Une conversation humoristique sur les langues sans typage fort:

https://www.destroyallsoftware.com/talks/wat

Le contrôle du type

CSI 3520

Amy Felty

Université d'Ottawa

Dernier Cours

Une histoire des langages de programmation fonctionnels

- Church et le lambda calcul
- Scheme
- ML (OCaml)
- Maintenant: F#, Clojure, Scala, Map-Reduce, ...

OCaml

- Langage fonctionel: on analyse les données existantes et on produit de *nouvelles données immuables*
- Langage typé et basé sur le lambda calcul
- Les données immuables sont les données par défaut. Les données mutables sont possibles (impératif, objets)

Le contrôle du type

- Toutes les valeurs ont un type et toutes les expressions ont un type.
- Comme dans Java, mais plus important dans un langage fonctionel
- Le type d'une expression est déterminé par le type de ses sous-expressions
- Notation: (e : t) l'expression e a le type t.
- Par exemple:

2 : int	"bonjour" : string
2 + 2 : int	"Je dis " ^ "bonjour" : string

Règles du contrôle du type

- Il existe un ensemble de règles simples qui régissent le contrôle de type
  - les programmes qui ne suivent pas les règles ne passeront pas le contrôle du type et OCaml refusera de les compiler
  - au début, cela semblera être un obstacle …

- Mais les types sont très utiles :
  - ils nous *aident à réfléchir* à *la façon de construire* nos programmes
  - ils nous aident à *trouver des erreurs bête*
  - ils nous aident à détecter rapidement les erreurs de compatibilité lorsque nous modifions et *entretenons notre code*
  - ils nous permettent *d'appliquer de puissants invariants* sur nos structures de données

Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

(4) si e1 : int et e2 : int
alors e1 * e2 : int

Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

(4) si e1 : int et e2 : int
alors e1 * e2 : int

(5) si e1 : string et e2 : string
alors e1 ^ e2 : string

(6) si e : int
alors string_of_int e : string

Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

(4) si e1 : int et e2 : int
alors e1 * e2 : int

(5) si e1 : string et e2 : string
alors e1 ^ e2 : string

(6) si e : int
alors string_of_int e : string

- application des règles

2 : int et 3 : int. (par règle 1)

Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

(4) si e1 : int et e2 : int
alors e1 * e2 : int

(5) si e1 : string et e2 : string
alors e1 ^ e2 : string

(6) si e : int
alors string_of_int e : string

- application des règles

2 : int et 3 : int. (par règle 1)
donc (2 + 3) : int (par règle 3)

Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

(4) si e1 : int et e2 : int
alors e1 * e2 : int

(5) si e1 : string et e2 : string
alors e1 ^ e2 : string

(6) si e : int
alors string_of_int e : string

- application des règles

2 : int et 3 : int. (par règle 1)
donc (2 + 3) : int (par règle 3)
5 : int (par règle 1)

Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour t… haîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

(5) si e1 : string et e2 : string
alors e1 ^ e2 : string

…tring_of_int e : string

Ceci est une preuve formelle que l'expression est bien typée!

- application des règles

2 : int et 3 : int. (par règle 1)
donc (2 + 3) : int (par règle 3)
5 : int (par règle 1)
donc (2 + 3) * 5 : int (par règle 4 et conclusions précédentes)

Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

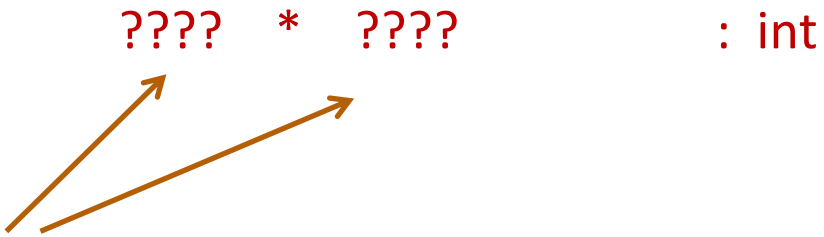
(4) si e1 : int et e2 : int
alors e1 * e2 : int

(5) si e1 : string et e2 : string
alors e1 ^ e2 : string

(6) si e : int
alors string_of_int e : string

- une autre perspective:

règle (4) exprime que
toute expression de
type int peut remplacer ????



Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

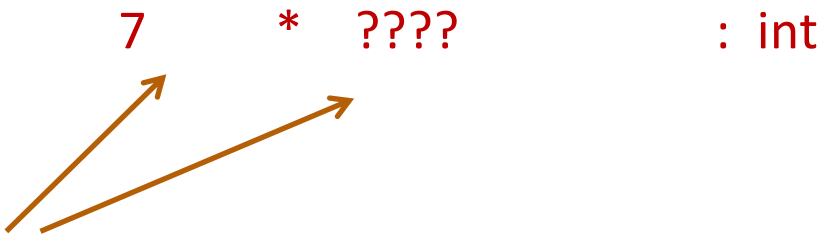
(4) si e1 : int et e2 : int
alors e1 * e2 : int

(5) si e1 : string et e2 : string
alors e1 ^ e2 : string

(6) si e : int
alors string_of_int e : string

- une autre perspective:

règle (4) exprime que
toute expression de
type int peut remplacer ????



Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

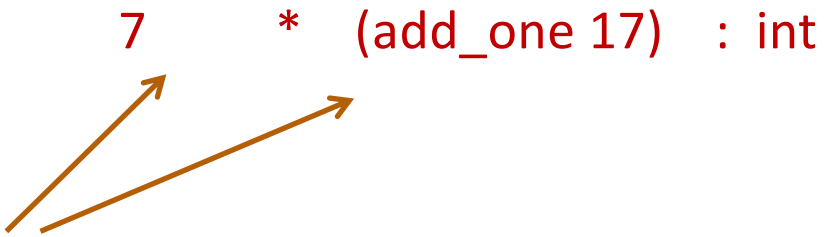
(4) si e1 : int et e2 : int
alors e1 * e2 : int

(5) si e1 : string et e2 : string
alors e1 ^ e2 : string

(6) si e : int
alors string_of_int e : string

- une autre perspective:

règle (4) exprime que
toute expression de
type int peut remplacer ????



Règles du contrôle du type

- Il est toujours possible de démarrer l'interpréteur OCaml pour trouver un type d'expression simple:

```
$ ocaml
        Objective Caml Version 4.07.0
#
```

Règles du contrôle du type

- Il est toujours possible de démarrer l'interpréteur OCaml pour trouver un type d'expression simple:

```
$ ocaml
        Objective Caml Version 4.07.0
# 3 + 1;;
```

Règles du contrôle du type

- Il est toujours possible de démarrer l'interpréteur OCaml pour trouver un type d'expression simple:

```
$ ocaml
        Objective Caml Version 4.07.0
# 3 + 1;;
- : int = 4
#
```

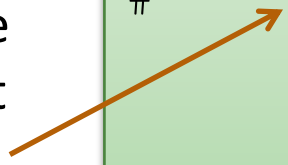
pour connaître le type et la valeur

Règles du contrôle du type

- Il est toujours possible de démarrer l'interpréteur OCaml pour trouver un type d'expression simple:

```
$ ocaml
        Objective Caml Version 4.07.0
# 3 + 1;;
- : int = 4
# "hello " ^ "world";;
- : string = "hello world"
#
```

pour connaître le type et la valeur



Règles du contrôle du type

- Il est toujours possible de démarrer l'interpréteur OCaml pour trouver un type d'expression simple:

```
$ ocaml
        Objective Caml Version 4.07.0
# 3 + 1;;
- : int = 4
# “hello ” ^ “world”;;
- : string = “hello world”
# #quit;;
$
```

Règles du contrôle du type

- exemples de règles

(1) 0 : int (et de même pour toute autre constante entière n)

(2) "abc" : string (et de même pour toute autre constante de chaîne"…")

(3) si e1 : int et e2 : int
alors e1 + e2 : int

(4) si e1 : int et e2 : int
alors e1 * e2 : int

(5) si e1 : string et e2 : string
alors e1 ^ e2 : string

(6) si e : int
alors string_of_int e : string

- violer les règles :

"hello" : string (par règle 2)
1 : int (par règle 1)
1 + "hello" : ?? (PAS DE TYPE! La règle 3 ne s'applique pas!)

Règles du contrôle du type

- violer les règles :

```
# "hello" + 1;;
Error: This expression has type string but an
expression was expected of type int
```

- Le message d'erreur indique le type attendu et le type inféré pour la sous-expression
- Notez que ceci est un exemple du cours précédent
- C'est ***une bonne chose*** que le contrôle du type échoue sur cette expression.

"Well typed programs do not go wrong"

Robin Milner, 1978

Règles du contrôle du type

- violer les règles :

```
# "hello" + 1;;
Error: This expression has type string but an
expression was expected of type int
```

- Une correction possible :

```
# "hello" ^ (string_of_int 1);;
- : string = "hello1"
```

- *L'un des meilleurs moyens de devenir un bon programmeur ML est de comprendre les messages d'erreur de type*.

