Programmation – TD6-7 : MiniML

{ Jeremie.Detrey, Emmanuel.Jeandel } @ens-lyon.fr

8/10 et 17/18 novembre 2004

Ce TD est très long, et nous avons donc décidé de l'étaler sur deux séances. Une répartion correcte du boulot serait approximativement de faire l'évaluation et l'inférence de type lors de la première séance, puis l'unification lors de la seconde[1].

MiniML, un langage qu'il est bien pour s'amuser

Au cours de ce TD, nous allons écrire un mini typeur-interprète pour MiniML, un langage d'expressions proche du cœur fonctionnel d'OCaml.

Voici, de manière succinte, la grammaire de ce langage :

$expr$	::=	$expr'$ `;;`	
$expr'$	::=	x	entier
	|	`true`	booléen
	|	`false`	
	|	id	identificateur
	|	`(` $expr'$ `)`	parenthèses
	|	$expr'$ `+` $expr'$	addition
	|	$expr'$ `-` $expr'$	soustraction
	|	$expr'$ `*` $expr'$	multiplication
	|	$expr'$ `/` $expr'$	division
	|	$expr'$ `==` $expr'$	égalité
	|	$expr'$ `>` $expr'$	comparaison
	|	`not` $expr'$	négation
	|	$expr'$ `&&` $expr'$	ET logique
	|	`if` $expr'$ `then` $expr'$ `else` $expr'$ `end`	conditionnelle
	|	`let` id `=` $expr'$ `in` $expr'$ `end`	déclaration
	|	`let rec` id `=` $expr'$ `in` $expr'$ `end`	déclaration récursive
	|	`fun` id `->` $expr'$ `end`	fonction
	|	$expr'$ $expr'$	application

Ainsi, l'expression suivante, qui calcule $10!$:

```
let rec fact = fun n -> if n > 1 then n * (fact (n-1)) else n end end
in fact 10 end;;
```

[1] Si vous parvenez à tout faire en une séance, ne vous inquiétez pas, vous aurez du rab' à la suivante.

Pour manipuler ces expressions, on adopte une représentation canonique par les types somme suivants, définis dans le module `Expr` :

```
type id = string

type operator = Add | Sub | Mul | Div | Not | And | Eq | Gt | If

type t =
  | Int of int
  | Bool of bool
  | Var of id
  | Op of operator * (t list)
  | Let of id * t * t
  | LetRec of id * t * t
  | Fun of id * t
  | App of t * t
```

Pour vous faciliter la tâche, vos TDmen (décidément trop bons envers leurs étudiants) ont écrit les modules `Lexer` et `Parser` qui s'occupent de lire un fichier (ici l'entrée standard) et de convertir chaque expression en un objet de type `Expr.t`.

Allez donc à l'adresse habituelle `http://perso.ens-lyon.fr/jeremie.detrey/04_prog/` pour récupérer l'archive `miniml.tar.gz`. Une fois extraite (grâce à la commande magique `tar xzvf miniml.tar.gz`), vous y trouverez les fichiers suivants :

- `Makefile` : très utile pour gérer les dépendances. Vous aurez bien entendu à la modifier pour y ajouter vos propres modules.
- `expr.ml` : le module `Expr`, qui ne contient que les définitions des types vus précédemment, ainsi qu'une fonction d'affichage `print` très rudimentaire permettant d'afficher une expression sous sa forme arborescente.
- `lexer.mll` : le module de *lexing*, qui convertit un fichier en une suite de *tokens* (ou objets du langage). Il s'agit d'un fichier `.mll`, qui doit être compilé avec `ocamllex` pour générer `lexer.ml`.
- `parser.mly` : le module de *parsing*, qui convertit les tokens en expressions de type `Expr.t`. Il s'agit d'un fichier `.mly`, qui doit être compilé avec `ocamlyacc` pour générer `parser.ml` et `parser.mli`.
- `miniml.ml` : le module principal, qui contient la boucle de l'interprète. Pour l'instant, l'interprète se contente de lire des expressions sur l'entrée standard puis de les afficher à l'aide de la fonction `Expr.print`.

