Systèmes de Types

INF 441 : Programmation Avancée

Xavier Rival

12 avril 2017

Quelques définitions et leurs types

Considérons un exemple vu lors du dernier cours tel que nous l'avons saisi dans le toplevel :

```
# let x = "abc" ^ " 78" ;;
val x : string = "abc 78"
# let f = fun x -> x + 1 ;;
val f : int -> int = <fun>
# let g x = x - 2 ;;
val g : int -> int = <fun>
# let y = g (f 8) ;;
val y : int = 7
```

Pour chaque déclaration, le compilateur renvoie deux éléments :

- un **type** : `string`, `int -> int`, `int`...
- une **valeur** : `"abc 78"`, `<fun>`, `7`

Mais à quoi ce type peut il bien servir ?

Des programmes rejetés par la faute des types

Voyons un autre exemple :

```
# let x = "abc " ;;
val x : string = "abc "
# let y = 78 ;;
val y : int = 78
# x ^ y ;;
Characters 4-5:
  x ^ y ;;
      ^
Error: This expression has type int but an expression
    was expected of type
         string
```

- en **Java**, nous aurions fait `x + y` et cela aurait marché...
- mais **OCaml rejette ce programme**... Pourquoi ?

Des programmes rejetés par la faute des types (2)

Le code suivant semble moins absurde :

```
# let a = 8 ;; (* a est un entier *)
val a : int = 8
# a.(0) ;;      (* utilisons le comme un tableau... *)
Characters 0-1:
  a.(0) ;;
  ^
Error: This expression has type int but an expression
    was expected of type
           'a array
```

- ce cas semble moins inattendu, car **on ne peut donner aucun sens à ce programme...**
- d'ailleurs, Java aurait fait de même (essayer...)

Des types et des programmes

Les types garantissent une forme de sûreté

Well typed programs do not go wrong

Dixit Robin Milner, Turing Award, un des pères fondateurs des types présents dans les langages modernes

Les points principaux de ce cours :

- la forme de **sûreté** garantie par les types
- **l'utilisation** des types pour écrire des programmes **plus clairs** et **plus robustes**
- les **principales constructions de types** en OCaml

En bref, nous voulons **comprendre les types pour mieux les utiliser**

Vers une définition intuitive

Dans ce cours, nous considérons principalement le langage OCaml, puis discutons d'autres langages dans un second temps...

On définit un ensemble de **valeurs de base** :

- intuitivement, une valeur est un terme "déjà calculé"
- exemples : 72, `true`, `"abc"` mais aussi `(fun x -> x + 1)` sont des valeurs
- `(if b && x < 8 then x + y else y - z)` n'est pas une valeur

L'évaluation d'un programme est une suite d'étapes de calcul, dont on attend **qu'elle produise une valeur**

- exemple : l'évaluation de `"abc" ^ " 78"` produit la valeur `"abc 78"`
- l'évaluation d'un programme pourrait aussi **bloquer** / **planter** (on ne sait pas comment calculer) ou **ne pas terminer**

Exemple : un langage réduit

On considère un **petit sous-ensemble** de OCaml avec **entiers** et **booléens**, pour mieux comprendre les différentes notions...

p	::=	`true` | `false`	valeurs booléennes
	|	`0` | `1` | `...`	valeurs entières
	|	`p + p`	addition entière
	|	`p && p`	conjonction
	|	`p < p`	comparaison de deux entiers
	|	`if p then p else p`	condition

On appelle une telle définition une **grammaire**.

- **valeurs** (programmes complètement évalués) : `true`, `false`, `0`, `1`, `...`
- **programmes qui ne sont pas des valeurs** :

```
1 + 2          8 + false          if 2 < 8 then 12 + 3 else 18
1 < true       true + 1 < 2       if 2 + 3 then false else 4
```

nb : **certains de ces programmes n'ont pas de sens**...

Exemple : évaluation et erreurs

Comment caractériser un programme qui a du sens, et un programme qui n'a pas de sens ?

