Rapport projet 2, où l'on parle de Fouine

François Pitois et Simon Fernandez

1 Présentation

Ce projet est un interprete, compilateur, machine à pile du langage Fouine.

Il a été programmé en CamL (souvent entre minuit et 3h) et aussi en partie en Fouine (de 3h à 5h)

2 Organisation du code

Le code est structuré de la manière suivante :

— main : Point d'entrée du programme, gère les options passées à l'executable et appelle les différentes fonctions
— parser/lexer : Parsent l'entrée. Fournissent un arbre du code
— uncoment : fichier permettant de supprimer les commentaires d'un fichier donné en entrée
— types : concentre tous les types qui seront utilisés partout
— interprete : interprete de l'arborescence générée par lexer/parser, en vérifiant les types
— zinc_compiler : compile l'arborescence vers une suite de lexems qui seront passés à la machine ZINC
— zinc_machine : la machine ZINC qui va traiter les lexems, faire les calculs et renvoyer le resultat. Elle est détaillée ici [1]
— machine_types : les types et les tokens utilisés dans la machine à pile
— de_bruijn : traduit toute une arborescence pou utiliser les indices de De Bruijn
— convert : le convertisseur de Caml + extenstion vers Caml
— exception.fou.ml : Fichier Fouine qui décrit les transformations de Caml + exception vers Caml

3 Options possibles

Nous avonos implémenté les options suivantes :

— Deux niveaux de debug : –verbose/-v –debug/-d
— L'interprete qui vérifie aussi en direct la validité du typage
— La machine ZINC, avec récursivité terminale
— La traduction vers des indices de De Bruijn

— L'export des lexems compilés vers un fichier exterieur
— La traduction Fouine + Exceptions vers Fouine
— Les fonctions prInt et feedback qui donnent des informations en cours d'interpretation

4 Objets traités

— Entiers
— Booléens
— Fonctions
— Recursivité
— Couples

5 Spécificité de l'interprete : les patterns

Pout gèrer les types dans l'interpreteur, nous avons ajouté l'idée des patterns. Les patterns permettent de traiter les fonctions, les paires, et le typage de variables. Ainsi, tout `Let ... = ... in` associe une exception à un pattern.

Donc les expressions du type `let a, b = 1, 2 in ...` fonctionnent parfaitement puisqu'on souhaite matcher la gauche du let avec la droite, et ainsi faire des définitions multiples

L'ajout des patterns a aussi permis de simplifier l'expressions du type `let f x y z = ...` Et éviter la surcharge d'un `fun x -> fun y -> fun z ->...`

Ces patterns ont aussi été utilisés par la ZINC

6 Spécificités de la machine : Zinc et De Bruijn

6.1 Zinc

La machine se base sur l'implémentation ZINC, c'est à dire qu'on simplifie les `let f x y z = ...` pour éviter d'avoir 3 RETURN de fonctions. On utilise la récursivité terminale.

Exemple : En machine classique la fonction de z rende la main à y qui rende la main à x qui rende enfin la main à l'appelant

Mais en recursion terminale, voyant que dès que y aura récupéré la main il la rendra à x, on préfère rendre directement la main à l'appelant, à partir de z

6.2 De Bruijn

On utilise aussi la methode de De Bruijn pour les variables.

L'idée est d'éviter des comparaisons entre chaines de caracteres lors de la recherche c'une variable dans l'environnement.

Tout se passe lors de la compilation, on garde trace de l'environnement à chaque instant et on remplace les `ACCESS(x)` par des `ACCESS(n)` où n est l'indice

de la variable dans l'environnement. On a ainsi une recherche bien plus rapide et une machine plus légère puisqu'elle ne traite plus du tout de chaine de caracteres. Tout est nettoyé lors de la compilation

Références

[1] Xavier Leroy. From krivine's machine to the caml implementations. 2016.