Les fichiers lexer et du parser sont tordus, donc vous n'êtes pas obligés d'y fourrer le nez (mais si vous êtes curieux ne vous gênez surtout pas). Par contre, regardez bien l'articulation de tous les modules, ainsi que le fonctionnement du `Makefile`. Il est très important que lors de ce TD vous appreniez à gérer un ensemble de modules et pour cela à vous servir correctement d'un `Makefile`. **Ainsi vous devrez écrire un fichier `.ml` (ainsi qu'un `.mli` optionnel) par module.**

1 Évaluation des expressions

Pour ceux qui frissonnent encore à l'évocation du dernier TD et des continuations, ne vous inquiétez pas : ce coup-ci, plus besoin de continuations, vu qu'on n'a pas d'exceptions dans MiniML. Ça devrait donc être bien plus facile.

Mais avant toute chose, il faut pouvoir gérer un environnement (ou un *contexte*) d'évaluation. C'est en effet dans ce contexte qu'iront les variables définies par des `let ... in` par exemple. Comme nous allons aussi avoir besoin d'un environnement lors de l'inférence de type, nous allons définir un environnement générique, associant un identificateur de type `Expr.id` à un objet de type `'a`.

1. Écrivez donc un module `Env` permettant de gérer des environnements génériques de type `'a Env.t`. Ce module devra fournir des fonctions comme l'ajout de paires (identificateur,objet) à l'environnement, ainsi que la recherche de l'objet correspondant à un identificateur. Vous avez libre choix quant à la représentation de ces environnements.

Il va aussi nous falloir de quoi manipuler les valeurs :

2. Écrivez un module `Val`, dans lequel vous définirez leur type `t`, ainsi qu'une fonction d'affichage `print`.
   Indication : vous n'allez avoir que trois types de valeurs : les entiers, les booléens et les fonctions. Si les deux premiers sont faciles, réfléchissez bien pour les fonctions.

Voilà, tout devrait être bon maintenant. Vous pouvez passer à l'attaque :

3. Écrivez un module `Eval`, qui contiendra une fonction `eval : Expr.t -> Val.t` évaluant des expressions.

2 Typage[2] (monomorphe)

Nous souhaitons effectuer le typage de nos expressions. Ceci est se fait en deux temps : tout d'abord la génération des contraintes de types, puis la résolution de ces contraintes par unification.

Le typage monomorphe (le seul vu en cours pour l'instant) n'est pas assez puissant pour typer tous les termes de MiniML. Ainsi l'expression :

```
let f = (fun x -> x end) in (f f) end;;
```

n'est pas typable avec notre simple algorithme. D'une façon plus générale, le typage monomorphe ne permet pas de typer les constructions `let` et `let rec`.

Dans la suite, pensez donc bien que vous n'avez PAS à gérer les cas de `let` et `let rec`.

2.1 Générer les contraintes

Comme pour l'évaluation, il nous faut dans un premier temps définir un type pour les types de nos expressions.

1. Pour cela, écrivez un module `Type` permettant de manipuler ces types. Définissez-y entre autres le type `t` suivant, ainsi qu'une fonction d'affichage.

[2] En raison du nombre considérable de jeux de mots possibles avec "type", il a été décidé par pitié envers les élèves de n'en faire aucun.

```
type t =
  | Bool
  | Int
  | Fun of t * t
  | Var of int
```

Une contrainte sera alors représentée par un couple `Type.t * Type.t`.
Ainsi, si l'on reprend un exemple du cours, la contrainte :

$$A_1 \to (\texttt{int} \to A_2) \stackrel{?}{=} (A_3 \to \texttt{bool}) \to A_1$$

sera représentée par le couple :

```
Fun (Var 1, Fun (Int, Var 2)), Fun (Fun (Var 3, Bool), Var 1)
```

2. Écrivez alors un module `Constr` dans lequel vous définirez une fonction de génération des contraintes `build : Expr.t -> Type.t * ((Type.t * Type.t) list)`. Cette fonction renverra ainsi le type de l'expression et la liste des contraintes générées par le typage de cette expression.
Remarque : vous aurez besoin d'une fonction `new_var : unit -> Type.t` vous permettant de créer des nouveaux types `Var i` à la volée.

