

# Analyse sémantique-I

## Rôle de l'analyse sémantique:

**Entrée** : arbre de syntaxe abstraite issu de l'analyse syntaxique

**Sortie** :

- un arbre de syntaxe abstraite **décoré**: résultat de la **compréhension du programme**
- une erreur si le programme est **incorrect**

**Outils théoriques** : descriptions sémantiques

**Outils logiciels** : programmes récursifs sur les arbres de syntaxe abstraite

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
- 4 Autres analyses statiques

- Repérer les utilisations d'objets non déclarés.
- Relier les déclarations d'objets (variables, types, procédures, fonctions ... ) et leur utilisation.
- Une fois ce lien réalisé, le sauvegarder pour les autres phases.

- Des informations doivent être collectées pour chaque identificateur introduit:
  - Nature de l'objet: variable globale, locale, paramètre, nouveau type ...
  - Type/profil d'une variable ou fonction
  - Allocation en mémoire (emplacement, taille ...)
- Ces informations seront utiles à chaque utilisation de l'objet
- Création d'un lien direct vers les informations pour chaque utilisation

Plusieurs représentations possibles pour lier les utilisations et les déclarations :

- Chaque utilisation est un pointeur sur la déclaration
- Chaque utilisation est un nom unique ou bien un numéro faisant référence à une table annexe qui contient les informations sur les déclarations.
  - Possibilité d'utiliser différentes tables suivant l'étape de compilation.
- L'analyse de portée va permettre de construire cette nouvelle représentation de l'arbre.

- **Entrée** : l'arbre de syntaxe abstraite issu de l'analyse syntaxique
- **Sortie** : un nouvel arbre de syntaxe abstraite avec des identificateurs uniques et une table qui stocke des informations sur les variables (globale ou locale).
- **Erreur** : si une variable a été utilisée sans être déclarée ou bien en dehors de son **scope** (portée).
- **Technique** : utiliser une liste d'associations intermédiaire entre les noms visibles et les déclarations de l'environnement identifiées par un nom unique.

# Exemple sur le langage ARITHC

```
type ident = string
type binop = Sum | Diff | Prod | Quot
type expr =
    Cst of int
  | Var of ident
  | Op of expr * binop * expr
  | Letin of ident * expr * expr
type instr = Set of ident * expr | Print of expr
type prg = instr list
```

L'analyse syntaxique construit un objet de type `prg`.

L'analyse sémantique produira un nouvel objet de type `prg` + un tableau identifiant chaque déclaration.

# Exemple de programme

```
set x = 4
set xx = let y = 2 * x + 5 in - (y * y)
```

Arbre issu de l'analyse syntaxique:

```
[Set("x", Cst 4);
  Set("xx", Letin("y", Op(Op(Cst 2, Mult, Var "x"), Plus, Cst 5),
                  Op(Cst 0, Diff, Op(Var "y", Mult, Var "y"))))] ]
```

Après analyse de portée:

```
[Set("x_0", Cst 4);
  Set("xx_2",
      Letin("y_1", Op(Op(Cst 2, Mult, Var "x_0"), Plus, Cst 5),
              Op(Cst 0, Diff, Op(Var "y_1", Mult, Var "y_1"))))] ]
[("x_0", "x", Global); ("y_1", "y", Local); ("xx_2", "xx", Global)] ]
```

## Autre exemple

```
set x_0 = 4  
set x_2 = (let x_1 = 2 * x_0 in (x_1 * x_1)) + x_0  
set y_3 = x_2
```

```
[Set ("x_0", Cst 4);  
  Set ("x_2", Op(Letin("x_1", Op(Cst 2, Mult, Var "x_0"),  
                               Op(Var "x_1", Mult, Var "x_1")),  
               Plus, Var "x_0"));  
  Set ("y_3", Var "x_2")  
]  
[ ("x_0", "x", Global); ("x_1", "x", Local);  
  ("x_2", "x", Global); ("y_3", "y", Global) ]
```

# Règles de portées

$\rho \vdash e$  **ok** dans l'environnement où les variables de  $\rho$  sont visibles, l'expression  $e$  respecte les règles de portées.

$$\frac{\rho \vdash e_1 \text{ ok} \quad \rho \vdash e_2 \text{ ok}}{\rho \vdash \text{Op}(e_1, op, e_2) \text{ ok}} \qquad \frac{x \in \rho}{\rho \vdash \text{Var}(x) \text{ ok}}$$

$$\frac{\rho \vdash e_1 \text{ ok} \quad \rho + x \vdash e_2 \text{ ok}}{\rho \vdash \text{Letin}(x, e_1, e_2) \text{ ok}}$$

$$\frac{\rho \vdash e \text{ ok} \quad \rho + x \vdash p \text{ ok}}{\rho \vdash (\text{Set}(x, e) :: p) \text{ ok}}$$

$$\frac{\rho \vdash e \text{ ok} \quad \rho \vdash p \text{ ok}}{\rho \vdash (\text{Print}(e) :: p) \text{ ok}}$$

# Reconstruire l'ast

$\rho \vdash e \rightsquigarrow e', d$

- $e$  respecte les règles de portées dans l'environnement  $\rho$
- $e'$  ast correspondant avec des identifiants uniques pour les variables
- $d$  ensemble des déclarations: pour chaque variable  $x$ , le nom dans le programme original et la nature locale/globale.

$$\frac{\rho \vdash e \rightsquigarrow e', d_1 \quad \rho \vdash p \rightsquigarrow p', d_2}{\rho \vdash (\text{Print}(e) :: p) \rightsquigarrow (\text{Print}(e') :: p'), d_1 \cup d_2}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow e'_1, d_1 \quad \rho \vdash e_2 \rightsquigarrow e'_2, d_2}{\rho \vdash \text{Op}(e_1, \text{op}, e_2) \rightsquigarrow \text{Op}(e'_1, \text{op}, e'_2), d_1 \cup d_2}$$

$$\frac{(x \rightsquigarrow n) \in \rho}{\rho \vdash \text{Var}(x) \rightsquigarrow \text{Var}(n), \emptyset}$$

$$\frac{\rho \vdash e_1 \rightsquigarrow e'_1, d_1 \quad (n, d) = \text{new}(x, \text{Loc}) \quad \rho + (x \rightsquigarrow n) \vdash e_2 \rightsquigarrow e'_2, d_2}{\rho \vdash \text{Letin}(x, e_1, e_2) \rightsquigarrow \text{Letin}(n, e'_1, e'_2), \{d\} \cup d_1 \cup d_2}$$

$$\frac{\rho \vdash e \rightsquigarrow e', d_1 \quad (n, d) = \text{new}(x, \text{Glob}) \quad \rho + (x \rightsquigarrow n) \vdash p \rightsquigarrow p', d_2}{\rho \vdash (\text{Set}(x, e) :: p) \rightsquigarrow (\text{Set}(n, e') :: p'), \{d\} \cup d_1 \cup d_2}$$

Il faut bien distinguer:

- La **table des symboles** qui contient l'ensemble des déclarations locales et globales incluses dans le programme et qui fait partie de la représentation du programme
- L'**environnement local**  $\rho$  qui conserve les variables visibles en un point de programme et n'est plus utilisé après l'analyse de portée.

## Table des symboles

peut être globale

```
type annot = Glob | Loc  
(* stocke la déclaration dans la table ,  
   renvoie l'identifiant unique *)  
val add_var : ident -> annot -> ident
```

## Variables visibles

liste d'associations ou structure de **map fonctionnelle** qui associe un entier à une chaîne.

```
type t  
val add_vis : ident -> ident -> t -> t  
val mem_vis : ident -> t -> bool  
val find_vis : ident -> t -> int
```

# Fonctions d'analyses

```
let rec expr_scope vis = function
| Cte _ as x    -> x
| Op(e1,op,e2) -> let e'1 = expr_scope vis e1
                  and e'2 = expr_scope vis e2
                  in Op(e'1,op,e'2)
| Var x         -> if mem_vis x vis then Var(find_vis x vis)
                  else raise ScopeError
| Letin(x,e1,e2) -> let e'1 = expr_scope vis e1
                  in let n = add_var x Loc
                  in let e'2 = expr_scope (add_vis x n vis) e2
                  in Letin(n,e'1,e'2)

let rec prg_scope vis = function
[] -> []
| Print(e)::p -> let e' = expr_scope vis e
                  in Print(e')::prg_scope vis p
| Set(x,e)::p -> let e' = expr_scope vis e
                  in let n = add_var x Glob
                  in Set(n,e')::prg_scope (add_vis x n vis) p
```

- On peut aussi avoir deux constructeurs différents `GVar` et `LVar` pour distinguer les variables locales et globales.
- On peut introduire des **tables différentes** en fonction de la nature des déclarations (variables, fonctions, types, ...) si l'utilisation permet de choisir la bonne table.
- L'analyse de portée peut aussi être faite **en même temps que le typage**.
- On s'est placé dans le cadre où la portée est résolue de manière syntaxique. Dans le cas de surcharge, des informations de typage peuvent être nécessaires.

