Université Paris 6
Licence d'informatique

Module de Programmation
ANNÉE 2000-2001

Travaux Pratiques n° 3
SYNTHÈSE DE TYPES POUR UN PETIT LANGAGE FONCTIONNEL

Le but du TP est d'implanter un algorithme de synthèse de types pour un petit langage fonctionnel dont la syntaxe est très proche de celle de Caml Light. Les règles de typage de ce langage sont celles qui ont été vues en cours.

1 Définition syntaxique du langage

Un programme est une suite de déclarations de la forme :

```
<decl> :: = def <nom> = <expr>;;
```

où `<expr>` désigne n'importe quelle expression. Un `<nom>` est une séquence ininterrompue de lettres de l'alphabet. Les expressions sont définies par la grammaire suivante au format BNF (Bachus Naur Form) :

```
<expr> ::= si <expr> alors <expr> sinon <expr>
        |  soit <nom> = <expr> dans <expr>
        |  fonction <nom> -> <expr>
        |  rec <nom> <nom> = <expr>
        |  <expr> <expr>
        | ( <expr> )
        | <nom>
        | i
        | vrai | faux
        | <expr> <op> <expr>
<op>   ::= + | - | * | / | < | > | <= | >= | != | =
```

Une expression peut être, une expression conditionnelle (`si`), une déclaration locale (`soit`), une fonction (`fonction`), une fonction récursive (`rec`), une application de fonction, une expression entre parenthèses, une variable (`nom`), une constante entière (`i`), un booléen (`vrai` et `faux`) ou une expression arithmétique.

Exemple 1 Les expressions suivantes sont des expressions valides du langage :

```
def x = 42;;

def y =
  soit z = x * x dans
  soit k = z + 1 dans
  z + 2;;

def f =
  fonction x -> fonction y -> si y = x alors soit x = 2 dans x+y sinon 3;;

def resultat = f 2 3;;

def g = rec fact x = si x = 0 alors 1 sinon x * (fact (x-1));;

let v = g 4

def app = fonction f -> fonction g -> fonction x -> f (g x);;
```

Les constructions sont essentiellement celles de Caml Light (à un renommage près des mots clefs pour éviter les ambiguïtés). La construction de fonctions récursives est cependant un peu différente : l'expression

```
rec fact x = si x = 0 alors 1 sinon x * (fact (x-1))
```

définit directement une fonction récursive et est équivalente à l'expression de Caml Light suivante :

```
let rec fact x = if x = 0 then 1 else x * (fact (x-1)) in fact
```

2 Organisation du logiciel

Le typage d'une expression est obtenu en enchaînant les étapes suivantes :

1. Une première étape d'**analyse syntaxique** qui construit une structure représentant la déclaration donnée en entrée.
2. Une étape de typage qui associe un type à chaque déclaration.
3. Une étape d'**affichage** du résultat et de **mise à jour** de l'environnement global. Un message d'erreur sera émis en cas d'erreur durant le typage.

Pour réaliser ce logiciel, un certain nombre de modules vous sont donnés. L'organisation globale est détaillée ci-dessous.

2.1 Représentation abstraite des expressions et des types

Les expressions du langage et les types sont définis dans le module :

Module syntaxe_abstraite Ce module définit les déclarations, les expressions, les valeurs dites immédiates (entiers, booléens), les types du langage, les variables de type et les schémas de type. Il déclare les types suivants :

```
type decl = ...
and expr = ...
and immediat = ..
and op = ...

and typ = ...
and var_type = ...
and schema_de_type = ...
```

Ce module vous est donné (voir fichier `syntaxe_abstraite.mli`).

Module global Le module **global** déclare des fonctions de manipulation de l'environnement global (ajout d'une liste de déclaration et type associé à un identificateur). Il déclare également une exception qui sera levée lorsqu'une erreur est survenue lors de l'analyse du programme (erreur de syntaxe ou erreur de typage). Il déclare les valeurs suivantes :

```
exception Erreur;;
value ajout_global : (string * schema_de_type) list -> unit;;
value chercher_env_global : string -> schema_de_type;;
  (* peut lever l'exception Not_found *)
```

Ce module vous est donné.

