# Stage d'informatique
# Implémentation de la géométrie de l'interaction

BLANCHET Bruno
Sous la direction de M. Ian MACKIE

22 septembre 1995

## 1 Introduction

Ce stage consiste à implémenter une extension de PCF [1] avec les listes et le filtrage à l'aide de la géométrie de l'interaction [6]. On procède ensuite à diverses optimisations de la machine initiale et on compare les performances obtenues par les différentes versions de la géométrie de l'interaction et de compilateurs existants pour les langages fonctionnels (on a pris comme exemple Caml-Light, copyright INRIA 1991-1995) et la réduction optimale (Bologna Optimal Higher-order Machine de A.Asperti, C.Giovannetti, C.Laneve et A.Naletto, Dipartimento di Matematica, Bologna).

## Première partie

# Le langage implémenté

Le programme `sflcomp` peut être utilisé de deux façons : comme compilateur proprement dit, ou comme toplevel avec l'option `-top`. De plus, des scripts facilitent le travail : `run` permet d'enchaîner la compilation et l'exécution d'un programme, et `runall` le lance de la même façon, mais sur toutes les versions de machines disponibles. L'annexe A donne la liste complète de toutes les options du compilateur et les commandes du toplevel. Le compilateur est implémenté en Caml-Light, et produit du code C à compiler avec `gcc`.

Le langage utilisé, appelé SFL (Small Functional Language), est une extension de PCF [1] avec des opérations supplémentaires sur les entiers et les booléens, les listes et le filtrage. L'annexe B en donne une grammaire simplifiée. Les expressions (notées $< \mathit{expr} >$) comprennent les constructions suivantes :

- `fn` $< \mathit{var} >$ `=>` $< \mathit{expr} >$ pour l'abstraction
- $< \mathit{expr} >$ $< \mathit{expr} >$ pour l'application
- `let` $< \mathit{var} >$ `=` $< \mathit{expr} >$ `in` $< \mathit{expr} >$
- $< \mathit{var} >$ référence à une variable
- `(`$< \mathit{expr} >$`)` pour passer outre les priorités

comme constructions de base. On y a ajouté des constantes :

- `[]`, `::`, `hd`, `tl`, `isnil` pour les listes
- `+`, `-`, `*` , `/`, `%`, `iszero`, `pred`, `succ`, `=`, `<=`, `<`, `<>` et $<$ *constantes entières* $>$ pour l'arithmétique
- `tt`, `ff`, `and`, `or`, `xor`, `not` comme fonctions logiques
- `cond` et la construction plus élégante `if` $< \mathit{expr} >$ `then` $< \mathit{expr} >$ `else` $< \mathit{expr} >$ pour l'expression conditionnelle
- `Y` pour la récursivité.

Les opérateurs `::`, `+`, `-`, `*`, `/`, `%`, `=`, `<=`, `<`, `<>`, `and`, `or`, `xor` sont naturellement infixes. Les constantes ci-dessus sont des mots-clés qui ne peuvent être utilisés comme identificateurs de variables.

Pour faciliter l'écriture, on a aussi ajouté les constructions suivantes :

`fn` $< \mathit{pattern} >$ `=>` $< \mathit{expr} >$<br>
  $\mid < \mathit{pattern} >$ `=>` $< \mathit{expr} >$<br>
  ...

et

`match` $< \mathit{expr} >$ `with`<br>
  $< \mathit{pattern} >$ `=>` $< \mathit{expr} >$<br>
  $\mid < \mathit{pattern} >$ `=>` $< \mathit{expr} >$<br>
  ...

pour le filtrage. Les motifs ($<\mathit{pattern}>$) disponibles pour le filtrage sont les suivants :

- $<\mathit{constantes\ entières}>$
- `[]` liste vide
- $<\mathit{pattern}>::<\mathit{pattern}>$ liste à au moins un élément
- `[`$<\mathit{pattern}>;<\mathit{pattern}>;....;<\mathit{pattern}>$`]` liste dont les éléments filtrent les motifs internes.
- $<\mathit{var}>$ filtre n'importe quoi et le met dans la variable
- `_` filtre n'importe quoi.

Seul le filtrage linéaire est accepté (c'est-à-dire que chaque variable ne doit apparaître qu'une fois). Ceci ressemble beaucoup au filtrage tel qu'il est implémenté dans Caml par exemple. En cas de filtrage non exhaustif, un message d'avertissement est affiché, et une erreur se produit à l'exécution si on rencontre un cas non prévu.

On a aussi ajouté :

- `let rec` $<\mathit{var}>$ `=` $<\mathit{expr}(\mathit{var})>$ `in` $<\mathit{expr}>$ pour la récursivité
- `[`$<\mathit{expr}>;<\mathit{expr}>;...;<\mathit{expr}>$`]` pour les listes
- `>`, `>=` pour l'arithmétique.
- $<\mathit{expr}>$ `:` $<\mathit{type}>$ pour pouvoir forcer le type d'une expression. Ceci est surtout utile dans le cas non typable (voir ci-dessous).

Ces dernières formes sont traduites dans les constructions précédentes après le typage.

