# *Traduction et Sémantique*
# *Contrôle de type, typage*

Luigi Santocanale
LIF, Université de Provence
Marseille, FRANCE

22 avril 2010

## *Plan*



## *Plan*

Théorie des types et typage

Contrôle de type pour `lse`

Les types dans les langages de programmation
- Instructions, tuplet et fonctions
- Noms et équivalence des types
- Surcharge
- Fonctions polymorphes

Inférence de type et unification



## *Les origines : la théorie des types*

L'ensemble

$$\{ x \mid x \notin x \}$$

n'est pas bien typé.

Les fonctions

$$\Delta = \lambda x.x(x) \quad \text{et} \quad \Omega = \Delta(\Delta)$$

ne sont pas bien typées.

## Typage

Technique pour filtrer des objets syntaxiques
ayant des bonnes propriétés sémantiques.

Exemples :

- Une expression d'ensembles typable,
  n'entraîne pas de contradictions.
- Si une expression de fonction est typable,
  alors l'évaluation de telle fonction se termine.
- Un programme typable
  ne peut pas produire des erreurs de type à l'exécution.
- Sécurité (non-interférence) :
  une procédure typable
  ne produit aucune fuite d'information.



## Typage statique et dynamique

Certaines programmes ne peuvent pas se typer statiquement.
Exemple :

```
void f(int i){
  int tab[10];
  tab[i] = 0;
}

int main(int argc,char *argv[]){
  f(atoi(argv[1]));
  printf("Programme termine\n");
}
```

```
$ gcc tab_n.c
$ a.out 9
Programme termine
$ a.out 13
Programme termine
```

```
$ a.out 14
Programme termine
Erreur de segmentation
$ a.out 15
Erreur de segmentation
```



## Système de type

- Ensemble de règles d'inférences
  permettant d'inférer un jugement de type.
- Ensemble de contraintes sur les expressions.
  Si cet ensemble possède une solution unique,
  alors la solution d'une expression est son type.



## Plan

## Typage pour `lse`

Le jugement

$$E \vdash e : \tau$$

où $\tau \in \{\, bool, nat \,\}$, à lire :

dans l'environnement $E$ l'expression $e$ appartient au type $\tau$.

La bonne propriété sémantique :

$$\emptyset \vdash e : \tau \text{ implique } \exists v.\ \emptyset \vdash v : \tau \text{ et } e \Downarrow v,$$



## Rappel : environnement

- Environnement $E$ :
  suite finie de contraintes de type sur les variables :

  $$x_1 : \tau_1,\ x_2 : \tau_2,\ \ldots,\ x_n : \tau_n$$

  tel que $x_i \neq x_j$ si $i \neq j$.
- $\emptyset$ : environnement vide.
- Environnement $E[\tau/x]$ :
  même environnement que $E$,
  sauf pour $x$ qui a type $\tau$ dans $E[\tau/x]$.



## Le système de types pour `lse`

$$\frac{n \in \mathbb{N}}{E \vdash n : nat} \qquad \frac{v \in \{\, true, false \,\}}{E \vdash v : bool}$$

$$\frac{x \text{ a type } \tau \text{ dans } E}{E \vdash x : \tau} \qquad \frac{E \vdash e_1 : nat \quad E \vdash e_2 : nat}{E \vdash e_1 + e_2 : nat}$$

$$\frac{E \vdash e : bool \quad E \vdash b_1 : \tau \quad E \vdash b_2 : \tau}{E \vdash if\ e\ then\ b_1\ else\ b_2 : \tau}$$

$$\frac{E \vdash e : \tau' \quad E[\tau'/x] \vdash b : \tau}{E \vdash let\ x = e\ in\ b : \tau}$$