- 1 Analyse de portée
- 2 Typage
  - Principes du typage
  - Règles de typage
  - Vérification de type
  - Surcharge
- 3 Synthèse de type et polymorphisme
- 4 Autres analyses statiques

- 1 Analyse de portée
- 2 Typage
  - Principes du typage
  - Règles de typage
  - Vérification de type
  - Surcharge
- 3 Synthèse de type et polymorphisme
- 4 Autres analyses statiques

Diviser l'espace des valeurs en collections: un type est une collection de valeurs partageant une même propriété.

**Exemple:** représentation des flottants ou de structures chaînées.

Les types sont toujours plus ou moins vérifiés:

- soit à l'exécution (typage dynamique)
- soit à la compilation (typage statique)

- détection précoce d'erreurs;
- documentation, structuration (une première information sur le programme);
- élimination de certaines erreurs d'exécution; plus de sûreté;
- une manière générale de décrire des analyses de programmes (e.g., analyses d'exception, effets de bords, sécurité du code).

# Typage statique vs dynamique

- le typage dynamique est plus précis  
(`if B then 1 else 2+"a"`)  
avec `B` qui s'évalue en `true`
- mais les erreurs de type dépendent des valeurs à l'exécution  
(et sont difficiles à détecter)
- les langages typés **statiquement** peuvent être compilés plus efficacement  
(e.g., pas de test de représentation à l'exécution), facilite les optimisations (enregistrements, accès, représentations,...)
- il est possible d'utiliser une même représentation pour deux valeurs de types différents sans risque de confusion à l'exécution, e.g.:
  - représenter `true` et `false` par des entiers;
  - les constructeurs de valeur de ocaml par des entiers consécutifs.

# Propriété attendue du typage statique

le typage statique doit être **conservatif** :

- si le programme est bien typé statiquement alors il n'a pas d'erreur de type à l'exécution :

$$\frac{e : \tau \quad \text{le calcul de } e \text{ termine sans erreur}}{e \rightsquigarrow v \in [\tau]}$$

- Contre exemple:

```
union test {
    int champ1;
    float champ2;
};
main() {union test s;
        s.champ2 = 3.3e15;
        printf("%d\n", s.champ1);}
```

La confusion entier/adresse donne des **segmentation fault** à l'exécution

- Trouver des systèmes de types les plus riches possibles
- Des langages comme Java associent du typage statique et dynamique (le type dynamique fait partie de la valeur de l'objet)

## Dans ce cours

- On étudie ici les principes du typage **statique**.
- Comment formuler les règles de typage ?
- Principes des algorithmes pour les mettre en oeuvre.

## Qu'est qu'un type?

- Un type est une expression d'un langage de types
- Déclarer un type = introduire un nom de type  
**Exemple** : déclarations de **struct** ou **enum** . . .
- Définir un type = associer un nom à une exp. de type  
**Exemple** : utilisation de **typedef**.

## Typer une expression

- Typer = associer un type à une expression
- Typage **fort** = toute expression a un type **principal**

**Vérification des types** le programmeur associe un type à chaque déclaration d'identificateur et le compilateur vérifie la correction (e.g, C, C++, Pascal, Ada,...)

```
int addition (int x1, int x2)
  int temp;
  temp = x1 + x2;
  return(temp);
```

**Synthèse des types** les types sont devinés (on dira synthétisés ou inférés) par le compilateur par une analyse des contraintes d'utilisation des variables (e.g, Caml, SML,...)

```
let addition x1 x2 =
  let temp = x1 + x2 in temp
val addition : int -> int -> int = <fun>
```

- 1 Analyse de portée
- 2 Typage
  - Principes du typage
  - Règles de typage
  - Vérification de type
  - Surcharge
- 3 Synthèse de type et polymorphisme
- 4 Autres analyses statiques

La vérification de types se fait par rapport à des **règles de typage**.

“le programme  $P$  est bien typé de type  $ty$ ”

$$P : ty$$

## Vérification

- **vérifier** que  $ty$  est un type valide pour  $P$

## Synthèse

- **trouver** un  $ty$  valide

**Typer une expression:** parcourir l'expression en associant un type à chaque sous-expression

- des axiomes:

$$P : ty$$

- des règles d'inférence

$$\frac{P_1 : ty_1 \quad P_n : ty_n}{C(P_1, \dots, P_n) : ty}$$

- typer = construire un arbre où les feuilles sont des axiomes et les noeuds, des règles d'inférence
- Un programme est bien typé lorsque l'on peut construire une telle déduction

$$\frac{\begin{array}{c} \vdots \\ P_1 : ty_1 \end{array} \quad \begin{array}{c} \vdots \\ P_n : ty_n \end{array}}{C(P_1, \dots, P_n) : ty}$$

## Expressions arithmético-logiques

Axiomes:

`Cte(i) : int true : bool false : bool`

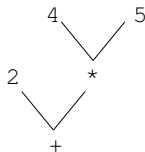
Une règle de déduction:

$$\frac{e_1 : \text{int} \quad e_2 : \text{int} \quad op \in \{\text{Plus}, \text{Mult}, \text{Div}, \text{Minus}\}}{\text{Op}(e_1, op, e_2) : \text{int}}$$

# Exemples

**Expression 1 :**  $2 + (4 * 5)$

$$\frac{2 : \text{int} \quad \frac{4 : \text{int} \quad 5 : \text{int}}{4 * 5 : \text{int}}}{2 + (4 * 5) : \text{int}}$$



**Expression 2 :**  $2 + (\text{true} * 5)$

$$\frac{2 : \text{int} \quad \frac{\text{true} : \text{bool} \quad 5 : \text{int}}{\text{true} * 5 : ?}}{2 + (\text{true} * 5) : ?}$$

**Question:** Que faire en présence d'identificateurs?

```
int x = 1;
int y;
y = 2 + (4 * x);
```

On introduit l'environnement de typage : *Env*.

- contient des liaisons (*idf*, *type*)
- $\langle (y : \text{int})(x : \text{int}) \rangle = \text{Env}$

“Dans l’environnement  $Env$ , le programme  $P$  a le type  $ty$ ”

$$Env \vdash P : ty$$

## L’environnement de typage

- liste d’association  $\langle (x_1 : ty_1), \dots, (x_n : ty_n) \rangle$
- une fonction d’accès  $acces$  tq:  
$$acces(x, \langle (x_1 : ty_1), \dots, (x_n : ty_n) \rangle)$$
$$= \begin{cases} ty_1 & \text{si } x = x_1 \\ acces(x, \langle (x_2 : ty_2), \dots, (x_n : ty_n) \rangle) & \text{sinon} \end{cases}$$
- L’analyse de portée faite préalablement assure l’unicité des identifiants.

## 1 Analyse de portée

## 2 Typage

- Principes du typage
- Règles de typage
- **Vérification de type**
- Surcharge

## 3 Synthèse de type et polymorphisme

## 4 Autres analyses statiques

- *Types de base*: `int`, `float`, `char`, `string`,...
- *Types construits*
  - constructeur de type (opérateur sur les types). e.g, `int[]`

```
type typ = Tbool | Tint | Tfloat | Tchar | Tstring ...  
         | Tarr of typ
```

On pourrait aussi avoir:

- des abréviations de type `Tdef of ident`  
des types définis par l'utilisateur `Tclass of ident`
- gestion de l'égalité, d'une table des symboles de types

```
type cte = Int of int | Float of float | Bool of bool ...
type primop = Sum | Diff | Prod | Quot | ...
type oper = Prim of primop | User of ident
type expr =
    Cst of cte
  | Var of ident
  | Op of oper * expr list
  | Letin of ident * expr * expr
type instr = Set of ident * expr | Print of expr
type prg = instr list
```

On associe à une fonction une **signature** qui spécifie

- le nombre d'arguments attendus
- le type de chaque argument
- le type de retour

Plusieurs sortes d'opérateurs:

- opérateurs primitifs (arithmétiques) : la signature est fixée
- opérateurs génériques (accès dans un tableau) : la signature va dépendre du type des arguments
- opérateurs définis par l'utilisateur : la signature est donnée par le programme.