2.2 Analyse syntaxique

Module syntaxe Déclare les fonctions d'analyse syntaxique permettant de construire une déclaration et une expression à partir d'une chaîne de caractères. La troisième fonction d'analyse lit le contenu d'une chaîne de caractères stockée dans un buffer dit buffer lexical. Ce buffer devra avoir été créé au préalable. Cette création est expliquée dans le module `lexing` du manuel de référence de Caml Light (page 151).

```
value lire_decl_chaine : string -> decl;;
value lire_expr_chaine : string -> expr;;
value lire_decl : lexing__lexbuf -> decl;;
```

Ce module vous est donné.

2.3 Typage

Comme vous l'avez vu en cours, le typage d'une expression consiste en un parcours récursif de celle-ci en appliquant à chaque construction rencontrée, la règle de typage appropriée. Durant ce typage, il peut être nécessaire d'introduire des inconnues de type et de résoudre alors des équations entre types.

Module Typer Ce module définira une fonction de typage d'une expression.

Cette partie est à faire (reportez vous aux sections 3 et 4 de ce document).

2.4 Affichage des types, traitement des erreurs et module principal

Une fois le type obtenu, celui-ci doit être affiché et en cas d'erreur, vous devrez indiquer la source de l'erreur.

Module afficher_type Ce module définit des fonctions permettant d'afficher un type ou un schéma de type. Il exporte les fonctions :

```
value afficher_type : out_channel -> typ -> unit;;
value affiche_type_schema : out_channel -> schema_de_type -> unit;;
value afficher_decl_type : out_channel -> (string * schema_de_type) list
                                        -> unit;;
```

`afficher_type oc typ` affiche le type `typ` dans la sortie `oc` (`oc` est un descripteur de fichier de sortie qui devra avoir été créé auparavant). `afficher_type` permet d'afficher un schéma de type. Enfin, `afficher_decl_type` permet d'afficher une liste de déclaration d'identificateurs globaux et leurs schémas de type associés.

Ce module vous est donné.

Module erreur Définit divers types d'erreurs. Il exporte les fonctions :

```
value erreur_de_syntaxe : unit -> unit;;
value erreur_identificateur : string -> unit;;
value erreur_de_type : unit -> unit;;
```

`erreur_de_syntaxe ()` affiche un message en cas d'erreurs de syntaxe.
`erreur_identificateur x` indique que `x` n'est pas déclaré.
`erreur_de_type ()` indique qu'une erreur de type est survenue.

Ce module vous est donné.

Module main Le module principal permet d'enchainer les différentes phases de traitement (analyse syntaxique, typage et affichage des types). Il devra exporter les fonctions suivantes :

```
main_decl : unit -> unit
main_string : string -> unit
```

Ce module est à écrire.

La fonction `main_decl` lance l'analyse syntaxique et le typage de la suite des déclarations apparaissant dans un fichier. Les déclarations sont analysées les unes après les autres. L'analyse d'un fichier s'arrête dès que l'analyse d'une déclaration échoue. La fonction `main_string` a le même comportement mais peut être appliquée à une chaîne de caractères.

Le travail à réaliser est décomposé en deux parties. Dans la première partie, vous implanterez un algorithme de typage pour un sous ensemble du langage où le typage peut être fait sans manipuler d'inconnus de type. Dans la deuxième partie, vous implanterez un algorithme de typage pour l'ensemble du langage.

3 Première partie : vérification des types

3.1 Question préliminaire

Pour vous familiariser avec l'analyse syntaxique, vous pouvez regarder le résultat de cette analyse sur les exemples donnés à la page 1 de ce document. Par exemple :

```
>       Caml Light version 0.74

#include "load.ml";;
#lire_decl_chaine "def x = si c alors 1 sinon y + 3;;";;
- : decl =
 Ldef
  ("x",
   Lconditionnelle
    (Lvar "c", Lconst (Lint 1), Lbinop (Bplus, Lvar "y", Lconst (Lint 3))))
```

Dans cette première partie, l'objectif est d'implanter un algorithme permettant de typer les expressions arithmétiques, les variables, les constantes, les conditionnelles ainsi que les déclarations locales.

Vous définirez pour cela deux modules :

Module typer Ce module devra exporter une fonction `typer` permettant de typer une expression. Elle sera basée sur les règles de typage que vous avez vu en cours.

