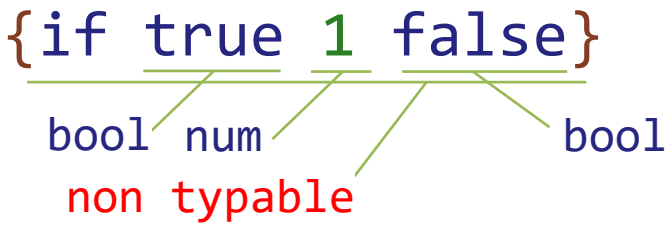
La condition est booléenne.
Les deux branches ont le même type.
On peut assigner un type à l'expression.



La condition n'est pas booléenne.
On ne peut pas assigner de type à l'expression.



Les deux branches n'ont pas le même type.
On ne peut pas assigner de type à l'expression.



UNIVERSITÉ CÔTE D'AZUR

Règles de typage : branchement conditionnel

$$\frac{\texttt{expr}_1 : \texttt{bool} \qquad \texttt{expr}_2 : \tau \qquad \texttt{expr}_3 : \tau}{\{\texttt{if } \texttt{expr}_1\ \texttt{expr}_2\ \texttt{expr}_3\} : \tau}$$

$$\frac{\texttt{true} : \texttt{bool} \qquad \texttt{1} : \texttt{num} \qquad \texttt{2} : \texttt{num}}{\{\texttt{if true 1 2}\} : \texttt{num}}$$

$$\frac{\{\texttt{+ 1 2}\} : \texttt{num} \qquad \texttt{1} : \texttt{num} \qquad \texttt{2} : \texttt{num}}{\{\texttt{if } \{\texttt{+ 1 2}\}\ \texttt{1 2}\} : \texttt{non typable}}$$

$$\frac{\texttt{true} : \texttt{bool} \qquad \texttt{1} : \texttt{num} \qquad \texttt{false}: \texttt{bool}}{\{\texttt{if true 1 false}\} : \texttt{non typable}}$$

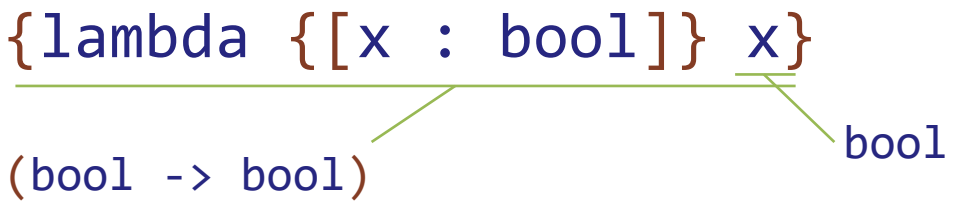
UNIVERSITÉ CÔTE D'AZUR

Règles de typage : identificateurs et fonctions

- Une expression élémentaire peut être un identificateur. Quel est son type ?

```
x : ?
```

- On ne peut pas assigner de type à `x`.
- Il est nécessaire d'avoir des informations supplémentaires sur les identificateurs. Ceux-ci sont nécessairement introduits par des lambda-expressions, on va donc ajouter l'information de type à ce moment et s'en rappeler.



UNIVERSITÉ CÔTE D'AZUR

Règles de typage : identificateurs et fonctions

- On ajoute un environnement de typage pour se souvenir du type des identificateurs.

$$[..., (id, \tau), ...] \vdash id : \tau$$

Si `id` est lié au type τ dans l'environnement, on peut retrouver son type.

- Quand on cherche à assigner un type au corps d'une fonction, il faut au préalable enrichir l'environnement de son paramètre formel.

$$\frac{(id, \tau_1), \Gamma \vdash expr : \tau_2}{\Gamma \vdash \{lambda\ \{[id : \tau_1]\}\ expr\} : (\tau_1 \rightarrow \tau_2)}$$

- Pour les règles précédentes, l'environnement est simplement transmis aux prémisses.