Enfin, on peut éventuellement indiquer explicitement les contractions et les affaiblissements par la syntaxe suivante :

- `[`$x$`=_]` pour l'affaiblissement : la variable $x$ n'est en fait pas utilisée.
- `[`$x$`=<`$y$`,`$z$`>]` pour la contraction : la variable $x$ est maintenant disponible sous deux noms, $y$ et $z$.

Dans ce cas, il faut préciser l'option `-a1` (pour le compilateur) et `#auto 1` (pour le toplevel) et bien contracter toutes les variables des constructions `fn` $<\mathit{var}>$ `=>` $<\mathit{expr}>$ (mais pas celles de `let` $<\mathit{var}>$ `=` $<\mathit{expr}>$ `in` $<\mathit{expr}>$). On peut aussi utiliser l'option `-a0` ou `#auto 0`, et contracter dans ce cas les variables de `fn` $<\mathit{var}>$ `=>` $<\mathit{expr}>$ et de `let` $<\mathit{var}>$ `=` $<\mathit{expr}>$ `in` $<\mathit{expr}>$. Cela alourdit la rédaction des programmes et, comme le compilateur s'oblige à suivre les indications des contractions données, cela l'empêche de les placer au meilleur endroit, et nuit donc à l'optimisation des programmes. C'est pourquoi je déconseille plutôt l'utilisation de cette possibilité.

Un programme complet est une succession de déclarations et d'expressions, où les déclarations sont de la forme :

`let` $<\mathit{var}>$ `=` $<\mathit{expr}>$`;;`

`let rec` $<\mathit{var}>$ `=` $<\mathit{expr}(\mathit{var})>$`;;` pour une expression récursive et les expressions sont de la forme :

$< expr >$;;

Remarque : la récursivité peut être utilisée aussi bien pour des structures de données que pour des fonctions.

On peut à tout moment insérer des commentaires de la forme `(*  .... *)`, et ces commentaires peuvent être imbriqués.

Après l'analyse syntaxique, le système type le programme par un typage analogue à celui de ML. De plus, en même temps qu'il type, il vérifie l'exactitude des contractions et affaiblissements. Ce typage s'inspire du fonctionnement de Caml Light, comme d'ailleurs l'affichage des messages d'erreur. L'option `-t` (pour le compilateur) ou `#notyping` (pour le toplevel) permet de faire passer des termes non typables : les erreurs de contractions ne sont plus indiquées, les erreurs de type deviennent de simples messages d'avertissement, et le type des expressions non typables est considéré comme quelconque (`'V`). L'utilisateur peut fixer lui-même le type des expressions par la construction $e$ : $t$. Dans ce cas, si l'expression $e$ est typable et de type $t'$ compatible avec $t$, le type résultant est l'unifié de $t$ et $t'$, sinon le type résultant est $t$, et aucun message n'est affiché. Mais alors, attention aux comportements étranges ! Tout peut arriver puisque le contrôle du programme n'est que très partiel.

Le compilateur tente ensuite d'optimiser l'expression entrée en éliminant les redex existants de la façon suivante : les expressions `(fn` $x$ `=>` $e_2$`)` $e_1$ et `let` $x$ `=` $e_1$ `in` $e_2$ deviennent:

- si $x$ n'apparaît pas dans $e_2$, $e_2$ seulement;
- s'il apparaît une fois, on remplace $x$ par son expression dans $e_2$;
- s'il apparaît plusieurs fois, soit $e_1$ est une variable et on renomme simplement $x$, sinon on garde la construction `let` $x$ `=` $e_1$ `in` $e_2$.

Les affaiblissements, les contractions inutiles et les mentions `rec` inutiles sont supprimés. L'expression optimisée est affichée par le compilateur après le type.

Ensuite, la phase de génération de code peut commencer. Celle-ci dépend de la machine choisie, et est décrite dans la partie suivante.

A l'exécution, les différentes expressions sont calculées successivement et leurs résultats sont affichés. On peut obtenir une trace latex avec l'option `-l` (compilateur) ou `#latex` (toplevel). Elle est placée dans le fichier `a.tex`, à compiler avec `latex a.tex` puis à visualiser avec `xdvi a.dvi &`. Dans le toplevel, la commande `#tex` permet d'effectuer en une seule opération compilation et visualisation de la trace.

Par défaut, le programme compte chaque type d'opération effectuée et la quantité de mémoire utilisée et affiche un bilan après chaque calcul d'expression. On peut désactiver ceci avec l'option `-c` (compilateur) ou `#nocount` (toplevel).

Si le comptage et la trace latex sont désactivées, le programme chronomètre l'exécution de chaque expression. Il n'y a pas de chrométrage dans les autres cas car les opérations de comptage et de trace prennent du temps (peu pour le comptage, mais beaucoup pour la trace).

