



Second demi-projet, rendu final (4)

Pour le 16 mai à 23h59

par mail à

aurore.alcolei@ens-lyon.fr,
daniel.hirschhoff@ens-lyon.fr,
bertrand.simon@ens-lyon.fr

1 Préambule

Que vous soyez **D**, **I** ou **A**, ce sujet est écrit en faisant l'hypothèse que tout a été traité lors du rendu 3.

Si tel n'est pas le cas, envoyez rapidement un mail à daniel.hirschhoff@ens-lyon.fr pour indiquer ce qui manque dans votre rendu 3 par rapport au menu escompté.

Le but étant de ne pas trop "charger la barque", et de maintenir une exigence raisonnable.

2 Réparer **D I A**

Continuez à tester votre programme, et réparez les bugs que vous découvrez ou qui ont été communiqués, afin que le rendu final soit le plus correct possible.

3 **D** : L'interprète

Ajoutez à votre implémentation de **fouine** le traitement des exceptions. Reportez-vous aux énoncés des rendus précédents pour la description de cette extension de **fouine**.

À noter qu'une exception **fouine** ne pourra être traduite vers une exception Caml. Il faudra au contraire ajouter un argument à la fonction qui fait l'interprétation, afin de gérer les exceptions. Cet argument fonctionne comme une pile : lorsqu'on fait un **try**, on empile... quelque chose, on ne vous dit pas quoi. Et on ne vous dit pas non plus quand est-ce qu'on dépile.

Si tout marche, vous pouvez vous attaquer aux références, mais cette partie est optionnelle.

4 **I** : Combiner interprète et machine

Dans un premier temps, ajoutez un environnement à la machine que vous avez construite pour le rendu 3, afin de pouvoir gérer les expressions arithmétiques avec **let.. in** (seulement pour les variables entières, pas de **let f = fun x -> ..**). Ajoutez aussi le traitement de la construction **prInt**.

On appelle dans ce qui suit "*pur*" un programme **fouine** qui ne comporte que les expressions arithmétiques, **let.. in** et **prInt**.

Ensuite, il vous est demandé de modifier l'option **-machine** telle que fournie pour le rendu 3. Lorsqu'on exécute un programme **fouine** avec cette option, c'est en fait l'interprète qui tourne, *sauf pour les sous-programmes purs*, où l'on s'appuiera sur la machine.

En d'autres termes, il vous faudra être capable de détecter si un bout de code est pur (au sens où il est exécutable avec la machine). Il serait fort naïf de faire cela au cours de l'interprétation : il vous est demandé au contraire de faire une analyse préalable, où vous annotez le code reçu en entrée de manière à ce que les

sous-arbres purs soient identifiés. Pour ce faire, vous devrez sans doute modifier légèrement le type qui sert à représenter les programmes `fouine`. L'enjeu étant bien entendu que ces modifications ne fassent pas trop de dégâts dans l'ensemble du code.

Une fois cette “analyse de programme” élémentaire implémentée, vous modifierez le code de l'interprète de manière à lancer une exécution de la machine sur les sous-arbres purs. Pensez, comme toujours, à tester vos programmes. Vous pouvez aussi avoir les deux versions de l'interprète qui coexistent, pour que `fouine` puisse vérifier tout seul que le résultat est le même dans les deux cas (si le résultat est le même, on ne dit rien, sinon tout s'arrête avec un message d'erreur indiquant les deux résultats obtenus).

À noter que l'option `-interm` est fondamentalement inchangée par rapport au rendu 3, à ceci près que l'ensemble de départ s'est légèrement accru : si le programme est pur, on compile ; sinon, on renvoie un message d'erreur.

5 A : `fouine` typée

En principe, le langage `fouine` est non typé (contrairement à Caml), et le programme `if 3>2 then (fun x -> x*x) else 12` s'exécute normalement.