UNIVERSITÉ CÔTE D'AZUR

Règles de typage : identificateurs et fonctions

$$[\ldots, (id, \tau), \ldots] \vdash id : \tau$$

$$\frac{(id, \tau_1), \Gamma \vdash expr : \tau_2}{\Gamma \vdash \{lambda\ \{[id : \tau_1]\}\ expr\} : (\tau_1 \to \tau_2)}$$

- Exemples :

$$[] \vdash x : \text{non typable}$$

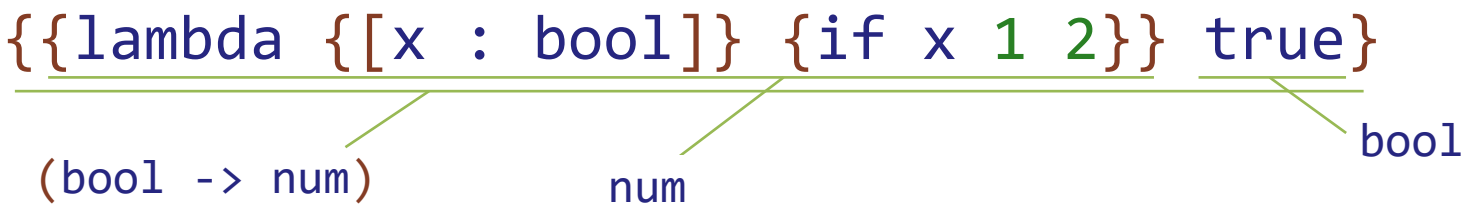
$$\frac{[(x, bool)] \vdash x : bool}{[] \vdash \{lambda\ \{[x : bool]\}\ x\} : (bool \to bool)}$$

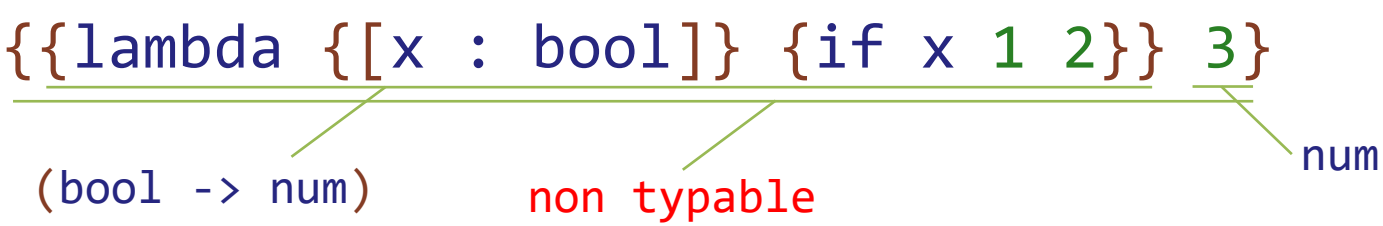
$$\frac{\dfrac{[(x, bool)] \vdash x : bool \qquad [(x, bool)] \vdash 1 : num \qquad [(x, bool)] \vdash 2 : num}{[(x, bool)] \vdash \{if\ x\ 1\ 2\} : num}}{[] \vdash \{lambda\ \{[x : bool]\}\ \{if\ x\ 1\ 2\}\} : (bool \to num)}$$

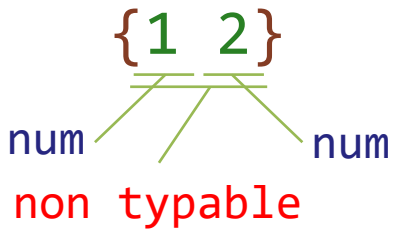
UNIVERSITÉ CÔTE D'AZUR

Règles de typage : appels de fonction

- Il faut vérifier la cohérence entre le type de l'argument et celui du paramètre. Le type de l'expression est le type de retour de la fonction.







