Rapport Projet 2
Où l'on parle de rongeurs

Guillaume Coiffier - Léo Valque

2017

Table des matières

Première partie

Généralités

Remarques générales

- **Niveau du binôme :** Intermédiaire
- **Adresse du dépôt Git :** `https://github.com/GCoiffier/Projet-2`
- Les fichiers de tests sont situés dans le dossier Programs. Les fichiers contenant du code fouine ont une extension `.ml`
- Les fichiers compilés pour la machine SECD sont situés dans le dossier Stack_Programs. Ils ont une extension `.code`

1 Comment utiliser notre programme

- Pour compiler le programme, utilisez simplement la commande `make`. Celle-ci crée un exécutable appelé *fouine*.
- Pour nettoyer le répertoire de travail, utilisez la commande `make clean`.
- `./fouine fichier` exécute le code contenu dans *fichier* et renvoie le résultat de ce code (qui doit être un entier)
- `./fouine -debug fichier` commence par afficher le code parsé dans la console, puis exécute le code et affiche le résultat.
- `./fouine -interm sortie fichier` compile le code parsé et le stocke dans sortie. Si aucun fichier de sortie n'est spécifié, le programme affichera le code dans la console.
- `./fouine -machine fichier` compile le code parsé et effectue l'interprétation mixte : ce qui peut être exécuté sur la machine à pile y est exécuté, le reste fait appel à l'interpréteur standard. Le code en entrée doit être un code fouine.
- `./fouine -execute fichier` compile le code parsé et l'exécute sur la machine à pile. Ce code doit être dans le langage de la machine à pile.

NB : il est dans tous les cas possibles de ne pas donner de fichier d'entrée à fouine. Le programme s'éxecute alors en mode interactif et il faut entrer un programme dans la console.

Exemples :

```
> ./fouine
> ./fouine Programs/factorielle.ml
> ./fouine -debug Programs/function.ml
> ./fouine -interm
> ./fouine -interm toto.code Programs/prog1.ml
> ./fouine -execute toto.code
> ./fouine -machine Programs/prog2.ml
```

```
> ./fouine -debug Programs/function.ml
```

2 Fouine

2.1 Fouine

La fouine (Martes foina) est une espèce de mammifères carnivores d'Europe et d'Asie, au pelage gris-brun, courte sur patte et de mœurs nocturnes. C'est une martre (ou marte) faisant partie de la famille des Mustélidés, au même titre que la belette, le blaireau, la loutre, le putois ou le furet, petits mammifères carnivores se caractérisant souvent par leur odeur forte.

FIGURE 1 – Fouine photographiée en hiver en République tchèque.

Les fouines sont des animaux solitaires, comme la plupart des autres espèces de martres. Elles évitent leurs congénères en dehors des périodes de reproduction. Il s'agit d'animaux territoriaux qui marquent leur territoire avec des secrétions et le défendent au moins contre d'autres fouines de même sexe. La grandeur du territoire est variable, mais reste inférieure à celui de la Martre des pins. Leur grandeur va de 12 à 21cm et varie en fonction du sexe (les territoires des mâles sont plus grands que ceux des femelles), de la saison (ils sont plus petits en hiver), de l'habitat (ils sont plus grands en campagne qu'en ville) et de la nourriture disponible. Leur activité est surtout nocturne. L'espérance de vie de la fouine est d'approximativement douze ans.

Extraits de l'article de Wikipedia [1]

2.2 Le langage fouine

Le langage fouine est un langage de programmation constituant un sous-ensemble du langage Ocaml. Il comprend notamment :

- Les expressions arithmétiques et booléennes
- Les déclarations de variables et de fonction (syntaxe `let ... = ... in ...`)
- Les branchements conditionnels
- Les fonctions récursives (syntaxe `let rec ... = ... in ...`)

Mais, à la différence d'Ocaml, il ne comprend pas :

- De typage : toutes les fonctions prennent en argument des entiers, et renvoient des entiers
- De structures de données telles que les listes
- De filtrage par motif
- La couche orientée objet

Cependant, nous verrons que fouine peut être étendu avec certaines fonctionnalités, notamment un système de gestion d'exceptions et des références.
Ce langage ne figure cependant pas dans la liste des 700 langages de programmation proposée par Peter J. Landin [7].

