Evaluateur-Typeur de Lambda-Calcul

Réalisé par : Floria Lim (28706087)

Enseignant :
Romain Demangeon

MU5IN555 TAS

Année universitaire : 2024/2025

Table des matières

1 Introduction

L'objectif de ce projet est d'écrire, dans un langage libre, un typeur et un évaluateur pour un λ-calcul intégrant différentes fonctionnalités. Le projet a été codé en Java et réalisé seul, avec l'aide occasionnelle d'une IA générative (ChatGPT) pour clarifier certains aspects difficiles à comprendre, résoudre des erreurs et générer des tests.
Voici quelques exemples de prompts qui ont été utilisés :

- *"Donne des exemples de Pterm que je pourrais tester pour réduire et typer."*
- *"À quelle réduction correspond <Pterm> ?"*
- *"À quel type correspond <Pterm> ?"*

2 Points forts

Toutes les parties du projet ont été implémentées jusqu'à la partie 4, c'est-à-dire l'évaluation et le typage des termes simples et complexes dans le λ-calcul. De nombreux tests ont été conçus pour couvrir le plus de cas possible, y compris ceux nécessitant des conversions alpha.

En plus de cela, plusieurs extensions ont été réalisées :

- **intégration de la gestion de quelques exceptions**, tel que la vérification des occurrences dans l'unification des types (`OccurCheckException`), les erreurs d'unification de types (`UnificationException`) et les timeouts (`TimeoutException`), avec un délai limite réglé à 100ms.
- **prise en charge des types produits et des types sommes**, améliorant ainsi la flexibilité du système de types.
- **implémentation d'un parser** pour l'évaluation et le typage d'un fichier texte contenant des expressions en λ-calcul.

Expression	Description
`(x1 x2)`	Application d'un terme `x1` à `x2`
`fun x -> expr`	Définition d'une fonction anonyme qui prend un argument `x` et renvoie `expr`.
`cons(x1, x2)`	Crée une liste avec en tête `x1` et en queue `x2`.
`nil`	Représente une liste vide.
`head(list)`	Renvoie le premier élément de la liste.
`tail(list)`	Renvoie la liste sans son premier élément.
`ifEmpty(list, x, y)`	Teste si la liste est vide. Si oui, renvoie `x`, sinon `y`.
`fix f = fun x -> expr`	Définition d'une fonction récursive `f`.
`let x = expr in body`	Définition d'une variable `x` dans un corps d'expression.
`pair(x1, x2)`	Crée une paire contenant `x1` et `x2`.
`proj1(pair)`	Accède au premier élément de la paire.
`proj2(pair)`	Accède au deuxième élément de la paire.
`switch(branch, x: x, y: y)`	Renvoie `x` si `branch` vaut `left(x)`, `y` si `branch` vaut `right(x)`.

TABLE 1 – Syntaxe des expressions en λ-calcul pour le parser

3 Points faibles

Un des points faibles du rendu concerne les tests de typage sur les listes de listes, qui échouent avec l'erreur *"Constructeurs incompatibles entre [T2] et N"* dans des cas tels que `((1 :: 2) :: ((3 :: 4) :: ()))`, en raison de la difficulté à implémenter correctement la gestion des

types de listes imbriquées. Une condition claire permettant de surmonter cette incompatibilité entre les types attendus dans les fonctions récursives n'a pas été trouvée dans le cadre de l'implémentation actuelle.

De plus, la partie 5, concernant l'ajout de traits impératifs dans le langage, n'a pas été réalisée. Cette section devait inclure des opérateurs comme `!e`, `ref e` et `e1 := e2`, ainsi que la mise à jour des types pour prendre en compte ces nouveaux constructeurs. Le professeur avait précisé qu'il était possible de remplacer cette partie par l'implémentation d'autres fonctionnalités abordées en cours, c'est pourquoi cette section n'a pas été incluse dans le projet final afin de se concentrer sur d'autres extensions jugées plus accessibles.

Aussi, le parser ne prend pas correctement en charge les applications imbriquées de fonctions, comme dans l'expression `Pterm SKK = new App(new App(S, K), K)`, où la dernière application est omise.

4 Difficultés rencontrées

Les erreurs rencontrées tout au long de l'implémentation ont été nombreuses et parfois complexes à résoudre. La gestion de l'unification des types, en particulier la résolution des erreurs liées aux variables de type et aux substitutions, a été particulièrement difficile, surtout en raison de la difficulté à visualiser chaque étape de l'unification pour vérifier le bon fonctionnement du typage.

De plus, pour tester l'exactitude de l'implémentation, il n'y avait aucun moyen fiable de valider le typage, à l'exception de comparaisons manuelles des résultats obtenus avec des attentes théoriques. Cette approche a introduit une marge d'erreur et une certaine incertitude concernant la validité des résultats obtenus.

Enfin, un autre défi a été la reconstruction du type final à partir du `HashMap` des substitutions. Il m'a été nécessaire d'écrire une fonction `inferType(Ptype t, Map<String, Ptype> substitutions)` pour reconstituer le type final à partir des substitutions, car dans l'unification des types, une substitution peut affecter récursivement plusieurs niveaux d'un type. Contrairement à d'autres langages où les types peuvent être substitués directement, mon implémentation Java gère cette substitution de manière explicite.

5 Conclusion

En conclusion, ce projet a constitué un défi majeur, notamment dans la mise en œuvre du système de typage qui m'a demandé beaucoup de temps et de réflexion. La gestion de l'unification des types et des substitutions a été complexe, mais elle m'a permis de mieux comprendre le fonctionnement de l'inférence de types dans le cadre du λ-calcul, un sujet qui m'était incompréhensible au début de l'année. Cette expérience a ainsi enrichi ma compréhension des concepts abordés en cours et m'a permis de saisir les nombreuses subtilités de l'inférence de types dans le λ-calcul.