UNIVERSITÉ CÔTE D'AZUR

Règles de typage : appels de fonction

- Règle :

$$\frac{\Gamma \vdash expr_1 : (\tau_1 \to \tau_2) \qquad \Gamma \vdash expr_2 : \tau_1}{\Gamma \vdash \{expr_1\ expr_2\} : \tau_2}$$

- Exemple :

$$\frac{[] \vdash \{\text{lambda } \{[x : bool]\} \{\text{if } x\ 1\ 2\}\} : (bool \to num) \qquad [] \vdash true : bool}{[] \vdash \{\{\text{lambda } \{[x : bool]\} \{\text{if } x\ 1\ 2\}\}\ true\} : num}$$

$$\frac{[] \vdash \{\text{lambda } \{[x : bool]\} \{\text{if } x\ 1\ 2\}\} : (bool \to num) \qquad [] \vdash 3 : num}{[] \vdash \{\{\text{lambda } \{[x : bool]\} \{\text{if } x\ 1\ 2\}\}\ 3\} : \text{non typable}}$$

$$\frac{[] \vdash 1 : num \qquad [] \vdash 2 : num}{[] \vdash \{1\ 2\} : \text{non typable}}$$

UNIVERSITÉ CÔTE D'AZUR

Grammaire du langage typé

```
<Exp>  ::= <Number>
         | <Symbol>
         | {+ <Exp> <Exp>}
         | {* <Exp> <Exp>}
         | {lambda {[<Symbol> : <Type>]} <Exp>}
         | {<Exp> <Exp>}

<Type> ::= num
         | bool
         | (<Type> -> <Type>)
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define-type Exp
  [numE (n : number)]
  [idE (s : symbol)]
  [plusE (l : Exp) (r : Exp)]
  [multE (l : Exp) (r : Exp)]
  [lamE (par : symbol) (par-type : Type) (body : Exp)]
  [appE (fun : Exp) (arg : Exp)])

(define-type Type
  [numT]
  [boolT]
  [arrowT (par : Type) (res : Type)])

(define-type TypeBinding [tbind (name : symbol) (type : Type)])
(define-type-alias TypeEnv (listof TypeBinding))
```

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(numE n) (numT)]
    ... ))
```

$\Gamma \vdash$ <Number> : num

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(idE s) (type-lookup s env)]
    ... ))
```

$$[..., (id, \tau), ...] \vdash id : \tau$$

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(plusE l r)
     (type-case Type (typecheck l env)
       [(numT)
        (type-case Type (typecheck r env)
           [(numT) (numT)]
           [else (type-error r "num")])]
       [else (type-error l "num")])]
    ... ))
```

$$\frac{\Gamma \vdash expr_1 : num \qquad \Gamma \vdash expr_2 : num}{\Gamma \vdash \{+\ expr_1\ expr_2\} : num}$$

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(lamE par par-type body)
     (arrowT par-type
             (typecheck body
                        (extend-env (tbind par par-type) env)))]
    ... ))
```

$$\frac{(\text{id}, \tau_1), \Gamma \vdash \text{expr} : \tau_2}{\Gamma \vdash \{\text{lambda } \{[\text{id} : \tau_1]\} \text{ expr}\} : (\tau_1 \text{ -> } \tau_2)}$$

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(appE fun arg)
     (type-case Type (typecheck fun env)
       [(arrowT par-type res-type)
        (if (equal? par-type (typecheck arg env))
            res-type
            (type-error arg (to-string par-type)))]
       [else (type-error fun "function")])]
    ... ))
```

$$\frac{\Gamma \vdash expr_1 : (\tau_1 \text{ -> } \tau_2) \qquad \Gamma \vdash expr_2 : \tau_1}{\Gamma \vdash \{expr_1\ expr_2\} : \tau_2}$$

UNIVERSITÉ CÔTE D'AZUR

Cas d'étude : paires vs listes

- En `Racket`, qui est un langage non typé, le mot clé `cons` permet de créer à la fois des paires et des listes.
- En fait une liste est soit la liste vide, soit une paire dont le deuxième élément est une liste.

- En `plait`, qui est typé, le mot clé `cons` sert uniquement à créer des listes. Les paires sont créées à l'aide du mode clé `pair`.

Pourquoi cette différence ?

UNIVERSITÉ CÔTE D'AZUR

Cas d'étude : paires vs listes

- Quels sont les types des expressions suivantes ?
  - `(fst (pair 1 true)) : Number`
  - `(fst (pair 1 2)) : Number`
  - `(snd (pair 1 true)) : Boolean`
  - `(snd (pair 1 2)) : Number`
  - `(pair 1 true) : (Number * Boolean)`
  - `(pair 1 2) : (Number * Number)`

Pourquoi ne pas faire cela pour les listes ?

UNIVERSITÉ CÔTE D'AZUR

Cas d'étude : paires vs listes

- En programmation fonctionnelle, les listes sont les éléments de base pour la plupart des algorithmes récursifs.
- En particulier, lors de l'appel d'une fonction `f` sur une liste `L`, il est courant d'appeler `(f (rest L))`.
- Comme le type du paramètre de `f` est fixé, cela signifie que `L` et `(rest L)` doivent avoir le même type.
- Les listes ne sont donc pas des paires du point de vue du typage.
- Pour les mêmes raisons, tous les éléments d'une liste doivent partager le même type.

UNIVERSITÉ
CÔTE D'AZUR

INFÉRENCE DE TYPE

UNIVERSITÉ CÔTE D'AZUR

Expressions et types

- **Question :** Quel est le type de l'expression suivante ?

