Second demi-projet : interpréter, compiler

Rendu 4 : pour le 20 mai à 23h59

par mail à `henning.basold@ens-lyon.fr`,
`daniel.hirschkoff@ens-lyon.fr`,
`bertrand.simon@ens-lyon.fr`

NB : il y aura une séance finale le 25 mai à 10h15, assez "détendue"
— cf. la fin de ce document

Préambule

Ce sujet est écrit en faisant l'hypothèse que tout a été traité (plus ou moins bien...) lors du rendu 3. Si tel n'est pas le cas, envoyez rapidement *un mail à* `daniel.hirschkoff@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 : si le rendu 3 est insuffisant, il n'y a pas de raison que le rendu 4 le soit automatiquement.

1 Débutants : de fouine à Caml

Ajoutez à votre interprète exceptions, références et couples (cf. énoncés des rendus précédents).

Il vous est recommandé que chaque moitié de binôme traite une extension de fond en comble, la troisième extension étant faite par la personne la plus rapide, ou alors à deux. Tout cela dans l'ordre que vous voulez.

2 Intermédiaires

2.1 Combiner interprète et machine

On appelle dans ce qui suit "*pur*" un programme fouine qui ne comporte que les expressions arithmétiques, let `let.. in` (seulement pour les variables entières, pas de `let f = fun x -> ..`, pas d'appel de fonctions comme `(f x)`) et `prInt`. De plus, un programme ne peut être pur que s'il ne consiste pas en un simple atome (constante, ou variable) ; ainsi, dans `ref 3`, le sous-programme `3` n'est pas pur, en revanche `3` est pur dans, par exemple, `3*4` ou `let x = 2 in 3+x` (cf. ci-dessous pour un commentaire à propos de cette contrainte).

Il vous est demandé de traiter les options `-machine` et `-stackcode` de `fouine`, comme décrit dans ce qui suit.

Option `-machine`. On prend en entrée un programme fouine. Les sous-programmes purs sont exécutés avec la machine, le reste à l'aide de l'interprète. Pour exécuter un programme avec la machine, on le traduit (ou "compile") vers un programme pouvant s'exécuter sur une machine à pile (comme vu en cours), puis on fait tourner ladite machine, qui renvoie son résultat (entier) à la fin. Pour cette raison, dans la définition donnée plus haut, on ne considère pas que `3` tout seul est pur : on veut que la machine "bouge un minimum" avant de renvoyer son résultat, et on ne fera pas appel à la machine si on doit évaluer par exemple `ref 3`.

À noter que la construction `prInt` doit également être traitée : on affiche l'entier, et on passe à la suite.

Il vous faudra donc être capable de détecter si un bout de code est pur (et donc 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 n'est pas le même, tout s'arrête avec un message d'erreur indiquant les deux résultats obtenus).

Option `-stackcode`. Si le programme entré est pur, vous affichez le programme résultant de la compilation, qui peut s'exécuter sur une machine à pile. À vous de spécifier la syntaxe du langage cible (et de la décrire dans le README, bien sûr).

Si on combine les options `-stackcode` et `-machine` (dans n'importe quel ordre), on affiche à l'écran le code compilé, on saute une ligne, puis on exécute le code compilé à l'aide de la machine.

Avec l'option `-stackcode`, si le programme entré n'est pas pur, `fouine` s'arrête après avoir affiché un message d'erreur.

Organisation du travail Voici comment nous vous suggérons de découper le travail :

- Vous vous mettez d'accord sur les grandes lignes, et les modifications du type servant à représenter les programmes fouine en entrée.
- Une moitié du binôme implémente l'analyse permettant de réaliser l'"aiguillage" entre interprète et machine.

  Pour ce travail, vous pouvez, si vous vous sentez suffisamment à l'aise, essayer d'avoir en tête ce qui vous est demandé à la partie 2.2. *N'ignorez pas, cependant, l'avertissement qui débute cette partie.*
- Pendant ce temps, l'autre moitié implémente l'interprète pour la machine à pile, sur lequel tournera le résultat de la compilation.
- Vous mettez tout cela ensemble, nettoyez, testez, etc.

Pensez au passage à la répartition des différents composants dans des fichiers.

2.2 fouine typée : vérification de types

Cette partie n'est à traiter que si vous avez traité complètement la partie qui précède. Si vous n'avez pas le temps d'aborder cette partie, eh bien tant pis, mais ne travaillez pas sur ces deux parties en parallèle.

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.

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).

Suivant le temps dont vous disposez pour traiter cette partie, vous pouvez décliner les choses de manière plus ou moins sophistiquée. L'idée est d'améliorer fouine en étant capable de rejeter *certains* programmes non typés.

Vous pouvez par exemple ne traiter que certaines formes d'annotations de types. Vous pouvez programmer une vérification de types plus ou moins astucieuse et riche, l'idée étant que soit on proteste lorsqu'on est sûr qu'il y a un problème, soit on est sûr que le typage est correct, soit on laisse tomber et on espère que ça ira bien. En particulier, un programme que l'on saisit sans annotation de types est toujours accepté par le vérificateur de types.

3 Avancés

3.1 La machine SECD

- Programmez une compilation vers la machine SECD telle que vue en cours, pour le langage de départ (fonctions récursives exclues, `prInt` inclus).
- Ajoutez dans un second temps les fonctions récursives.
- Complétez la machine SECD, en traitant références et exceptions (dans l'esprit de ce qui a été fait pour l'interprète : pas de `ref` pour traiter les références, pas de "vraies exceptions" pour traiter les exceptions — vous pourrez en particulier enrichir la machine pour traiter les exceptions).