Une signature peut se représenter par le type ocaml suivant :

```
type sign = typ list * typ
```

Chaque expression a un type unique qui peut être inféré.  
Une table associe à chaque déclaration son type.

```
val add_typ : ident → typ → unit  
val find_typ : ident → typ
```

# Algorithme de typage

```
let rec type_expr = function
  Cte c -> type_cte c
| Var x -> find_typ x
| Op(op,le) -> let (lt,t) = find_sign op in
                check_lexpr lt le; t
| Letln(x,e1,e2) -> let t1 = type_expr e1 in
                    add_typ x t1; type_expr e2

and check_lexpr = function
  [],[] -> ()
| t::lt,e::le -> if eqt (type_expr e) t then check_lexpr lt le
                 else raise TypeError
| _,_ -> raise TypeError

let rec type_prog = function
  [] -> ()
| Set(x,e)::p -> let t = type_expr e
                 in add_typ x t; type_prog p
| Print e::p -> let _ = type_expr e in type_prog p
```

# Opérateurs génériques

Certains opérateurs n'ont pas une signature unique:

- conditionnelle **if**  $b$  **then**  $e_1$  **else**  $e_2$  a pour type  $\tau$  si  $b : \text{bool}$  et  $e_1, e_2 : \tau$ .
- accès dans un tableau  $t[n] : \tau$  si  $t$  de type  $\tau \text{ array}$  et  $n : \text{int}$ .

On leur associe souvent des constructeurs explicites dans la syntaxe abstraite.

```
| If(b, e_1, e_2) ->
  let lb = type_expr b
  and t1 = type_expr e1 and t2 = type_expr e2
  in if eqt lb Tbool && eqt t1 t2 then t1
     else raise TypeError
| Gop(op, le) -> let lt = List.map type_expr le in
                 find_and_check_sign op lt
```

On verra que le typage polymorphe à la ocaml permet de traiter uniformément ce cas.

- Il est nécessaire de comparer les types suivant des règles qui dépendent du langage quand le système de type est complexe.  
Utilisation d'une fonction spécifique `eqt` et pas l'égalité structurelle.
- Notre algorithme utilise le fait que la portée a été préalablement résolue, sinon il faut introduire un environnement  $\rho$  qui contient les variables visibles.
- Il n'y a pas d'annotations de type dans le langage car les variables sont initialisées (leur type est calculé à partir du type de l'expression d'initialisation)

# Déclaration de fonctions

```
Def f (x:int,y:int) : bool = 2 * x < y
```

Le type de retour est nécessaire dans le cas de fonction récursive

Ajout dans le type des ast:

```
and instr = ...  
  Fun of ident * (ident * typ) list * typ * expr
```

**Exemple :**

```
Fun (f, [x, Tint; y, Tint], Tbool,  
      Op (Inf, [Op (Mult, [Cte (Int (2)) ; Var (x)]) ; Var y]))
```

```
prog_scope = ...
| Fun(f,lxt,t,e)::p ->
  let vis',lvt = List.fold_left
    (fun (x,t) (vis,lvt) ->
      let n = add_var x Par
      in (add_vis x n vis,(n,t)::lvt))
    (vis,[]) lxt
  in let sign = (List.map snd lxt,t)
  in let fn = add_fun f sign
  let e' = expr_scope (add f fn vis') e
  in Fun(fn,List.rev lvt,t,e')::prog_scope (add_vis f fn vis) p
```

Il faudrait aussi vérifier que deux paramètres n'ont pas le même nom.

```
type_prog = ...  
| Fun(f, lxt, t, e)::p ->  
  List.iter (fun (x, t) -> add_typ x t) lxt;  
  add_fun f (List.map snd lxt, t);  
  let t' = type_expr e in  
  if eqt t t' then type_prog p  
  else raise TypeError
```

1 Analyse de portée

2 Typage

- Principes du typage
- Règles de typage
- Vérification de type
- **Surcharge**

3 Synthèse de type et polymorphisme

4 Autres analyses statiques

- La portée n'est pas toujours résolue de manière statique.
- La surcharge permet d'utiliser le même nom pour des constructions différentes
  - opérations arithmétiques (entiers, flottants)
  - fonctions utilisateurs (Java, ada)
- La surcharge peut être résolue de manière sémantique: le contexte d'utilisation d'une expression permet de déterminer sans ambiguïté l'opération concernée.

- A l'issue de l'analyse syntaxique, une constante ou une fonction peut représenter des objets différents.
  - Une constante numérique pourra représenter un flottant ou un entier
  - Les opérations arithmétiques telles que  $+$  peuvent représenter l'addition sur les entiers ou sur les flottants et même des opérations mixtes (en introduisant des coercions).
- A l'issue de l'analyse sémantique on souhaite avoir résolu ces ambiguïtés : la référence à une opération correspond à un code machine précis.

## Problème

- $+$  :  $([int; int], int)$  et  $([float; float], float)$
- les constantes telles que 3, 4 ont les types `int` et `float`
- résoudre  $(1 + 4) + 3.2$

## Algorithme

- On procède des feuilles vers la racine en collectant les types possibles pour les expressions et les objets

$$\frac{1 : int, float \quad 4 : int, float \quad + : ([int; int], int), ([float; float], float)}{(1 + 4) : int, float}$$

$$\frac{(1 + 4) : int, float \quad 3.2 : float \quad + : ([float; float], float)}{(1 + 4) + 3.2 : float}$$

- S'il y a plus d'un type possible pour l'expression, la surcharge ne peut pas être résolue statiquement.
- Sinon il faut s'assurer qu'il y a un seul profil possible pour chaque sous-expression. Cela est fait lors d'une phase descendante:

$$\frac{(1 + 4) + 3.2 : \text{float}}{(1 + 4) : \text{float} \quad 3.2 : \text{float} \quad + : ([\text{float}; \text{float}], \text{float})}$$

$$\frac{(1 + 4) : \text{float}}{1 : \text{float} \quad 4 : \text{float} \quad + : ([\text{float}; \text{float}], \text{float})}$$

Les types ne sont pas uniques

- Dans Java, les types sont soit des types numériques, soit des tableaux, soit des classes.
- Les classes sont organisées suivant une notion d'héritage simple  $A \leq B$  si  $A$  est une sous-classe de  $B$ .
- Si  $A \leq B$  alors tout objet de type  $A$  est aussi de type  $B$ .
- On distingue le **type statique** déterminé à la compilation du **type dynamique** à l'exécution.
- Le type statique est une **approximation** du type dynamique de l'objet dans le sens que si  $e$  a pour type  $A$  alors toute exécution de  $e$  donne un objet dans un type  $B \leq A$ .
- Le type statique représente un ensemble de types possibles pour une expression.

- On ne regarde que le type des arguments mais on prend en compte le sous-typage entre classe.
- Une méthode  $e.m(e_1, \dots, e_n)$  avec  $e : A$  et  $e_i : A_i$ . Son profil est  $[A; A_1; \dots A_n]$ .
- On cherche toutes les méthodes de noms  $m$  définies dans les classes  $B$  dont  $A$  hérite et qui attendent  $n$  arguments. Cela nous donne une collection de profils:  $\{B; B_1; \dots B_n\}$
- Le profil  $B; B_1; \dots B_n$  est adapté à l'appel  $e.m(e_1, \dots, e_n)$  ssi  $A \leq B$  et  $A_i \leq B_i$ .  
Parmi tous les profils adaptés possibles on cherche s'il y en a un plus petit que les autres.

# Exemple

- Deux classes  $A \leq B$ ,  
une méthode  $m_B$  dans  $B$  avec des arguments  $B\ x; B\ y$   
deux méthodes  $m_{AB}$  et  $m_{BA}$  dans  $A$  avec des arguments  $A\ x; B\ y$  et  
 $B\ x; A\ y$
- Soit  $e.m(e_1, e_2)$  avec  $e : A$ 
  - Si  $e_1, e_2 : B$  une solution  $m_B$ .
  - Si  $e_1 : A\ e_2 : B$ , deux méthodes applicables  $m_{AB}$  et  $m_B$  et une seule meilleure  $m_{AB}$ .
  - Si  $e_1 : A\ e_2 : A$ , trois méthodes applicables  $m_{AB}$ ,  $m_{BA}$  et  $m_B$  mais deux incomparables  $m_{AB}$  et  $m_{BA}$  : **echec**.