## Deuxième partie

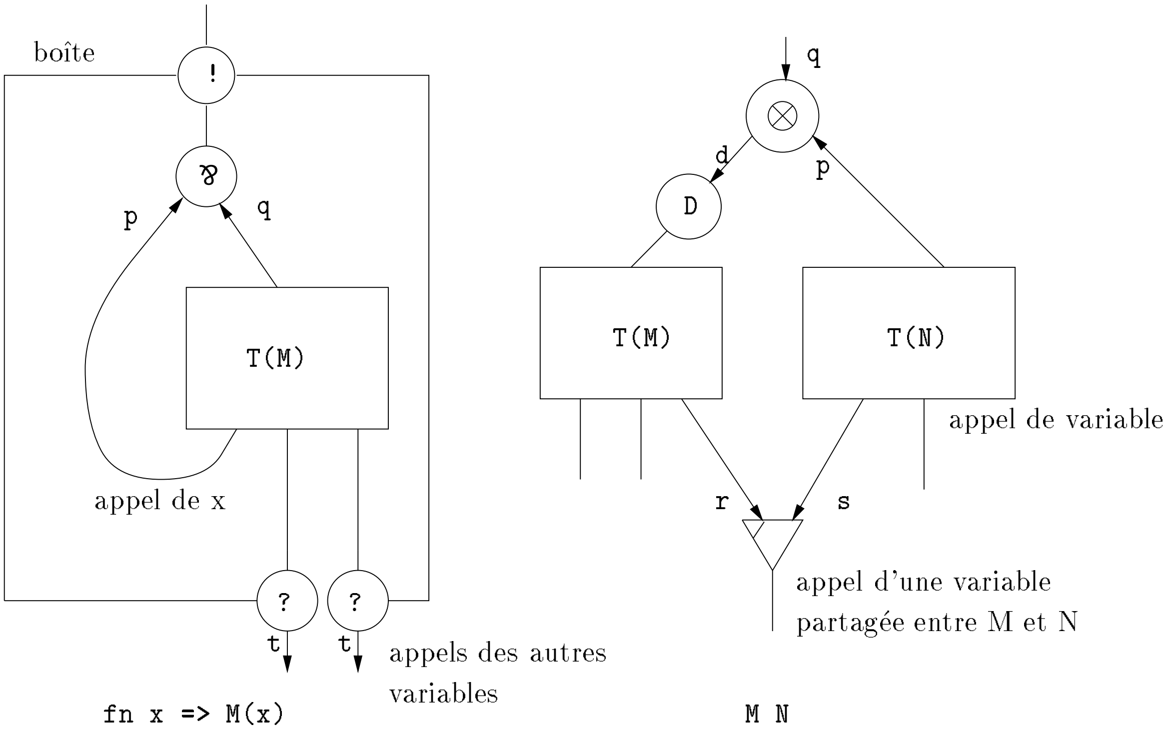
# Les différentes machines utilisées

## 1 La géométrie de l'interaction (GOI)

Cette machine s'appuie sur celle décrite dans la thèse de M. MACKIE [7]. Je me contenterai ici de donner un résumé rapide du codage effectué, sans le justifier.

La géométrie de l'interaction est une sémantique à mi-chemin entre la sémantique dénotationelle et opérationnelle. Elle est dénotationelle puisqu'elle est indépendante de la syntaxe (par exemple, de la notion de substitution), et elle est opérationnelle car les règles données indiquent comment le calcul doit être effectué.

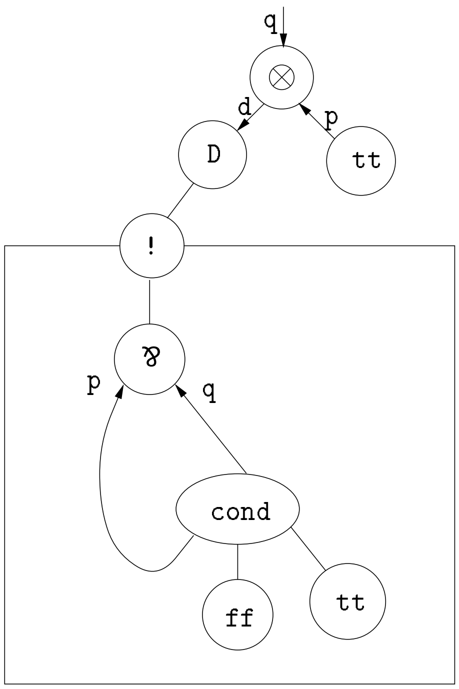
Chaque terme est traduit par un graphe selon les schémas suivants :




*Codage de l'abstraction et de l'application*

Seuls sont donnés ici les schémas concernant le $\lambda$-calcul pur; les constantes, listes et fonctions prédéfinies sont représentées par des nœuds qui seront donnés plus loin.

La sémantique du programme consiste en un ensemble de chemins dans ce graphe, certains chemins étant interdits par la sémantique et d'autres autorisés. On peut commencer le calcul des chemins n'importe où, mais on s'intéressera ici uniquement aux résultats de types simples, qui sont obtenus par les chemins qui partent de la racine et y reviennent. Il existe un seul chemin de ce type car le comportement à chaque nœud est entièrement déterminé par l'état de la machine.




