Analyse statique

Les tables des symboles sont lourdement utilisées dans la phase d'analyse statique dans le compilateur, qui comporte au moins:

- vérification de la déclaration et initialisation des identificateurs
- vérification des types

Déclaration des identificateurs

La plupart des langages modernes, dont OCaml, exigent qu'un identificateur soit déclaré avant son usage, et le compilateur se doit de vérifier cette contrainte.

Cela s'effectue par un simple parcours de l'AST[1]:

- à chaque déclaration d'identificateur, on l'ajoute à la table des symboles, pour toute sa portée
- à chaque utilisation d'identificateur, on vérifie s'il est bien défini dans la table des symboles, sans quoi, on envoie une erreur

Cette vérification est tellement simple, que l'on la fusionne souvent avec la phase de typage.

Initialisation des identificateurs

Certains langages, comme Ocaml, exigent aussi que l'identificateur soit initialisé au moment de la déclaration: on est obligés par la syntaxe à le faire:

```
let x = 33 in ...
```

Sans cette contrainte, detecter si un identificateur sera toujours utilisé après initialisation n'est pas possible en général:

Exemple: dans le programme Perl suivant

```
if ( test complexe) {$i=2} else {$j=3};
if ( autre test complexe) {print $i} else {print $j};
```

les identificateurs $i et $j sont toujours utilisés après initialisation seulement si *test complexe* et *autre test complexe* sont des programmes équivalents (ce qui n'est pas décidable en général).

Pourtant, certains langages largement utilisés, comme Perl et Python, permettent de n'initialiser ni déclarer un identificateur avant son usage, en lui associant une valeur par défaut dépendante du type du contexte.

Exemple (Perl):

Le programme

[1] Abstract Syntax Tree


```
#!/bin/perl
print "\$toto dans un contexte de type string : <", $toto, ">\n";
print "\$toto dans un contexte de type int :<", $toto+0, ">\n";
```

imprime (notez que la variable `$toto` n'est ni déclarée ni initialisée)

```
$toto dans un contexte de type string : <>
$toto dans un contexte de type int :<0>
```

Cela complique énormement la tache de la phase d'analyse statique, au point qu'il n'y a pas, à ce jour, d'analyseurs statiques satisfaisants pour ces langages.

Système de types

Chaque langage est équipé d'un système de type plus ou moins sophistiqué, dont le but est de structurer l'ensemble des objets qui peuvent être manipulés par un programme.

Un ensemble de *règles de typage*, qui peuvent être décrites informellement dans un manuel de référence, ou formalisès précisement dans un formalisme mathématique, indiquent, en ne regardant que le *type* des objets, quand il est licite de composer et/ou manipuler des objets.

Un programme qui ne viole aucune des règles de typage est dit *bien typé*.

Exemple:

En C, on peut former la somme d'un entier avec un flottant, et le résultat est un flottant, ainsi le fragment de programme C suivant est bien typé:

```
int i=0;
printf("%f\n", i+3.45)
```

En Ocaml, on ne peut former la somme d'un entier avec un flottant, ainsi le fragment de programme Ocaml suivant donne, au moment de la compilation, un erreur de type:

```
# 1+1.0;;
Characters 2-5:
  1+1.0;;
    ^^^
This expression has type float but is here used with type int
```

Typage statique, dynamique, etc.

À quel moment on vérifie que les règles de typage ne sont pas violés? On parle de

typage statique[2] si la vérification se fait au moment de la *compilation*

typage dynamique si la vérification se fait au moment de l'*exécution*

En fonction de la conception du langage, et de ses règles de typage, il peut être possible de vérifier *statiquement* plus ou moins de fragments de programmes. Exemple:

En Ocaml, toute expression du programme est typée statiquement.

En Perl, il est possible d'ecrire le programme suivant (il ne demande pas que les deux branches d'une conditionnelle aient le même type) :

```
if ( test complexe ) {$i=""} else {$i=3};
if ( autre test complexe ) {$j=$i."toto"} else {$j=$i+3};
```

mais sa correction (du point de vue des types) depend du résultat des deux test complexes, et leur équivalence est indécidable[3]. Perl s'en sort en appliquant automatiquement des conversions implicites entre tous les types de bases (`string` de et vers `int`, etc.).