Options : pour exécuter la machine au lieu de lancer l'interprète, on utilisera l'option `-machine`. L'option `-stackcode` permet d'afficher le programme résultant de la compilation (il vous faudra définir la syntaxe du langage cible, et la décrire dans le fichier Readme du rendu). `fouine` se contente de l'affichage si on n'invoque que l'option `-stackcode`. La combinaison des deux options (dans n'importe quel ordre) a pour effet d'afficher le code compilé, sauter une ligne, puis lancer l'exécution du code compilé.

Vous pouvez aussi, pour faciliter le debug, implémenter une option qui exécute la machine et l'interprète, et compare les résultats, en poussant des cris s'il y a une différence. Ceci est facultatif.

Enfin, vous pouvez vous intéresser à des raffinements de la machine vue en cours, comme l'indique le paragraphe suivant.

Documentation en ligne Vous trouverez ici les transparents d'un cours de Xavier Leroy d'où vient ce qui vous a été raconté en cours (qui est disponible ici).

3.2 Et aussi : améliorer, typer, ou prouver

Cette partie n'est à traiter que si vous avez traité complètement la partie 3.1. De plus, il vous est demandé de traiter l'une des trois propositions ci-dessous, pas les trois (!). Si l'inférence de types est déjà traitée au rendu 3, traitez autre chose.
Enfin, si vous n'avez pas le temps d'aborder cette partie, eh bien tant pis, mais ne travaillez pas sur 3.1 et 3.2 en parallèle.

Améliorations possibles de la machine. Une première amélioration consiste à passer en *indices de De Bruijn* pour la représentation des lieurs : plutôt que d'avoir `let x = 3 in let y = 4 in x+y`, on a quelque chose comme `let 3 in let 4 in 1+0` (pas dans le source, bien sûr, mais dans la représentation interne du code). Ceci permet d'avoir des ACCESS(i) plutôt que des ACCESS(x) dans la machine.

Bonus : si vous êtes motivés, vous pouvez aller regarder la machine Zinc, afin d'améliorer les performances de la SECD (mieux gérer les fonctions à plusieurs variables). Idéalement, vous comparerez les performances des deux machines sur une batterie d'exemples. Des liens à propos de la Zinc: ici et ici.

Typer. On reprend ici la partie 2.2, mais en allant plus loin, et en faisant quelque chose de différent par rapport à 2.2, dans la mesure où l'on fait de l'*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. 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).

Comme spécifié en 2.2, 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 par endroits plus restrictif, et par endroits plus permissif, que celui de Caml (au sens où il rejette des programmes que Caml accepte, ou l'inverse). Racontez ce que vous pensez de la chose (exemples à l'appui) dans le README.

Prouver. Si vous connaissez Coq, vous pouvez formaliser la machine pour un programme en entrée ne comportant pas de composante "complexe" (fonctions, références, exceptions).

Il vous faudra :

- Définir la syntaxe de départ.

  Vous pouvez utiliser des entiers pour représenter les variables, pas comme des indices de De Bruijn, mais comme des identificateurs. Exemple : `let 1 = 3 in let 2 = 5 in 2*7+1`

- Définir la syntaxe pour la machine, et la fonction de compilation du source vers l'objet.

- Définir l'interprète (la fonction qui exécute la machine, quoi).

Après quoi, une option `-coq` de `fouine` permettra de récupérer en sortie :

- l'énonce d'un lemme de la forme `run p = k`, où `p` est la représentation Coq du programme en entrée, et `k` est le résultat de l'exécution ;

- un script de preuve permettant de prouver ce lemme.

Bonus : vous pourrez ensuite, si vous êtes motivés et avez le temps :

- Définir en Coq la sémantique à grands pas du langage source, et prouver qu'elle calcule le même résultat que l'exécution de la version compilée du programme.

- Prouver des propriétés sur l'exécution des programmes, comme par exemple :

  . $\texttt{n} \neq \texttt{m} \Rightarrow$ `let n = i in let m = j in p` $\simeq$ `let m = j in let n = i in p` pour tout "programme" `p`, où $\texttt{p} \simeq \texttt{q}$ signifie que `p` et `q` renvoient le même résultat.

  . si `n` n'apparaît pas dans `p`, alors `let n=k in p` $\simeq$ `p`

4 Spécification : exécutable, options, etc.

Options. Mêmes options que pour les rendus précédents, avec, pour les intermédiaires et les avancés, les options supplémentaires `-machine` et `-stackcode` : voir les parties correspondantes pour la description de ces options.

Pas trop d'interférences entre les options : `fouine` renonce à s'exécuter s'il y a un "mélange d'options" entre transformations de programmes et machine à pile. Dit autrement, seule l'option `-debug` peut être combinée avec n'importe quelle autre option, mais sinon, soit on fait tourner la transformation de programmes, soit on fait tourner la partie où l'on compile vers machine. Et tant pis pour la séduisante idée consistant à transformer puis compiler (vous pouvez inventer une solution pour cela, mais c'est facultatif).

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.

Le rendu. Vous pouvez vous reporter aux rendus 1 à 3 : l'esprit est le même, en ce qui concerne la propreté du rendu, les explications, etc. N'oubliez pas d'expliquer dans le README comment vous avez procédé, notamment si vous traitez la partie sur les transformations de programmes.

5 Le README qui termine le 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.

6 . . . et ça n'est pas tout à fait terminé (pour le 25 mai)

 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 LaTeX, qui sera accompagné d'une présentation de 4 minutes, *avec des transparents,* lors de la séance du 25 mai (mais tout est à rendre pour le 20 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.

References

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