Évaluation partielle

I. Présentation

1. Définition d'un évaluateur partiel

Le but d'un évaluateur partiel est d'optimiser un programme dont on connaît une partie des arguments. Plus précisément, il s'agit d'une fonction `specialize` vérifiant : pour toute fonction `f`, pour toute entrée `x`, `y` de cette fonction, `(specialize(f, x))(y) = f(x, y)`. On souhaite naturellement que `(specialize(f, x))(y)` s'éxécute plus rapidement que `f(x, y)`, en profitant du fait que `x` est déjà connu au calcul de `specialize(f, x)`.

2. Applications

L'évaluation partielle a de nombreuses applications, dont on pourrait par exemple citer (c.f. [1] p.xi) :

- Génération de code et compilation d'un langage en disposant uniquement d'un interpréteur de celui-ci,
- Entrainement de réseaux de neurones : un évaluateur partiel permet de transformer un programme général en un adapté pour entrainer un réseau donné, et qui sera donc plus rapide,
- Simulation numérique de circuits.

Je me suis plus particulièrement intéressé au premier point, pour plusieurs raisons : en plus de son intérêt pratique (un interpréteur est bien plus simple à écrire qu'un compilateur), il permet en effet de faire des tests de performance de l'évaluateur partiel en comparant le résultat au programme qui serait naturellement écrit dans le langage cible ou bien compilé à partir du programme initial.

II. Conception d'un langage simple

1. Choix de conception

Pour pouvoir écrire un évaluateur partiel, il me fallait commencer par un interpréteur. J'ai donc créé un langage assez simple, dont la syntaxe est assez proche du Lisp (par souci de simplicité d'implémentation). Les langages impératifs conduisant à des effets de bord, ils ne sont pas souhaitables pour l'écriture d'un évaluateur partiel, car les effets de bord conduisent à une complexité accrue, en particulier lors des phases d'optimisation. Au contraire, un langage purement fonctionnel vérifie des propriétés comme le fait que résultat ne dépend pas de l'ordre d'évaluation ou qu'une fonction peut être mémoïzée, qui rendent les optimisations plus simples et dont la correction est plus simple à montrer. Un autre

choix important est la portée des variables : ayant commencé par utiliser une portée dynamique, pour faciliter l'implémentation, je me suis vite aperçu de mon erreur (c.f. note 1)

Note 1 (Un problème avec la portée dynamique)

```
(let m 1)
(let (f x)
     (+ x m))
(let (g m)
     (f m))
```

Ce code possède un problème majeur : Dans le calcul de `(g 2)`, on appelle `(f 2)`. Ce calcul cherche la valeur de `m`; or celle-ci était 1 lors de la définition de `f`, mais quand on appelle, `(g 2)`, la valeur de `m` est 2 dans l'environnement de `g`, puis dans celui de `f` *à cause de la portée dynamique*. On obtient donc le résultat 4 au lieu de 3 attendu : en particulier, le résultat de `(f 2)` *dépend de l'environnement* dans lequel cet expression est calculée.

Après avoir choisi d'utiliser une portée statique, un autre problème se présentait : celui des fonctions récursives. En effet, avec la portée dynamique, une fonction est dans son propre environnement au moment de son appel; mais avec la portée statique, comme l'environnement d'une fonction est réduit à celui qui existait au moment de sa définition plus ses arguments, elle n'est pas dans son propre environnement ! J'avais alors deux possibilités à ma disposition : la première était d'utiliser un combinateur de point fixe, la deuxième étant d'utiliser un environnement circulaire. J'ai choisi la première possibilité, bien que pour des raisons de confort (aussi bien au niveau de l'écriture d'un programme que de l'écriture de l'interpréteur), ce combinateur de point fixe soit implémenté dans l'interpréteur lui-même (c.f. note 2). En effet, un environnement circulaire possède le problème qu'il nécessite de créer une structure de données circulaire (ce qui n'est par exemple pas possible dans le langage lui-même), qui peut ensuite poser des problèmes (lors d'un test d'égalité structurelle par exemple).