# Analyse sémantique II

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
  - Rappels
  - Le cadre fonctionnel
  - Résolution des contraintes de type
  - Polymorphisme
- 4 Autres analyses statiques

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
  - **Rappels**
  - Le cadre fonctionnel
  - Résolution des contraintes de type
  - Polymorphisme
- 4 Autres analyses statiques

# Rappels : vérification de types

## Types et signatures:

```
type typ = Tbool | Tint | Tfloat | Tchar | Tstring ...  
          | Tarr of typ  
type sign = typ list * typ
```

## Expressions:

```
type cte = Int of int | Float float | Bool of bool ...  
type primop = Sum | Diff | Prod | Quot | ...  
type oper = Prim of primop | User of ident  
and expr =  
  Cst of cte  
  | Var of ident  
  | Op of oper * expr list  
  | Letin of ident * expr * expr
```

# Rappel : typage

## Instruction

```
type instr = Set of ident * expr
           | Print of expr
           | Fun of ident * (ident * typ) list * typ * expr
type prg = instr list
```

## Environnement de typage:

- associe des **types** à des variables et des **signatures** à des opérateurs

## Jugement de typage:

- un programme est correct dans un environnement
- une expression  $e$  a pour type  $\tau$  dans un environnement

# Algorithme de typage

- Etant donné un environnement et une expression: **calcule** son type.
- Etant donné un environnement et un programme vérifie qu'il est **bien formé**.

$$\frac{\rho + (x_i : \tau_i)_{i=1..n} + (f : [\tau_1; \dots; \tau_n], \tau) \vdash e : \tau \quad \rho + (f : [\tau_1; \dots; \tau_n], \tau) \vdash p \text{ ok}}{\rho \vdash (\text{Fun}(f, [x_1 : \tau_1; \dots; x_n : \tau_n], \tau, e) :: p) \text{ ok}}$$

Dans certains cas, il faut vérifier l'**égalité entre types**:

- application d'un opérateur: conditionnelle, fonction définie par l'utilisateur  
...
- déclaration d'une fonction (adéquation entre le type du corps et le type de retour déclaré)

- Ne pas écrire de types dans les déclarations de fonctions:

$$\frac{\rho + (x_i : \tau_i)_{i=1..n} + (f : [\tau_1; \dots; \tau_n], \tau) \vdash e : \tau \quad \rho + (f : [\tau_1; \dots; \tau_n], \tau) \vdash p \text{ ok}}{\rho \vdash (\text{Fun}(f, [x_1; \dots; x_n], e) :: p) \text{ ok}}$$

- Comment deviner  $\tau_i$  et  $\tau$  ?
- Idée : utiliser les contraintes d'égalité.
- Technique : introduire des variables de type.

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
  - Rappels
  - **Le cadre fonctionnel**
  - Résolution des contraintes de type
  - Polymorphisme
- 4 Autres analyses statiques

- Un langage dans lequel les fonctions sont des expressions comme les autres: **fun**  $x \rightarrow e$ 
  - Pas de distinction déclaration de variable/déclaration de fonction
  - Pas de distinction type/signature : type fonctionnel  $\tau_1 \rightarrow \tau_2$ .
- Fonction unaire:
  - **Def**  $f(x, y) = 2 * x < y$  devient  
**let**  $f = \mathbf{fun} x \rightarrow \mathbf{fun} y \rightarrow 2 * x < y$
  - La signature  $([int;int], bool)$  devient  
le type fonctionnel  $int \rightarrow int \rightarrow bool$   
associativité droite  $int \rightarrow (int \rightarrow bool)$ .
- Application binaire:  $f(4, 2)$  devient  $f\ 4\ 2$   
associativité gauche  $((f\ 4)\ 2)$
- pas de distinction entre expression et instruction

- On introduit des **opérateurs de type** pour les types de bases (entiers, chaînes, `unit` ...) ou les types paramétrés (tableaux, produit, ...) ou encore les types définis par l'utilisateur.
- Chaque opérateur de type a une arité (nombre de types pris en arguments)
  - type constant (`int`,...): arité 0
  - type (tableau, liste,...) : arité 1
  - type produit (`int*bool`) : arité 2

```
type typop = Tbool | Tint | Tfloat | Tunit ...  
          | Tprod | Tarr | Tdef of ident | ...
```

```
type typ = Tfun of typ * typ | Tconst of typop * typ list
```

# Description du langage

```
type primop = Sum | Diff | Prod | Quot | ...
type cte = Int of int | Float of float | Bool of bool ...
      | Oper of primop
and expr =
      Cst of cte
      | Var of ident
      | App of expr * expr
      | Fun of ident * expr
      | Letin of ident * expr * expr
type instr = Set of ident * expr | Print of expr
type prg = instr list
```

# Règles de typage-expressions

Constante

$$\frac{\text{type\_cte}(\mathbf{c}) = \tau}{\rho \vdash \text{Cst}(\mathbf{c}) : \tau}$$

Variable

$$\frac{(\mathbf{x} : \tau) \in \rho}{\rho \vdash \text{Var}(\mathbf{x}) : \tau}$$

Application de fonction

$$\frac{\rho \vdash \mathbf{f} : \tau' \rightarrow \tau \quad \rho \vdash \mathbf{e} : \tau'}{\rho \vdash \text{App}(\mathbf{f}, \mathbf{e}) : \tau}$$

Fonction

$$\frac{\rho + (\mathbf{x} : \tau') \vdash \mathbf{e} : \tau}{\rho \vdash \text{Fun}(\mathbf{x}, \mathbf{e}) : \tau' \rightarrow \tau}$$

Déclaration locale

$$\frac{\rho \vdash \mathbf{e}_1 : \tau_1 \quad \rho + (\mathbf{x} : \tau_1) \vdash \mathbf{e}_2 : \tau_2}{\rho \vdash \text{Letin}(\mathbf{x}, \mathbf{e}_1, \mathbf{e}_2) : \tau_2}$$

Expression

$$\frac{\rho \vdash e : \tau \quad \rho \vdash p \text{ ok}}{\rho \vdash (\text{Print}(e) :: p) \text{ ok}}$$

Déclaration

$$\frac{\rho \vdash e : \tau \quad \rho + (x : \tau) \vdash p \text{ ok}}{\rho \vdash (\text{Set}(x, e) :: p) \text{ ok}}$$

- Ajout de **variables de type**  $\alpha, \beta, \dots$  notées  $'a, 'b, \dots$  en ocaml.

```
type typ = Tfun of typ * typ
          | Tconst of typop * typ list
          | Tvar of ident
```

- Introduction de contraintes  $\alpha = \tau$ .

```
let rec type_prog = function  
  [] -> ()  
| Set(x,e)::p -> let t = type_expr e  
                  in add_typ x t; type_prog p  
| Print e::p -> let _ = type_expr e in type_prog p  
  
let rec type_expr = function  
  Cte c -> type_cte c  
| Var x -> find_typ x  
| LetIn(x,e1,e2) -> let t1 = type_expr e1 in  
                    add_typ x t1; type_expr e2
```

# Algorithme de typage : le cas des fonctions

| Fun (x, e) ->

- introduire une nouvelle variable de type  $\alpha$
- ajouter  $(x : \alpha)$  dans l'environnement
- typer le corps de la fonction e (type résultat  $\tau$ )
- collecter les contraintes sur  $\alpha$ 
  - Si une solution  $\alpha = \tau'$  renvoyer  $\tau' \rightarrow \tau$
  - Sinon erreur de typage

```
| App(f, e) -> let tyf = type_expr f
                and tye = type_expr e in
                (* vérifier que tyf de la forme Tfun(tye, tyr),
                 renvoyer tyr
                 *)
```

Appliquer l'algorithme de typage aux expressions:

**fun**  $x \rightarrow$  **fun**  $y \rightarrow 2 * x < y$

**fun**  $f \rightarrow f 2 + f 3$

**fun**  $f \rightarrow f 2 + f \text{ true}$

Dans des déclarations locales ou globales, possibilité d'introduire des objets récursifs:

**let rec**  $f = e$

**let rec**  $f = e_1$  in  $e_2$

Règles de typage:

$$\frac{\rho + (f : \tau) \vdash e_1 : \tau \quad \rho + (f : \tau) \vdash e_2 : \tau_2}{\rho \vdash \text{LetRIn}(f, e_1, e_2) : \tau_2}$$

Inférence:

- Introduire pour  $\tau$  une nouvelle variable de type et résoudre les contraintes.