On **peut évaluer** (`if 2 < 8 then 12 + 3 else 18`) vers une valeur :

```
if 2 < 8 then 12 + 3 else 18
  ⟶  if true then 12 + 3 else 18
  ⟶  12 + 3
  ⟶  15
```

On **ne peut pas évaluer** (`if 2 + 3 then false else 4`) vers une valeur :

```
if 2 + 3 then false else 4
  ⟶  if 5 then false else 4
  ⟶  ??? on ne peut interpréter un entier
      comme condition booléenne...
```

Exemple : informations fournies par les types

Comparons ces deux expressions :

`if 2 < 8 then 12 + 3 else 18`	la condition s'évalue vers un booléen l'expression s'évalue vers un entier
`if 2 + 3 then false else 4`	la condition s'évalue vers un entier l'évaluation s'arrête

On peut **caractériser chaque construction du langage**:

- `+` s'applique à deux entiers et renvoie un entier
- `&&` s'applique à deux booléens et renvoie un booléen
- `<` s'applique à deux entiers et renvoie un booléen
- `if ... then ... else ...` requiert une condition booléenne

C'est l'intuition même des types

Exemple : types et évaluation

Considérons de nouveau `if 2 < 8 then 12 + 3 else 18` :

- 2 et 8 sont entiers
- donc `(2 < 8)` s'évalue correctement et produit un booléen
- 12 et 3 sont entiers
- donc `(12 + 3)` s'évalue correctement et produit un entier

Principe fondamental

De proche en proche, on peut montrer que chaque sous terme de `if 2 < 8 then 12 + 3 else 18` s'évalue et produit une valeur d'un type permettant la poursuite de l'évaluation

Ce raisonnement dépend seulement de la structure de l'expression, et est indépendant de la valeur produite

Pour `(if 2 + 3 then false else 4)`, **ce raisonnement échoue**

Les types : un mécanisme d'abstraction

Au premier cours, nous avons parlé de deux aspects de la notion d'abstraction en programmation :

- paramétrisation
- masquage d'informations destinées à rester locales

L'information de type permet de raisonner sur un (fragment de) programme sans en connaître tous les détails :

- afin de **s'assurer qu'il s'exécute correctement**
  ex : `p` s'exécute correctement et renvoie un entier
- afin de **synchroniser** un travail de programmation collaboratif
  ex : le programmeur 1 écrit `fact: int -> int`
  le programmeur 2 utilise `fact`
- afin de permettre au compilateur de **choisir la représentation des données**

Application I: la sûreté

Les types fournissent **une garantie**
à la fois sur **la bonne exécution d'un programme**
et sur **la forme du résultat**

Autrement dit certaines erreurs ne peuvent jamais se produire lorsqu'on exécute un programme bien typé :

- accès à un **champ indéfini** en Java
- **opérateur** appliqué à des valeurs pour lesquelles il n'est pas défini en Java
- idem en OCaml
  plus d'autres : **pas de pointeur nul**...

Abstraction : caractérisation des résultats par des types

Application II: le découpage du code

Les types permettent de **définir des interfaces**
entre **fragments de programmes**

En **Java** : classe, sous classe et code utilisateur peuvent être écrits par des programmeurs différents

```
class G {
  G add( G g )
  { ... }
  G inv( )
  { ... }
  G simp( Expr e )
  { ... }
}
```

```
class M extends G
    {
  M add( M x )
  { ... }
  M inv( )
  { ... }
}
```

```
class Main {
  // utilise
  //   M.add,
  //   M.inv,
  //   G.simp
  ...
}
```

OCaml : fournit des mécanismes similaires (**fonctions**, **modules**, **classes**)
les **types** permettent de **spécifier les interfaces**

Application III: la compilation

Les types fournissent parfois **des informations critiques sur la représentation des données pour compiler les programmes**

Exemple :

- **tableaux** et **records** (enregistrements) ont une représentation similaire...
- mais les **accès** doivent être traités de manières différentes
  pour un tableau, il faut calculer un index
  pour un record, il faut connaître l'offset d'un champ
- le typage statique permet de **résoudre** cela **statiquement**

Types statiques : OCaml, Java

Définition : typage statique

Les types **sont vérifiés avant exécution du programme**. On parle d'**inférence** lorsque le compilateur calcule les types.