2.2 Unification

Il faut maintenant résoudre les contraintes, et, pour cela, unifier. L'unification est un principe général, et on va donc le faire dans le cas général. Plus précisément, au lieu d'unifier des termes de type :

```
type Type.t =
  | Bool
  | Int
  | Fun of t * t
  | Var of int
```

on va travailler sur des termes de type :

```
type Unif.t = Cons of string * (t list) | Var of int
```

L'idée est que `Cons` sert à représenter n'importe quelle construction de type : ainsi, par exemple, `Cons ("Toto", [`x_1 `; ... ;` x_n`])` représente la construction du type `Toto`, qui est une construction d'arité n.

Dans ce contexte, on a donc le schéma de correspondance suivant :

`Type.t`		`Unif.t`
`Int`	$\leftrightarrow$	`Cons ("Int", [])`
`Bool`	$\leftrightarrow$	`Cons ("Bool", [])`
`Fun (x,y)`	$\leftrightarrow$	`Cons ("Fun", [x;y])`
`Var i`	$\leftrightarrow$	`Var i`

Les règles d'unification, dans ce contexte général, sont rappelées Fig. 1.

Trivial
$\{s \stackrel{?}{=} s\} \cup P, S \;\Rightarrow\; P, S$

Décomposition
$\{\texttt{Cons}\ C\ [s_1 \dots s_n] \stackrel{?}{=} \texttt{Cons}\ C\ [t_1 \dots t_n]\} \cup P, S \;\Rightarrow\; \{s_1 \stackrel{?}{=} t_1\} \cup \dots \cup \{s_n \stackrel{?}{=} t_n\} \cup P, S$

Conflit
$\{\texttt{Cons}\ C\ [s_1 \dots s_n] \stackrel{?}{=} \texttt{Cons}\ D\ [t_1 \dots t_p]\} \cup P, S \;\Rightarrow\; \textit{échec}$ $\qquad C \neq D$

Élimination de variable
$\{\texttt{Var}\ x \stackrel{?}{=} t\} \cup P, S \;\Rightarrow\; P_{[x \leftarrow t]}, S_{[x \leftarrow t]} \cup \{x \rightsquigarrow t\}$ $\qquad x \notin \mathrm{Var}(t)$

Circularité
$\{\texttt{Var}\ x \stackrel{?}{=} t\} \cup P, S \;\Rightarrow\; \textit{échec}$ $\qquad x \in \mathrm{Var}(t) \wedge x \neq t$

Orientation
$\{t \stackrel{?}{=} \texttt{Var}\ x\} \cup P, S \;\Rightarrow\; \{\texttt{Var}\ x \stackrel{?}{=} t\} \cup P, S$ $\qquad t$ n'est pas une variable

FIG. 1 – Algorithme d'unification : règles pour passer du problème P dont la solution est S au problème P' avec la solution S'.

Vous avez donc besoin de deux structures de données pour représenter :
- les ensembles de contraintes : on appelle ce type `pb` ;
- les substitutions : on appelle ce type `sub` ;

Le choix de la structure de données la plus adaptée vous est laissé.

3. Écrivez un module `Unif` dans lequel vous définirez le type `Unif.t` vu précédemment, une fonction `unif : (pb * sub) -> (pb * sub)` ainsi que toutes les fonctions qui vous semblent nécessaires pour écrire l'unification.

2.3 Putting it all together

4. Réunissez convenablement tout ce que vous avez écrit de façon à obtenir un interprète MiniML qui, avant d'évaluer un terme, vérifie qu'il est bien typé.

2.4 Pour ceux qui n'ont rien d'autre à faire

5. Enrichissez le langage MiniML à votre guise : boucle `while`, séquence d'expressions ($expr'; expr'$), exceptions, ... Arrêtez-vous quand vous commencez à être plus puissant qu'OCaml.

2.5 Pour ceux qui n'ont *vraiment* rien d'autre à faire

6. À l'aide de votre module d'unification, écrivez un interpréteur Prolog.