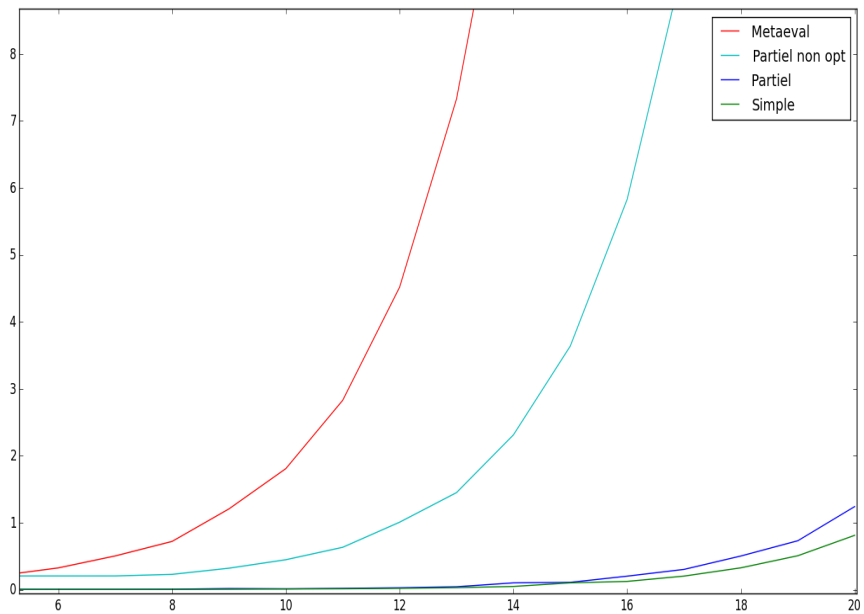
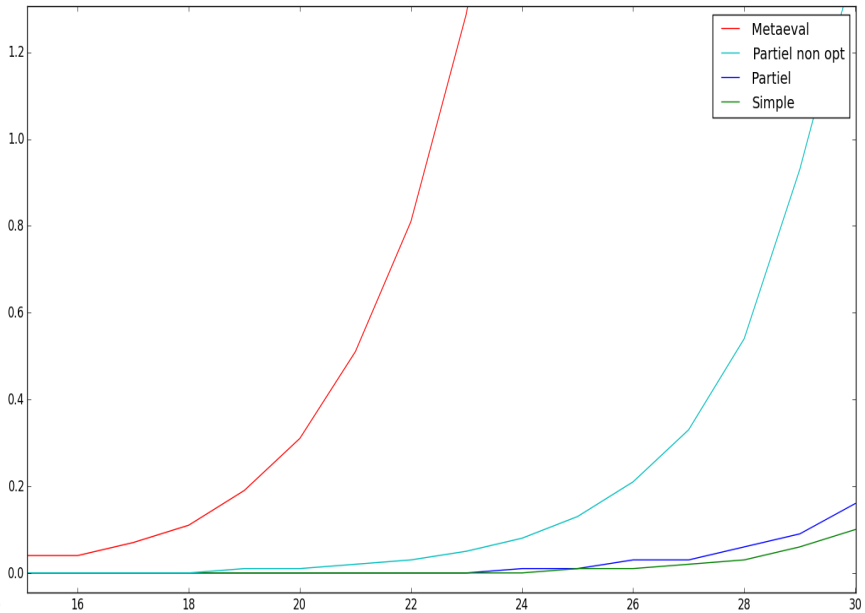
Temps d'éxécution





L'interpréteur : évaluer

```
evaluer(expr, env)
  Si expr est une constante:
    retourner expr
  Si expr est une variable:
    retourner trouver_variable(env, expr)
  Si expr est une instruction conditionnelle:
    resultat_test = evaluer(test(expr), env)
    Si resultat_test:
      retourner evaluer(si_vrai(expr), env)
    Sinon:
      retourner evaluer(si_faux(expr), env)
  Si expr est un appel:
    fct = evaluer(fonction(expr), env)
    arguments = [evaluer(arg, env)
        pour arg dans arguments(expr)]
    appeler(fct, arguments)
```