```
{lambda {x} {+ x 1}}
```

- **Réponse :** Ce n'est pas une expression de notre langage, il manque le paramètre de type.

- Mais on aurait envie de répondre `(num -> num)`.

UNIVERSITÉ CÔTE D'AZUR

Inférence de type

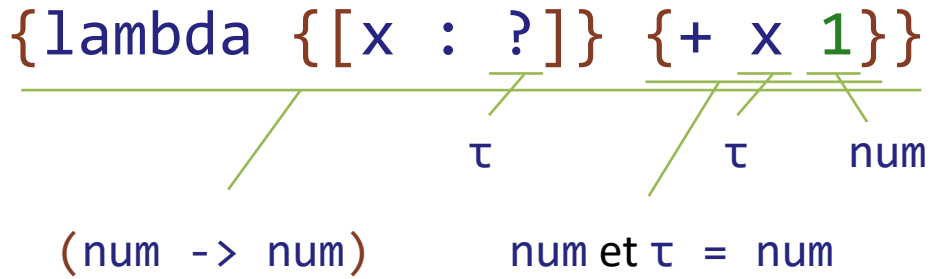
- L'**inférence de type** est un procédé qui consiste à ajouter automatiquement les annotations de type lorsque le programmeur les omet.
- Dans notre langage, nous allons ajouter un type qui indique explicitement l'omission.

