Rapport Projet TAS - Implémentation d'un langage de programmation avec typage

Typeur, Evaluateur , Analyseur Syntaxique

Dibassi Brahima 21210230

2023-11-19

Table des matières

1 Langage de programmation choisi

Afin d'implémenter ce projet, nous avons choisi le langage de programmation *OCaml*. Ayant déjà expérimenté l'implémentation de l'interpréteur d'un langage en OCaml suite au cours d' **APS**, nous avons pu constater que ce langage était très adapté à ce genre de projet.
Grâce à la somme de produits au pattern-matching associé, et au support des paradigmes, impératif et fonctionnel, nous avons pu nous servir des différents traits du langage afin de faciliter notre implémentation de l'évaluateur et du typeur.

2 Lancement du projet

Si vous souhaitez lancer votre fichier utilisez la commande suivante :

```
dune exec ProjetTAS *chemin vers le fichier à tester*/*fichier*.ml
```

Nous proposons également un lancement automatisé de notre batterie de tests. La commande est :

```
dune test --force
```

3 Structure du projet

- Repertoire `Lib/`
  - `lexer.mll` : Création des tokens
  - `parser.mly` : Parser afin de construire notre arbre syntaxique abstrait depuis les tokens
  - `ast.ml` : Structure de l'AST du langage avec les types des expressions
  - `baselib.ml` : Définitions des primitives du langage et leur types
  - `evaluator.ml` : Partie sémentique du langage, évaluation des expressions
  - `prettyprinter.ml` : Fonctions d'affichage
  - `typeur.ml` : Partie analyse statique, vérification des types
  - `typingEnv.ml` : Gestion de l'environnement des déclarations de types par l'utilisateur
- Repertoire `Bin/`
  - `main.ml` : Contient notre script principal, lancé lors de la commande `dune exec ProjetTAS *fichier*.ml`. Il vérifie si le programme .ml ciblé est bien typé et l'évalue si c'est le cas en faisant tous les affichages sur le terminal. Nous avons choisi d'afficher également l'état de la mémoire à la fin de l'évaluation du programme.
- Repertoire `Test/`
  - `testing.ml` : Script de test, permet de tester tous les fichiers .ml présents dans le dossier `test` et de vérifier si le typeur et l'évaluateur fonctionnent correctement.
  - `yamlHelper.ml` : Fonctions d'aide pour la lecture des fichiers de test au format yaml
  - `template.yaml` : Template de fichier de test au format yaml

Les tests sont dans le dossier `test` classés en fonction des extentions du langage auxquels ils correspondent. Les extensions du langage doivent également passer les tests des versions précédentes.

4 Syntaxe du langage

Une grande partie de la syntaxe du langage est reprise d'un projet précédent, le langage MiniML.

4.1 Programme

On définit un programme comme une expression précédée de zéro ou plusieurs définitions néccessaires à son évaluation.

```
<Prog> ::=  |  <Expr>
            |  <Def> ';;' <Prog>
```

4.2 Elément nommés

On appelle éléments nommés, les identificateurs **<Id>** pour les variables, motifs et types, les identificateurs de constructeurs **<ConstructeurId>** et les variables de types **<Vartype>**

```
          <Id> ::=  [ 'a - z' 'A - Z' '0 - 9' '_' ]+
<ConstructeurId> ::=  [ 'A - Z' ] <Id>
     <Vartype> ::=  ''' [ 'a - z' ] [ '0 - 9' ]*
```

4.3 Types

- Les **Types polymorphique**.
- Les **Utilisation de variables de types**.
- Les **Applications de types**.
- Les **Lambda**.
- Les **Tuples**.

```
<Type> ::=  |  <Vartype>
            |  <Id>
            |  <Type> <Type>
            |  <Type> '->' <Type>
            |  <Type> '*' <Type>
```

4.4 Déclarations de types

‘*type*‘ <**Vartype**>* <**Id**> ‘=‘ <**NewContructors**>

4.4.1 Nouveaux constructeurs

<**NewContructors**> ::= | <**ConstructeurId**> ‘*of*‘ <**Type**>
| <**NewContructors**> ‘|‘ <**NewContructors**>

4.5 Expressions

- Les **Littéraux**.
- Les **Utilisations de Variables**.
- Les **Appels d'opérateurs** unaire et binaire.
- Les **Appels de fonctions**.
- Les **Tuples**.
- Les **Lambda**.
- Les **Fonctions Récursives**.
- Les **Constructions**.
- Les **Correspondance de motifs**.