En pratique dans Ocaml, les définitions récursives sont limitées aux définitions de fonctions **let rec**  $f\ x = e$  qui est la même chose que:

**let rec**  $f = \mathbf{fun}\ x \rightarrow e$

$$\frac{\rho + (f : \tau \rightarrow \tau')(x : \tau) \vdash e_1 : \tau' \quad \rho + (f : \tau \rightarrow \tau') \vdash e_2 : \tau_2}{\rho \vdash \text{LetRIn}(f, \mathbf{fun}\ x \rightarrow e_1, e_2) : \tau_2}$$

On introduit deux variable  $\alpha$  et  $\beta$  pour  $\tau$  et  $\tau'$ .

```
let rec fact1 = fun n ->  
  if n <= 0 then 1  
  else n * fact1 (n-1)
```

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
  - Rappels
  - Le cadre fonctionnel
  - **Résolution des contraintes de type**
  - Polymorphisme
- 4 Autres analyses statiques

# Contraintes de type

- Les types ont une structure de termes du premier ordre:

```
type fot = Var of ident | Term of ident * fot list
```

deux termes sont égaux s'ils sont structurellement égaux

- La résolution des contraintes de type est un problème d'**unification** du premier ordre:
  - Soient deux termes  $t$  et  $u$  avec des variables,
  - on cherche s'il existe une manière de remplacer les variables de  $t$  et de  $u$  par d'autres termes qui rend les termes résultant **égaux**.
- L'association de termes à des variables s'appelle une **substitution**
- Une substitution peut être appliquée à un terme pour obtenir un nouveau terme

```
let rec subst s = fun  
  Var x -> List.assoc x s  
  | Term(f, l) -> Term(f, List.map (subst s) l)
```

# Exemple

$\sigma$  =  $(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta \text{ list} \rightarrow \alpha$   
 $\mathbf{s}$  =  $\{\alpha \mapsto \text{bool}, \beta \mapsto \alpha \times \text{int}\}$   
 $\mathbf{s}(\sigma)$  =  $(\text{bool} \rightarrow \alpha \times \text{int} \rightarrow \text{bool})$   
 $\rightarrow \text{bool} \rightarrow (\alpha \times \text{int}) \text{ list} \rightarrow \text{bool}$

- Un terme avec variables représente un **ensemble de termes** : tous les termes obtenus par substitution
- Les termes peuvent être ordonnés  $t_2 \preceq t_1$  s'il existe une substitution  $s$  telle que  $t_2 = s(t_1)$ .  $t_2$  est alors moins général que  $t_1$ .

**Exercice** ordonner les types suivants:

- $\alpha \rightarrow \alpha$
- $(\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \beta$
- $\text{int} \rightarrow \text{int}$
- $\alpha \rightarrow \text{int}$

Soient deux termes  $t$  et  $u$

- On peut décider s'il existe une substitution  $s$  telle que  $s(t) = s(u)$
- Si  $s(t) = s(u)$  alors pour toute substitution  $s_0$  on a  $s_0 \circ s(t) = s_0 \circ s(u)$ .
- Si une substitution  $s$  telle que  $s(t) = s(u)$  existe, il en existe une **principale**.  
 $s$  est principale si pour toute substitution  $s'$ , si  $s'(t) = s'(u)$  alors il existe  $s_0$  tel que  $s' = s_0 \circ s$ .

## Unificateur principal

- `int et  $\alpha$`   
`{ $\alpha \mapsto \text{int}$ }`
- `int et bool`  
`Echec`
- `$\alpha$  et  $\beta$`   
`{ $\alpha \mapsto \beta$ }`
- `$\alpha$  et array`  
`{ $\alpha \mapsto \beta$  array}`
- `$\alpha$  et array`  
`Echec (pas de solution finie)`

# Algorithme d'unification

Algorithme naïf : on traite un ensemble d'équations entre termes, on renvoie la substitution résultat

```
let rec unif = fun  
  [] -> []  
  |(Var x, Var y)::E when x = y -> unif E  
  |(Var x, t)::_ when occur_term x t -> raise UnifError  
  |(Var x, t)::E -> (x,t)::unif (subst_eqs (x,t) E)  
  |(Term(f1, lt1), Term(f2, lt2))::_  
    when f1 <> f2 || List.length lt1 <> List.length lt2  
    -> raise UnifError  
  |(Term(f1, lt1), Term(f2, lt2))::E -> unif (List.combine lt1 lt2@E)
```

- critère de terminaison complexe
- autres algorithmes plus efficaces

- On se donne un terme  $t$  à typer
- On résoud les contraintes de type à l'aide d'unificateurs principaux
- On obtient un type qui peut contenir des variables
- C'est le type le plus général (tout autre type valide est une instance)

# Exemple 1

Typier **fun**  $n \rightarrow$  **fun**  $f \rightarrow$  **fun**  $x \rightarrow f (n f x)$

①  $n : \alpha, f : \beta, x : \gamma \vdash f : \beta_1 \rightarrow \beta_2$

Solution  $\{\beta = (\beta_1 \rightarrow \beta_2)\}$

②  $n : \alpha, f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash n f x : \beta_1$

①  $n : \alpha, f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash n f : \alpha_1 \rightarrow \beta_1$

①  $n : \alpha, f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash n : \alpha_2 \rightarrow (\alpha_1 \rightarrow \beta_1)$

Solution  $\{\alpha = (\alpha_2 \rightarrow (\alpha_1 \rightarrow \beta_1))\}$

②  $n : \alpha_2 \rightarrow (\alpha_1 \rightarrow \beta_1), f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash f : \alpha_2$

Solution  $\{\alpha_2 = (\beta_1 \rightarrow \beta_2)\}$

②  $n : (\beta_1 \rightarrow \beta_2) \rightarrow (\alpha_1 \rightarrow \beta_1), f : \beta_1 \rightarrow \beta_2, x : \gamma \vdash x : \alpha_1$

Solution  $\{\gamma = \alpha_1\}$

③  $n : (\beta_1 \rightarrow \beta_2) \rightarrow (\alpha_1 \rightarrow \beta_1), f : \beta_1 \rightarrow \beta_2, x : \alpha_1 \vdash f (n f x) : \beta_2$

# Exemple 2

L'expression **fun**  $x \rightarrow (x\ x)$  n'est pas typable.

$x : \alpha \vdash x\ x : ?$

①  $x : \alpha \vdash x : \beta_1 \rightarrow \beta_2$

Solution :  $\{\alpha = \beta_1 \rightarrow \beta_2\}$

②  $x : \beta_1 \rightarrow \beta_2 \vdash x : \beta_1$

Solution ?  $\beta_1 \rightarrow \beta_2 \simeq \beta_1$

Echec : pas de solution finie!

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
  - Rappels
  - Le cadre fonctionnel
  - Résolution des contraintes de type
  - Polymorphisme
- 4 Autres analyses statiques

- Certaines constantes sont polymorphes:
  - conditionnelle :  $\text{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$
  - accès dans un tableau  $\alpha \text{ array} \rightarrow \text{int} \rightarrow \alpha$
  - listes  $[] : \alpha \text{ list}$  et  $::$  de type  $\alpha \rightarrow \alpha \text{ list} \rightarrow \alpha \text{ list}$
  - ...
- A chaque utilisation d'une de ces constantes, on peut utiliser ce type avec des **variables fraîches**.

## Exemples

```
1 :: []  
true :: []  
1 :: true :: []
```

- Lors du typage, certaines variables de type restent indéterminées.

```
let singl x = x::[]
```

- On souhaite également les utiliser de manière polymorphe

```
singl 1;;  
singl true;;
```

- Intérêt du polymorphisme
  - généralité du code, concision
  - réutilisabilité (bibliothèques)

Les variables de type non déterminées sont généralisées.

$$\rho \vdash \mathbf{fun} \ x \rightarrow x : \forall \alpha. \alpha \rightarrow \alpha$$

**Schéma de type:** expression de type dont certaines variables sont quantifiées universellement en tête de l'expression:  $\forall \alpha. \alpha \rightarrow \alpha$ .

## Remarques

- $\alpha$  est une variable **liée**:  $\forall \alpha. \alpha \rightarrow \alpha$  représente le même schéma que  $\forall \beta. \beta \rightarrow \beta$ .
- Une variable qui n'est pas liée est dite **libre**.

**Type:** schéma de type sans quantificateur.

**Type monomorphe:** type ne contenant pas de variable de type.

**Type polymorphe:** schéma de type.

**Instancier un schéma de type:** remplacer les variables universellement quantifiées par une expression de type.

Un schéma de type représente un **ensemble de types** (toutes les instanciations possibles).