Module main Le module principal enchaînera la phase d'analyse syntaxique, de typage, l'affichage des types sur la sortie standard et le traitement des erreurs.

La construction de ces deux modules sera faite pas à pas, selon les instructions suivantes. Vous devrez développer votre logiciel en utilisant (et en complétant) le `Makefile` qui vous est donné.

3.2 Constantes, expressions arithmétiques et conditionnelles

1 Écrire la fonction de typage lorsque les expressions sont réduites à des constantes entières ou booléennes.

2 Enrichissez la fonction de typage pour qu'elle traite les expressions arithmétiques de la forme `e1 op e2`. En cas d'erreur de type, vous afficherez un message approprié.

3 Enrichissez la fonction de typage pour qu'elle traite également les conditionnelles. On rappelle ici qu'une expression conditionnelle `si c alors e1 sinon e2` est bien typée si `c` est un booléen et si `e1` et `e2` sont de même type.

3.3 Variables, déclarations locales, déclarations globales

Le typage d'une expression contenant une variable nécessite la définition d'un environnement permettant de retrouver le type associé à une variable. Un environnement est une liste d'associations (nom, schema_de_type). La signature de la fonction `typer` sera :

```
typer : (string * schema_de_type) list -> expr -> typ
```

La fonction `typer` sera telle que : `typer Env e` rend le type `ty` lorsque `Env` $\vdash$ `e` : `ty`.

4 Définir la fonction de typage des variables et des définitions locales. On fait l'hypothèse ici que les types ne sont pas polymorphes. Il n'y a donc pas de règle d'instanciation ni de règle de généralisation pour l'instant.

5 Un programme est une suite de déclarations qui sont évaluées dans un environnement global (voir le module **global**).

Définir la fonction de typage des déclarations globales dans le module `typer`. Vous définirez ensuite la fonction principale du module `main`. Il faudra pour cela ouvrir le fichier en lecture, lancer l'analyse syntaxique puis le typage et enfin, mettre à jour l'environnement de type global.

4 Deuxième partie

La fonction de typage que vous avez réalisé ne permet pas de typer les autres constructions du langage pour lesquelles il est nécessaire d'introduire durant le typage des inconnues de type et de résoudre des équations sur ces types. Elle ne permet pas non plus d'introduire du polymorphisme.

L'objectif de cette partie est donc d'étendre la fonction de typage. Pour ce faire, nous vous donnons un module s'occupant de la résolution d'équations. L'algorithme de synthèse des types est basé sur un algorithme d'unification que vous verrez en TD et dont le principe est rappelé ici :

Unifier deux types t et t' consiste à trouver comment remplacer les variables de ces types par des types, de telle sorte qu'après remplacement, les deux types obtenus soient égaux. Autrement dit, il s'agit de trouver une substitution (une liste d'association) $[\alpha_1\backslash t_1, ..., \alpha_n\backslash t_n]$ telle que :

$$t[\alpha_1\backslash t_1, ..., \alpha_n\backslash t_n] = t'[\alpha_1\backslash t_1, ..., \alpha_n\backslash t_n]$$

Cette substitution est désignée `subst_type` dans la suite du texte. Par exemple, les deux types :

$$\texttt{int} \to (\alpha_1 \to \texttt{bool} * \alpha_2) \text{ et } \alpha_3 \to (\texttt{int} \to \texttt{bool} * (\texttt{int} * \texttt{bool}))$$

sont unifiables en utilisant la substitution :

$$[\alpha_3\backslash\texttt{int}, \alpha_1\backslash\texttt{int}, \alpha_2\backslash(\texttt{int} * \texttt{bool})]$$

Une substitution sera représentée par une liste d'association de type `(var_type * typ) list` associant un type `typ` à une variable de type `var_type`.

L'unification est réalisée par le module suivant :

Module unification Définit les types de données et les valeurs suivantes :

```
exception Unification;;

value reinit_var_de_type : unit -> unit;;
value gen_var_type : unit -> var_type;;

value vars : var_type list -> var_type list -> typ -> var_type list;;
value var_libres : typ -> var_type list;;
value var_libres_env : (string * schema_de_type) list -> var_type list;;

value substituer : (var_type * typ) list -> typ -> typ;;
value unifier : (var_type * typ) list -> typ -> typ -> (var_type * typ) list;;

value generaliser :
  (string * schema_de_type) list -> typ * (var_type * typ) list
                                  -> schema_de_type;;
value instancier : schema_de_type -> typ;;
```

La fonction `gen_var_type` rend une nouvelle variable de type et `reinit_var_de_type` permet d'initialiser cette fonction.