- **Avantage** : les garanties fournies par les types sont vraies pour **toutes les exécutions** (pas besoin de tester ou de prouver quoi que ce soit). en effet, tous les calculs / vérifications ont lieu avant exécution, et en se fondant sur la sémantique du langage
- **Inconvénient** : un peu contraignant, au moins dans un premier temps

Exemples :

- **Java** : les annotations fournies sont vérifiées à la compilation
- **OCaml** : idem, de plus de nombreux types sont inférés...

Types dynamiques : JavaScript

Définition : typage dynamique

Le langage a une notion de types, mais les informations de type ne sont pas calculées avant exécution du programme

Essentiellement le contraire du typage statique :

- **Avantages** :
  plus permissifs (*mais est-ce toujours bien ?*) et plus légers (*au moins dans un premier temps*)
- **Inconvénients** :
  pas de garantie statique ; on peut rencontrer pour la première fois un bug très tard, on n'a aucune information à la compilation...

Exemples :

- **JavaScript** : typage **faible** et **dynamique**, pas de vérification statique des propriétés des objets... (pratique en programmation web)
- **Python** : typage **fort** mais dynamique

Outline

Programmes et valeurs

On va maintenant formaliser un peu tout cela :

- **définir évaluation et typage** de manière plus formelle
- **prouver** les **théorèmes principaux**, décrivant les intuitions vues jusqu'ici

Grammaire des valeurs :

`b`	::=	`true`	|	`false`		valeur booléennes
`n`	|	`0`	|	`1`	| `...`	valeur entières

Grammaire des programmes :

`p`	::=	`b` | `n`	valeur
	|	`p + p`	addition entière
	|	`p < p`	comparaison de deux entiers
	|	`if p then p else p`	condition

Évaluation des programmes

Définition : évaluation

On note `p0` $\rightarrow$ `p1` lorsqu'un pas de calcul transforme `p0` en `p1`

- `n0 + n1` $\rightarrow$ `n` où `n` décrit la somme des entiers `n0` et `n1`
- si `p0` $\rightarrow$ `p0'`, alors `p0 + p1` $\rightarrow$ `p0' + p1`
- si `p1` $\rightarrow$ `p1'`, alors `p0 + p1` $\rightarrow$ `p0 + p1'`
- `n0 < n1` $\rightarrow$ `true` si `n0` est plus petit que `n1`
- `n0 < n1` $\rightarrow$ `false` si `n0` est plus grand que `n1`, ou si `n0` est égal à `n1`
- si `p0` $\rightarrow$ `p0'`, alors `p0 < p1` $\rightarrow$ `p0' < p1`
- si `p1` $\rightarrow$ `p1'`, alors `p0 < p1` $\rightarrow$ `p0 < p1'`
- si `p0` $\rightarrow$ `p0'`,
  alors `if p0 then p1 else p2` $\rightarrow$ `if p0' then p1 else p2`
- `if true then p1 else p2` $\rightarrow$ `p1`
- `if false then p1 else p2` $\rightarrow$ `p2`
- note : pas de règle `p` $\rightarrow$ `p'` où `p` est une valeur

Types, typage des programmes

Notations :

- `p` **a le type** `t` se note $\vdash$ `p : t`
- **sous l'hypothèse** `H` **on déduit la conclusion** `C` se note :

$$\frac{\texttt{H}}{\texttt{C}}$$

Types : `int` et `bool` (pour l'instant...)

