Tables des symboles et portée des identificateurs

- Portée des identificateurs
- Tables de symboles: structures de données
  - approche impératif: hash et pile de undo
  - approche fonctionnel: arbres binaires équilibrés…
  - en Ocaml
- Typage

Portée statique des identificateurs: I

Dans les langages modernes, les identificateurs ont presque toujours une portée *statique* limitée au bloc dans lequel ils sont définis. En CTigre, (comme en Pascal, C++ ou ML, quoique avec une syntaxe différente), dans un programme contenant le fragment

```
let a := e1
in e2
```

l'identificateur `a` est lié à la valeur `e1` dans le corps de l'expression `e2`, et cette liaison (*binding* en anglais) est *détruite* dés que l'on sorte de `e2`.

Portée des identificateurs: II

Définition 1.1 (Liaison) *On notera dans la suite $x \mapsto v$ la* liaison *entre un identificateur x et un objet v.*

Définition 1.2 (Environnement) *On appelle environnement un ensemble de liaisons. Selon la nature des valeurs associées aux identificateurs, (constantes, cases mémoire, fonctions, types, etc.) on aura diverses environnements.*
Un environnement s'écrira $\{x_1 \mapsto v_1, \ldots x_n \mapsto v_n\}$.

Lors de la compilation on a besoin de connaître, quand on rencontre un usage d'un identificateur, quelle est la définition *active* pour cet usage.

Portée des identificateurs: III

Un même nom d'identificateur, défini à l'entrée d'un bloc, peut être redéfini dans un bloc plus interne, sans faire référence au même objet.
Exemple: dans le programme

```
0 type a = int in
1 let a := 3
2 in let f(x:a) =
3       let b :=
4          let a:= 'A'
5          in ord(a)+x
6       in print("Ord(A)+"); printint(x); print (" = ");
7          printint(b); print(" a vaut "); printint(a)
8    in f(a)
```

l'identificateur `a` défini comme une case mémoire entière contenant l'entier 3 à la ligne 1 est en principe visible dans toute l'expression entre les lignes 3 et 8, mais à la ligne 4 on trouve une redéfinition qui fait en sorte que la liaison entre `a` et 3 est *cachée* dans la ligne 5 par la liaison entre `a` et `'A'` définie à la ligne 4.
Cependant, la redéfinition de la ligne 1 ne cache pas la définition de type de la ligne 0!

Exemple

Suivons l'évolution de l'environnement des liaisons de type et de l'environnement qui donne le type des identificateurs sur le programme de l'exemple

Ligne	Prog	Tenv	TVEnv
0	`type a = int in`	$\emptyset$	$\emptyset$

`let a := 3`	$\{a \mapsto int\}$	$\emptyset$
`in let f(x:a) =`	$\{a \mapsto int\}$	$\{a \mapsto int\}$
`    let b :=`	$\{a \mapsto int\}$	$\{a \mapsto int, x \mapsto a\}$
`       let a:= 'A'`	$\{a \mapsto int\}$	$\{a \mapsto int, x \mapsto a\}$
`       in ord(a)+x`	$\{a \mapsto int\}$	$\{a \mapsto char, x \mapsto a\}$
`    in . . .`	$\{a \mapsto int\}$	$\{a \mapsto int, x \mapsto a, b \mapsto int\}$
`  in f(a)`	$\{a \mapsto int\}$	$\{a \mapsto int\}$

Portée des identificateurs: IV

À retenir pour les langages avec structures de blocs:

portée statique la portée d'une définition d'identificateur (variable, fonction, procédure, type), i.e. la partie du programme où la liaison pour cet identificateur créée par la définition est active, peut-être obtenu *statiquement* en analysant le texte du programme.

redéfinitions/hiding une liaison peut être cachée temporairement par une nouvelle liaison *de même nature* pour un même identificateur, mais seulement dans un bloc plus interne du bloc sur lequel porte la première définition.

LIFO l'ordre dans lequel les liaisons sont introduites est détruites est du gendre "Last In First Out", ce qui suggère une pile comme structure de donnée adaptée tant à la compilation qu'à l'exécution.