Vérification des types

La vérification des types s'effectue aussi par le biais d'un parcours de l'AST:

- pour chaque noeud, on calcule le type des fils et on vérifie si le type du noeud est correct, par rapport aux règles du langage.
- pour cela on utilise un environnement qui donne le type des identificateurs de valeurs, et aussi (pour les langages avec types définis par l'utilisateur) un environnement qui donne le type associé à un identificateur de type.

Formalisation des règles de typage

Les règles de typage peuvent être décrites par un programme, ou dans un manuel, mais on peut faire mieux, notamment en donnant le *système de type* à l'aide de règles d'inférence, comme on en trouve pour décrire des systèmes logiques (ce n'est pas un hasard).
Dans cette formalisation, on écrit des assertions

$$\Gamma \vdash e : t$$

qui signifie *"dans l'environnement Γ l'expression e a type t"*.
Et on trouve des règles de la forme:

$$\frac{\Gamma \vdash e_1 : t_1 \quad \ldots \quad \Gamma \vdash e_n : t_n}{\Gamma \vdash f(e_1, \ldots, e_n) : t}$$

qui signifie "si les assertions en haut de la barre (les *prémisses*) sont établies, l'assertion en bas (la *conclusion*) l'est aussi".
Ou, lu a l'inverse, si on veut établir l'assertion en bas, on doit établir d'abord les assertions en haut.
Une règle sans prémisses s'appelle un *axiome*.

[3] Voir cours de calculabilité

Typage structurel ou génératif

Le langage CTigre dispose de types de bases, et des enregistrements (record) et tableaux (array) pour permettre à l'utlisateur de construire des types complèxes, et même mutuellement récursifs.

Dès que l'on dispose d'un système de type avec des types construits par l'utilisateur, se pose une question naturelle:

Quand doit-on considérer équivalents deux types?

On a deux choix:

équivalence structurelle on considère un type `t1` et un type `t2` équivalents si leur *structure* est identique. Sin on fait ce choix, le programme suivant est bien typé:

```
type  rectype = {name:string, age:int} in
let rec1:rectype := rectype {name="Nobody", age=1000}
in
  type rectype' = {name:string, age:int} in
  let rec2:rectype' := rectype' {name="Somebody", age=2000}
  in
  rec1 := rec2
```

Mais cela demande un effort considérable au compilateur, en particulier si on permet des types récursifs (on sait en décider l'équivalence, mais cela sort du cadre du cours de cette année)

types génératifs on considère *tous* les types distincts, même s'ils ont la même structure[4].
Cela signifie que chaque nouvelle définition d'un type utilisateur doit être distinguée de toutes les précédentes.

Pour obtenir cet effet, on associe à chaque définition de type une valeur unique (cela peut être un numero de série, en litérature on parle de "time-stamp"), qui permettra de le distinguer facilement et rapidement des autres.

On parle aussi de types *génératifs*, parce-que chaque déclaration de type produit (génére) une nouvelle valeur unique.

Le choix fait dans les langages

[4] Les langages qui font ça comportenten général une notion d'*abréviation*pour introduire des nouveaux nomspour le même type

Langage	types génératifs	Notes
Ocaml	oui	mais pas pour les modules
C/C++		oui pour structures, union, tableaux
		non pour le reste
Pascal	oui	sauf pour SET
Ada	oui	
Algol 68	non	le langage non génératif le plus complexe
Modula-3		génératif sur les types abstrait,
		structurel sur les types concrets

Soustypage

Si on dispose d'un enregistrement

```
salarié={name="Toto", age="23", sex = "F"}
```

et d'une fonction qui ne fait pas de discrimination sexuelle,

```
function paye (s:{name: string, age:int}):int
```

peut-on calculer le salaire? En C on ne peut pas le faire, mais dans d'autres langages, qui sont dotés de "sous-typage", on peut (Galileo, Modula-3, C++, Java, Ocaml), même si on doit utiliser des "objets" plutot que des enregistrements.

Le typage en présence de sous-typage/héritage et/ou polymorphisme est un sujet intéressant, mais qui sort du cadre du cours de cette année.