```
{lambda {[x : ?]} {+ x 1}}
```

```
<Type> ::= num
         | bool
         | (<Type> -> <Type>)
         | ?
```

UNIVERSITÉ CÔTE D'AZUR

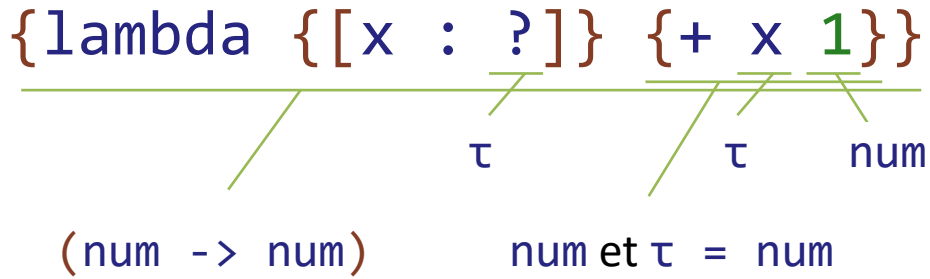
Inférence de type

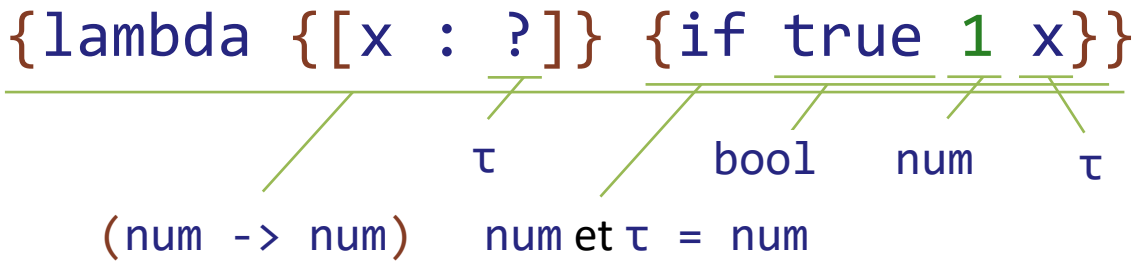


- On va créer une nouvelle variable de type pour chaque `?`.
- Le type `τ` doit être `num`.
- On change la comparaison des types pour mettre en place une équivalence entre types.

UNIVERSITÉ CÔTE D'AZUR

Inférence de type

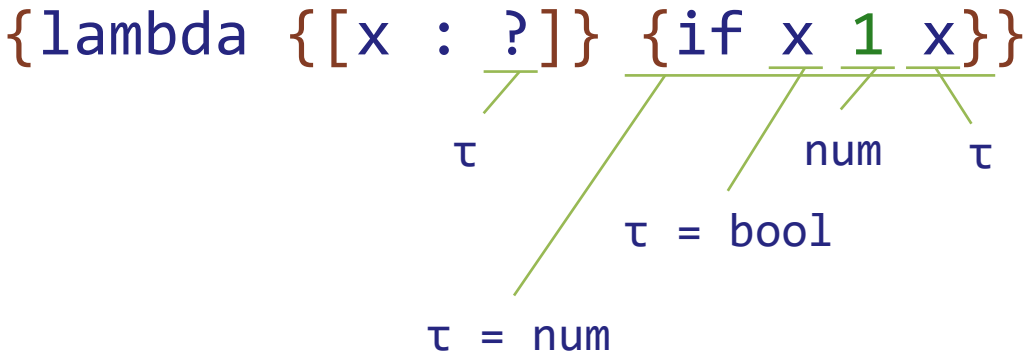




- Le type τ doit être num.

UNIVERSITÉ CÔTE D'AZUR

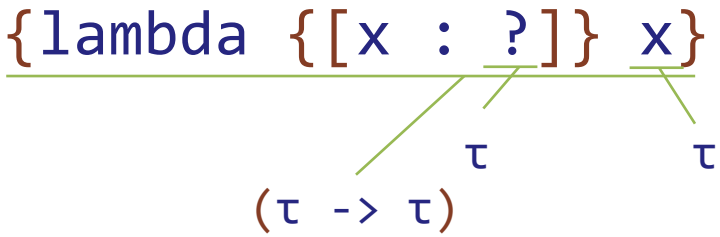
Inférence de type : contradictions



- Le type `τ` doit être `bool`.
- Le type `τ` doit être `num`.
- Impossible, `τ` ne peut pas être à la fois `num` et `bool`.

UNIVERSITÉ CÔTE D'AZUR

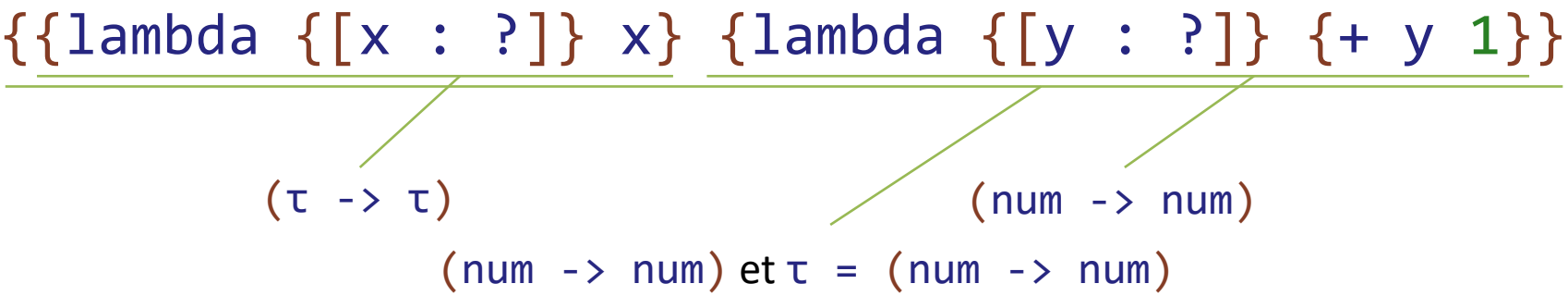
Inférence de type : possibilités multiples



- Dans certains cas, plusieurs types peuvent convenir.
  - `(num -> num)`
  - `(bool -> bool)`
