SCIENCES DU NUMÉRIQUE

S7 - Programmation Fonctionnelle

Projet

Inférence de type pour MiniML

Auteurs :
Aicha BENNAGHMOUCH
Chloé HOSÉ WENDLING
Yvan Charles KIEGAIN DJOKO
Maëlis MARCHAND

22 janvier 2023

Systèmes Logiciels - Groupe L-34
Département Sciences du Numérique - Deuxième année
2022-2023

Table des matières

Table des figures

1 Introduction

L'inférence de types de OCAML permet de déterminer les types des expressions manipulées par le langage, ou de détecter une potentielle mauvaise utilisation des données qui provoquerait une erreur à l'exécution.
Dans ce projet, notre but était de réaliser l'inférence de types de OCAML pour le langage MINIML, qui est un sous-ensemble du langage OCAML.

Le projet a été découpé en différentes tâches :

— nous avons réalisé un **parser** du langage MiniML, à partir d'une grammaire simplifiée de celui-ci et d'un lexer fourni ;

— puis, nous avons ensuite réalisé un **typer** : il s'agit d'un algorithme qui définit le type d'une expression en appliquant consécutivement plusieurs règles de typage et en accumulant des équations entre type qui garantissent le bon typage ;

— nous avons ensuite réalisé un **normalizer** : à partir des équations renvoyées par le typer et d'un ensemble de règles, il définit le type de l'expression ;

— enfin, nous avons défini un **programme principal** qui lit un programme MiniML dans un fichier, applique les règles de typage afin d'obtenir son type et les équations associées, détermine la forme la plus générale des solutions ou l'absence de solution, puis affiche le type obtenu ou une erreur de typage.

2 Reconnaissance du langage MiniML

2.1 Grammaire du langage MiniML

Pour commencer, nous avons réalisé un parser pour reconnaître des expressions du langage MiniML, dont voici la grammaire :

Expr	→	**let** *Liaison* **in** *Expr*	*Liaison*	→	*ident* = *Expr*
	|	**let rec** *Liaison* **in** *Expr*	*Binop*	→	*Arithop* | *Boolop* | *Relop* | @ | ::
	|	(*Expr Binop Expr*)	*Arithop*	→	+ | − | ∗ | /
	|	(*Expr*)	*Boolop*	→	&& | ||
	|	(*Expr Expr*)	*Relop*	→	= | <> | <= | < | >= | >
	|	**if** *Expr* **then** *Expr* **else** *Expr*	*Constant*	→	*entier* | *booleen* | [] | ()
	|	(**fun** *ident* −> *Expr*)			
	|	*ident*			
	|	*Constant*			

FIGURE 1 – Grammaire des expressions de MiniML

Dans cette grammaire, nous avons donc identifié plusieurs règles à parser.

2.2 Choix de conception pour le parser

Pour réaliser le parser, nous avons dû définir plusieurs types. Comme vu en TP/TD, nous avons utilisé le type **('a, 'b) result**. Notre parser est de type **('a, 'b) parser**, et plus précisément de type **(token, expr) parser** : il prend en paramètre un flux entrant de tokens (**token FluxEnt.t**) et renvoie un résultat de type **(token, expr) result**.

```
(* types des parsers *)
type ('a,'b) result = ('b * 'a FluxEnt.t) Solution.t;;
type ('a, 'b) parser = 'a FluxEnt.t -> ('a, 'b) result;;
```

FIGURE 2 – Types utilisés par le parser

Nous avons ensuite écrit une monade Parsing dans laquelle nous définissons des opérations permettant de combiner des parsers (telles que map, bind (»=), (++)), et d'autres permettant de créer des parsers simples comme par exemple pvide ou ptest.

Nous avons ensuite créé un module Parser qui implémente cette interface Parsing à l'aide des fonctions du module Solution, comme nous l'avons vu en TP.
Ensuite, nous avons défini un parser pour chaque mot-clé (comme let, rec, in, etc) et chaque symbole (parenthèses, crochets, opérateurs...).

