Rapport
Projet de programmation avancée
Un interpréteur pour le langage Lua

Hugo BARREIRO LDD3 Magistère Info

19 mai 2024

Table des matières

1 Introduction

L'ensemble du sujet à été traité. Le projet compile et passe tous les tests sans problème.

Seulement certains fichiers de tests ont été modifiés afin d'en ajouter des supplémentaires (en plus de celui proposé pendant la séance d'aide au projet), hormis celles-ci, aucunes ressources de l'archive n'ont été modifiées.

J'ai essayé autant que possible d'écrire du code simple, concis et élégant tout en restant efficace.

2 Choix

2.1 Affichage

Pour l'affichage j'ai choisi de construire complètement la chaîne de caractères avant de l'afficher. Ainsi, on peut utiliser les fonctions `map` ainsi que `concat` ou `join` respectivement en OCaml et en Rust.

En OCaml CPS, je fais exactement la même chose sauf que je n'utilise pas la fonction `map` pour évaluer les arguments mais une fonction annexe que j'ai écrite pour l'occasion.

2.2 Les opérateurs binaires

Le traitement des opérateurs binaires étaient plus subtil que prévu pour obtenir un code à la fois simple, concis et élégant.

En effet, on pourrait d'abord penser à matcher sur l'opérateur pour savoir quelle opération effectuer puis dans chaque cas interpréter chaque valeur et effectuer l'opération. Cette solution permettrait de s'en sortir avec un seul match. Néanmoins, elle n'est pas satisfaisante car on duplique du code en quelque sorte car dans chaque cas on interprète les deux valeurs.

On pense alors à interpréter d'abord les valeurs puis à matcher sur les opérateurs. Or, cela est incorrect à cause du `and` et du `or`. En effet, ces opérateurs doivent être paresseux.

Ainsi, j'ai choisi d'interpréter la première valeur puis de matcher sur ces deux opérateurs, puis dans tous les autres cas de re-matcher sur les autres opérateurs et pour les cas de `and` et `or` de faire `assert false` car ces deux cas sont inatteignables.

On se retrouve alors avec un match imbriqué ce qui n'est pas totalement satisfaisant mais cela permet quand même d'obtenir un code plus concis et lisible.

En Rust la construction `if let` est bien pratique ici car elle permet de traiter les cas `and` et `or` en dehors du match et ainsi d'en avoir qu'un seul. Cette solution est la plus satisfaisante.

2.3 Interprétation des expressions

Lorsque l'on passe à la forme CPS, on ne doit pas renvoyer les valeurs des expressions mais les donner en argument de la continuation k. Ainsi, pour les expressions simples comme les entiers ou encore les booléens, l'appel à k se fait immédiatement et non dans une autre continuation.

Ainsi, on pourrait avoir envie de séparer les expressions avec un appel à k "externe" et "interne". Or, cela implique de se retrouver avec un match imbriqué, avec dans celui-ci un autre match imbriqué pour les opérateurs binaires.

C'est pour ces soucis de simplicité et de lisibilité que j'ai choisi de ne pas séparer les appels à k "externe" et "interne" et d'appeler k dans chaque cas au lieu de l'écrire qu'une fois après le match pour les appels à k "externe".

Cependant pour les opérateur binaires, j'ai choisi d'effectuer ce choix car ils sont déjà tous regroupés ensemble dans un match imbriqué. Il est de même pour les opérateurs unaires.

2.4 Interprétation des arguments

Lors de l'interprétation d'une closure, on doit interpréter les arguments. En OCaml simple et en Rust cela est facile grâce à la fonction `map` mais en OCaml CPS elle ne fonctionne plus.

Ainsi, j'ai implémenté une fonction annexe qui prend en paramètre l'environnement, la coroutine courante, la liste des arguments à évaluer et la continuation courante. Cette fonction évalue les arguments un par un, puis appel la continuation courante avec la liste dont les éléments ont été évalués.

Cette fonction me permet de récupérer de manière satisfaisante l'évaluation des arguments dans l'implémentation des closures, de print, et des coroutines.

2.5 Création de l'environnement local

En OCaml et en OCaml-CPS cela consiste à implémenter la fonction `create_scope`. Au départ on pourrait penser qu'il suffit d'utiliser `List.iter2` pour remplir la `Hashtbl` mais cela n'est pas correct car en Lua il est autorisé d'appeler une fonction avec plus ou moins d'arguments que de nombre de paramètres qu'elle possède.

Ainsi, ma solution est de parcourir les deux listes en parallèle et de s'arrêter dès que la liste des noms des variables est vide ; et tant qu'elle n'est pas vide, d'ajouter une

valeur si la liste des valeurs n'est pas vide ou d'ajouter `Nil` si ce n'est pas le cas.