  - `((bool -> num) -> (bool -> num))`
- Le vérificateur de type préserve alors la variable de type.

UNIVERSITÉ CÔTE D'AZUR

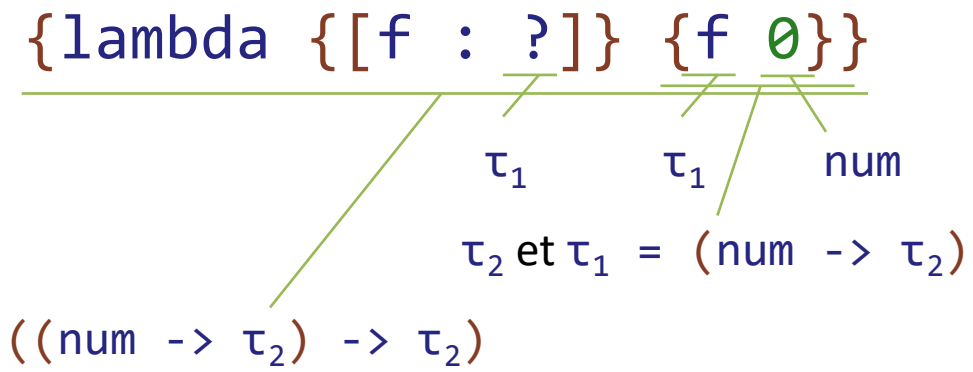
Inférence de type : appels de fonctions



- Le type du paramètre et celui de l'argument doivent être équivalents.

UNIVERSITÉ CÔTE D'AZUR

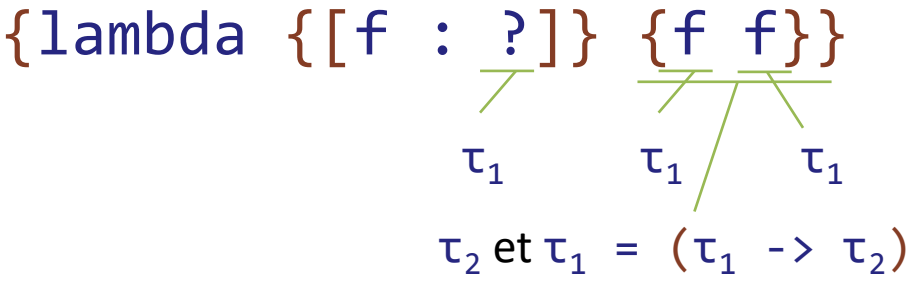
Inférence de type : appels de fonctions



- On sait que `f` est une fonction mais on ne connait pas son type de retour.
- On crée une variable de type τ_2 pour celui-ci et on regarde les équivalences qui en découle.

UNIVERSITÉ CÔTE D'AZUR

Inférence de type : équations cycliques



- Mais alors $\tau_1 = (\tau_1 \rightarrow \tau_2) = ((\tau_1 \rightarrow \tau_2) \rightarrow \tau_2) = \ldots$
- Il n'y a pas de solution.
- De manière générale, on ne peut pas rendre équivalent un type τ_1 et un autre type qui contient τ_1.

UNIVERSITÉ CÔTE D'AZUR

Unification

- Auparavant, pour comparer les types, on utilisait une simple égalité.