## Typage comme traduction dirigée par la syntaxe : les e_expressions de `lse`

| Production | Règle sémantique |
|---|---|
| $e \rightarrow$ `id` | $e.type :=$ *if* $e.env(\texttt{id}.lexeme)\downarrow$ *then* $e.env(\texttt{id}.lexeme)$ *else error* |
| $e \rightarrow$ `const` | $e.type :=$ *if* $\texttt{const}.lexeme \in \{\,\texttt{true},\texttt{false}\,\}$ *then bool else nat* |
| $e \rightarrow e_1$+$e_2$ | $e_1.env,\ e_2.env := e.env$<br>$e.type :=$ *if* $e_1.type = e_2.type = nat$ *then nat else error* |
| $e \rightarrow$ ( $e_1$ ) | $e_1.env := e.env$<br>$e.type := e_1.type$ |

## Typage comme traduction dirigée par la syntaxe : les b_expressions de `lse`

| Production | Règle sémantique |
|---|---|
| $start \rightarrow b$ | $b.env := \emptyset$ |
| $b \rightarrow e$ | $e.env := b.env$ |
| | $b.type := e.type$ |
| $b \rightarrow$ `if` $e$ `then` $b_1$ `else` $b_2$ | $e.env,\ b_1.env,\ b_2.env := b.env$ |
| | $b.type :=$ *if* $e.type = bool$ |
| | *and* $b_1.type = b_2.type$ |
| | *then* $b_1.type$ |
| | *else error* |
| $b \rightarrow$ `let id =` $e$ `in` $b_1$ | $e.env := b.env$ |
| | $b_1.env := b.env[e.type/x]$ |
| | $b.type := b_1.type$ |
| $b \rightarrow$ `(` $b_1$ `)` | $b_1.env := b.env$ |
| | $b.type := b_1.type$ |



## Implantation avec Bison

```
%union {
  char *str;
  Val val;
}

%%
S: {$<val>$.env=NULL;} b_expression {
  printf("Le type de cette expression est :\n\t");
  switch($2.type){
  case BOOL:
    puts("bool");
    break;
  case NAT:
    puts("nat"); break;
  default:
    puts("erreur : cette expression n'est pas typable"
  }
 }
;
```



```
b_expression:  e_expression {
   $$.type=$1.type;
  }
| LPAR {$<val>$.env=$<val>0.env;} b_expression RPAR {
   $$.type=$3.type;
  }
| LET ID EQ {$<val>$.env=$<val>0.env;} e_expression
   IN {
   //
   Pair *p=NEW(Pair);
   p->str=strdup($2);
   p->type=$5.type;
   //
   $<val>$.env=list_add(p,$<val>0.env);
  } b_expression {
    $$.type= $8.type;
  }
;
| IF {$<val>$.env=$<val>0.env;} e_expression THEN
{$<val>$.env=$<val>0.env;} b_expression ELSE {$<val>$.env=$<
b_expression {
   if(($3.type == BOOL) && ($6.type == $9.type))
     $$.type= $6.type;
   else
     $$.type= ERROR;
  }
;
```



```
e_expression: ID {
   Pair *p=NEW(Pair);
   List l;
   p->str=$1;
   l= list_find(p,$<val>0.env);
   if(l == NULL)
     $$.type=ERROR;
   else
     $$.type=l->content->type;
   free(p);
  }
| CONST {
   if((strcmp($1,"true") == 0) || (strcmp($1,"false")==0))
     $$.type=BOOL;
   else
     $$.type=NAT;
   }
|  e_expression PLUS
    {$<val>$.env=$<val>0.env;}  e_expression {
   if(($1.type == NAT) && ($4.type == NAT))
     $$.type = NAT;
   else
     $$.type = ERROR;
  }
| LPAR {$<val>$.env=$<val>0.env;} e_expression RPAR {
   $$.type=$3.type;
  }
;
```

## *Plan*

Théorie des types et typage

Contrôle de type pour `lse`

Les types dans les langages de programmation
- Instructions, tuplet et fonctions
- Noms et équivalence des types
- Surcharge
- Fonctions polymorphes

Inférence de type et unification



## *Expressions de type*

Dans les langages de programmation
les types ont d'habitude une structure plus complexe.