Le cas de CTigre: un module pour les types

Pour décrire les types de CTigre, on utilisera la définition suivante, qui oublie les détails syntaxiques. CTigre est génératif!
Le `unit ref` qui apparaît dans la définition de RECORD et ARRAY est le "time-stamp" unique utilisé pour garantir la générativité des types enregistrement et tableau... qui est caractéristique du langage.
Le constructeur ref produit en effet une case de mémoire différente à chaque fois (on aurait tout aussi bien pu mettre un `int` à la place, et s'assurer après que l'on met des entiers différents à chaque fois, mais la solution choisie ici est plus solide).

```
module Types =
struct
  type unique = unit ref
  type ty =
            RECORD of (Symbol.symbol * ty) list * unique
          | NIL
          | INT
          | STRING
          | ARRAY of ty * unique
          | NAME of Symbol.symbol * ty option ref
          | UNIT
end
```


Typage pour CTigre, sur l'AST: I

Expressions simples:

$$\Gamma_t;\ \Gamma_v, id : t \vdash VarExp(SimpleVar(id)) : t \quad (Var)$$

$$\Gamma_t;\ \Gamma_v \vdash NilExp : NIL \quad (Nil)^5$$

$$\Gamma_t;\ \Gamma_v \vdash IntExp : INT \quad (IntConst)$$

$$\Gamma_t;\ \Gamma_v \vdash StringExp : STRING \quad (StringConst)$$

Typage pour CTigre, sur l'AST: I

Expressions simples (suite et fin):

$$\frac{\Gamma_t;\ \Gamma_v \vdash VarExp(v) : RECORD([\ldots;(n,t);\ldots],\_)}{\Gamma_t;\ \Gamma_v \vdash VarExp(FieldVar(v,n)) : t}$$

$$\frac{\Gamma_t;\ \Gamma_v \vdash VarExp(v) : ARRAY(t,\_) \qquad \Gamma_t;\ \Gamma_v \vdash e : Int}{\Gamma_t;\ \Gamma_v \vdash VarExp(SubscriptVar(v,e)) : t}$$

Typage pour CTigre, sur l'AST: II

Expressions complexes:

$$\frac{\Gamma_t;\ \Gamma_v \vdash b : Bool \qquad \Gamma_t;\ \Gamma_v \vdash e_1 : t \qquad \Gamma_t;\ \Gamma_v \vdash e_2 : t}{\Gamma_t;\ \Gamma_v \vdash IfExp(b, e_1, e2) : t}$$

$$\frac{\Gamma_t;\ \Gamma_v \vdash b : Bool \qquad \Gamma_t;\ \Gamma_v \vdash e : t}{\Gamma_t;\ \Gamma_v \vdash WhileExp(b,e) : t}$$

$$\frac{\Gamma_t;\ \Gamma_v \vdash v : Int \qquad \Gamma_t;\ \Gamma_v \vdash l : Int \qquad \Gamma_t;\ \Gamma_v \vdash h : Int \qquad \Gamma_t;\ \Gamma_v \vdash e : t}{\Gamma_t;\ \Gamma_v \vdash ForExp(v,l,h,d,e) : t}$$

[5] Attention à l'usage de NIL.

Typage pour CTigre, sur l'AST: II

Expressions complexes (suite et fin):

$$\frac{\Gamma_t;\ \Gamma_v \vdash e_1 : t_1 \qquad \Gamma_t;\ \Gamma_v \vdash e_2 : t_2}{\Gamma_t;\ \Gamma_v \vdash SeqExp(e_1, e2) : t_2}$$

$$\frac{\Gamma_t;\ \Gamma_v \vdash v : t \qquad \Gamma_t;\ \Gamma_v \vdash e : t}{\Gamma_t;\ \Gamma_v \vdash AssignExp(v, e) : Unit}$$

$$\frac{\Gamma_t;\ \Gamma_v \vdash op : t_1 \to t_2 \to t^6 \qquad \Gamma_t;\ \Gamma_v \vdash e_1 : t_1 \qquad \Gamma_t;\ \Gamma_v \vdash e_2 : t_2}{\Gamma_t;\ \Gamma_v \vdash Opexp(op, e_1, e_2) : t}$$

Typage pour CTigre, sur l'AST: III

Enregistrements...

$$\frac{\Gamma_t;\ \Gamma_v \vdash e_1 : t_1 \quad \ldots \quad \Gamma_t;\ \Gamma_v \vdash e_k : t_k \qquad \{id \mapsto RECORD([id_1, t_1; \ldots; id_k, t_k])\} \in \Gamma_t}{\Gamma_t;\ \Gamma_v \vdash RecordExp([id_1, e_1; \ldots; id_k, e_k], id) : id}$$

La liaison $\{id \mapsto RECORD([id_1, t_1; \ldots; id_k, t_k])\}$ est introduite dans l'environnement Γ_t au moment de la *décaration* du type id.

Faite de même pour les tableaux.