Règles de typage pour notre fragment simplifié :

$$\frac{}{\vdash \texttt{n} : \texttt{int}} \qquad \frac{}{\vdash \texttt{b} : \texttt{bool}}$$

$$\frac{\vdash \texttt{p0} : \texttt{int} \qquad \vdash \texttt{p1} : \texttt{int}}{\vdash \texttt{p0 + p1} : \texttt{int}} \qquad \frac{\vdash \texttt{p0} : \texttt{int} \qquad \vdash \texttt{p1} : \texttt{int}}{\vdash \texttt{p0 < p1} : \texttt{bool}}$$

$$\frac{\vdash \texttt{p0} : \texttt{bool} \qquad \vdash \texttt{p1} : \texttt{t} \qquad \vdash \texttt{p2} : \texttt{t} \qquad \texttt{t} \in \{\texttt{int}, \texttt{bool}\}}{\vdash \texttt{if p0 then p1 else p2} : \texttt{t}}$$

Typage d'un programme complet

On peut combiner ces règles pour former un **arbre de typage** (ou une **dérivation de type**) :

$$\dfrac{\dfrac{\dfrac{}{\vdash \texttt{2} : \texttt{int}} \quad \dfrac{}{\vdash \texttt{8} : \texttt{int}}}{\vdash \texttt{2 < 8} : \texttt{bool}} \quad \dfrac{}{\vdash \texttt{9} : \texttt{int}} \quad \dfrac{}{\vdash \texttt{7} : \texttt{int}}}{\vdash \texttt{if 2 < 8 then 9 else 7} : \texttt{int}}$$

Définition : programme typable

On dit qu'un programme `p` est typable, de type `t` si et seulement si il existe une dérivation, dont la conclusion est :

$$\vdash \texttt{p} : \texttt{t}$$

- OCaml essaye de construire un tel arbre
- lorsqu'il échoue, le message d'erreur (pas toujours simple...) pointe vers le premier échec, et donne des informations sur les types attendus

Évaluation d'un programme bien typé

Théorème : progression

Supposons que `p` est bien typé et de type `t`. Alors :

- soit `p` est une valeur (de type `t`)
- soit il existe `p'` tel que $\texttt{p} \rightarrow \texttt{p'}$

Intuition :

- si on peut typer un programme, alors soit il s'agit d'un calcul correctement terminé (valeur), soit le calcul peut se poursuivre
- autrement dit, pas de blocage...

Évaluation d'un programme bien typé

Théorème : progression

Supposons que `p` est bien typé et de type `t`. Alors :

- soit `p` est une valeur (de type `t`)
- soit il existe `p'` tel que `p` $\rightarrow$ `p'`

Preuve par induction sur la structure des programmes, et (longue mais simple) analyse de cas. Voyons quelques cas :

- cas où `p` est une valeur : trivial
- cas où `p` est de la forme `p0 + p1` :
  alors `p`, `p0`, `p1` sont typables de type `int` (d'après les règles)
  - si `p0` et `p1` sont des valeurs, alors il s'agit de valeurs entières et on peut calculer la somme `n`, donc `p0 + p1` $\rightarrow$ `n`
  - si `p0` n'est pas une valeur, on peut appliquer l'hypothèse de récurrence et il existe `p0'` tel que `p0` $\rightarrow$ `p0'`, donc `p0 + p1` $\rightarrow$ `p0' + p1`
  - si `p1` n'est pas une valeur : idem

Préservation des types

On a prouvé qu'un programme bien typé **peut faire UN pas de calcul**, mais **ça n'est pas suffisant**, puisqu'une exécution complète comprend en général de nombreux pas de calcul... Il faut donc un second théorème :

Théorème : auto-réduction

Supposons `p` bien typé de type `t` et supposons que `p` $\rightarrow$ `p'`.
Alors `p'` est aussi bien typé, de type `t`.

Intuition : l'exécution d'un programme préserve le type...