Par exemple :

- Types de base (ou atomiques) :

$$\texttt{bool}, \texttt{char}, \texttt{int}, \ldots$$

- Tableaux :

$$\texttt{tableau}\,[\,\texttt{nb}\,]\,\texttt{de}\ T$$

- Pointeurs :

$$\uparrow T$$

D'où une grammaire pour les types :

$$T \rightarrow \texttt{bool} \mid \texttt{char} \mid \texttt{int} \mid \texttt{tableau}\,[\,\texttt{nb}\,]\,\texttt{de}\ T \mid \uparrow T$$



## *Syntaxe abstraite pour les types*

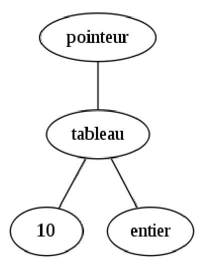
Par exemple le type

$$\uparrow \texttt{tableau}\,[\,10\,]\,\texttt{de int}$$

peut se representer par

$$\mathit{pointeur}(\mathit{tableau}(10, \mathit{entier}))$$

i.e. l'arbre :






## *Un autre langage (un peu moins simple)*

$$\begin{array}{l} P \rightarrow D\,;\,E \\ D \rightarrow D\,;\,D \mid \texttt{id} : T \\ T \rightarrow \texttt{bool} \mid \texttt{char} \mid \texttt{int} \mid \texttt{tableau}\,[\,\texttt{nb}\,]\,\texttt{de}\ T \mid \uparrow T \\ E \rightarrow \texttt{litt} \mid \texttt{id} \mid \texttt{nb} \mid E\ \mathit{mod}\ E \mid E\,[\,E\,] \mid E\uparrow \end{array}$$

Contrainte :
tout identificateur doit être déclaré avant être utilisé.

Remarques :

- il s'agit d'une règle contextuelle,
- on peut vérifier que cette règle est respectée
par une analyse sémantique.

## Contrôle de type comme analyse sémantique

| Production | Règle sémantique |
|---|---|
| $P \rightarrow D\,;\,E$ | |
| $D \rightarrow D\,;\,D$ | |
| $D \rightarrow \texttt{id} : T$ | $\mathit{AjouterType}(\texttt{id}.\mathit{entree}, T.\mathit{type})$ |
| $T \rightarrow \texttt{char}$ | $T.\mathit{type} := \mathit{caractère}$ |
| $T \rightarrow \texttt{int}$ | $T.\mathit{type} := \mathit{entier}$ |
| $T \rightarrow \texttt{tableau}\,[\,\texttt{nb}\,]\,\texttt{de}\ T$ | $T.\mathit{type} := \mathit{tableau}(\texttt{nb}.\mathit{val}, T_1.\mathit{type})$ |
| $T \rightarrow \uparrow T$ | $T.\mathit{type} := \mathit{pointeur}(T_1.\mathit{type})$ |



## Contrôle de type des expressions

| Production | Règle sémantique |
|---|---|
| $E \rightarrow \texttt{litt}$ | $E.\mathit{type} := \mathit{caractère}$ |
| $E \rightarrow \texttt{nb}$ | $E.\mathit{type} := \mathit{entier}$ |
| $E \rightarrow \texttt{id}$ | $E.\mathit{type} := \mathit{RechercherType}(\texttt{id}.\mathit{entree})$ |
| $E \rightarrow E_1\ \mathit{mod}\ E_2$ | $E.\mathit{type} := \mathit{si}$<br>$E_1.\mathit{type} = \mathit{entier}$ *et* $E_2.\mathit{type} = \mathit{entier}$<br>*alors entier sinon err_de_type* |
| $E \rightarrow E_1\,[\,E_2\,]$ | $E.\mathit{type} := \mathit{si}$<br>$E_2.\mathit{type} = \mathit{entier}$ *et* $E_1.\mathit{type} = \mathit{tableau}($ |
| | *alors* $t$ *sinon err_de_type* |
| $E \rightarrow E_1\uparrow$ | $E.\mathit{type} := \mathit{si}$<br>$E_1.\mathit{type} = \mathit{pointeur}(t)$<br>*alors* $t$ *sinon err_de_type* |