Avec cette solution il est facile de gérer la possible différence entre le nombre de paramètres et d'arguments. Il suffit juste d'appeler `create_scope` avec nos données et la fonction gère ce problème. Ainsi, il n'est pas nécessaire de regarder la longueur des listes des noms et des valeurs afin d'ajuster si besoin avant d'appeler `create_scope`, ce qui permet de gagner en efficacité et en lisibilité.

Pour ce qui est de Rust, j'ai adopter la même idée à quelques détails techniques près liés à la programmation en Rust. La fonction `extend` ne prend pas deux listes en paramètre mais un tableau pour les noms et un itérateur pour les valeurs. Afin de reprendre la même idée pour les mêmes raisons de simplicités, je créer un itérateur infini de `Nil` que je concatène à celui des valeurs. Puis, grâce à la fonction `zip`, j'itère sur les noms et les valeurs dans une boucle afin d'ajouter toutes ces paires dans la `hashmap`. On obtient le bon comportement grâce à la fonction `zip` car elle coupe l'itérateur à la plus petite longueur entre celle des noms et des valeurs. Or, comme l'itérateur des valeurs est infini, on parcourt exactement les noms auxquels on associe la valeur correspondante ou `Nil` s'il n'y en a pas.

Ce code doit être un peu moins efficace que celui réalisé en OCaml car on doit travailler sur le tableau et l'itérateur avant de les parcourir. En effet, on doit concaténer l'itérateur des valeurs avec un itérateur infini, puis transformer le tableau des noms en itérateur puis zip les deux ensemble. La solution reste, à mon avis, élégante et agréable à utiliser en pratique.

3 Difficultés

3.1 Lisibilité

Une première difficulté a été de rendre le code lisible.

En effet, on peut arriver très vite à des matchs imbriqués ce qui rend moins compréhensible le code. Pour éviter d'en arriver là, on peut utiliser en Rust des `if let` et aussi les fonctions `is_none` et `unwrap`. En OCaml, j'ai par exemple utilisé la fonction `Option.value` qui permet de renvoyer, si l'option est `Some` alors sa valeur, sinon une valeur par défaut.

La forme CPS peut ne pas faciliter la lecture, notamment lorsque l'on doit imbriqués plusieurs continuations. Heureusement qu'il existe l'opérateur `@@` qui permet de supprimer les parenthèses autour des continuations. Cela permet aussi à ocamlformat de proposé une bien meilleur indentation.

Parfois l'on souhaite appliquer plusieurs transformation à une même donnée, c'est par exemple le cas avec l'affichage où l'on doit evaluer les arguments, les transformer en chaîne de caractères puis les concaténer. On peut effectuer toutes les transformations en imbriquant des fonctions mais cela ajoute beaucoup de parenthèses et rend moins

lisible le code. On peut sinon utiliser des `let` pour chaque étape intermédiaire mais cela introduit beaucoup de variables superflues. J'ai donc décidé d'utiliser l'opérateur `|>` qui permet de chaîner les opérations de manière lisible.

3.2 Forme CPS et coroutine

Une autre difficulté a été de comprendre le fonctionnement des continuations.

Cela a pu paraître compliqué au départ notamment au niveau de la création d'une coroutine ; mais une fois compris, c'est plutôt simple.

Néanmoins la forme CPS peut avoir un bug un peu pervers. En effet, les codes suivants ne sont pas équivalents :

```
| FunctionCall f ->
  interp_funcall env co f (fun _ -> ());
  k ()

| FunctionCall f ->
  interp_funcall env co f (fun _ -> k ());
```

Ces deux codes passent les tests sans coroutines, mais le premier rend un résultat erroné pour les coroutines.

Au départ, je pensais que ces deux implémentations étaient équivalentes, et ainsi dans `interp_stat` j'évaluais les stats puis je faisais l'appel `k ()` dans le but d'écrire une fois cet appel et non dans chaque cas du pattern matching. J'ai dû corriger ce bug pour que les tests incluant les coroutines passent.

4 Conclusion

Ce projet a été fort enrichissant pour plusieurs raisons.

Tout d'abord, cela a été l'occasion de se familiariser davantage avec Rust. En effet, contrairement aux premiers TP, l'écriture de l'interpréteur à été agréable, notamment grâce à la structure du code donnée dans l'archive.

De plus, ce fût ma première fois avec la forme CPS. Bien que parfois moins lisible que du code plus naïf, elle est particulièrement intéressante pour transformer du code récursif non-terminal en code récursif terminal. Cela me sera plus qu'utile lors de mon stage consistant à implémenter des machines abstraites de manière efficace en OCaml via la mise en forme CPS et la défonctionnalisation.

Enfin, j'ai pu approfondir mes connaissances en langage de programmation et en programmation plus globalement ; ce qui n'est pas pour me déplaire puisqu'il s'agit d'un domaine de l'informatique qui m'intéresse énormément.