Note 2 (Le combinateur de point fixe et son utilisation)
Un combinateur de point fixe `Y` vérifie : pour tout `f`, `Y f = f (Y f)`. Il permet de faire des récursions en transformant une expression du type :

```
(letrec (f x) <definition de f>)
```

en:

```
(let (f_rec f x) <definition de f>)
(let f (Y f_rec))
```

Le combinateur de point fixe est implémenté ici directement dans l'interpréteur, ainsi `(Y f_rec)` retourne une valeur de type point fixe, qui ne sera développé que lorsque l'on essaiera de l'appliquer à des arguments. Une version légèrement généralisée du combinateur de point fixe, qui prend plusieurs arguments est utilisée pour les fonctions mutuellement récursives.

2. L'interpréteur : la boucle évaluer / appeler

L'interpréteur se base sur deux fonctions mutuellement récursives, "évaluer" et "appeler". "évaluer" prend en entrée une expression et un environnement, tandis que "appeler" prend en entrée une fonction et ses arguments, et calcule le résultat de l'application de cette fonction à ces arguments (elle prend donc des valeurs et non des expressions en entrée).

III. L'évaluateur partiel

1. Modification de l'interpréteur en évaluateur partiel

On remarque que un interpréteur et un évaluateur partiel sont en fait très proches : on va donc modifier l'interpréteur. La première modification est le type des valeurs : on ajoute un type spécial `<inconnu>` qui signifie qu'on peut remplacer cette valeur par n'importe quelle autre.

> Définition 1
>
> On dit qu'une valeur v est *compatible* avec une valeur $\tilde{v}$ utilisant des `<inconnu>` si pour chaque occurence de `<inconnu>` dans $\tilde{v}$ il existe une valeur de manière à ce que remplacer chaque occurence de `<inconnu>` dans $\tilde{v}$ par la valeur associée donne v.
>
> Par exemple, `(cons 1 2)` est compatible avec `(cons 1 <inconnu>)`.

Uniquement modifier l'interpréteur pour qu'il accepte et retourne des valeurs de ce type étendu n'est cependant pas suffisant: on obtient bien une valeur du type étendu qui correspond à la valeur de retour de l'expression, mais on ne sait pas comment obtenir ce résultat ! On modifie donc également la valeur de retour de l'interpréteur pour qu'il retourne également une expression calculant la valeur, et tenant compte des restrictions imposées aux arguments.

Une fois le calcul de l'évaluateur partiel terminé, on effectue une phase d'optimisation : en effet, on se retrouve avec beaucoup plus de fonctions que l'on en avait au départ, mais de nombreuses fonctions sont triviales et se réduisent à l'appel d'une autre fonction, ou n'utilisent pas certains de leurs arguments. On simplifie donc cela, puis on recrée le code permettant de calculer le résultat en mettant bout-à-bout les définitions des différentes fonctions (après calcul des composantes fortement connexes du graphe d'appel des fonctions pour savoir quelles définitions récursives sont nécessaires).

2. Vérification de la validité

On suppose ici que le calcul de l'évaluateur partiel termine (en pratique, on ajoute des limites au nombre de fois où une fonction peut être spécialisée pour assurer cela). On peut alors montrer le résultat suivant :

> Théorème 1
>
> Pour tout appel de `specialize(expr, env)` *lors de l'éxécution de l'évaluateur partiel sur un programme donné*, si on a `(v, code) = specialize(expr, env)`

alors tout appel de `eval(expr, e)` *qui termine* où `e` est un environnement compatible avec `env` (i.e. chaque valeur de `e` est compatible avec la valeur correspondante de `env`) vérifiera : `eval(expr, e)` est compatible avec `v`, et de plus, `eval(expr, e) = eval(code, e)`.