## D'autres constructions et d'autres types

- Les instructions et le type `vide` :

$$x := 31 : \texttt{vide}$$

- Les produits :

$$(x, y) : T_1 \times T_2$$

- Les structures :

$$\texttt{struct}\,\{\,\dots\ \texttt{champ}_\texttt{i} : \texttt{T}_\texttt{i};\ \dots\,\}$$

- Les fonctions :

$$\text{si}\quad T_1\ f\,(\,T_2\ x\,)\quad \text{alors}\quad f\ :\ \texttt{T}_2 \rightarrow \texttt{T}_1$$

- Les noms (ou définitions) ...



## Contrôle des instructions

| Production | Règle sémantique |
|---|---|
| $I \rightarrow \texttt{id} := E$ | $I.\mathit{type} := \mathit{si}$<br>$\texttt{id}.\mathit{type} = E.\mathit{type}$<br>*alors vide sinon err_de_type* |
| $I \rightarrow \texttt{si}\ E\ \texttt{alors}\ I_1$ | $I.\mathit{type} := \mathit{si}$<br>$E.\mathit{type} = \mathit{bool}$ *et* $I_1.\mathit{type} = \mathit{vide}$<br>*alors vide sinon err_de_type* |
| $I \rightarrow \texttt{tant que}\ E\ \texttt{faire}\ I_1$ | $I.\mathit{type} := \mathit{si}$<br>$E.\mathit{type} = \mathit{bool}$ *et* $I_1.\mathit{type} = \mathit{vide}$<br>*alors vide sinon err_de_type* |
| $I \rightarrow I_1\,;\,I_2$ | $I.\mathit{type} := \mathit{si}$<br>$I_1.\mathit{type} = \mathit{vide}$ *et* $I_2.\mathit{type} = \mathit{vide}$<br>*alors vide sinon err_de_type* |

## Contrôle des tuplets et fonctions

| Production | Règle sémantique |
|---|---|
| $E \rightarrow (E_1 , E_2)$ | $E.type := E_1.type \times E_2.type$ |
| $E \rightarrow E_1 ( E_2 )$ | $E.type :=$ *si* $E_1.type = T_d \rightarrow T_c$ *et* $E_2.type = T_d$ *alors* $T_c$ *sinon* *err_de_type* |



## Équivalence des expressions de type

Nous avons utilisé l'égalité = entre expressions de types.

Considérez la définition de type suivant :

```
type complexe = reel × reel
```

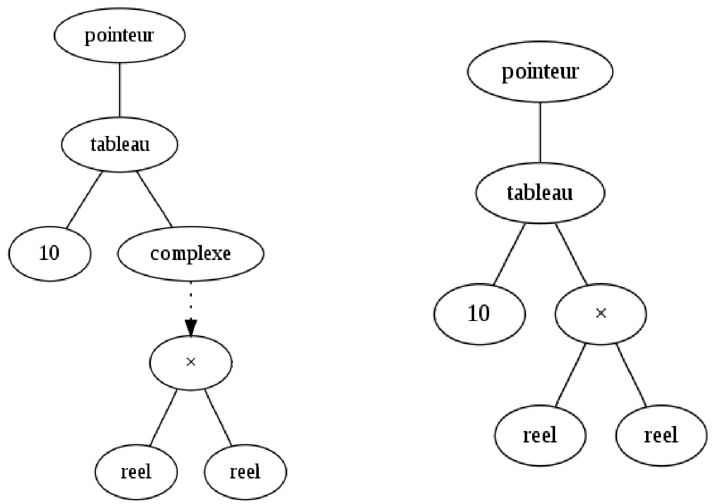
Est-ce que `complexe` est

- un nouveau type, ou
- un synonyme de `reel` × `reel` ?

Pascal ne répond pas à cette question.



