Évaluation partielle

Sommaire

Introduction

Conception d'un langage simple

L'évaluateur partiel

Résultats

Améliorations envisageables

Bibliographie

Présentation

On donne :

- Le code d'une fonction `f`
- Un argument `x` de `f`

But : Créer une fonction `g` telle que :

- $\forall$`y`, `f(x, y) = g(y)`
- Le calcul de `g(y)` est plus rapide que celui de `f(x, y)`

Applications

De nombreuses applications existent, dont :

- Génération de code et compilation d'un langage en disposant uniquement d'un interpréteur de celui-ci,
- Entrainement de réseaux de neurones,
- Simulation numérique de circuits.

Pourquoi créer un nouveau langage ?

- Disposer d'un langage minimaliste
- Disposer d'un interpréteur pour celui-ci

Choix de conception

- Syntaxe : proche du Lisp
- Typage dynamique
- Portée statique
- Utilisation d'un combinateur de point fixe pour la récursion

L'interpréteur - la boucle évaluer / appeler

- Basé sur deux fonctions mutuellement récursives : `évaluer` et `appeler`
- `évaluer(expr, env)` retourne le résultat de l'évaluation de `expr` dans l'environnement `env`
- `appeler(fct, args)` retourne le résultat de l'appel de `fct`
  avec les arguments `args`

Modification de l'interpréteur

On ajoute une valeur spéciale `<inconnu>` et on définit :

Définition

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 (cons 2 3))` est compatible avec `(cons <inconnu> (cons 2 <inconnu>))`.

Modification de l'interpréteur

On modifie ensuite l'interpréteur en une fonction `specialize` pour que la propriété suivante soit vérifiée :

Propriété

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)`.

Méthode d'expérimentation

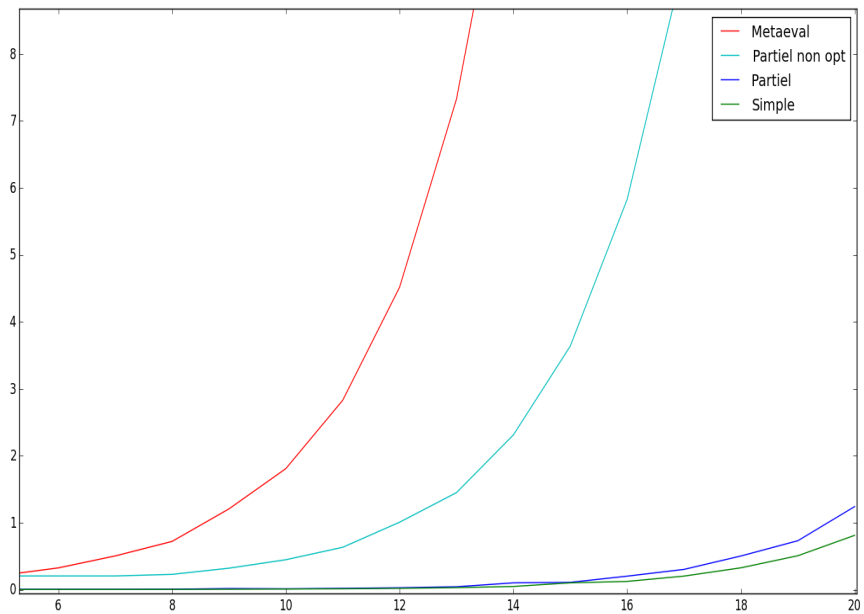
On compare les programmes suivants :

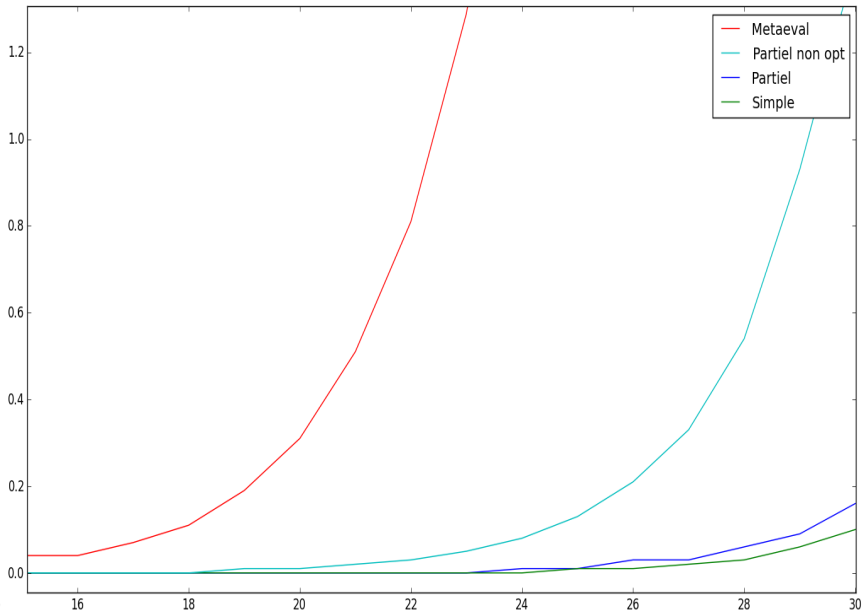
- Un programme calculant la suite de Fibonacci naïvement
- Le metaévaluateur exécutant ce programme
- Ce dernier évalué partiellement, avec et sans optimisations

On compare également deux méthodes d'éxécution différentes :

- L'évaluateur (écrit en Python) qui éxécute ces programmes,
- Les programmes, compilés vers OCaml puis vers du code natif.

Temps d'éxécution





Améliorations envisageables

- Gérer les fonctions comme :

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

- Améliorer la phase d'optimisation ou utiliser un compilateur optimisant déjà existant

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.