Typage pour CTigre, sur l'AST: IV

Application de fonctions ...

$$\frac{\Gamma_t;\ \Gamma_v \vdash e_1 : t_1 \quad \ldots \quad \Gamma_t;\ \Gamma_v \vdash e_k : t_k \qquad \Gamma_t;\ \Gamma_v \vdash f : ([id_1, t_1; \ldots; id_k, t_k], t)}{\Gamma_t;\ \Gamma_v \vdash Apply(f, [e_1; \ldots; e_k]) : t}$$

Typage pour CTigre, sur l'AST: V

Déclarations des variables ...

$$\frac{\Gamma_t;\ \Gamma_v \vdash e_1 : t_1 \quad \ldots \quad \Gamma_t;\ \Gamma_v \vdash e_k : t_k \qquad \Gamma_t;\ \Gamma_v, id_1 : t_1, \ldots, id_k : t_k \vdash e : t}{\Gamma_t;\ \Gamma_v \vdash LetExp([id_1, e_1; \ldots; id_k, e_k], e) : t}$$

Typage pour CTigre, sur l'AST: VI

Déclarations des fonctions (cas d'une fonction) ...

$$\frac{\Gamma_t;\ \Gamma_v, id_1 : t_1, \ldots, id_k : t_k \vdash c : t_r \qquad \Gamma_t;\ \Gamma_v, (f : ([id_1, t_1; \ldots; id_k, t_k], t_r)) \vdash e : t}{\Gamma_t;\ \Gamma_v \vdash LetExp((f, [id_1, t_1; \ldots; id_k, t_k)], t_r, c), e) : t}$$

Typage pour CTigre, sur l'AST: VII

Déclarations des fonctions *recursives* (cas d'une fonction) ...

$$\frac{\Gamma_t;\ \Gamma_v, (f : ([id_1, t_1; \ldots; id_k, t_k], t_r)), id_1 : t_1, \ldots, id_k : t_k \vdash c : t_r \qquad \Gamma_t;\ \Gamma_v, (f : ([id_1, t_1; \ldots; id_k, t_k], t_r)) \vdash e : t}{\Gamma_t;\ \Gamma_v \vdash LetExp((f, [id_1, t_1; \ldots; id_k, t_k)], t_r, c), e) : t}$$

Typage pour CTigre, sur l'AST: abréviations de types

Tout ce que l'on a vu jusque là est parfait si on ne permet pas des abréviations de types, mais doit être modifié si on admets des déclarations (récursives ou non).

En effet, la version spécialisée pour la somme de la règle des opérateurs,

$$\frac{\Gamma_t;\ \Gamma_v \vdash e_1 : INT \qquad \Gamma_t;\ \Gamma_v \vdash e_2 : INT}{\Gamma_t;\ \Gamma_v \vdash Opexp(PLUS, e_1, e_2) : t}$$

n'est pas preparée à traiter le cas ou par exemple e_1 soit de type non pas INT, mais `a`, dans un programme comme

```
type a = int
in let f(x:a,y:a) = x+y
   in f(3,5)
```

Typage pour CTigre, sur l'AST: abréviations de types

Pour traiter les abréviations, il est nécéssaire de modifier toutes les règles, pour taverser les NAME() jusqu'au type sous-jacent.

Par exemple, la version spécialisée pour la somme de la règle des opérateurs deviendrait

$$\frac{\Gamma_t;\ \Gamma_v \vdash e_1 : t_1 \qquad actual\_ty(t_1) = INT \qquad \Gamma_t;\ \Gamma_v \vdash e_2 : t_2 \qquad actual\_ty(t_2) = INT}{\Gamma_t;\ \Gamma_v \vdash Opexp(PLUS, e_1, e_2) : t}$$

où la fonction `actual_ty` est définie comme suit

```
let rec actual_ty =
  function
      NAME(s,{contents=Some(t)}) -> actual_ty t
    |  t                         -> t;;
```

Typage pour CTigre, sur l'AST: et les types récursifs?