<**Litteral**> ::= | [‘*0 - 9*‘]+
| ‘*(*‘ ‘*)*‘

<**Expr**> ::= | <**Litteral**>
| <**Id**>
| <**UnaryOperator**> <**Expr**>
| <**Expr**> <**BinaryOperator**> <**Expr**>
| <**Expr**> <**Expr**>
| <**Expr**> ‘*,*‘ <**Expr**>
| ‘*fun*‘ <**Id**> ‘*->*‘ <**Expr**>
| ‘*let*‘ <**Id**> <**Id**> ‘=‘ <**Expr**> ‘*in*‘ <**Expr**>
| ‘*let*‘ ‘*rec*‘ <**Id**> ‘=‘ <**Expr**> ‘*in*‘ <**Expr**>
| <**ConstructeurId**> <**Expr**>
| ‘*match*‘ <**Expr**> ‘*with*‘ <**MatchCase**>

4.6 Filtrage de motifs

- Les patterns sur **Littéraux**.
- Les patterns sur **Variables**.
- Les patterns sur **Tuple**.
- Les patterns sur **Constructeurs**.

<MatchCase> ::= | <Pattern> '->' <Expr>
| <MatchCase> '|' <MatchCase>

<Pattern> ::= | <Litteral>
| <Id>
| <Pattern> ',' <Pattern>
| <ConstructeurId> <Pattern>

5 Let Polymorphique

Durant la réalisation de ce projet, nous avons rencontré une difficulté majeure, celle de la gestion du polymorphisme qui a été un véritable challenge pour nous.

En effet, nous avons dû faire face à plusieurs problèmes :

1. La gestion de l'environnement des types
2. La généralisation des types
3. La remontée des substitutions induite par l'unification

Afin de résoudre ces problèmes, nous avons dû revoir notre gestion de l'environnement des types plusieurs fois.

Dans un premier temps, notre environnement des types était un simple set de variables de types. Collectant les variables de types lors de leur création. Hélas cette implémentation ne nous permettait pas de récupérer les nouvelles variables de type créées lors de la récupération des substitutions.

La solution que nous avons mise en place afin de résoudre ce problème a été de récupérer toutes les substitutions induites par le typeur et de les appliquer à l'environnement des types. De plus, nous ajoutons toutes les substitutions induites par la généralisation à la génération d'équations courante.

6 Extensions

Dans cette partie nous allons présenter les différentes extensions que nous avons implémentées dans notre langage. Pour chacune d'entre elles une implémentation de l'évaluateur et du typeur a été réalisée.

6.1 Annotations de type

Afin de simplifier les tests sur notre typeur, nous avons implémenté la possibilité d'annoter les expressions avec leur type attendu.

```
let intId a = (a : int) in (intId 1) (* Type *)
let intId a = (a : int) in (intId (fun a -> a)) (* Type Error *)
```

6.2 Types Utilisateurs

Pour generaliser les cas listes chainées et option, nous avons implémenté la possibilité de définir des types utilisateurs.

```
type 'a list =
  | Nil
  | Cons of ('a * ('a list))

type 'a option =
  | None
  | Some of 'a
```

Au sein du typeur, ces définitions de types sont traduites en source d'instances de type. Ainsi, lors de la vérification de type, les types utilisateurs sont traités comme des types issus d'une généralisation. Pour pouvoir utiliser ces définitions, un environnement est créé au début du programme afin de mettre en association les constructeurs avec leur définitions et le type qu'ils construisent.

6.3 MatchPattern Profond

Afin de pouvoir utiliser correctement les types utilisateurs, nous avons implémenté la possibilité de faire du filtrage de motifs profonds.