Règles du contrôle du type

si e1 : bool
et e2 : t et e3 : t (le même type t, pour un certain type t)
alors if e1 then e2 else e3 : t (encore, le même type t)

Règles du contrôle du type

- Les erreurs de type pour les instructions "if" ne sont pas toujours claires. Exemple: construisez une chaîne de s en la concaténant n fois:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

Règles du contrôle du type

- Les erreurs de type pour les instructions "if" ne sont pas toujours claires. Exemple: construisez une chaîne de s en la concaténant n fois:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

OCaml:

```
Error: This expression has type int but an
expression was expected of type string
```

Règles du contrôle du type

- Les erreurs de type pour les instructions "if" ne sont pas toujours claires. Exemple: construisez une chaîne de s en la concaténant n fois:

```
let rec concatn s n =
  if n <= 0 then
    ...
  else
    s ^ (concatn s (n-1))
```

OCaml:

```
Error: This expression has type int but an
expression was expected of type string
```

Règles du contrôle du type

- Les erreurs de type pour les instructions "if" ne sont pas toujours claires. Exemple: construisez une chaîne de s en la concaténant n fois:

les types ne sont pas les mêmes!

```
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

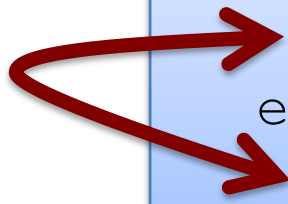
OCaml:

```
Error: This expression has type int but an
expression was expected of type string
```

Règles du contrôle du type

- Les erreurs de type pour les instructions "if" ne sont pas toujours claires. Exemple: construisez une chaîne de s en la concaténant n fois:

les types ne sont pas les mêmes!



```
let rec concatn s n =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

Le contrôle du type pointe sur la branche correcte comme cause de erreur parce que son type n'est pas
le même que le type de la branche précédente.
En réalité, l'erreur est dans la branche précédente.

Conclusion: *Parfois, il faut chercher une erreur dans une branche precedente,* même si le vérificateur de type pointe vers une branche qui apparaît plus tard.
Le contrôle du type ne sait pas ce que veut l'utilisateur.

Une stratégie: ajouter des annotations de type

```
let rec concatn (s:string) (n:int) : string =
  if n <= 0 then
    0
  else
    s ^ (concatn s (n-1))
```

Error: This expression has type int but an expression was expected of type string

LES EXCEPTIONS:
SONT-ILS LA CAUSE D'UN PROGRAMME QUI "GOES WRONG"?

Règles du contrôle du type

- Considérez l'expression suivante :

```
# 3 / 0 ;;
Exception: Division_by_zero.
```

- Pourquoi le contrôle du type de ML ne nous indique-t-il pas que l'expression mènera à une exception.

Règles du contrôle du type

- Considérez l'expression suivante :

```
# 3 / 0 ;;
Exception: Division_by_zero.
```

- Pourquoi le contrôle du type de ML ne nous indique-t-il pas que l'expression mènera à une exception.
  - En général, pour détecter une erreur de division par zéro, il faut savoir que le diviseur évaluera à 0.
  - En général, décider si le diviseur évaluera à 0 nécessite de résoudre le problème de l'arrêt:

```
# 3 / (if turing_machine_halts m then 0 else 1);;
```

Est-ce qu'on triche?

"Well typed programs do not go wrong"

Robin Milner, 1978

(3 / 0) est bien typé.

Est-ce que c'est le cas que « it goes wrong »? Réponse: Non.

"Go wrong" est un terme technique qui signifie qu'il n'y a pas de sémantique bien définie. Lever une exception a une sémantique bien définie sur laquelle on peut raisonner, et on peut appliquer un gestionnaire d'exceptions.

Donc, on ne triche pas.

Sûreté et sécurité

"Well typed programs do not go wrong"

Les langages de programmation qui satisfont cette propriété ont des systèmes de type *sécurisés*. Ils s'appellent des langages *sûres*.

Les langages sûres n'ont pas de vulnérabilités de « buffer overrun », de pointeur non initialisées, etc.
(Mais il peut encore y avoir des bugs!)

langages sûres : ML, Java, Python, …

langages non sûres : C, C++, Pascal

Well typed programs do not go wrong

Robin Milner

Turing Award, 1991

"For three distinct and complete achievements:

1. LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction;

2. ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism;

3. CCS, a general theory of concurrency.

In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics."

"Well typed programs do not go wrong"

Robin Milner, 1978

RÉSUMÉ

OCaml

OCaml est un langage de programmation *fonctionel*

- Java fonctionne principalement en *modifiant* les données.
- OCaml fonctionne principalement en produisant de *nouvelles données immuables*

- OCaml est un langage de programmation *typé*
  - *le type* d'une expression *prévoit correctement* la sorte de *la valeur* qui sera calculée
  - il y a des règles qui définissent si une expression (ou programme) passe le contrôle du type
    - ces règles forment une logique formelle ... ce n'est pas par hasard que les langages comme ML sont utilisés dans les démonstrateurs de théorèmes
  - Le système de type est *sécurisé*. Le langage est *sûre*.