## Doit-on considérer ces deux types égaux ?






## Équivalence structurelle et par nom

- équivalence structurelle :
  - la réponse est oui,
  - un type défini est un synonyme.
- équivalence par nom :
  - la réponse est non,
  - un type défini est un nouveau type.

## L'équivalence dans le langage C

Dans *C* :

- tout nom de type doit être défini avant être utilisé,
- on utilise l'équivalence structurelle :

```
typedef int entier;

int main (){
  int i = 0;
  entier j = i;
  return j;
}
```



## L'équivalence dans le langage C (II)

Dans *C* :

- . . . sauf pour les pointeurs à des structures,
- on utilise dans ce cas l'équivalence par nom.

Exercice :
quelles sont les définitions de type correctes ?

```
typedef struct cellule {
  int info;
  Cellule *suivant;
} Cellule;
```

```
typedef struct cellule {
  int info;
  struct cellule *suivant;
} Cellule;
```

```
struct cellule {
  int info;
  struct cellule *suivant;
};
typedef struct cellule Cellule;
```

```
struct cellule {
  int info;
  struct autre * suivant;
};
```



## Surcharge des opérateurs

Le type de $*$ peut à la fois avoir les types :

$$\texttt{int} \times \texttt{int} \to \texttt{int}$$
$$\texttt{float} \times \texttt{float} \to \texttt{float}$$

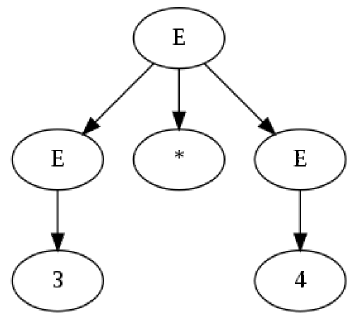
| Production | Règle sémantique |
|---|---|
| $E \to E_1 * E_2$ | $E.type :=$ *si* $E_1.type = E_2.type = entier$ *alors entier sinon si* $E_1.type = E_2.type = reel$ *alors reel sinon err_de_type* |



Cette solution peut n'est pas être suffisante : si

$$* \;:\; \texttt{int} \times \texttt{int} \to \texttt{float}$$

aussi, alors l'expression




possède plusieurs types.

# Fonctions polymorphes

Le fichier `lists.h` :

```
#ifndef LISTS_H
#define LISTS_H

#define LIST_CONTENT_TYPE int
#define LIST_CONTENT_EGAL(x,y) (x == y)

typedef struct lnode{
  LIST_CONTENT_TYPE *content;
  struct lnode *next;
} * List;

extern List
list_add(LIST_CONTENT_TYPE *, List);

extern int
list_length(List);

extern List
list_find(const LIST_CONTENT_TYPE *, List);

#endif /*  LISTS_H */
```



# Un exemple (I)

...est son implémentation `lists.c` :

```
#include <stdio.h>
#include <assert.h>
#include "lists.h"

#define NEW(type) (calloc(1,sizeof(type)))

List list_add(LIST_CONTENT_TYPE *c, List old){
  List new=NEW(struct lnode);
  assert(new != NULL);

  new->content=c;

  if(old == NULL)
    return new;

  new->next=old;

  return new;
}
```



# Un exemple (II)

```
int list_length(List l){
  int i;

  for(i = 0; l!= NULL; l=l->next,i++)
    ;
  return i;
}

List
list_find(const LIST_CONTENT_TYPE *c, List l){

  List start=l;

  for(;l!= NULL; l=l->next)
    {
      if(LIST_CONTENT_EGAL(c,l->content))
        return l;
    }

  return NULL;
}
```



# Fonctions polymorphes

- Les listes peuvent « porter » n'importe quel contenu ...
- ...d'habitude, il s'agit d'un contenu uniforme
  donc, du même type.
- Des fonctions sur les listes
  - ne dépendent pas du contenu, ou
  - peuvent se paramétrér sur le contenu.

  Il s'agit de fonctions polymorphes.

Le code précédent est une simulation du polymorphisme :

- à l'exécution, le même code `list_add`
  ne peut pas s'utiliser pour deux types différents.