3 Avancement du projet en fonction du temps

Rendu 2

- **Semaine 1**
  - Lexer et parser de base
  - Définition d'un type programme pour les programmes fouine
  - Interprétation des fichiers fouine sans les fonctions (ie expressions arithmétiques, booléennes, `if ... then ... else` et `let ... in`)
- **Semaine 2**
  - Interprétation des fichiers fouine avec des fonctions
  - Interprétation des fichiers fouine avec des fonctions récursives
- **Semaine 3**
  - Gestion des exceptions
  - Gestion des références
  - Gestion du `begin...end` et du `let _ = ...`
  - Ajout de primitives "bonus" `prStr` (qui renvoie 0 et affiche une string) et `prNl` (qui passe une ligne et renvoie aussi 0)

Rendu 3

- **Semaine 4**
  - Correction des exceptions (on utilise plus `try ... with` de Caml)
  - Implémentation des tableaux
  - Compilation des expressions arithmétiques vers une machine à pile
  - Exécution des expressions arithmétiques sur une machine à pile
- **Semaines 5 et 6**
  - Corrections de bugs du retour du rendu 2 (réferences, ordre d'execution des fonctions, exceptions)
  - Travail sur le main. Le programme peut désormais lire l'entrée standard dans le cas où on ne donne pas de fichier en argument
  - Extension de la machine à pile qui gère les variables et les branchements conditionnels

Rendu 4

- **Semaine 6**
  - Ajout d'un lexer, parser pour les instructions de la machine à pile, on peut desormais compiler puis executez plus tard
  - Ajout des fonctions dans la machine à pile
- **Semaine 7**
  - Ajout des fonctions recursives dans la machine à pile
  - Implémentation des indices de bruijn dans la machine à pile
  - Ajout de l'interpréteur mixte qui envoie sur la machine quand il peut et sinon éxecute normalement
- **Semaine 8**
  - Correction de bugs sur l'interprétation mixte. Implémentation du transfert d'environnement entre fouine et SECD
  - Rédaction du présent rapport.

Deuxième partie

Le projet

4 Organisation du code

La plupart des fonctionnalités de notre projet sont encapsulées dans des modules. Ainsi, l'interpréteur fouine et la machine SECD constituent deux modules que l'on charge dans le fichier main. Nous avons privilégié cette approche pour deux raisons. D'une part par souci de propreté : on minimise le nombre de primitives que l'on peut appeler depuis l'extérieur du module à l'aide des mécanismes de signature (ce qui permet de plus d'inclure de la documentation dans les dites signatures), d'autre part pour rendre le code plus facilement maintenable, dans ce projet où certaines fonctions ont été réécrites plusieurs fois.

4.1 Liste des fichiers

- **main.ml :** Fichier principal. Lit les argument envoyés au programme et fait les différents appels aux différentes parties du code.
- **fouine_type.ml** Fichier contenant les définitions des types représentant un programme fouine dans Caml.
- **interpreteur.ml :** Le fichier contenant le module au coeur de fouine . Contient ma grosse fonction d'interprétation d'un programme fouine , la fonction `debug`, qui permet d'afficher un programme fouine parsé, ainsi que de quoi réaliser l'exécution mixte.
- **machine.ml :** Module implémentant la machine à pile.
- **environnement.ml** La définition du module `Environnement`, construit à l'aide d'un dictionnaire.
- **dictionnaire.ml** La classe de dictionnaire utilisée. Basée sur une table de hachage.
- **lexParInterface.ml :** Fichier faisant l'interface entre les parser et le reste du code. Fournit simplement des primitives `read_prgm <fichier>`.
- **parser.mly , lexer.mll** Ces fichiers permettent de lire la formule donnée en entrée par le programme, et de construire un objet de type programme représentant le code à exécuter.
- **parsMachine.mly , lexMachine.mll** Ces fichiers permettent de lire les instructions donnée en entrée par le programme, et de construire un objet de type `instruction list` représentant le code à exécuter sur la machine à pile.