L'interpréteur : appeler

```
appeler(fonction, arguments)
  Si fonction est une fonction de base:
    appeler_fonction_de_base(fonction, arguments)
  Sinon:
    env = environnement_fonction(fonction)
    nouveau_env = ajout_environnement(env,
        noms_arguments(fonction), arguments)
    retourner evaluer(code(fonction), nouveau_env)
```

L'évaluateur partiel

```
specialiser(expr, env)
 [...]
 Si expr est une instruction conditionnelle:
   (resultat_test, code_test) =
       specialiser(test(expr), env)
   Si resultat_test est connu:
       Si resultat_test:
         retourner specialiser(si_vrai(expr), env)
       Sinon:
         retourner specialiser(si_faux(expr), env)
   Sinon:
      (valeur1, code1) = specialiser(si_vrai(expr), env)
      (valeur2, code2) = specialiser(si_faux(expr), env)
      retourner (fusion_valeurs(valeur1, valeur2),
                    creer_test(code_test, code1, code2))
 [...]
```

Exemple d'évaluation partielle

Entrée :

```
(letrec (rev_concat l r)
    (if (= l [])
        r
        (rev_concat
            (tail l)
            (cons (head l) r))))
(rev_concat [a b c d] [])
```

Sortie :

```
[d c b a]
```

Résultats de l'optimisation

Sans optimisation :

```
(letrec* (
((eval_expr—6 env—5 expr—4) (let (cons type—2 ex—2) globalvar—12
  (eval_keyword—1 env—5 globalvar—13)))
((eval_call—1 func—3 args—4) (let (cons code—1 (cons argnames—1 (cons fenv—1 []))))
  globalvar—23 (eval_expr—6 (add_list_to_env—1 globalvar—7 globalvar—24 args—4)
    globalvar—12)))
((eval_expr—10 env—12 expr—10) (let (cons type—6 ex—6) globalvar—28 (let (cons ft—4 func—5)
  globalvar—29 (let (cons ft2—1 func2—1) globalvar—30 (eval_call—1 globalvar—23
    (eval_args—5 env—12 globalvar—31))))))
((eval_expr—12 env—18 expr—13) (let (cons type—7 ex—7) globalvar—37 (let (cons ft—5 func—6)
  globalvar—29 (let (cons ft2—2 func2—2) globalvar—30 (eval_call—1 globalvar—23
    (eval_args—7 env—18 globalvar—38))))))
((eval_args—9 env—27 args—12) (cons (eval_expr—12 env—27 globalvar—37) globalvar—17))
((eval_args—4 env—90 args—42) (cons (eval_expr—10 env—90 globalvar—28)
  (eval_args—9 env—90 globalvar—46)))
((eval_expr—5 env—11 expr—9) (let (cons type—5 ex—5) globalvar—11 (let (cons ft—3 func—4)
  globalvar—1 (let args—5 (eval_args—4 env—11 globalvar—27) (tagged_+—1 args—5)))))
((eval_keyword—1 env—4 expr—3) (let keyword—1 globalvar—8 (let (cons ct—1 condition—1)
  (eval_expr—3 env—4 globalvar—9) (if condition—1 (eval_expr—4 env—4 globalvar—10)
    (eval_expr—5 env—4 globalvar—11)))))
))
```

Avec optimisation :

```
(letrec (eval_keyword—1 env—465—h—t—t—1) (if (<= env—465—h—t—t—1 1) (cons 0 env—465—h—t—t—1)
  (cons 0 (+ (tail (eval_keyword—1 (— env—465—h—t—t—1 2)))
             (tail (eval_keyword—1 (— env—465—h—t—t—1 1)))))))
```