Si on écrit un programme comme

```
type tree ={key: int, children: treelist}
and  treelist = {hd: tree, tl: treelist}
and  intlist =  {hd: int, tl: intlist} in

let lis:intlist := intlist { hd=0, tl= nil } in
let a:tree = { . . . }

in  (lis.tl.tl.tl.hd)+(a.children.tl.tl.key)
```

on doit pouvoir determiner en phase d'analyse des types si `lis.tl.tl.tl.hd` et `a.children.tl.tl.key` sont *corrects*:

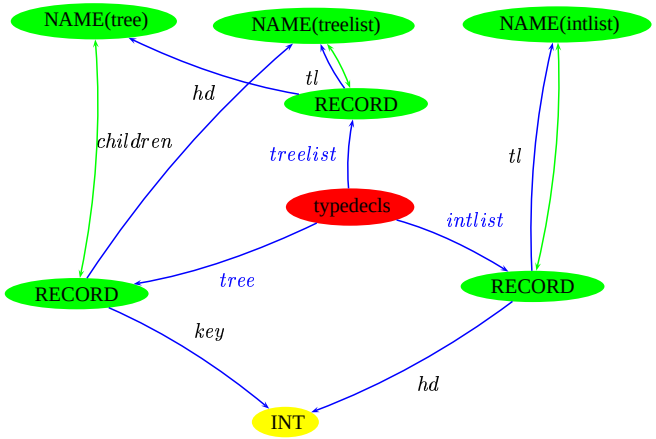
- `a` as-t-il un champ `children`?
- `a.children` as-t-il un champ `tl`?
- …

et quel est leur type.

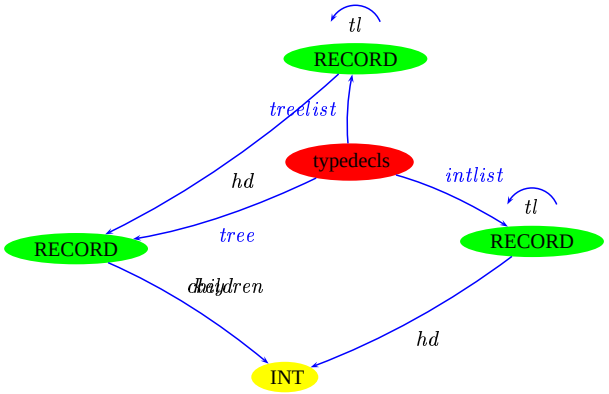
Compilation des types recursifs

Pour cela, on construit une réprésentation interne des types recursifs, qui convertit un type recursif en un graphe cyclique.
Determiner la correction d'une suite de selections et indices reviendra a parcourir le graphe sans rester bloques.
Voyons, sur cet exemple, comment la compilation est faite, et comment on verifie bien la correction.

Compilation des types recursifs



Compilation des types recursifs: sans NAME



Utilisation

Maintenant, pour determiner si `lis.tl.tl.tl.hd` et `a.children.tl.tl.key` sont bien corrects, il suffira de suivre le chemin sur le graphe, en court-circuitant les `NAME()`, comme pour les types declarés.
Donc, l'algorithme de typage, adapté aux déclarations, n'a besoin d'aucun changement pour traiter les types recursifs, une fois que l'on les a compilés comme graphes cycliques.

Possible réalisation en Ocaml

```
(* La representation interne des types pour l'analyse semantique *)
module Types :
  sig
    type unique = unit ref
    and ty = RECORD of (Symbol.symbol * ty) list * unique
           | NIL
           | INT
           | STRING
           | ARRAY of ty * unique
           | NAME of Symbol.symbol * ty option ref
           | UNIT
  end
(* Les environnements utilises par le verificateur de types *)
module Env :
  sig
    and enventry = VarEntry of Types.ty
    | FunEntry of Types.ty list * Types.ty
    val base_tenv : Types.ty Symbol.table
    val base_venv : enventry Symbol.table
  end
```

Possible réalisation en Ocaml

```
(* suit une chaine de definitions de types *)
(* jusqu'au type sous-jacent               *)
val actual_ty : Types.ty -> Types.ty
(* traduction de syntaxe abstraite a representation interne *)
(* d'un type non recursif                                   *)
val transTy :
  Types.ty Symbol.table -> Absyn.core_type -> Types.ty

(* Pour les types (mutuellement) recursifs, c'est plus dur: *)
(* on doit traiter *simultanement* toutes les declarations  *)
val transTyRecDecl :
  Types.ty Symbol.table ->
```



```
    (Symbol.symbol * Absyn.core_type) list ->
      (Symbol.symbol * Types.ty) list

(* Le verificateur de types. Attention, il renvoye un type *)
(* "actuel", pas un NAME(_,_)                              *)
val typecheck :
  Types.ty Symbol.table ->
    Env.enventry Symbol.table -> Absyn.exp -> Types.ty
```