Il prend la forme suivante :

```
let t = Some (Some (1)) in
match t with
| Some (Some x) -> x
| Some (Some x) -> (x : (int ref)) (* Ici ne type pas *)
| None -> 0
```

6.3.1 Analyse Statique

L' analyse statique vérifie ici 4 informations :

1. Le type des patterns est bien conforme au type de l'expression filtrée
2. Le type de retour de chaque branche est bien conforme au type de retour de l'expression filtrée
3. Que l'expression de retour de chaque branche n'utilise pas de variables non définies
4. L'absence de doublons dans les variables de motifs

6.3.2 Evaluation

L'évaluation s'est avérée un peu plus complexe, car il a fallu mettre en place un parcours de graphe afin de pouvoir associer les variables de motifs aux valeurs correspondantes dans l'expression filtrée.

6.4 Gestion des erreurs

Nous avons implémenté une gestion des erreurs afin de pouvoir afficher des messages d'erreurs plus explicites.
Il est important de noter qu'on ne parle pas ici d'erreurs définies explicitement par l'utilisateur, mais d'erreurs liées à une mauvaise utilisation, à une détection par analyse statique ou à une erreur d'évaluation.

6.4.1 Analyse Statique

En plus des garenties de typage, nous avons implémenté des erreurs de typage afin de pouvoir afficher des messages d'erreurs plus explicites.

- Utilisation de variables non définies
- Utilisation de constructeurs non définis
- Double utilisation de variables au sein d'un même pattern

Lorsque l'une de ces erreurs est détectée, ou lorsque l'unification échoue, nous affichons un message d'erreur explicite mentionnant la ligne et la colonne de début et de fin de l'expression concernée, de plus nous affichons l'équation qui a échoué.

Exemples :

```
let x = a in ()
```

Affichera :

```
Type Inference:
Error from line 1 col 8 to line 1 col 9:
  Unbound variable a.
```

```
let x = (fun a -> a) in (x : int)
```

Affichera :

```
Type Inference:
Error from line 1 col 33 to line 1 col 36:
  int = (lambda int,int).
```

6.4.2 Erreurs d'évaluation

Nous avons également implémenté des erreurs d'évaluation afin de pouvoir afficher des messages d'erreurs plus explicites.

- Fuite dans les cas de filtrage
- Division par zéro

Comme pour les erreurs de typage, lorsqu'une erreur d'évaluation est détectée, nous affichons un message d'erreur explicite mentionnant la ligne et la colonne de début et de fin de l'expression concernée.

Exemples :

```
(1 / ((fun a -> 0) () )  )
```

Affichera :

```
Evaluation:
Error from line 1 col 0 to line 1 col 26:
  Division by zero : (1 / ((fun a -> 0) ())).
```

```
(match 2 with
 1 -> ()
)
```

Affichera :

```
Evaluation:
Error from line 1 col 0 to line 1 col 22:
  Non exhaustive pattern match on :
(match 2 with
 1 -> ()
)
```

7 Améliorations possibles

Dans cette partie, nous allons présenter les différentes améliorations que nous aurions pu implémenter dans notre langage.

Toutes ces améliorations ont été pensées et spécifiées, mais par manque de temps, nous n'avons pas pu les mettre en place.

Erreurs Utilisateur:

La possibilité de définir des erreurs et d'en générer au sein du programme aurait pu être une amélioration intéressante.

Egalité structurelle :

Nous aurions pu implémenter l'égalité structurelle afin de pouvoir comparer des valeurs de type complexe comme les constructions.

Analyse Statique :

Voici les différentes améliorations que nous aurions pu implémenter dans l'analyse statique :

1. Exhaustivité du filtrage de motif Vérifier que le filtrage de motif était exhaustif. C'est-à-dire que toutes les valeurs possibles de l'expression filtrée sont couvertes par les patterns du filtrage de motifs.
2. Détection de pattern inatteignable Détecter que le filtrage de motif ne contient pas de pattern inatteignable. C'est-à-dire qu'un pattern n'est pas atteignable, car il est précédé d'un pattern qui couvre toutes les valeurs possibles de l'expression filtrée.
3. Détection des variables non utilisées Lors de l'analyse statique, nous aurions pu vérifier que toutes les variables de motifs sont utilisés dans l'expression de retour de chaque branche.
4. Détection des types non utilisés Vérifier que tous les types définis par l'utilisateur sont utilisés dans le programme.

ifier que tous les types définis par l'utilisateur sont utilisés dans le programme. \end{enumerate}