**Généraliser un type:** quantifier universellement les variables d'un type pour obtenir un schéma de type.

**Représentation possible:** un type avec variables + une liste de variables généralisées.

# Les règles de typage

- on associe un type (**sans quantificateurs**) à chaque expression
- Les constantes mais aussi les variables dans l'environnement sont associées à des **schéma de types**

$\forall \alpha_1 \dots \alpha_n. \text{typ}$

instancier chaque  $\alpha_i$  par une *nouvelle* variable de type.

On modifie la règle de typage des constantes/variables.

$$\frac{\text{type\_cte}(\mathbf{c}) = \forall \alpha_1, \dots, \alpha_n. \tau \quad \alpha_i \text{ fraîches}}{\rho \vdash \text{Cte}(\mathbf{c}) : \tau}$$

$$\frac{(x : \forall \alpha_1, \dots, \alpha_n. \tau) \in \rho \quad \alpha_i \text{ fraîches}}{\rho \vdash \text{Var}(x) : \tau}$$

## Exemple

```
let id = fun x  $\rightarrow$  x  
(id true, id 1)
```

$$\frac{\frac{(id : \forall \alpha. \alpha \rightarrow \alpha) \in Env}{Env \vdash id : bool \rightarrow bool} \quad Env \vdash true : bool}{Env \vdash id\ true : bool}$$

```
(fun f  $\rightarrow$  f true, f 1) (fun x  $\rightarrow$  x)
```

## Quand et comment généraliser ?

Au moment des déclarations locales et globales

```
let idf = exp
let idf = exp1 in exp2
```

On voudrait modifier un peu la règle de typage des déclarations.

$$\frac{\rho \vdash e_1 : \tau_1 \quad \rho + (x : \forall(\alpha_j)_{j=1..n} \tau_1) \vdash e_2 : \tau_2}{\rho \vdash \text{Let in}(x, e_1, e_2) : \tau_2}$$

```
fun x → let f y = [y] in f (f x)
```

```
fun x → let f y = [x;y] in f (f x)
```

- Seules les variables **non libres** dans l'environnement peuvent être généralisées.

## Traits impératifs : exemples

```
let counter = ref 0
val counter : int ref = {contents = 0}
let add_counter n =
  counter := !counter + n
val add_counter : int -> unit = <fun>
let afficher mess =
  print_string mess;
  print_newline ()
val afficher : string -> unit = <fun>
exception Division_par_zero
let division x y =
  if y = 0 then raise Division_par_zero
  else x / y}
val division : int -> int -> int = <fun>
```

# Traits impératifs : typage

**Référence** (*ref exp*)

$$\frac{Env \vdash exp : \tau}{Env \vdash \text{ref } exp : \tau \text{ ref}}$$

**Affectation** (*exp1 := exp2*)

$$\frac{Env \vdash exp1 : \tau \text{ ref} \quad Env \vdash exp2 : \tau}{Env \vdash exp1 := exp2 : \text{unit}}$$

**Séquence** (*exp1 ; exp2*)

$$\frac{Env \vdash exp1 : \tau_1; \quad Env \vdash exp2 : \tau_2}{Env \vdash exp1; exp2 : \tau_2}$$

**Exceptions** (*exception idf of typ*)

$$Env \vdash idf : typ \rightarrow \text{exn}$$

$$Env \vdash \text{raise} : \forall \alpha. \text{exn} \rightarrow \alpha$$

Problème dit des **références polymorphes**

Sans restriction, le programme suivant serait bien typé alors qu'il produit une erreur de type à l'exécution:

```
let lr = ref []  
let add x =  
  lr := x :: !lr;!lr
```

```
add 1; add 2;  
add true;;
```

# Généralisation et effets de bord

- Une expression est **expansive** si elle est de la forme  $e_1 e_2$ .
- Le type d'une expression expansive n'est pas généralisé.
- La généralisation du type des seules expressions *non expansives* garantit la correction du typage: "tout programme bien typé ne fait pas d'erreur de type à l'exécution".

```
let lr = ref []
val lr : '_a list ref = {contents = []}
let add x = lr := x :: !lr;!lr
val add : '_a -> '_a list = <fun>
add 1;;
- : int list = [1]
lr;;
- : int list ref = {contents = [1]}
add;;
- : int -> int list = <fun>
add true;;
```

This expression has **type** bool but is here used **with type** int

- compromis pour pouvoir définir des fonctions génériques sur les références sans casser le typage

```
let id = fun x -> x
val id : 'a -> 'a = <fun>
let f = id id
val f : '_a -> '_a = <fun>
let g = fun x -> (id id) x
val g : 'a -> 'a = <fun>
```

- d'autres solutions ont été proposées pour résoudre ce problème
- la généralisation des types des expressions non expansives est un **critère simple** pour l'utilisateur

# Extension du polymorphisme

Un schéma de type est un type.

Possibilité de généraliser le type des paramètres des fonctions:

$$(\forall \alpha. \tau) \rightarrow \sigma$$

$$\frac{x : \forall \alpha. \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

modifie profondément le système de type.

Ainsi on a :

$$\frac{x : \forall \alpha. \alpha \rightarrow \alpha \vdash x : (\beta \rightarrow \beta) \rightarrow \beta \rightarrow \beta \quad x : \forall \alpha. \alpha \rightarrow \alpha \vdash x : \beta \rightarrow \beta}{\vdash \mathbf{fun} \ x \rightarrow (x \ x) : \forall \beta (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta}$$

Système connu sous le nom de lambda-calcul du second ordre ou encore lambda-calcul polymorphe ou encore système F.

Il a été montré que l'inférence de type dans ce système n'était pas décidable.

- ML adopte un polymorphisme limité
- On ne généralise que les schémas des variables introduites dans les **let** (et pas les paramètres de fonctions)
- Il s'agit d'un polymorphisme **paramétrique**: le code est le même quelque soit le type manipulé.

- Déclaration du type des identificateurs/synthèse du type
- Typage fort: une expression a un type préservé par calcul
- Polymorphisme : description unique d'un ensemble de types
- Garanties apportées par le typage fort

# Autres analyses statiques

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
- 4 Autres analyses statiques
  - Vérifications élémentaires
  - Graphe de flots
  - Analyse de flot
  - Interprétation abstraite
  - Techniques de preuve
  - Conclusion

Le typage n'est qu'un exemple particulier d'analyse **statique** (effectuée à la compilation).

Il existe d'autres techniques d'analyse statique:

- Contraintes de bonne formation des programmes
- Analyse de flots
- Interprétation abstraite
- Preuve de programmes

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
- 4 **Autres analyses statiques**
  - **Vérifications élémentaires**
  - Graphe de flots
  - Analyse de flot
  - Interprétation abstraite
  - Techniques de preuve
  - Conclusion

## En C/Java

- les échappements **break** ou **continue** associés à des labels doivent se trouver sous des instructions de même label.
- les différents cas d'une instruction `switch` doivent être calculables à la compilation.
- Une variable "constante" ne peut être affectée.
- Les exceptions levées (**throws**) sont déclarées dans les profils de méthodes Java `throws`.
- ...

Vérifications qui ne peuvent pas être simplement faites à l'analyse syntaxique.

- détecter si une expression fait des **effets de bord**
- liens entre **affectations** et **utilisations** de variables
- recherche de **code mort**
- ...

# Exemple de propriété de programmes

Tester si un programme a un **break** ou **continue** hors d'une boucle **while**.

On peut définir une fonction `check_break` qui renvoie vrai si un `break` ou `continue` apparaît hors d'une boucle :

```
let rec check_break = function  
  While (cond, prog) -> check_break cond  
| Break | Continue   -> true  
| Op(_, le)          -> List.exists check_break le
```

Dans le cas de Java, les boucles peuvent être associées à des labels et les `break/continue` peuvent également être associés à des labels.

On peut définir une fonction `trans_break` qui transforme l'arbre pour associer chaque `break` ou `continue` au **while** correspondant.

- Chaque boucle est annotée par un label (qui peut être engendré à l'analyse syntaxique s'il n'est pas donné par l'utilisateur)
- La fonction prend en argument la liste des labels des boucles traversées.

```
let rec trans_break labs = function
  While(lab, cond, prog) ->
    let cond' = trans_break labs cond
    and prog' = trans_break (lab::labs) prog
    in While(lab, cond', prog')
| Break(None)    -> if labs = [] then raise BreakError
                   else Break(List.hd labs)
| Break(Some l) as x -> if List.mem l labs then x
                       else raise BreakError
| Continue      -> (* meme chose que BREAK *)
| Op(op, le)    -> Op(op, List.map (trans_break labs) le)
```

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
- 4 **Autres analyses statiques**
  - Vérifications élémentaires
  - **Graphe de flots**
  - Analyse de flot
  - Interprétation abstraite
  - Techniques de preuve
  - Conclusion

# Graphe de flot de contrôle

- Le graphe de flot de contrôle s'intéresse à l'enchaînement de l'exécution des instructions d'un programme.
- Représentation différente de l'arbre de syntaxe abstraite qui fait apparaître l'enchaînement des opérations.
- Dans une représentation arborescente  $Op(op, e_1, e_2)$  les deux expressions sont au même niveau.  
Le graphe de flots fera apparaître le calcul de  $e_1$  avant le calcul de  $e_2$ .
- La présence de conditionnelle introduit un branchement (2 suites possibles)
- La présence de boucle introduit des cycles

On perd les informations sur la structure de haut niveau du programme au bénéfice d'une représentation plus opérationnelle propice aux optimisations.