```
equal? : (Type Type -> Boolean)
```

- Maintenant, on utilise des relations d'équivalence qui unifient des types.
- Soit on peut unifier les types et on ne renvoie rien.
- Soit ce n'est pas possible et on produit une erreur.

```
unify! : (Type Type -> Void)
```

UNIVERSITÉ CÔTE D'AZUR

Unification

- Pour connaître le type réel d'une variable de type, il faudra résoudre les équivalences où elle intervient.

$$\texttt{resolve : (Type -> Type)}$$

- Par exemple, si τ_1 et τ_2 sont des nouvelles variables de type, alors :

$$\texttt{(resolve } \tau_1\texttt{)} = \tau_1 \text{ et } \texttt{(resolve } \tau_2\texttt{)} = \tau_2$$

- Si on unifie τ_1 et num, alors :

$$\texttt{(resolve } \tau_1\texttt{)} = \texttt{num et (resolve (}\tau_1\texttt{ -> }\tau_2\texttt{))} = \texttt{(num -> }\tau_2\texttt{)}$$

- Si on unifie τ_1 et τ_2, alors :

$$\texttt{(resolve } \tau_2\texttt{)} = \texttt{num}$$

UNIVERSITÉ CÔTE D'AZUR

Représentation

```
(define-type Type
  [numT]
  [boolT]
  [arrowT (par : Type) (res : Type)]
  [varT (is : (Boxof (Optionof Type)))])
```

- La boîte dans le type `varT` indique si une équivalence a été trouvée.
- Une variable fraîche sera donc de la forme :

```
(varT (box (none)))
```

- Si cette même variable a été unifié avec le type num, elle sera alors de la forme :

```
(varT (box (some (numT))))
```

UNIVERSITÉ CÔTE D'AZUR

Représentation

```
(define-type Type
  [numT]
  [boolT]
  [arrowT (par : Type) (res : Type)]
  [varT (is : (Boxof (Optionof Type)))])

(define (unify! [t1 : Type] [t2 : Type]) : Void
  (type-case Type t1
    [(varT is)
     ... (set-box! is (some t2)) ... ]
    ... ))
```

UNIVERSITÉ CÔTE D'AZUR

Unification : exemples

- Dans certains cas, l'unification ne fait que vérifier la cohérence des types.

```
(test (unify! (numT) (numT)) (void))

(test (unify! (boolT) (boolT)) (void))

(test/exn (unify! (numT) (boolT)) "unable to unify")
```

UNIVERSITÉ CÔTE D'AZUR

Unification : exemples

- Dans d'autres, elle met en place les équivalences.

```
(test (unify! (varT (box (none))) ; le contenu de la variable devient (some (numT))
              (numT))
      (void))
```

- Ce qui permet de faire des vérifications ultérieurement.

```
(test (unify! (varT (box (some (numT))))
              (numT))
      (void))

(test/exn (unify! (varT (box (some (boolT))))
                  (numT))
          "unable to unify")
```

UNIVERSITÉ CÔTE D'AZUR

Unification : exemples

- Elle utilise énormément les effets de bords

```
(test/exn (let ([t (varT (box (none)))])
            (begin
              (unify! t (numT))
              (unify! t (boolT))))
          "unable to unify") ; impossible d'unifier (numT) et (boolT)

(test (let ([t (varT (box (none)))])
        (begin
          (unify! t (numT))
          (unify! t (numT))))
      (void)) ; ici, il n'y a pas de problème
```

UNIVERSITÉ CÔTE D'AZUR

Unification : exemples

- Cas des fonctions

```
(test (let ([t (varT (box (none)))])
        (begin
          (unify! (arrowT t (boolT))
                  (arrowT (numT) (boolT)))
          (unify! t (numT))))
      (void))

(test/exn (let ([t (varT (box (none)))])
            (begin
              (unify! (arrowT t (boolT))
                      (arrowT (numT) (boolT)))
              (unify! t (boolT))))
          "unable to unify")
```

UNIVERSITÉ CÔTE D'AZUR

Unification : exemples

- Types cycliques

```
(test/exn (let ([t (varT (box (none)))])
            (unify! t (arrowT t (boolT))))
          "unable to unify")
```

UNIVERSITÉ CÔTE D'AZUR

Unification : exemples

- Cohérence des variables

```
(test (let ([t1 (varT (box (none)))]
            [t2 (varT (box (none)))])
        (begin
          (unify! t1 t2)
          (unify! t1 (numT))
          (unify! t2 (numT))))
      (void))

(test/exn (let ([t1 (varT (box (none)))]
                [t2 (varT (box (none)))])
            (begin
              (unify! t1 t2)
              (unify! t1 (numT))
              (unify! t2 (boolT))))
          "unable to unify")
```

UNIVERSITÉ CÔTE D'AZUR

Unification : exemples

- Cohérence des variables