En particulier, il se peut qu'un programme qui ne termine pas soit transformé en programme qui termine (c.f. note 3). On ne montre *pas* ce résultat par induction structurelle (car l'appel d'une fonction ne correspond pas à un cas d'induction structurelle) mais par récurrence par ordre croissant de fin d'appel (c'est pour cela que l'on a supposé que le calcul se terminait).

Note 3 (Programme dont la terminaison n'est pas respectée)

```
(letrec (f x) (f x))
(let (g x) (tail (cons (f x) 0)))
```

Le calcul de `(g 1)` ne termine pas, mais après évaluation partielle, on obtient :

```
(let (g x) 0)
```

qui termine. On aurait cependant obtenu le même résultat si le programme initial terminait, par exemple si il était évalué paresseusement.

IV. Résultats

Pour tester l'efficacité de l'évaluateur partiel, j'ai écrit un meta-évaluateur du langage, et je l'ai évalué partiellement en lui fixant un programme. J'ai utilisé la fonction naïve calculant la suite de Fibonacci comme programme afin d'obtenir de nombreux appels de fonctions pour des entrées assez petites: en effet, pour avoir des temps mesurables, il fallait que les temps d'éxécution soient très grands. Ceux-ci restent cependant exponentiels après évaluation partielle : celle-ci ne remplace pas un mauvais algorithme.

J'ai également comparé quatre versions différentes :

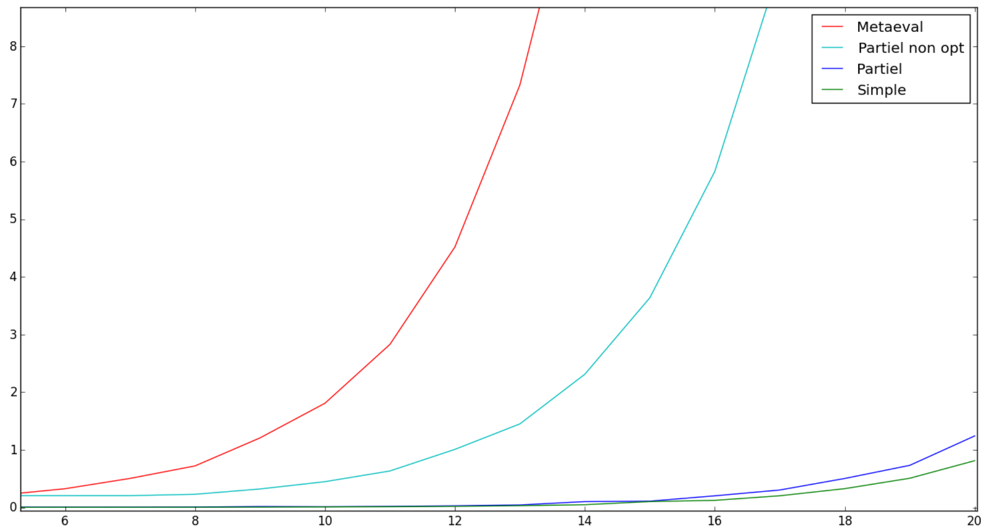
- Le metaévaluateur calculant la suite de Fibonacci [Metaeval]
- Le même programme évalué partiellement [Partiel]
- Le même programme évalué partiellement, mais sans la phase d'optimisations [Partiel non opt]
- La programme calculant la suite de Fibonacci qui était donné comme entrée au metaévaluateur. [Simple]

De plus, j'ai comparé deux méthodes d'exécution :