# Un langage simplifié

Pour le graphe de flots, on considère un langage simplifié (code à trois adresses):

Principes:

- ensemble infini de variables temporaires;  
 $a$ ,  $b$ ,  $c$  désignent des variables temporaires ou des constantes;
- Les instructions peuvent être associées à des labels  $L$ .

Instructions:

- $a = b \text{ op } c$
- $a = \text{op } b$
- $a = b$
- $a [ b ] = c$
- $a = b [ c ]$
- **goto**  $L$
- **iffalse**  $x$  **goto**  $L$
- **if**  $x \text{ relop } y$  **goto**  $L$

Un programme est une liste d'instructions.

- Les noeuds du graphe sont les points de programme étiquetés par les instructions.
- Une arête va de  $n$  à  $m$  s'il est possible de passer à l'instruction  $n$  juste après l'exécution de l'instruction  $m$ .
- On regroupe parfois les instructions qui s'exécutent de manière séquentielle sous forme de bloc.

# Exemple

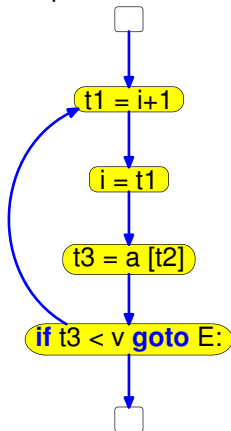
Programme initial:

**do**  $i = i+1$  ; **while** ( $a[i] < v$ )

Code 3 adresses:

E:  $t1 = i+1$   
 $i = t1$   
 $t2 = i$   
 $t3 = a[t2]$   
**if**  $t3 < v$  **goto** E:

Graphe de flot:



# Construction du graphe

Fonction qui prend en argument un noeud d'entrée, un noeud de sortie et un programme et qui construit le graphe (version naive).

Pour les expressions, le résultat est stocké dans une variable temporaire.

```
let rec flot_expr nin nout t = function
```

```
Binop(e1,op,e2) ->
```

```
    let t1 = new_temp () and t2 = new_temp()
```

```
    and nint = new_node () and newn = new_node ()
```

```
    in flot_expr nin nint t1 e1;
```

```
        flot_expr nint newn t2 e2;
```

```
        add_node newn "t=t1_op_t2";
```

```
        add_edge newn nout
```

```
| Var _ | Cte _ as x ->
```

```
    add_node nin "t=x"; add_edge nin nout
```

# Construction du graphe : instructions

```
let rec flot_instr nin out = function  
  Aff(x,e) -> let t = new_temp () and nint = new_node()  
              in flot_expr nin nint t e;  
              add_node nint "x=t";  
              add_edge nint nout  
|Seq(i1 ,i2) -> let nint = new_node() in  
              flot_instr nin nint i1;  
              flot_instr nint nout i2;
```

# Construction du graphe : instructions

```
| If(e,i1 ,i2) -> let t = new_temp () and nint1 = new_node()  
                and nint2 = new_node() and nint = new_node()  
                in flot_expr nin nint t;  
                in add_node nint "iffalse_t_goto_L";  
                add_edge nint nint1;  
                flot_instr nint1 nout i1;  
                add_edge nint nint2;  
                flot_instr nint2 nout i2  
| While(e, i) -> let t = new_temp () and nint = new_node()  
                and nint1 = new_node()  
                in flot_expr nin nint t;  
                add_node nint "iffalse_t_goto_L";  
                add_edge nint nout; add_edge nint nint1;  
                flot_instr nint1 nin i
```

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
- 4 **Autres analyses statiques**
  - Vérifications élémentaires
  - Graphe de flots
  - **Analyse de flot**
  - Interprétation abstraite
  - Techniques de preuve
  - Conclusion

À chaque point du graphe de flot on calcule certaines informations relatives au programme.

- Ces informations sont caractérisées par des équations locales.
- Un calcul de point fixe permet de gérer les boucles dans l'exécution.

- détecter du "code mort"  
morceaux de code qui ne seront jamais exécutés
- déterminer la durée de vie des variables  
(point de programme à partir duquel la variable n'est plus accédée et donc l'emplacement mémoire peut être récupéré)
- calculer les alias possibles d'un programme,  
variables qui peuvent pointer sur la même case mémoire  
(en cas de passage de variables par référence, de manipulation de pointeurs, de langage objet)

# Un théorème de point fixe

- Soit  $X$  un ensemble et  $F$  une fonction de  $X$  dans  $X$ .
- On cherche un **point-fixe**, c'est-à-dire  $p$  tel que  $F(p) = p$

## Remarques

- Une fonction arbitraire peut avoir 0 ou plusieurs points fixes
- Typiquement on aura  $X = X_1 \times \cdots \times X_n$  lorsque l'on a  $n$  inconnues.

# Théorème de Tarski

- On suppose que  $X$  est un ensemble (partiellement) ordonné
- Tout sous-ensemble  $Y \subseteq X$  a une **borne inférieure** ie il existe  $x_0 \in X$  tel que (plus grand des minorants)
  - ①  $\forall y \in Y, x_0 \leq y$
  - ② Si  $\forall y \in Y, x \leq y$  alors  $x \leq x_0$
- En particulier il existe un plus petit élément  $\perp$  qui est la borne inférieure de l'ensemble  $X$  et un plus grand élément  $\top$  qui est la borne inférieure de l'ensemble vide.

## Exemples

- $X$  est l'ensemble des parties d'un ensemble  $A$  avec l'ordre inclusion.
- Les booléens avec  $\text{false} \leq \text{true}$
- Ensemble  $D$  avec deux élément  $\perp$  et  $\text{top}$  et l'ordre  $\perp \leq d \leq \top$ .

**Théorème** Toute fonction  $F$  monotone sur  $X$  admet un plus petit point fixe.

- Par hypothèse si  $x \leq y$  alors  $F(x) \leq F(y)$
- Soit  $Y = \{y \mid F(y) \leq y\}$
- Soit  $p$  la borne inférieure de  $Y$ .
  - On montre :  $F(p) \leq p$   
Il suffit de montrer que  $\forall y \in Y, F(p) \leq y$  (borne-inf (2)). Or si  $y \in Y$ , alors  $F(y) \leq y$  (par déf de  $Y$ ) donc  $p \leq y$  (borne-inf 2) donc  $F(p) \leq F(y)$  (monotonie) et finalement  $F(p) \leq y$  (transitivité)
  - On montre  $p \leq F(p)$   
Il suffit de montrer que  $F(p) \in Y$  ie  $F(F(p)) \leq F(p)$  (borne-inf (1)). On utilise la monotonie et la preuve précédente de  $F(p) \leq p$ .
  - Si  $y$  est un point fixe alors  $F(y) \leq y$  dont  $p \leq y$  :  $p$  est le plus petit point fixe.

# Autre théorème de point fixe

- On suppose que  $X$  est un ensemble (partiellement) ordonné
- Toute suite croissante  $(x_n)_{n \in \mathbb{N}}$  a une **borne supérieure** ie il existe  $x \in X$  tel que (plus petit des majorants)
  - ①  $\forall n, x_n \leq x$
  - ② Si  $\forall n, x_n \leq y$  alors  $x \leq y$
- Il existe un plus petit élément  $\perp$ .
- On dit qu'une fonction monotone  $F$  est **continue** si pour toute suite croissante  $(x_n)_{n \in \mathbb{N}}$ , soit on a  $F(\mathbf{sup} (x_n)_n) = \mathbf{sup}(F(x_n))_n$

**Théorème de point fixe** Toute fonction croissante et continue admet un **plus petit point fixe**:  $p = \mathbf{sup} (F^n(\perp))_{n \in \mathbb{N}}$

- $\perp \leq F(\perp)$  dont  $F^n(\perp) \leq F^{n+1}(\perp)$
- $F(p) = \mathbf{sup} F^{n+1}(\perp) = \mathbf{sup} F^n(\perp) = p$
- si  $y$  est un point fixe alors  $\perp \leq y$  donc pour tout  $n$ ,  $F^n(\perp) \leq F^n(y) = y$  donc  $p \leq y$

Propriétés duales (en prenant l'ordre inverse):

- On suppose que toute partie  $Y$  de  $X$  admet une borne sup.  
La fonction croissante  $F$  a un plus grand point fixe qui est la borne sup de l'ensemble des  $y$  tels que  $y \leq F(y)$ .
- On suppose que toute suite décroissante a une borne inférieure et que  $X$  a un plus grand élément  $\top$ .  
La fonction croissante continue  $F$  a un plus grand point fixe qui est la borne inférieure de la suite  $F^n(\top)$ .