On peut simuler le polymorphisme par

```c
#ifndef LISTS_H
#define LISTS_H

typedef struct lnode{
  void *content;
  struct lnode *next;
} * List;

extern List
list_add(void *, List);

extern int
list_length(List);

extern List
list_find(const void *, List);

#endif /*  LISTS_H */
```

Nous ne disposons plus du mécanisme de contrôle de type.



## Plan





## Polymorphisme et inférence de type

Dans des langages fonctionnels (tels que Caml)
nous disposons de définitions de types paramétrées :

```ocaml
type 'a list = Nil | Cons of 'a * 'a list
```

- Le type d'une expression peut contenir des variables.
- On parle d'inférence de types :
  le type le plus général est inféré.



## Un exemple

```ocaml
let x = 4 in Cons(3,Cons(x,Nil))
;;

let rec len l =
  match l with
      Nil -> 0
    | Cons(t,q) -> 1 + len q
;;

# let e = 4 in
  Cons(3,Cons(e,Nil));;
  - : int list = Cons (3, Cons (4, Nil))
# let rec len l =
  match l with
      Nil -> 0
    | Cons(t,q) -> 1 + len q;;
      val len : 'a list -> int = <fun>
```

## Inférence de types comme de solution de contraintes

$\alpha_e$ est le type (inconnu) de l'expression $e$ .

Les contraintes pour

```
let x = 4 in Cons(3,Cons(x,Nil))
```

sont engendrés dans un parcours de l'arbre abstrait :

$\alpha_{\texttt{let}\ x=4\ \texttt{in Cons}(3,\texttt{Cons}(x,\texttt{Nil}))} = \alpha_{\texttt{Cons}(3,\texttt{Cons}(x,\texttt{Nil}))}$

$\alpha_x = \alpha_4 , \alpha_4 = \texttt{int}$

$\alpha_{\texttt{Cons}(3,\texttt{Cons}(x,\texttt{Nil}))} = \texttt{list}(\alpha_3), \quad \texttt{list}(\alpha_3) = \alpha_{\texttt{Cons}(x,\texttt{Nil})}$

$\alpha_3 = \texttt{int}$

$\alpha_{\texttt{Cons}(x,\texttt{Nil})} = \texttt{list}(\alpha_x), \quad \texttt{list}(\alpha_x) = \alpha_{\texttt{Nil}}$

$\alpha_x = \beta$

$\alpha_{\texttt{Nil}} = \texttt{list}(\gamma)$



## Substitutions

Nous obtenons un système d'équations de la forme

$$\{ t_i = s_i \mid i \in I \}$$

où $t_i, s_i$ sont des expressions de type avec variables.
Substitution :
fonction $\sigma$ qui associe à chaque variable une expression.

Example

$$\alpha \to \texttt{int} \qquad\qquad \sigma(\alpha) = \texttt{int}.$$

Substitution dans un terme : notée $t[\sigma]$.

Example

$$\texttt{list}(\alpha)[\sigma] = \texttt{list}(\texttt{int})$$



## Unificateurs

Unificateur :
substitution $\sigma$ telle que

$t_i[\sigma]$ est le même terme de que $s_i[\sigma]$

pour tout $i$.

Example

$$\sigma(\alpha) = \texttt{int} \quad \sigma(\beta) = \texttt{char}$$

est un unificateur des expressions de type

$$\texttt{arbre}(\alpha, \texttt{char}) \qquad \texttt{arbre}(\texttt{int}, \beta)$$



## Le Théorème sur l'unification

La composition de substitution :

$$\sigma(\alpha) = \texttt{list}(\beta) \quad \tau(\beta) = \texttt{int}$$
$$(\tau \circ \sigma)(\alpha) = \texttt{list}(\texttt{int}).$$

Théorème. S'il existe un unificateur de

$$\{ \ t_i = s_i \ \mid i \in I \}$$

alors il existe un unificateur $\sigma$ le plus général :

$$\tau = \tau' \circ \sigma$$

pour tout autre unificateur $\tau$.