*Exemple de codage* `(fn x => if x then ff else tt) tt`

Cet état se compose des structures de données suivantes :

- Un arbre binaire avec pointeur sur le père (`prev`) et sur les fils (`left`, `right`), qui permet de stocker l'environnement. Il est repéré par deux pointeurs, un à sa racine (`context`) et un qui se déplace le long de sa branche droite (`ep`). A chaque nœud de cet arbre se situe une pile (implémentée par une liste chaînée) qui contient l'état des contractions (une contraction est un nœud représenté par un triangle, qui apparaît dans le schéma du codage de l'application, et qui permet l'utilisation répétée des variables).

- Une pile pour stocker les valeurs de types simples (entiers et booléens) pointée par `vstack`. Cette pile sera notée $v$.

- Une pile multiplicative $M$, qui est stockée dans un simple registre 32 bits `R0`. En effet, cette pile sert à stocker des informations d'un bit 0 ou 1, et on montre que le nombre maximal de valeurs à stocker est borné par la taille maximale du type d'un sous-terme. 32 bits suffisent donc largement.

- Une pile $d$ pour se repérer dans les déplacements dans les listes, pointée par `dstack`.

L'exécution du programme consiste donc en un parcours du graphe, qui part de la racine, exécute à chaque nœud un morceau de code C qui modifie l'état puis passe au nœud suivant. Les opérations effectuées sont réversibles, c'est-à-dire que quand on passe sur une arête dans le sens de la flèche on effectue l'opération indiquée, et que quand on passe dans le sens contraire, on effectue l'opération inverse (notée avec une *). Seules les opérations concernant la pile de valeurs ne sont pas réversibles (et sont exécutées seulement sur le chemin de retour).

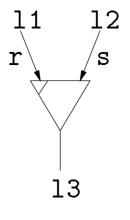
Au retour à la racine, la pile de valeurs contiendra un unique élément qui sera le résultat de l'évaluation du programme.

L'effet des instructions élémentaires sur l'état est le suivant :

| Opération | Etat initial | | Etat final | Op. inverse |
|---|---|---|---|---|
| | | Opérations multiplicatives | | |
| p | $M$ | $\Longleftrightarrow$ | p:$M$ | p* |
| q | $M$ | $\Longleftrightarrow$ | q:$M$ | q* |
| | | Opérations sur les valeurs | | |
| $c$ | $v$ | $\longrightarrow$ | $c : v$ | |
| `succ` | $a : v$ | $\longrightarrow$ | $(a + 1) : v$ | |
| `iszero` | $a : v$ | $\longrightarrow$ | `tt`:$v$ si $a = 0$ | |
| | | | `ff`:$v$ si $a \neq 0$ | |
| `+` | $a : b : v$ | $\longrightarrow$ | $(a + b) : v$ | |
| `not` | `tt`:$v$ | $\longrightarrow$ | `ff`:$v$ | |
| | `ff`:$v$ | $\longrightarrow$ | `tt`:$v$ | |
| Les autres opérations sont analogues. | | | | |
| | | Opérations sur l'arbre exponentiel | | |
| Ces opérations concernent le sous-arbre de racine `ep`. | | | | |
| r | $< e_1, e_2 >$ | $\Longleftrightarrow$ | $< L : e_1, e_2 >$ | r* |
| s | $< e_1, e_2 >$ | $\Longleftrightarrow$ | $< R : e_1, e_2 >$ | s* |
| t | $< e_1, < e_2, e_3 >>$ | $\Longleftrightarrow$ | $<< e_1, e_2 >, e_3 >$ | t* |
| d | $e$ | $\Longleftrightarrow$ | $< \square, e >$ | d* |
| | | Opérations sur les listes | | |
| h | $d$ | $\Longleftrightarrow$ | h:$d$ | h* |
| l | $d$ | $\Longleftrightarrow$ | l:$d$ | l* |
| i | $d$ | $\Longleftrightarrow$ | i:$d$ | i* |

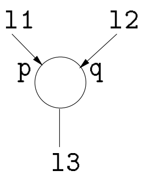
Chaque arête du graphe est représentée par un couple d'étiquettes (`l_in`, `l_out`) et les différents nœuds sont codés comme suit :

– contraction



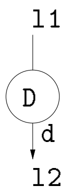

```
l1_in: r
       aller à l3_out;
l2_in: s
       aller à l3_out;
l3_in: dépiler le sommet de ep->left;
       si c'est L,
         aller à l1_out;
       sinon
         aller à l2_out;
```

– nœuds multiplicatifs. Il y a deux sortes de nœuds multiplicatifs : tenseur $\otimes$ et par $\parr$, qui sont codés de la même façon.



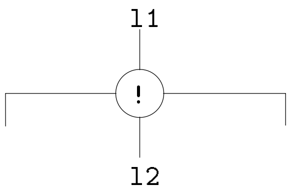

```
l1_in: p
       aller à l3_out;
l2_in: q
       aller à l3_out;
l3_in: si p est au sommet de M,
          p*
          aller à l1_out;
       sinon
          q*
          aller à l2_out;
```

– déréliction (D)



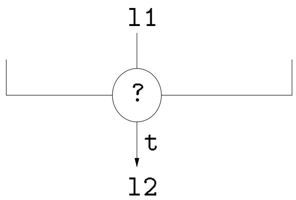

```
l1_in: d
       aller à l2_out;
l2_in: d*
       aller à l1_out;
```

– porte principale (!)