Préservation des types

On a prouvé qu'un programme bien typé **peut faire UN pas de calcul**, mais **ça n'est pas suffisant**, puisqu'une exécution complète comprend en général de nombreux pas de calcul... Il faut donc un second théorème :

Théorème : auto-réduction

Supposons `p` bien typé de type `t` et supposons que `p` $\rightarrow$ `p'`.
Alors `p'` est aussi bien typé, de type `t`.

Preuve par analyse de cas sur les transitions possibles :

- cas où `p` est une valeur : ne s'applique pas !
- cas où `p` est de la forme `p0 + p1` :
  alors `p`, `p0`, `p1` sont typables de type `int` (d'après les règles)
  - si `p0` et `p1` sont des valeurs entières, il en est de même pour le résultat du calcul
  - si `p0` n'est pas une valeur, on peut appliquer l'hypothèse de récurrence et il existe `p0'` aussi de type `int` tel que `p0` $\rightarrow$ `p0'`, donc `p0 + p1` $\rightarrow$ `p0' + p1`

Types et correction

Nous pouvons maintenant combiner les deux résultats :

"Well typed programs do not go wrong"

Supposons que `p` est bien typé et de type `t`. Alors :

- soit `p` produit une valeur de type `t` en un nombre fini d'étapes :

$$\texttt{p} \to \texttt{p'} \to \ldots \to \texttt{p'''} \to \texttt{v}$$

- soit `p` s'exécute perpétuellement

$$\texttt{p} \to \texttt{p'} \to \ldots \to \texttt{p'''} \to \ldots$$

Exercice :

- prouver ce théorème
- compléter les cas manquants pour les théorèmes de progression et d'auto-réduction

Types et correction

Nous pouvons maintenant combiner les deux résultats :

"Well typed programs do not go wrong"

Supposons que `p` est bien typé et de type `t`. Alors :

- soit `p` produit une valeur de type `t` en un nombre fini d'étapes :

$$\texttt{p} \to \texttt{p'} \to \ldots \to \texttt{p'''} \to \texttt{v}$$

- soit `p` s'exécute perpétuellement

$$\texttt{p} \to \texttt{p'} \to \ldots \to \texttt{p'''} \to \ldots$$

- `if 2 < 8 then 9 else 7` est typable, de type `int`, et peut s'évaluer
- mais `if 2 < 8 then 9 else true` ne peut être typé, et pourtant on pourrait l'évaluer
  le système de type est conservatif : certains programmes corrects sont rejetés...

Vers un fragment plus large : noms

On doit **étendre** le fragment vu précédemment :

`p`	::=	...	
	|	`x`	lecture de la valeur du nom `x`
	|	`let x = p0 in p1`	définition locale d'un nom `x`

Évaluation :

- il faut prendre en compte un environnement dans la relation d'évaluation, qui associe à chaque nom une valeur
- `let x = p0 in p1` évalue `p0`, étend l'environnement en associant la valeur obtenue à `x` puis évalue `p1` avec l'environnement étendu
- `x` récupère la valeur de `x` dans l'environnement

Nouvelles erreurs possibles :

- l'exécution serait bloquée si on devait évaluer `x` alors que `x` n'est pas présent dans l'environnement

Vers un fragment plus large : noms

On doit **étendre** le fragment vu précédemment :

`p`	::=	...	
	|	`x`	lecture de la valeur du nom `x`
	|	`let x = p0 in p1`	définition locale d'un nom `x`

Jugements de typage : on note `x0: t0, ..., xn: tn` $\vdash$ `p: t` pour dire "si `x0` a le type `t0`, ..., `xn` a le type `tn` alors `p` a le type `t`.