`substituer subst_type t` où `subst_type` est une substitution de type (associant des types à des variables de type) et `t`, un type, rend un type `t'` déduit de `t` en remplaçant les variables de `t` figurant dans `subst_type` par les types qui leur sont associés.

`unifier subst_type t1 t2`, où `subst_type` est une substitution de type, `t1` et `t2` sont des types, rend une nouvelle substitution `subst_type'` telle que `substituer subst_type' t1 = substituer subst_type' t2`. Lorsque les deux types ne sont pas unifiables, l'exception `Unification` est levée.

Ce module vous est donné. Il est recommandé cependant de l'examiner en détail et d'essayer plusieurs exemples avant de s'attaquer au typage des programmes.

Dans cette partie, le module **typer** sera enrichi. Il devra exporter les deux fonctions suivantes :

Module typer
```
value typer :
  (string * schema_de_type) list -> (var_type * typ) list
                                  -> expr -> typ * (var_type * typ) list;;
value typer_decl : decl -> unit;;
```

`typer env_de_type subst_type e` rend un couple `(ty, nouv_subst_type)` où `ty` est le type obtenu pour l'expression `e` et où `nouv_subst_type` est une nouvelle substitution de type (on peut en effet supposer que le typage de `e` a fait apparaître de nouvelles contraintes de type). Cette nouvelle substitution enrichie la substitution `subst_type` avec de nouvelles contraintes. La fonction `typer` prend en paramètre un environnement de type `env_de_type` (associant des schémas de type aux variables du programme) et `subst_type`.

`typer_decl` s'occupe du typage des déclarations : cette fonction type une déclaration, modifie l'environnement global et affiche le type obtenu.

6 Vous devrez ici reprendre le code de la fonction de typage vue dans la première partie. Pour cela, redéfinissez la nouvelle règle de typage des variables. La règle de typage des variables dans un environnement de type `env_type` est :

$$\text{si } \texttt{env\_type}(\mathrm{x}) = \forall\alpha_1, ..., \alpha_n.ty \text{ alors } x : \texttt{instancier}\ (\forall\alpha_1, ..., \alpha.ty)$$

où la fonction `instancier` remplace les variables α_1,...,α_n par des variables de type neuves.

7 Définissez la nouvelle règle de typage des déclarations locales et globales. On rappelle qu'une fois que le type d'une déclaration a été obtenu, celui-ci doit être généralisé. On a :

$$\texttt{generaliser env\_type}\ (ty) = \forall\alpha_1, ..., \alpha_n.ty$$

où les α_i n'appartiennent pas à `var_libres_env env_type`. La fonction `var_libres_env` est définie dans le module **unification**.

8 Revoyez la définition de la fonction de typage des expressions arithmétiques. Définir la fonction de typage des applications de fonctions. On rappelle qu'une application `f e` est bien typée et de type t_2 si le type de `f` peut s'unifier avec un type $t_1 \to t_2$ et si le type de `e` peut s'unifier avec t_1.

9 Définir la fonction de typage des fonctions puis des récusions.

5 Compte rendu de TP

Le compte rendu doit être rédigé de manière concise et précise, en soignant la présentation.

1. Vous rendrez le code source complet.
2. Donner un jeu d'exemples significatifs pour tester la fonction d'évaluation des expressions et des déclarations.

6 Documents fournis

Tous les fichiers nécessaires pour le TP se trouvent dans le catalogue `~pouzet/tp3`. Vous y trouverez un fichier `LISEZMOI` qui décrit les différents fichiers ainsi qu'un `Makefile` donnant l'architecture générale du logiciel. Ce fichier est à modifier suivant vos besoin. Enfin, le fichier `exemples.mml` vous donne quelques exemples de programmes du langage et le résultat rendu par le typage.

Vous disposez de l'exécutable `typer` qui prend en entrée un fichier `fichier.mml` et qui affiche sur la sortie standard le résultat du typage.