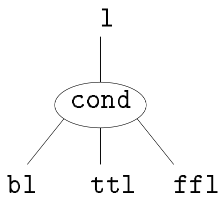

```
l1_in: ep = ep->right;
       aller à l2_out;
l2_in: ep = ep->prev;
       aller à l1_out;
```

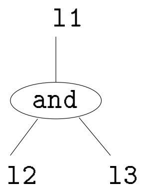
– portes auxiliaires (?t)




```
l1_in: ep = ep->prev;
       t
       aller à l2_out;
l2_in: t*
       ep = ep->right;
       aller à l1_out;
```

– condition, `and`, `or`

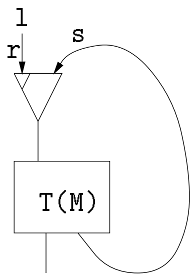



| | |
|---|---|
| `l_in : aller à bl_out;`<br>`bl_in: dépiler le sommet de` $v$`;`<br>`si c'est tt,`<br>`aller à ttl_out;`<br>`sinon`<br>`aller à ffl_out;`<br>`ttl_in:`<br>`ffl_in: aller à l_out;` | `l1_in: aller à l2_out;`<br>`l2_in: si le sommet de` $v$ `est tt,`<br>`le dépiler;`<br>`aller à l3_out;`<br>`sinon`<br>`aller à l1_out;`<br>`l3_in: aller à l1_out;` |

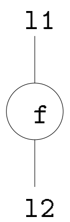
`or` se code comme `and` sauf que le test est "`si le sommet de` $v$ `est ff`".

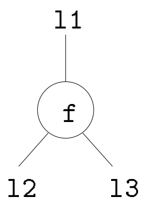
– récursivité (Y) : on se sert ici de nœuds déjà existants. Il est inutile d'en introduire de nouveaux.




Y (fn x =>M)

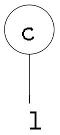
– fonctions arithmétiques : les opérations à un argument (`succ`, `pred`, `iszero`, `not`) sont codées suivant le schéma de gauche, alors que les opérations à 2 arguments (`+`, `-`, `*`, `/`, `%`, `<>`, `<=`, `<`, `=`, `xor`) utilisent celui de droite.



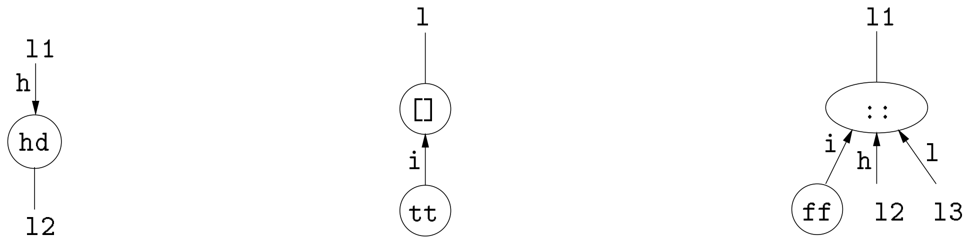

| | |
|---|---|
| `l1_in: aller à l2_out;`<br>`l2_in: f`<br>`aller à l1_out;` | `l1_in: aller à l2_out;`<br>`l2_in: aller à l3_out;`<br>`l3_in: f`<br>`aller à l1_out;` |

– constantes : les constantes entières, `tt` et `ff` sont codés suivant le même schéma.




```
l_in: c
     aller à l_out;
```

– listes : les fonctions `hd`, `tl` et `isnil` sont codées suivant le même principe, avec les opérations h, l et i respectivement. C'est pourquoi seul le code de `hd` est donné.




| | | |
|---|---|---|
| `l1_in:`<br>`  h`<br>`  aller à l2_out;`<br>`l2_in:`<br>`  h*`<br>`  aller à l1_out;` | `l_in:`<br>`  si le sommet de` $d$ `est i,`<br>`    tt`<br>`    aller à l_out;`<br>`  sinon`<br>`    erreur` | `l1_in:`<br>`  suivant le sommet de` $d$<br>`    h: h*`<br>`       aller à l2_out;`<br>`    l: l*`<br>`       aller à l3_out;`<br>`    i: ff`<br>`       aller à l1_out;`<br>`l2_in:`<br>`  h`<br>`  aller à l1_out;`<br>`l3_in:`<br>`  l`<br>`  aller à l1_out;` |

L'affichage des listes est effectué en exécutant une boucle qui teste d'abord si la liste est vide, puis si elle ne l'est pas, calcule et affiche sa tête, puis recommence avec la queue de la liste. Le code de cette partie est écrit directement en C.