Règles de typage :

$$\frac{}{\texttt{x0} : \texttt{t0}, \ldots, \texttt{xn} : \texttt{tn} \vdash \texttt{xi} : \texttt{ti}}$$

$$\frac{\texttt{x0} : \texttt{t0}, \ldots, \texttt{xn} : \texttt{tn} \vdash \texttt{px} : \texttt{tx} \qquad \texttt{x0} : \texttt{t0}, \ldots, \texttt{xn} : \texttt{tn}, \texttt{x} : \texttt{tx} \vdash \texttt{p} : \texttt{t}}{\texttt{x0} : \texttt{t0}, \ldots, \texttt{xn} : \texttt{tn} \vdash \texttt{let x = px in p} : \texttt{t}}$$

On peut alors étendre les deux théorèmes :

- à nouveau, les programmes bien typés s'exécutent correctement...
- si un nom utilisé n'est pas défini, le programme est rejeté

Types et correction à l'exécution

Certaines erreurs demeurent possibles malgré le système de types :

- Java / OCaml : une **division par zéro** engendre une exception
- un **accès hors bornes d'un tableau** également
- on peut construire des systèmes de types pour capturer de telles erreurs, mais les systèmes ainsi obtenus ne sont pas pratiques

Les erreurs qui ne peuvent survenir pour les programmes bien typés dépendent du langage :

- OCaml : pas de pointeurs nuls, donc pas de `NullPointerException`
- Java : les `NullPointerException`, mais pas de pointeurs invalides
- C : un programme, même bien typé, peut lire / écrire dans un pointeur invalide (et une telle erreur peut être catastrophique)

Types dynamiques : aucune garantie statique !

Outline

Utilisation des types en OCaml

Déclaration :

- syntaxe : `type t = ...`
  par exemple : `type mon_type = int`
- complètement séparée de toute opération
  (en Java : coïncide avec la déclaration d'une classe)

Inférence :

- OCaml **infère** le plus souvent tous les types
  inutile de donner le type des noms, références ou arguments de fonctions !
  exemple :
  ```
  let f = fun x -> 8 + x in fun y -> f y < 100
  ```
- il est parfois utile de donner des contraintes de type de la forme
  ```
  let f: int -> int = fun (x: int) -> x + 2
  ```
  (dans cette expression, la contrainte est bien sûr inutile, nous verrons plus loin des cas de contraintes utiles)

Types de base et types construits

Types de base déjà introduits au premier amphi :

entiers	`int`	`53`
booléens	`bool`	`true`, `false`
flottants	`float`	`3.1415`
caractères	`char`	`'a'`
chaînes de caractères	`string`	`"abc"`
unité (une seule valeur)	`unit`	`( )`

Nous allons maintenant voir plusieurs possibilités pour **définir des types plus complexes** :

- **construction** :
  formation d'une valeur à partir de valeurs plus simples
- **dé-construction** (ou filtrage) :
  exploration de la structure d'une valeur ainsi construite

Types produits : *n*-uples

Principe : une forme de **conjonction**

- construire une valeur combinant une valeur de type `t0` **et** une valeur de type `t1`
- exemple : définition de points dans l'espace comme paires de coordonnées

Type paire : noté `t0 * t1` (généralisation `t0 * t1 * t2 * ... * tn`)

Construction : `(v0, v1)`

Déconstruction / filtrage :

- les fonctions `fst` et `snd` permettent de récupérer les coordonnées
- on peut aussi écrire `let (a, b) = paire in ...` pour déconstruire l'élément `paire`

Exemple :

```
let sum_vect (x0, x1) (y0, y1) = x0 + y0, x1 + y1 ;;
sum_vect (1, 0) (2, 3) ;;
```

Types produits : enregistrements

Il s'agit d'une forme d'uplet, **dont chaque champs a un nom**.

Déclaration de type :

```
type t = { c0: t0; ...; cn: tn }
```

Construction d'une valeur :

```
{ c0 = v0; ...; cn = vn }
```

Lecteur d'un champ : `enreg.ci`

Exemple :

```
type point = { x : float; y : float }
type disc  = { center : point; rayon : float }
let p = { x = 1.0; y = 2.0 }
let d0 = { center = p; rayon = 1.5 }
let d1 = { center = d0.center; rayon = 1.2 }
let d2 = { d1 with rayon = 0.9 }
```

Immutabilité / persistance

Paires / **tuples** et **enregistrements** sont des types **immutables** :

Définition : type / valeur immutable (ou persistant)

Une **valeur immutable** est une valeur qu'on peut lire mais pas modifier.