Il vous est demandé pour le rendu final de munir `fouine` de types, ce que d'aucuns ont déjà fait. Votre menu est l'item le plus petit dans l'énumération qui suit qui ne soit pas déjà présent dans le rendu 3.

1. **Vérification de types.** Ajoutez la possibilité d'écrire des annotations de type au langage, ce qui pourra avoir pour effet de rejeter des programmes pour des raisons de typage.

Inspirez-vous pour ce faire de la syntaxe de Caml, qui permet par exemple d'écrire

```
let f (x : int) : int = x+1 in f 2.
```

Un programme comme `fun (f:int->int) -> (f 4)` sera accepté,

de même que `fun (g:int -> int -> int) -> fun x -> g x 4.`

Le programme `let f (x:int) = x in f (fun z -> z)` sera en revanche refusé.

À noter que l'utilisateur choisit de passer ou pas par la vérification de types : les programmes sans annotation de type se comportent comme précédemment (et des programmes mal typés peuvent être exécutés).

Vous ne devez pas traiter le polymorphisme (les “`'a`” de Caml), mais vous pouvez si ça vous chante.

2. **Inférence de types.** `fouine` sait “deviner” les types, et proteste lorsqu'un conflit lié au typage est découvert. On cherche ici à typer tous les programmes, et donc du code mal typé est rejeté. L'utilisateur n'a pas à saisir des annotations de type (vous pouvez ajouter cette possibilité si vous le souhaitez, mais ça n'est pas demandé).

Là aussi, vous pouvez ne pas traiter le polymorphisme. Deux remarques à ce propos : si l'utilisateur entre `fun x -> x`, vous pouvez décider d'accepter le programme car il a par exemple le type `int -> int`. Par ailleurs, il se peut que votre système de types soit plus restrictif que celui de Caml (au sens où il rejette des programmes que Caml accepte), du fait de l'absence de polymorphisme.

3. **Modules et Signatures.** Enrichissez `fouine` avec une fraction du système de modules de Caml, en ajoutant `sig` et `struct` (et `module` et `module type`).

Vous pouvez aussi songer à ajouter

— les modules paramétrés : il s'agit donc de programmer un petit langage de modules, une sorte de “`fouine` sur la `fouine`”.

— les types abstraits et la construction `with`.

Veillez *scrupuleusement*, en tout cas, à procéder par étapes, pour être sûrs d'avoir quelque chose qui fonctionne bien au moins sur un sous-ensemble du langage de modules que vous avez traité.

Autre extension possible : la compilation séparée et les `open` de Caml, avec le code engendré pour la machine SECD.

Là aussi, vous pouvez ne pas traiter le polymorphisme.

6 Spécification : exécutable, options, etc.

Le suffixe pour les fichiers en `fouine` est `.ml`. L'exécutable s'appelle `fouine`.

Options.

Sans options, le programme est exécuté avec l'interprète.

Il affiche le résultat de l'évaluation, à savoir :

- soit un entier,
- soit une fonction (ou plutôt une clôture) : si vous avez exécuté la machine à pile, vous afficherez le code (de la machine) correspondant ; sinon, vous afficherez un programme Caml correspondant au résultat du calcul.

Le programme sans options s'utilisera de la manière suivante : `./fouinetoto.ml`

-debug Cette option aura pour effet d'afficher le programme écrit en entrée, en plus de l'affichage qui est engendré si on n'appelle pas **-debug**.

Vous pouvez bien sûr afficher davantage d'informations lorsque l'utilisateur choisit cette option, qui permettront de suivre l'exécution du programme.

-machine compile le programme vers la machine, puis l'exécute. Si vous avez codé plusieurs versions de la machine, l'option **-machine** prendra un argument, et vous décrierez cela dans le fichier README.

-interm toto.code engendre le programme compilé, dans le fichier `toto.code`, sans l'exécuter.

-autotest exécute le programme `fouine` de deux (au moins) manières différentes (avec l'interprète et la machine, ou avec l'interprète et l'interprète hybride), et renvoie le résultat. Renvoie un message d'erreur *informatif* si les résultats diffèrent.