```
(test/exn (let ([t1 (varT (box (none)))]
                [t2 (varT (box (none)))])
            (begin
              (unify! t1 (arrowT t2 (boolT)))
              (unify! t1 (arrowT (numT) t2))))
          "unable to unify")
```

UNIVERSITÉ CÔTE D'AZUR

Unification : algorithme

- Unifier les types τ_1 et τ_2
  - Si τ_1 est une variable de type
    - Si τ_3 est assigné à τ_1, unifier τ_3 et τ_2
    - Si τ_2 est déjà équivalent à τ_1, ne rien faire
    - Si τ_2 contient τ_1, échouer
    - Sinon, assigner τ_2 à τ_1
  - Sinon
    - Si τ_2 est une variable de type, unifier τ_2 et τ_1
    - Si $\tau_1 = (\tau_3 \text{ -> } \tau_4)$ et $\tau_2 = (\tau_5 \text{ -> } \tau_6)$
      - Unifier τ_3 et τ_5
      - Unifier τ_4 et τ_6
    - Si τ_1 et τ_2 sont tous les deux `num` ou `bool`, ne rien faire
    - Sinon, échouer

`(resolve` τ_2`)` donne τ_1 ?

`(occurs?` τ_1 τ_2`)`

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (unify! [t1 : Type] [t2 : Type] [e : Exp]) : Void
  (type-case Type t1
    [(varT is1)
     (type-case (Optionof Type) (unbox is1)
       [(some t3) (unify! t3 t2 e)]
       [(none)
        (let ([t3 (resolve t2)])
          (cond
            [(eq? t1 t3) (void)]
            [(occurs t1 t3) (type-error e t1 t3)]
            [else (set-box! is1 (some t3))]))])]
    [else
     ... ]))
```

L'expression qui est à l'origine de l'unification. Permet de fournir des informations supplémentaires en cas d'erreur de typage.

`eq?` Pour l'égalité **en mémoire**.

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (unify! [t1 : Type] [t2 : Type] [e : Exp]) : Void
  (type-case Type t1
    [(varT is1)
     ... ]
    [else
     (type-case Type t2
       [(varT is2) (unify! t2 t1 e)]
       [(arrowT t3 t4)
        (type-case Type t1
          [(arrowT t5 t6)
           (begin (unify! t3 t5 e)
                  (unify! t4 t6 e))]
          [else (type-error e t1 t2)])]
       [else (if (equal? t1 t2)
                 (void)
                 (type-error e t1 t2))])]))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (resolve [t : Type]) : Type
  (type-case Type t
    [(varT is)
     (type-case (Optionof Type) (unbox is)
       [(none) t]
       [(some t2) (resolve t2)])]
    [(arrowT t1 t2) (arrowT (resolve t1) (resolve t2))]
    [else t]))
```

UNIVERSITÉ CÔTE D'AZUR

Implémentation

```
(define (occurs [t1 : Type] [t2 : Type]) : Boolean ; t1 est un varT
  (type-case Type t2
    [(arrowT t3 t4) (or (occurs t1 t3) (occurs t1 t4))]
    [(varT is)
     (or (eq? t1 t2)
         (type-case (Optionof Type) (unbox is)
           [(none) #f]
           [(some t3) (occurs t1 t3)]))]
    [else #f]))
```

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(numE n) (numT)]
    ... ))
```

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(idE s) (type-lookup s env)]
    ... ))
```

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(plusE l r)
     (begin
       (unify! (typecheck l env) (numT) l)
       (unify! (typecheck r env) (numT) r)
       (numT)))
    ... ))
```

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(lamE par par-type body)
     (arrowT par-type
             (typecheck body
                        (extend-env (tbind par par-type) env)))]
    ... ))
```

UNIVERSITÉ CÔTE D'AZUR

Vérification de type

```
(define (typecheck [e : Exp] [env : TypeEnv]) : Type
  (type-case Exp e
    ...
    [(appE fun arg)
     (let ([t1 (varT (box (none)))]
           [t2 (varT (box (none)))])
       (begin
         (unify! (typecheck fun env) (arrowT t1 t2) fun)
         (unify! (typecheck arg env) t1 arg)
         t2))]
    ... ))
```