On **ne modifie pas une paire ou un enregistrement**, on en définit une nouvelle / un nouveau, et les composantes réutilisées sont partagées ! (encombrement mémoire réduit)
À l'inverse, les **tableaux** sont **mutables**.

Exception : on peut inclure des champs de type `ref` ou déclarer un champ mutable :

```
# type point = { x : float; mutable y : float } ;;
let p = { x = 1.2 ; y = -3.2 } in
p.y <- p.x ;
p.y;;
- float = 1.2
```

Types sommes : définition et constructeurs

Paires, uples, enregistrements, tableaux correspondent à des **conjonctions** : élément 1 **et** élément 2 **et** ...

On a parfois besoin d'une forme de **disjonction** :

- une fonction de **recherche dans une base de données** peut trouver un élément ou non
- une **liste** est soit vide, soit non vide

Déclaration de type :

```
type t =
  | C0 of t0
  | C1 of t1
  | ...
```

Construction d'une valeur : `Ci v` où `v` est de type `ti`

Types sommes : exemples

Objets géométriques :

```
type forme =
  | Carre of float * float * float
  | Rectangle of float * float * float * float
  | Disque_centre of float
```

Transformations géométriques :

```
type transformation =
  | Symetrie_x
  | Symetrie_y
  | Translation of float * float
```

Nous allons voir **bien d'autres exemples**...

Types sommes : déconstruction par filtrage

Le **filtrage** d'une valeur `v` s'effectue à l'aide d'une construction spécifique :

```
match v with
| C0 x0 -> expr0
| C1 x1 -> expr1
```

- `xi` est un "**motif**" correspondant au type associé à `Ci` (voir ci-dessous)
- `expri` fait référence aux variables du motif `xi`

Par exemple :

```
let transfo tr (x, y) =
  match tr with
  | Symetrie_x -> (x, -. y)
  | Symetrie_y -> (-. x, y)
  | Translation (vx, vy) -> (x +. vx, y +. vy)
```

Erreurs et typage : un match partiel risque d'engendrer une erreur "match failure" ; on notera que le système de types tolère mais émet un **warning**, que l'on **déconseille fortement d'ignorer**

Types sommes : le type "option"

Un exemple particulièrement utile :
on a souvent besoin de renvoyer **soit une valeur**, **soit rien du tout**...

Définition :

```
type int_option = None | Some of int
```

Quelques remarques :

- en Java, on simule souvent ce type avec des **pointeurs nuls**
  en cas d'oubli de tester qu'un pointeur est non nul :
  `NullPointerException`
- en OCaml, pas de pointeur nul, un type option (plus général que `int_option`, voir plus loin), et pas d'erreurs de pointeurs nuls !

On verra également plus loin un type `option` **plus général**

Types sommes récursifs

Une limite :

- tous les types que nous avons définis sont de **taille bornée** (à part les tableaux)
- on ne peut définir de **structure dynamique** :
  une **liste** est soit vide, soit non vide

Types récursifs : dans la définition d'un type somme `t`, `t` peut apparaître dans le type associé aux constructeurs.

Exemple :

```
type intlist =
  | Empty
  | Element of int * intlist
```

Types sommes récursifs : arbres

On considère les **arbres binaires de recherche avec éléments flottants**.

Définition :

```
type float_tree =
  | Leaf
  | Node of float_tree * float * float_tree
```

Quelques fonctions :

```
let rec height a =
  match a with
  | Leaf -> 0
  | Node (a0, _, a1) -> max (height a0) (height a1)
let rec mem f a =
  match a with
  | Leaf -> false
  | Node (a0, x, a1) ->
      x = f || f < x && mem f a0 || mem f a1
```

Remarque : en Java, on utilise un pointeur nul pour décrire Leaf

Types sommes récursifs : arbres syntaxiques

Une autre application :

- **représenter des programmes** (dans le langage de votre choix)
- les **compiler**, les **interpréter**...
- ou, pourquoi pas, **calculer leur type**...

