

Rapport de Projet

Typeur de lambda calcul

Zakaria Djebbes

Typage et Analyse Statique



November 26, 2022

1 Introduction

Ce rapport a pour but d'expliquer ce qui a été fait et les difficultés rencontrées lors de la réalisation du projet, sans entrer dans les détails (comme demander par les enseignants).

2 Ce qui a été fait

Tous les points du sujet ont été réalisés. Le code est également commenté pour expliquer certains choix.

J'ai cependant ajouté certains éléments pour avoir un langage intéressant !

2.1 Caractères

J'ai dans mon langage la possibilité d'avoir des caractères, pour ce faire il y a un type *TChar* et le terme *Char*

2.2 Les booléens

J'ai rajouté le type *TBool*, ainsi que le terme *Bool*.

J'ai donc des booléens dans mon langage ce qui me permet d'avoir à la place de *IfZero* et *IfEmpty* un *IfThenElse*.

2.3 IfThenElse

Comme indiqué précédemment, j'ai ajouté le terme *IfThenElse* qui prend un booléen et fait un branchement en fonction !

2.4 Opérateurs

J'ai ajouté les opérateurs binaires suivants :

- Tous les opérateurs numériques existent à savoir : $+$, $-$, $*$, $/$, $\%$
- Les opérateurs de comparaison : $==$, $<$, $<=$, $>$, $>=$
- Les opérateurs sur les booléens : $\&\&$ et $||$

2.5 Fonctions internes au langage

J'ai également ajouté des fonctions internes au langage, (qui ne sont pas définies dans une grammaire) :

- **Head** : Tête d'une liste
- **Tail** : Queue d'une liste
- **Not** : Le non logique
- **Empty** : Fonction qui check si une liste est vide

- **NumToChar** : Prends un entier et le transforme en character (Encodage ASCII)
- **CharToNum** : Inverse de NumToChar
- **Fst** : Renvoie le premier élément d'une *Pair*
- **Snd** : Renvoie le deuxième élément d'une *Pair*

2.6 Pair

J'ai à la place des tuples des pairs avec un type *TPair* $< T1, T2 >$. Sachant que $T2$ lui-même peut être une *Pair*.

2.7 Exception

Il y a dans mon langage une forme basique d'exceptions, un terme peut donc se réduire en une exception !

J'ai défini trois exceptions de base, la division par zéro, la tête et la queue d'une liste vide.

On peut également lever une custom exception !

Une exception est de la forme `Exception(MLException, Term)` ou `MLException` est le genre d'exception (`DivideByZero` par exemple) et `Term` le terme qui a généré l'exception.

Ceci implique aussi l'ajout de *raise* pour lever des exceptions.

2.8 TryWith

L'existence des exceptions veut également dire *TryWith*, j'ai donc un bloc `try term with term`, qui si le premier terme est réduit en exception, le deuxième terme est exécuté à la place !

Notez qu'il n'y a pas de type exception, dans mon langage une exception est de type *Unit*. Les exceptions sont donc des erreurs de réduction et non de typage !

2.9 Grammaire

Le langage s'accompagne d'une grammaire à la Fsharp, il y a aussi du sucre syntaxique (Définition des listes avec `[1, 5, 3]` plutôt que `1 :: 5 :: 3 :: []` par exemple !)

3 Difficultés rencontrés

La première difficulté que j'ai rencontrée c'est le fait de n'avoir aucune connaissance préalable des langages fonctionnels et du lambda calcul.

Deuxièmement, le fait de typer le `Let` polymorphe, à savoir devoir ouvrir le `let` polymorphe et appliquer toutes les modifications engendrées dans l'environnement à chaque fois pour ne pas avoir d'erreurs lors du typage d'un `let` lui-même dans un `let`.