3 Règles de typage

3.1 Règles de typage du langage MiniML

Dans un second temps, nous avons réalisé un typer : c'est un algorithme qui donne le type d'une expression et accumule les équations entre types qui garantissent le bon typage. Pour cela, on se base sur les règles de typage suivantes :

$$Const\ \frac{c \in Const}{\Gamma \vdash c : Type(c)} \qquad Var_1\ \frac{}{(x:\tau) :: \Gamma \vdash x : \tau} \qquad Var_2\ \frac{\Gamma \vdash x : \tau}{(y:\sigma) :: \Gamma \vdash x : \tau}$$

$$Cons\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1 :: e_2) : \tau_2}\ (\tau_2 \equiv \tau_1\ \text{list}) \qquad Pair\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$Fun\ \frac{(x:\alpha) :: \Gamma \vdash e : \tau}{\Gamma \vdash (\textbf{fun}\, x \text{->} e) : \alpha \to \tau}$$

$$App\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \alpha}\ (\tau_1 \equiv \tau_2 \text{ -> } \alpha)$$

$$IfThenElse\ \frac{\Gamma \vdash b : \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \textbf{if}\, b\, \textbf{then}\, e_1\, \textbf{else}\, e_2 : \tau_1}\ (\tau \equiv \text{bool}, \tau_1 \equiv \tau_2)$$

$$Let\ \frac{\Gamma \vdash e_1 : \tau \quad \{x:\tau\} :: \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \textbf{let}\, x = e_1\, \textbf{in}\, e_2 : \tau'}$$

$$LetRec\ \frac{\{x:\alpha\} :: \Gamma \vdash e_1 : \tau \quad \{x:\alpha\} :: \Gamma \vdash e_2 : \tau'}{\Gamma \vdash \textbf{let rec}\, x = e_1\, \textbf{in}\, e_2 : \tau'}\ (\alpha \equiv \tau)$$

FIGURE 3 – Règles de typage

3.2 Choix de conception pour le typer

Afin de réaliser le typer, nous avons du faire un choix de types et de structures de données pour représenter un environnement. Nous avons décidé que l'environnement serait une liste de couples : un couple contient le nom de la variable et son type (**'a typ**) associé. Nous avons donc défini le type **'a env** ainsi :

```
type 'a env = (ident * 'a typ) list
```

FIGURE 4 – Type environnement

Certaines règles de typage (en l'occurence *Cons*, *App*, *IfThenElse* et *LetRec*) engendrent des équations entre types qui garantissent le bon typage. Notre typer doit donc renvoyer la liste des équations accumulées pendant le typage d'une expression. Nous avons donc défini le type **'a equ** pour représenter une équation entre deux types :

```
type 'a equ = ('a typ * 'a typ)
```

FIGURE 5 – Type équation

Nous avons donc défini ainsi notre typer :

```
let rec typer : expr -> 'a env -> ('a typ, 'a equ list) option
```

FIGURE 6 – Type de la fonction typer

Notre typer prend en paramètres :
— l'expression à typer de type **expr** ;
— l'environnement de type **'a env** dans lequel se trouvent les couples (variable, type) connus.

Il renvoie un couple avec :
— le type (**'a typ**) trouvé pour l'expression ;
— la liste d'équations de type accumulées (**'a equ list**) lors du processus de typage.

On utilise le type **option** car dans le cas où l'expression passée en paramètre du typer est mal définie, celui-ci renvoie *None*.