Exemple : **le fragment discuté plus haut**

```
type valeur =
  | Val_bool of bool
  | Val_int of int
type expr =
  | Valeur of valeur
  | Addition of expr * expr
  | Inferieur of expr * expr
  | Condition of expr * expr * expr
```

Types polymorphes : vers des types plus généraux

Quelques types ont semblé **peu généraux** :

- `int_option` : type "option" spécifique aux entiers, alors que la notion d'option est utile pour tout type...
- `float_tree` : arbres binaires de recherche, dont les éléments sont des flottants, mais on pourrait vouloir la même chose avec des entiers...

Il faut donc un **mécanisme de généralisation** :

Définition : types polymorphes

Un type est dit **polymorphe** si il est paramétré par un autre type

Définition d'un type polymorphe : `type 'a t = ...`
où la définition peut faire intervenir `'a`, qui est une sorte de variable de type.
Penser à une **quantification universelle** : $\forall \alpha,\ \texttt{t}(\alpha)$

Instantiation : `t0 t` correspond au type `t`, avec `'a = t0`

Types polymorphes : exemples

Types sommes polymorphes :

```
type 'a option =
  | None
  | Some of 'a
type 'a bst =
  | Leaf
  | Node of 'a bst * 'a * 'a bst
```

Ce mécanisme est **parfaitement transparent** à l'utilisation, et OCaml infère des types polymorphes :

```
# let identity x = x
val identity : 'a -> 'a = <fun>
# let fst (x, y) = x
val fst : 'a * 'b -> 'a = <fun>
# let cswap b (x, y) = if b then (x, y) else (y, x)
val cswap : bool -> 'a * 'a -> 'a * 'a = <fun>
```

Types algébriques

Une vision **plus abstraite** des types :

Définition : types algébriques

Un **type algébrique** se définit par une équation de la forme $T = E$ où E est de la forme

- le **type unité**, noté 1 (une seule valeur)
- un **type de base** (entier, booléen, flottant...)
- le **type récursif** T
- un **type produit** $E_0 * E_1$
- un **type somme** $E_0 + E_1$
- un **type polymorphe** $\forall \alpha \cdot E$

Exemples :

- **listes d'entiers** : $L = 1 + \texttt{int} * L$
- **option polymorphe** : $O = \forall \alpha \cdot (1 + \alpha)$

Outline

Principaux éléments à retenir : types et correction

La pierre angulaire d'un **système de types** repose sur les résultats fondamentaux :

- **Progression** :
  un programme bien typé peut progresser, sauf s'il est déjà complètement évalué (exécution terminée)
- **Auto-réduction** :
  la progression des calculs préserve les types
  (autrement dit les types forment un **invariant du programme**)

Principaux éléments à retenir : construction, utilisation

Les constructions fondamentales pour **exprimer des structures variées** :

- **types de base** : entiers, flottants, booléens
- **types produits** : paires, tuples, enregistrements, tableaux...
- **types sommes**
- **récursivité** et **polymorphisme**

Du point de vue du langage :

- **inférence** (OCaml) : types calculés
- par opposition à une **vérification** (Java)
- **catégorie d'erreurs évitées**

Types et expressivité

En termes d'expressivité, **nous n'avons pas tout vu**, loin de là :

- **types algébriques généralisés (GADT)** :
  ou comment écrire une fonction dont l'argument est de type somme et dont la forme de la valeur de retour dépend de la structure de l'argument
  (présents en OCaml : vous pouvez vous y initier...)
- **types dépendants** :
  ou comment définir un type paramétré par une valeur (par exemple pour ajouter une information sur la longueur d'un tableau)

Qui dit plus forte expressivité signifie :

- plus de **propriétés exprimables et vérifiables** en regardant les types (statiquement !)
- un système de type plus complexe à utiliser, voir une **inférence indécidable**