## Propriétés de grammaires

- On veut savoir si un non-terminal est productif (ie permet de dériver un mot formé de terminaux).
- Règle de grammaire  $X ::= a_1 \dots a_n$ .  
Cette règle est productive si tous les non-terminaux dans la partie droite sont productifs (conjonction).
- $X$  est productif s'il y a au moins une règle associée qui est productive (disjonction).

Exemple:

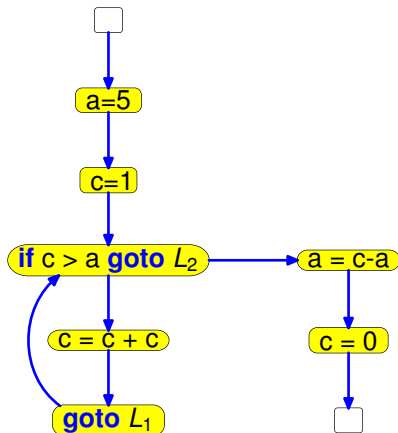
$$\begin{array}{l|l} S & ::= aAa \\ A & ::= bE \\ B & ::= CD \\ D & ::= d \end{array} \quad \begin{array}{l|l} S & ::= bBb \\ E & ::= \epsilon \\ C & ::= cC \end{array}$$

# Exemple sur les graphes de flots

- Calculer en chaque point de programme les affectations *actives* de la forme  $x = e$ , on les identifie par  $(x, n)$  si  $n$  est le point de programme.
- On calcule les affectations actives *avant* un point de programme  $n$  et les affectations actives *après*.
- On a *deux* équations:
  - ①  $\text{in}[n] = \bigcup_{p \in \text{pred}(n)} \text{out}[p]$ .
  - ②  $\text{out}[n] = (x, n) \cup (\text{in}[n] \setminus \{(x, k)\})$   
si l'instruction en  $n$  est de la forme  $x = e$
  - ③  $\text{out}[n] = \text{in}[n]$  sinon.

# Exemple

```
1:      a = 5
2:      c = 1
3:  L1  if c > a goto L2
4:      c = c + c
5:      goto L1
6:  L2: a = c - a
7:      c = 0
```



- De nombreuses techniques pour rendre l'analyse efficace (ordre de calcul des itérations)
- Nécessité de faire des analyses **inter-procédurales** (qui prennent en compte les appels de fonction et les passages de paramètres)

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
- 4 **Autres analyses statiques**
  - Vérifications élémentaires
  - Graphe de flots
  - Analyse de flot
  - **Interprétation abstraite**
  - Techniques de preuve
  - Conclusion

- Cette technique consiste à calculer une approximation finie d'un programme.
- L'idée est de simuler l'exécution du programme sur des données *abstraites*, plus simples que les entrées du programme et sur lesquelles on va pouvoir exécuter le programme.
- Une fois le calcul abstrait exécuté, on souhaite en déduire des informations sur le comportement du programme dans des cas concrets.

# Exemple : la preuve par 9

$$\begin{array}{r} 236 \\ \times 478 \\ \hline = 112808 \end{array}$$

**Problème** Comment détecter des erreurs ?

**Solution** Effectuer le même calcul modulo 9

Calculer rapidement :

- $236 \bmod 9 = 2 + 3 + 6 \bmod 9 = 2$
- $478 \bmod 9 = 4 + 7 + 8 \bmod 9 = 1$
- $236 \times 478 \bmod 9 = 236 \bmod 9 \times 478 \bmod 9 = 2$
- $112808 \bmod 9 = 1 + 1 + 2 + 8 + 8 \bmod 9 = 2$

Ne permet pas de prouver la correction mais détecte certaines erreurs.

- Ensemble  $A$  de données abstraites  
(exemple :  $[0, 8]$  )
- Ensemble  $C$  de données concrètes  
(exemple :  $\mathbb{N}$  )
- Relation entre ces données

$$c \sim a \text{ si et seulement si } c \bmod 9 = a$$

- Associer des opérations abstraites aux opérations concrètes

$$f_c : C \rightarrow C \rightarrow C \quad f_a : A \rightarrow A \rightarrow A$$

- Préservation de la relation :

$$\text{si } c_1 \sim a_1 \text{ et } c_2 \sim a_2 \text{ alors } f_c(c_1, c_2) \sim f_a(a_1, a_2)$$

Trouver les variables lues et écrites par un programme

- Ensemble concret : les instructions du langage
- Ensemble abstrait : couple  $(W, R)$  d'ensemble de variables écrites  $W$  et lues  $R$
- Relation :  $i \sim (W, R)$  l'exécution de l'instruction  $i$  ne peut accéder qu'aux variables de  $R$  et mettre à jour les variables de  $W$ .
- Interprétation des constructions :
  - si  $i_1 \sim (W_1, R_1)$  et  $i_2 \sim (W_2, R_2)$ 
    - $i_1; i_2 \sim (W_1 \cup W_2, R_1 \cup R_2)$
    - **if**  $(e)$   $i_1$  **else**  $i_2 \sim (W_1 \cup W_2, R_1 \cup R_2)$
- Fonctions mutuellement récursives : calcul de point fixe par itération

- Des bibliothèques pour des analyses d'intervalles
- Résultats approchés utiles pour la détection d'erreur
- Certaines analyses peuvent trouver toutes les erreurs, mais il y aura des faux positifs
- Techniques automatiques mais problèmes d'efficacité du calcul  
nombreuses techniques d'accélération pour le calcul itératif du point fixe.
- Des outils commerciaux utilisés dans le cas de systèmes critiques

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
- 4 **Autres analyses statiques**
  - Vérifications élémentaires
  - Graphe de flots
  - Analyse de flot
  - Interprétation abstraite
  - **Techniques de preuve**
  - Conclusion

Garantir que certaines erreurs ne se produiront pas.

Accès dans les bornes d'un tableau, déréférencement de pointeurs non-nuls

...

- Ajout de préconditions à certaines opérations.
- Calcul de plus faible préconditions (logique de Hoare).
- Génération d'obligations de preuve.
- Utilisation d'outils de démonstration.

# Exemple : JML

## Java Modeling Language

```
class Purse {
    //@ public invariant balance >= 0;
    int balance;

    /*@ public behavior
       @   requires s >= 0;
       @   modifiable balance;
       @   ensures s<=\old(balance) && balance==\old(balance)-s;
       @   signals (NoCreditException)
       @       s>\old(balance) && balance == \old(balance);
    @*/
    public void withdraw(int s) throws NoCreditException
        if (balance >= s) balance -= s;
        else throw new NoCreditException();
}
```

## Remarques

- La spécification documente le code.
- La spécification permet de garantir des propriétés fonctionnelles du code.
- Elle permet de dépasser les limites de l'analyse statique automatique (explicitation des invariants de boucle)
- La spécification peut être plus longue à écrire que le code (nécessité d'automatiser les annotations).

## Autres exemples

- La méthode B qui s'appuie sur des méthodes analogues a été utilisée pour spécifier et prouver des morceaux de code de Meteor.
- Outil Spec# développé par Microsoft research.
- Méthodes par model-checking qui produisent un modèle du code sous forme de système de transition qui est analysé.

- 1 Analyse de portée
- 2 Typage
- 3 Synthèse de type et polymorphisme
- 4 **Autres analyses statiques**
  - Vérifications élémentaires
  - Graphe de flots
  - Analyse de flot
  - Interprétation abstraite
  - Techniques de preuve
  - **Conclusion**

- Analyse élémentaire : visibilité, typage à la Pascal
- Typage qui influence le calcul : conversions de types entiers en C, surcharge à la Ada, méthodes Java . . .
- Typage pour documenter, structurer : aspects avancés de polymorphisme, structure des types
- Méthodes d'analyses statiques pour garantir des propriétés fines