On a réalisé une deuxième version de cette machine, qui comprend plusieurs optimisations :

– suppression des `goto` autant que possible, pour faciliter l'optimisation du code par `gcc`. Ceci fait perdre quelque peu la structure en nœuds du programme qui était très visible sur la version précédente (un morceau de code C pour chaque nœud du graphe), mais la compréhension du code C obtenu est plutôt plus facile car on regroupe les parties qui seront exécutées en séquence.

– amélioration des accès aux variables : les contractions sont regroupées en une seule grande contraction, dans laquelle à l'aller on stocke l'adresse où se brancher, et au retour on se branche à cette adresse. D'autre part, le groupe déréliction-sortie de boîte est optimisé en !*. Enfin, les successions de sorties de boîtes sont regroupées en une seule opération, qui prend approximativement le même temps qu'une seule sortie de boîte (tant que le nombre de sorties est assez faible, ce qui est le cas en général).

– diminution de la taille de la structure de données utilisée, par suppression des feuilles de l'arbre binaire. Ceci a aussi pour effet d'accélérer le fonctionnement, car la réservation et la libération de mémoire prennent une partie importante du temps.

On verra qu'au total ces optimisations permettent de gagner un facteur 2 ou 3 sur le temps d'exécution de beaucoup de programmes, d'autant plus quand il y a beaucoup de boîtes imbriquées. Dans ce cas, la part la plus importante du gain vient de la deuxième optimisation.

## 2 Une optimisation : la pile de questions

Cette nouvelle machine est décrite dans [7] et dans [9].

Cette optimisation de la machine précédente consiste à retourner immédiatement au nœud appelant quand on arrive à une valeur de type de base. Ainsi, chaque nœud qui cherche à obtenir une valeur de base pose une question, c'est-à-dire qu'il sauvegarde sur la pile de questions l'environnement et son adresse actuelle. Quand le programme rencontre la valeur correspondante, il restaure l'environnement et retourne à l'endroit où la question a été posée.

Ainsi, la longueur du chemin est divisée par deux (exactement pour le $\lambda$-calcul pur car on ne rencontrera qu'une valeur de type de base au milieu du calcul; un peu moins dans les autres cas, car on s'arrête sur le chemin de retour à chaque question posée). La contrepartie est un plus grand besoin de mémoire et le temps passé aux copies d'environnements.

## 3 La machine de Krivine

Cette optimisation est proposée dans [5].

Cette machine peut-être vue comme une généralisation de la machine à pile de questions : ce n'est plus seulement pour les valeurs de type simple qu'on pose les questions, mais pour toutes les valeurs. Cependant, ceci introduit une différence de conception assez importante, dans la mesure où il n'y a jamais de retour en arrière. Les directions de parcours des nœuds restent toujours les mêmes, et les données multiplicatives qui étaient nécessaires pour les machines précédentes ne le sont plus ici. L'accès aux variables est maintenant beaucoup plus simple : il se fait en une seule étape, par restauration globale de l'environnement et saut à l'adresse voulue. Cependant, cette opération peut être longue.

*Remarque :* Dans [5], le retour se fait au niveau de la déréliction quand on fait appel à une variable. Il faut alors retraverser le tenseur $\otimes$ pour aller chercher le paramètre voulu, et cela nécessite de conserver les données multiplicatives. Ici, on va directement à l'adresse du paramètre cherché, ce qui simplifie le processus.

On peut considérer que l'arbre binaire décrit dans la section 1 est séparé en deux parties : de la racine jusqu'à `ep` exclu on a l'environnement, alors que le sous-arbre de racine `ep` représente la pile. L'effet de chaque nœud peut alors être décrit par le tableau suivant en ce qui concerne le $\lambda$-calcul pur (les variables sont ici désignées par un numéro qui correspond au nombre de $\lambda$ à remonter pour trouver leur déclaration).

| Adresse | Env. | Pile | | Adresse | Env. | Pile |
|---|---|---|---|---|---|---|
| M N | e | s | $\longrightarrow$ | M | e | (N,e):s |
| $\lambda$M | e | u:s | $\longrightarrow$ | M | u:e | s |
| 1 | (N,e'):e | s | $\longrightarrow$ | N | e' | s |
| n+1 | (N,e'):e | s | $\longrightarrow$ | n | e | s |

Le tableau ci-dessous donne quelques exemples de codage des constantes, des fonctions prédéfinies et des listes. Par manque de place, les transitions sont ici disposées sous la forme d'un trait horizontal, avec au-dessus l'état initial et en-dessous l'état final.

| Adresse | Env. | Pile | Valeurs | Listes |
|---|---|---|---|---|
| $c$ | e | (N, e'):s | $v$ | |
| N | e' | s | $c$:$v$ | |
| `cond` be tte ffe | e | s | | |
| be | e | (`cond2` tte ffe, e):s | | |
| `cond2` tte ffe | e | s | `tt`:$v$ | |
| tte | e | s | $v$ | |
| `cond2` tte ffe | e | s | `ff`:$v$ | |
| ffe | e | s | $v$ | |
| M `+` N | e | s | | |
| M | e | (`+` N, e):s | | |
| `+` N | e | s | | |
| N | e | (`+`, NIL):s | | |
| `+` | e | (N, e'):s | $c_1$:$c_2$:$v$ | |
| N | e' | s | $(c_1 + c_2)$:$v$ | |
| M::N | e | (P, e'):s | $v$ | i:$d$ |
| P | e' | s | `ff`:$v$ | $d$ |
| M::N | e | s | $v$ | h:$d$ |
| M | e | s | $v$ | $d$ |
| M::N | e | s | $v$ | l:$d$ |
| N | e | s | $v$ | $d$ |
| `[]` | e | (P, e'):s | $v$ | i:$d$ |
| P | e' | s | `tt`:$v$ | $d$ |
| `hd` M | e | s | $v$ | $d$ |
| M | e | s | $v$ | h:$d$ |