Portée des identificateurs: structures de données

On doit trouver des structures de données adaptées à la mise en oeuvre des environnements lors de la compilation.
Voilà nos *desiderata*:

- accès rapide aux liaisons par le nom de l'identificateur
- gestion facile de l'évolution des environnements, notamment de l'opération d'ajout d'une liaison et de suppression d'une liaison en restaurant l'(éventuelle) liaison précédente

Définition 1.3 *Somme d'environnements Si σ est un environnement, on écrit $\sigma + \{x \mapsto v\}$ pour l'environnement qui associe v à x, mais se comporte autrement comme l'environnement σ.*
On écrira $\sigma_1 + \sigma_2$ pour l'environnement qui contient les liaisons de σ_1 et σ_2, mais en donnant priorité à celles de σ_2.

En particulier, l'opération + ci-dessus n'est pas commutative.

Solution *impérative*

On utilise une table de hachage spécifique, avec une pile de *undo*.

- quand on rentre dans un bloc, on empile un marquer, puis on insère les définitions locales
- l'insertion se fait en ajoutant une valeur en tête de la liste correspondante à la clef dans la table, et en empilant la clef sur la pile de undo.
- à la sortie d'un bloc, on dépile toutes les clefs jusqu'au marqueur de bloc, et on les supprime
- la suppression s'effectue en enlevant l'élément en tête de liste.

Cela peut être mis en place dans un module qui fournit une primitive `insert` qui fait l̀a fois l'insertion en table de hachage et sur la pile de undo, et deux primitives `begin_scope` (met le marqueur sur la pile) et `end_scope` (dépile en enlevant de la table jusqu'au marqueur compris).

Solution *fonctionnelle pure*

On utilise des arbres binaires balancés, sans pile de *undo*.

- quand on rentre dans un bloc, on construit un nouvel environnement à partir du précédent, tandis que à la sortie d'un bloc, on abandonne le nouveau environnement et on retrouve l'ancien

```
let checktype tenv venv = function
  . . .
  | LetExp(VarDecl([x,v]),e) -> let newvenv=insert(venv,x,type(v))
                                in checktype tenv newvenv e
  | SeqExp(e1,e2)            -> checktype tenv venv e1;
                                checktype tenv venv e2
  . . .
```

Pour que cela soit efficace, il faut savoir construire l'objet modifié sans trop "copier".
Voir cours.

Typage

Un premier exemple d'utilisation des tables des symboles est la phase de typage dans le compilateur:

- on examine l'AST
- pour chaque noeud, on vérifie si le type est correct (et on le retourne aussi, voir item suivant!)
- pour cela, on doit souvent calculer le type des fils
- 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

$$\Gamma \vdash e : t$$

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

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

ce qui signifie que 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'invers, 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*.

Un module pour les types

Pour décrire les types de CTigre, on utilisera la définition suivante, qui oublie les détails syntaxiques.

Le `unit ref` qui apparaît dans la définition de RECORD et ARRAY est utilisé pour garantir l'unicité des types enregistrement et tableau… qui est caractéristique du langage.
Le constructeur ref produit en effet une case de mémoire différente à chaque fois.

```
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:

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

$$\Gamma_t;\ \Gamma_v \vdash NilExp : NIL \quad (Nil)^1$$

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

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

[1] Attention à l'usage de NIL.

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:

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

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

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

Typage pour CTigre, sur l’AST: II

Expressions complexes (suite et fin):

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

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

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

Typage pour CTigre, sur l'AST: III

Enregistrements…

$$\frac{\begin{array}{c}\Gamma_t;\ \Gamma_v \vdash e_1 : t_1 \quad \ldots \quad \Gamma_t;\ \Gamma_v \vdash e_k : t_k \\ \Gamma_t;\ \Gamma_v \vdash id = RECORD([id_1, t_1; \ldots ; id_k, t_k])\end{array}}{\Gamma_t;\ \Gamma_v \vdash RecordExp([id_1, e_1; \ldots ; id_k, e_k], id) : id}$$

Faite de même pour les tableaux.

Typage pour CTigre, sur l'AST: IV

Application de fonctions …

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

Typage pour CTigre, sur l'AST: V

Déclarations des variables …

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

Typage pour CTigre, sur l'AST: VI

Déclarations des fonctions (cas d'une fonction) …

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

Typage pour CTigre, sur l'AST: VII

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

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

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

Tout ce que l'on a vu jusque là est parfait si on ne permet pas de déclarer des 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: déclarations de types

Pour traiter les types déclarés, 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

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

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?

Voir cours.