4.2 Liste des programmes fouine donnés en exemple

- exemples simples sur les différentes fonctionnalités de fouine (les noms des programmes sont a priori explicites)
- fibonacci récursif stupide
- fibonacci en temps linéaire avec références
- factorielle
- tours de Hanoi
- crible d'erathosthène (illustration des tableaux)

4.3 Bugs détectés et non corrigés

- la traduction d'environnement fouine vers SECD suppose qu'une fonction est récursive. Du coup quelquechose défini par `let f x = x in let f x = f x in f 2` ne s'exécutera pas correctement, il bouclera au lieu de renvoyer 2.

5 L'interpréteur fouine

5.1 L'interprétation

L'interprétation est réalisée dans la fonction execute du fichier `interpreteur.ml` . Cette fonction prend en argument un programme fouine parsé (de type programme) et renvoie un entier. On utilise une fonction récursive auxiliaire qui associe une valeur de type `ret` au programme. Ensuite, on appelle la petite fonction `return` qui renvoie un `int` à partir de ce `ret`.

Ce type intermédiaire `ret` est nécessaire pour pouvoir utiliser des fonctions dans les programmes fouine . C'est le type des éléments qui sont associés à nos programmes dans l'environnement (`Env.elt`). Cela permet de renvoyer en interne autre chose que des entiers (ce qui est nécessaire pour les références et les fonctions). De plus, un filtrage par motif dans la fonction return permet de détecter les erreurs dynamiques (entier + fonction) ou (entier + ref) et d'obtenir des messages explicites.

5.2 Structures de données

On utilisera des tables de hachage (Hastbl) qui associeront une valeur `Env.elt` à une valeur `programme`. Cette structure de donnée a le gros avantage de gérer parfaitement la portée des variables. En effet, d'après la documentation de Ocaml, lorsqu'une valeur y est assignée à la variable x dans une Hashtbl, l'ancienne valeur de x est remplacée par y. Lorsque l'on supprime l'association (x,y) dans la table, l'ancienne valeur associée à x est restaurée, ce qui est le comportement attendu.

Cette structure de donnée supporte l'ajout d'un couple d'élément, la suppression d'un élement et de son association (avec éventuelle restauration de la précédente association), la recherche d'un élement et la copie (pour pouvoir faire des clotures).

Le module `Environnement` contient de plus la fonction `transform_env`, qui transforme l'environnement de l'interpréteur en un environnement lisible par la machine à pile.

5.3 Fonctions et fonctions récursives

Pour l'implémentation des fonctions, on a ajouté un constructeur `Cloture` au type `Env.elt`. Un cloture comprend l'expression de la fonction (le A de `let f x = A in`) et une copie de l'environnement au moment de la définition de la fonction. Cette copie est "brutale" : on copie l'intégralité de l'environnement sans chercher à savoir quelles valeurs sont inutiles.

Dans le cas des fonctions récursives, on ajoute à la clôture crée... elle-même. Ainsi, on évite de faire autant de clôtures que d'appel récursif. Cela ne pose pas de problème tant qu'il existe un cas de sortie à la fonction récursive.

5.4 Les exceptions

Les exceptions sont gérées grâce à l'ajout d'un booléen au type de retour de l'interpréteur. Ce booléen vaut vrai si une exception a été levée durant l'éxecution d'un bout de programme. Ainsi, seule l'instruction `raise` renvoie un booléen vrai. Le `try ... with` éxecute son premier argument normalement. Si une exception a été levée, alors on revient à l'ancien environnement et on exécute la partie `with ...` en prenant soin d'ajouter la valeur renvoyée par le raise à l'environnement.

Tous les autres constructeurs ont été modifiés à cet effet : soit ils obtiennent récursivement un résultat sans exception, et tout se passe normalement, soit ce résultat renvoie une exception, et ils ne font alors que la propager. C'est ce qui permet à la valeur de l'exception de remonter jusqu'au dernier `try`.

Si un `raise` est executé à l'extérieur d'un `try`, on vérifie que le booléen est vrai en sortie de la fonction d'évaluation, et donc on peut planter en annonçant une exception non rattrapée.