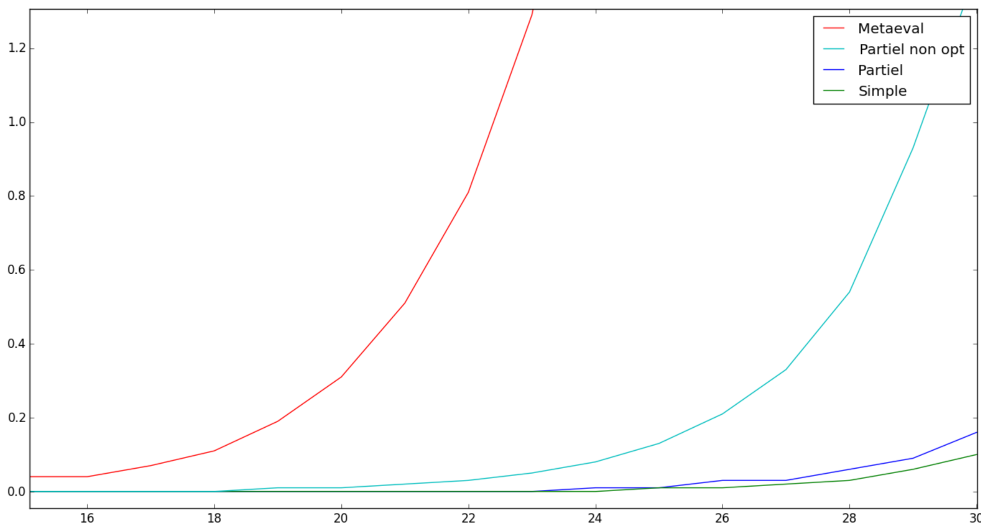
- L'évaluateur initial,
- Les programmes, compilés en OCaml puis vers du code natif.

Résultats :

Interprété :



Compilé :



Pour le calcul de `(fib 20)` :

	Metaeval	Partiel	Partiel non opt	Simple
Évalué	211.2 s	1.206 s	39.03 s	0.813 s
Compilé	0.304 s	0.005 s	0.015 s	0.003 s

On remarque en particulier que le temps d'exécution du programme partiellement évalué est proche de celui du programme donné en argument au metaévaluateur, ce qui était bien le résultat souhaité. On voit également que la phase d'optimisation, réduit de manière importante le temps d'éxécution. Sans celle-ci, le programme partiellement évalué reste assez lent mais est quand même bien plus rapide que le metaévaluateur. C'est aussi en supprimant cette phase d'optimisation qu'on trouve la seule différence significative de temps d'éxécution entre la version compilée et interprétée (le compilateur de OCaml faisant quelques optimisations).

V. Améliorations envisageables

Pour le cœur de l'évaluateur partiel, on pourrait envisager quelques améliorations par exemple lors de la spécialisation de fonctions récursives : en effet une fonction du type :

```
(letrec (f n) (if (= n 0) 0 (f (- n 1))))
```

qui retourne toujours 0 quand elle termine n'est pas détectée en tant que telle.

Une autre amélioration, cette fois-ci lors de l'optimisation pourrait être de simplifier les fonctions qui retournent une structure partiellement connue à l'évaluation (du type `(cons 0 <inconnu>)`). La valeur de retour de telles fonctions n'est en effet pas modifiée lors de l'optimisation. Une autre manière d'aborder ce problème serait de compiler le langage vers un langage fonctionnel doté d'un bon compilateur optimisant (possiblement Haskell), et de laisser ce compilateur faire toute la phase d'optimisation une fois la phase d'évaluation partielle faite (qui resterait utile même si le langage était compilé, le compilateur n'effectuant pas d'évaluation partielle).

VI. Bibliographie

[1] Jones, Neil D.; Gomard, Carsten K.; Sestoft, Peter. Partial Evaluation and Automatic Program Generation. 1993, 425 p.

[2] Futamura, Yoshihiko, Partial Evaluation of Computation Process — An Approach to a Compiler-Compiler. Higher-Order and Symbolic Computation 1999, n°12, pp. 381-391

[3] Abelson, Hal; Sussman, Jerry; Sussman, Julie. Structure and Interpretation of Computer Programs. 2e éd. MIT Press, 1996, pp. 362-397.