4 Résolution des équations

4.1 Règles de normalisation du langage MiniML

Une fois le type d'une expression obtenu avec le typer, il reste à résoudre les équations de type accumulées pour obtenir le type réel de l'expression. Ce processus de normalisation transforme progressivement les équations générales en définitions de variables de types. Voici les règles de normalisation utilisées :

$$\frac{Eqs \vdash_N \tau \Rightarrow \sigma}{\{\text{int} \equiv \text{int}\} \cup Eqs \vdash_N \tau \Rightarrow \sigma} \qquad \frac{Eqs \vdash_N \tau \Rightarrow \sigma}{\{\text{bool} \equiv \text{bool}\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}$$

$$\frac{Eqs \vdash_N \tau \Rightarrow \sigma}{\{\text{unit} \equiv \text{unit}\} \cup Eqs \vdash_N \tau \Rightarrow \sigma} \qquad \frac{\{\tau_1 \equiv \tau_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}{\{\tau_1 \text{ list} \equiv \tau_2 \text{ list}\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}$$

$$\frac{\{\tau_1 \equiv \sigma_1, \tau_2 \equiv \sigma_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}{\{\tau_1 -> \tau_2 \equiv \sigma_1 -> \sigma_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma} \qquad \frac{\{\tau_1 \equiv \sigma_1, \tau_2 \equiv \sigma_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}{\{\tau_1 * \tau_2 \equiv \sigma_1 * \sigma_2\} \cup Eqs \vdash_N \tau \Rightarrow \sigma}$$

$$\frac{Eqs \vdash_N \tau \Rightarrow \sigma}{\{\alpha \equiv \alpha\} \cup Eqs \vdash_N \tau \Rightarrow \sigma} \qquad \frac{[\alpha \mapsto \rho]Eqs \vdash_N \tau \Rightarrow \sigma \quad (\alpha \text{ non libre dans } \rho \neq \alpha)}{\{\alpha \equiv \rho\} \cup Eqs \vdash_N \tau \Rightarrow [\alpha \mapsto \rho]\sigma}$$

$$\frac{}{\emptyset \vdash_N \tau \Rightarrow \tau} \qquad \frac{\{\alpha \equiv \rho\} \cup Eqs \vdash_N \tau \Rightarrow \sigma \quad (\rho \text{ n'est pas une variable})}{\{\rho \equiv \alpha\} \cup Eqs \vdash_N \tau \Rightarrow [\alpha \mapsto \rho]\sigma}$$

FIGURE 7 – Règles de normalisation

4.2 Choix de conception pour le normalizer

Nous avons défini les fonctions relatives au normalizer dans le module Normalizer.

Pour notre programme normalizer, nous avons défini ainsi le type **'a jugementNorm** pour représenter un jugement de normalisation :

```
type 'a jugementNorm = JugementNorm of 'a equ list * 'a typ
```

FIGURE 8 – Type d'un jugement de normalisation

Un jugement de normalisation est ainsi composé :
— de la liste d'équations de types obtenues avec le typer ;
— et du type τ à normaliser.

Notre normalizer est une fonction récursive qui applique successivement plusieurs jugements de normalisation pour normaliser un type τ. Il renvoie ensuite le type normalisé : **'a typ option** car en cas d'erreur de type (par exemple un type récursif mal fondé), la fonction renvoie *None*.

5 Utilisation du programme principal

Nous avons défini un programme principal qui :
— lit un programme MINIML dans un fichier à l'aide de la fonction **readminimltokensfromfile** définie
— parse l'expression obtenue à l'aide la fonction **parseminiml**.
— applique les regles de typage afin d'obtenir son type en utilisont la fonction typer défini dans le module RegleTypage.
— applique les équations associées avec la fonction normalizer défini dans le module Normalizer.
— affiche le type obtenu ou une erreur de typage.

Afin d'utiliser ce programme il faut :
— compiler comme suit : dune build miniml_principal.exe
— lancer le fichier : _build/default/bin/miniml_principal.exe **tests/filename**

Nous avons fourni plusieurs fichiers de tests définis dans le répertoire tests.

6 Conclusion

En conclusion, ce projet nous a permis de mettre en application les différents concepts vus en programmation fonctionnelle : parsing, monades, flux, etc.
Cependant, nous avons eu du mal à nous lancer dans le projet car il nous a fallu du temps à nous approprier le sujet qui est relativement abstrait. Un ou plusieurs exemples nous auraient été utiles.