Historique de l'implémentation : Les exceptions ont jadis été implémentées à l'aide du mécanisme de gestion d'exceptions de Ocaml (aux alentours du rendu 2). Comme c'était un peu de la triche, nous avons, lors du rendu 3, implémenté une une gestion d'exceptions basé sur une pile d'environnement. Le soucis était que nous ne pouvions pas interrompre l'execution d'un code de cette façon. Par exemple, le code `try 3 + raise E 2 with E x -> 3;;` renvoyait 6, car le `raise` renvoyait le résultat du code situé après le `with` et ce qui se trouvait à l'intérieur du `try` continuait d'être exécuté.
Nous sommes donc passé à ce système de propagation de levée d'exception à l'aide de booléens qui allourdit certes la fonction d'interprétation, mais résout le problème.

5.5 Aspects impératifs et tableaux

Les références se font sur des entiers uniquement. Elles sont implémentées en ajoutant un constructeur `Ref` au type `Env.elt` et sont stockées en tant que références de Caml dans l'environnement. Les trois opérations (déclaration, assignation et déréférencement) se font alors naturellement.

Les tableaux ont été implémentés quasiment comme les références. Le type `Env.elt` a été enrichi d'un constructeur `Array` contenant un tableau. Les trois opérations (création de tableau, assignation à un indice, lecture d'une case) se font alors naturellement.

6 La machine à pile SECD

La machine à pile est implémentée par la fonction `step` du fichier `machine.ml`. Cette fonction prend en argument une pile d'instruction machine (de type `instruction list`) et affiche un entier lorsqu'il a terminé. Chaque `step` lit et exécute l'instruction au sommet de la pile.

La compilation du code parsé est effectuée par la fonction `build` situé dans le même fichier. Cette fonction prend en argument un programme fouine parsé (de type `programme`) et renvoie une pile d'instruction machine (de type `instruction list`).

La machine fonctionne avec des indices de Bruijn, calculés au moment de la compilation par la fonction `find_bruijn`

7 Interface et interprétation mixte

La machine à pile ne permettant pas d'exécuter n'importe quel code de fouine enrichi, il nous a été demandé d'implémenter un interpréteur mixte fouine /SECD. Cet interpréteur consistait à repérer dans le code les morceaux qui sont "purs" et les envoyer à la machine SECD, le reste étant géré normalement par l'interpréteur fouine .

7.1 Pureté du code

Pour détecter les morceaux purs d'un code, il a fallu ajouter un constructeur `Pure` dans le type `programme`. Ainsi, un code est pur s'il est de la forme `Pure(x)`.
Ensuite, la fonction `label_pure_code` du module `Interpreteur` permet de marquer les bouts de codes purs.

NB : Comme les fonctions étaient déjà implémentées sur la machine à pile avant le début du rendu 4, nous avons décidé de rendre les fonctions pures, contrairement à la définition de pureté donnée dans l'énoncé. Cela nous a vallu quelques difficultés supplémentaires Dans cette fonction, on propage une liste des variables et fonctions qui ont été déclarées comme pures, afin de déterminer quels appels de fonction sont purs. Les fonctions récursives sont gérées de façon particulière : comme un appel à cette fonction est présent dans son expression, on part du principe que la fonction est pure. Si l'appel récursif nous dit le contraire, c'est qu'elle ne l'était pas, et on ne l'ajoute donc pas à la liste des fonctions pures.

7.2 Interprétation mixte

Pour l'interprétation d'un programme où les codes purs ont été marqués, il a suffit d'ajouter un cas de filtrage par motif à l'interpréteur : si le code est pur, on l'envoie sur la machine à pile avec l'environnement courant au lieu de continuer l'interprétation récursivement.

Conclusion

Ce rapport est un peu long, et le mentionner ici le rend encore plus long.

Références

[1] Fouine. Wikipedia, `https://fr.wikipedia.org/wiki/Fouine`.

[2] Eddy Caron, editor. *Recueil des projets intégrés de l'année*, volume 1 of *Best sellers du DI*. ENS de Lyon, 2016.

[3] Les M1 du DI. Notre beau projet intégré. In Caron [2], pages 10–22.

[4] Donald Ervin Knuth. *The art of computer programming, Volume I : Fundamental Algorithms, 3rd Edition.* Addison-Wesley, 1997.

[5] Donald Ervin Knuth. *The art of computer programming, , Volume III, 2nd Edition.* Addison-Wesley, 1998.

[6] Donald Ervin Knuth. *The art of computer programming, Volume II : Seminumerical Algorithms, 3rd Edition.* Addison-Wesley, 1998.

[7] Peter J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3) :157–166, 1966.