-E Applique la transformation pour éliminer les exceptions, puis exécute le programme ainsi obtenu à l'aide de l'interprète (rendu 2).

En combinant **-E** et **-debug**, on peut visualiser le programme obtenu après avoir appliqué la transformation (même chose pour **-R** et **-ER**).

-R Même chose, pour la transformation pour éliminer les aspects impératifs.

-ER La combinaison des deux transformations

Les quatre dernières options peuvent être combinées avec l'option **-machine** pour faire tourner le programme résultant à l'aide de la machine plutôt qu'avec l'interprète.

Des tests. Fournissez un répertoire avec des programmes de test que vous avez soumis à `fouine`. Veillez à ce que les tests couvrent l'ensemble des aspects que vous traitez (lex-yacc, exécution des divers composants du langage `fouine`).

Indiquez s'il y a des tests qui échouent, en raison de bugs pas encore résolus.


7 Le README qui termine le demi-projet.

Le fichier README sera à préparer dans le même esprit que ce qui a été fait pour les rendus précédents.

En plus du retour sur le rendu, ajoutez quelques mots sur l'ensemble du demi-projet :

- ce qui a marché, ce qui n'a pas marché, au long des rendus 2, 3 et 4.
- Les bugs importants que vous avez rencontrés : ceux que vous avez pu corriger, ceux que vous n'avez pas été en mesure de corriger (vous pouvez, le cas échéant, indiquer qu'un bug s'est manifesté trop tard vis-à-vis de la date limite pour que vous ayez pu lui régler son compte).
- Toute autre remarque sur l'ensemble de votre demi-projet.

8 ...et ça n'est pas tout à fait terminé

 En plus du rendu, vous devrez faire un bref compte-rendu sur le projet. On vous demande d'envoyer avec le dernier rendu un court rapport (entre 2 et 5 pages), rédigé en L^AT_EX, qui sera accompagné

d'une présentation de 5 minutes, *avec des transparents*, lors de la séance du 17 mai.

Vous trouverez sur la page [www](#) du cours des exemples simples de fichiers desquels vous pourrez partir. *Lisez ces fichiers*, pour savoir comment s'en servir, et comment engendrer un fichier au format pdf.

Mettez rapport et présentation dans un sous-répertoire dédié.

Passage obligé. Il vous faudra nécessairement mentionner le nom de Peter Landin dans votre rapport, en faisant référence à son papier sur les 700 langages, par le biais d'une citation comme celle-ci [1].

Pour cela, il vous faudra éditer le fichier `ex-biblio.bib`, en y insérant les données au bon format (le format BibTeX) pour l'article de 1966. Comment trouver ces données ? Par exemple sur le site DBLP. Une fois trouvées les informations, copiez-collez-les directement dans le fichier `.bib` (des sites comme DBLP permettent d'exporter les données au format BibTeX).

Rien de bien difficile dans tout cela, mais il faut l'avoir fait au moins une fois avant de partir en stage.

Pourquoi ce compte-rendu ? Soyons lucides, l'objectif essentiel de ce compte-rendu est de s'assurer que vous avez une certaine familiarité avec les outils mis en jeu. Du point de vue du contenu, ne perdez pas de temps à raconter ce que tout le monde sait (ce qu'est une fouine, qu'est-ce qu'un environnement, quelles options propose votre programme). Concentrez-vous sur ce qui est propre à votre travail sur le programme **fouine** : comment il est fabriqué, ce qu'il sait bien faire, ce qu'il devrait savoir faire, ce qui a été facile/difficile à réaliser au cours du semestre, etc. (bref, rendez-vous compte vous-même de ce qu'il est pertinent d'exposer).

Cet aspect des choses ne comptera pas pour beaucoup dans l'évaluation du projet, mais jouez quand même le jeu : encore une fois, cela vous sera utile, entre autres pour l'évaluation du stage de L3.

Références

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