Les autres constantes sont codées de façon analogue. Dans la colonne adresse, j'ai en fait indiqué le terme de $\lambda$-calcul qui est évalué dans le programme à l'adresse en question. Le ":" dénote la concaténation d'éléments dans une liste.

En mettant un environnement vide NIL dans la pile quand on évalue `+` N, on évite une sauvegarde inutile (car on n'a pas besoin de l'environnement pour évaluer `+`).

# 4 L'appel par valeur

L'appel par valeur est une modification de la machine de Krivine, qui, elle, implémentait l'appel par nom comme la GOI. Le tableau précédent doit être remplacé par celui-ci :

| Adresse | Env. | Pile | | Adr. | Env. | Pile |
|---|---|---|---|---|---|---|
| M N | e | s | $\longrightarrow$ | M | e | L(N,e):s |
| $\lambda$M | e | L(N, e'):s | $\longrightarrow$ | N | e' | (M, e):s |
| | e | (N, e'):s | $\longrightarrow$ | N | ($\lambda$M, e):e' | s |
| 1 | (N, e'):e | s | $\longrightarrow$ | N | e' | s |
| 1 | v($c$):e | (N, e'):s | $\longrightarrow$ | N | v($c$):e' | s |
| n+1 | (N, e'):e | s | $\longrightarrow$ | n | e | s |
| $c$ | e | (N, e'):s | $\longrightarrow$ | N | v($c$):e' | s |
| `cond` be tte ffe | e | s | $\longrightarrow$ | be | e | (`cond2` tte ffe,e):s |
| `cond2` tte ffe | v(`tt`):e | s | $\longrightarrow$ | tte | e | s |
| | v(`ff`):e | s | $\longrightarrow$ | ffe | e | s |
| M `+` N | e | s | $\longrightarrow$ | M | e | (`+` N,e):s |
| `+` N | v($c$):e | s | $\longrightarrow$ | N | e | (`+`, NIL):v($c$):s |
| `+` | v($c_1$):e | v($c_2$):(N,e'):s | $\longrightarrow$ | N | v($c_1 + c_2$):e' | s |
| M::N | e | h:s | $\longrightarrow$ | M | e | s |
| | e | l:s | $\longrightarrow$ | N | e | s |
| | e | i:(P, e'):s | $\longrightarrow$ | P | v(`ff`):e' | s |
| | e | (P, e'):s | $\longrightarrow$ | P | (M::N, e):e' | s |
| `[]` | e | i:(N, e'):s | $\longrightarrow$ | N | v(`tt`):e' | s |
| | e | (N, e'):s | $\longrightarrow$ | N | (`[]`,e):e' | s |
| `hd` M | e | s | $\longrightarrow$ | M | e | h:s |

La lettre L représente un drapeau qui indique que la valeur est à évaluer. v(x) représente la valeur de type simple x.

# 5 L'appel par nécessité

Le système est encore analogue mais cette fois le tableau est le suivant :

| Adresse | Env. | Pile | Affectations |
|---|---|---|---|
| M N | e | s | |
| M | e | (N,e):s | |
| $\lambda$M | e | UPD($a_1$):...:UPD($a_n$):u:s | |
| M | u:e | s | $a_1$ := ... := $a_n$ :=($\lambda$M,e) |
| 1 | ($\lambda$M,e'):e | s | |
| $\lambda$M | e' | s | |

| Adresse | Env. | Pile | Affectations |
|---|---|---|---|
| 1 | v($c$):e | UPD($a_1$):...:UPD($a_n$):(N,e'):s | |
| N | e' | s | $a_1$ := ... := $a_n$ :=v($c$) |
| 1 | $a$=(N,e'):e | s | |
| N | e' | UPD($a$):s | |
| n+1 | (N,e'):e | s | |
| n | e | s | |
| $c$ | e | UPD($a_1$):...:UPD($a_n$):(N,e'):s | |
| N | e' | s | $a_1$ := ... := $a_n$ :=v($c$) |
| `cond` be tte ffe | e | s | |
| be | e | UPD($a$):(`cond2` tte ffe,e):$a$=v(`ff`):s | |
| `cond2` tte ffe | e | v(`tt`):s | |
| tte | e | s | |
| `cond2` tte ffe | e | v(`ff`):s | |
| ffe | e | s | |
| M `+` N | e | UPD($a$):s | |
| M | e | UPD($a$): (`+` N,e):UPD($a$):s | |
| `+` N | e | s | |
| N | e | UPD($a$):(`+`,e):$a$=v(0):s | |
| `+` | e | v($c_2$):UPD($a_1$):...:UPD($a_n$):(N,e'):s | |
| N | e' | s | $a_1$=v($c_1$);<br>$a_1$ := ...$a_n$ :=v($c_1 + c_2$) |
| M::N | e | UPD($a$):s | |
| M::N | e | s | $a$:=(M::N,e) |
| M::N | e | h:s | |
| M | e | s | |
| M::N | e | l:s | |
| N | e | s | |
| M::N | e | i:UPD($a_1$):...:UPD($a_n$):(P,e'):s | |
| P | e' | s | $a_1$ := ... := $a_n$ :=v(`ff`) |
| `[]` | e | UPD($a$):s | |
| `[]` | e | s | $a$:=(`[]`,e) |
| `[]` | e | i:UPD($a_1$):...:UPD($a_n$):(N,e'):s | |
| N | e' | s | $a_1$ := ... := $a_n$ :=v(`tt`) |
| `hd` M | e | s | |
| M | e | h:s | |

La valeur UPD($a$) indique que la valeur à l'adresse $a$ devra être mise à jour quand on aura obtenu un résultat. Ainsi, dès qu'un résultat est obtenu, il est enregistré et n'est plus recalculé. UPD($a_1$):...:UPD($a_n$) représente un nombre quelconque de valeurs UPD (éventuellement zéro). v(x) représente la valeur de type simple x. La notation $a$=x signifie que la valeur x se situe à l'adresse $a$, alors que la notation $a$:=x signifie qu'on affecte la valeur x à l'adresse $a$.

La première ligne du tableau concernant l'adresse "1" est une optimisation : si la valeur est déjà en forme normale en tête, on ne peut poursuivre son évaluation, donc il est inutile de rajouter une valeur UPD dans la pile. Dans le compilateur, cette optimisation est implémentée sous une forme légèrement plus restrictive, qui est que si la valeur a déjà été évaluée, on ne remet pas UPD sur la pile.

Les partages de données, nécessaires pour que les valeurs soient mises à jour partout où elles sont utiles, forcent à utiliser des pointeurs plutôt que des copies des environnements. Ceci empêche de savoir quand il faut libérer la mémoire. Il faudrait donc implémenter un "glaneur de cellules" (GC), ce qui n'a pas été fait ici. Cette machine consomme donc beaucoup de mémoire, et sa rapidité est quelque peu illusoire, puisque la présence d'un GC la ralentirait.

Du même coup, la structure arborescente de l'environnement n'est plus utilisable. Il a donc fallu passer à une structure de deux listes simplement chaînées, une pour l'environnement et une pour la pile.

## Troisième partie
# Evaluation des performances de ces machines

Les tests décrits ci-dessous ont été réalisés sur Sparc 20.

## 1 $\lambda$-calcul pur

Les exemples de $\lambda$-calcul pur sont extraits de [3] (sauf le dernier). Ils consistent en calculs arithmétiques avec les entiers de Church. Voici le programme utilisé :

```
let I = fn x => x;;

let zero = fn f => fn x => x;;
let one  = fn f => fn x => f x;;
let two  = fn f => fn x => f (f x);;
(* de même jusqu'à eleven ! *)

let Succ = fn n => fn x => fn y => (x (n x y));;
let Mult = fn n => fn m => fn x => (n (m x));;

let Pair = fn x => fn y => fn z => (z x y);;
let Fst  = fn x => fn y => x;;
let Snd  = fn x => fn y => y;;

let Nextfact = fn p => let n1 = (p Fst) in
               let n2 = (Succ (p Snd)) in
                     (Pair (Mult n1 n2) n2);;

let Fact = fn n => (n Nextfact (Pair one zero) Fst);;

let Add = fn m => fn n => fn x => fn y => (m x (n x y));;

let Nextfibo = fn p => let n1 = (p Fst) in
    let n2 = (p Snd) in (Pair (Add n1 n2) n1);;

let Fibo = fn n => (n Nextfibo (Pair zero one) Fst);;

let g = fn n => n two I tt;;
```

Les tableaux suivants donnent les temps d'exécution en ms dans différents exemples. La précision des mesures de temps est de 16 ms environ. Les petits temps ne sont donc pas très significatifs. Avec Caml-Light, ou les machines par valeur ou par nécessité, certains temps sont vraiment très courts. On a alors répété 100 fois l'évaluation pour obtenir une meilleure précision. Dans ce cas, le temps est donné au dixième de ms près.

D'autres tableaux indiquent le nombre de $\beta$-réductions effectuées. Pour l'appel par valeur, une $\beta$-réduction correspond à une instruction d et pour l'appel par nécessité et la machine de Krivine, à une instruction d et une !. Le nombre de $\beta$-réductions est le même pour toutes les machines à appel par nom, donc on n'a pas répété les valeurs, et elles se trouvent dans la colonne "Nom". Il se trouve que sur les exemples présentés tous les paramètres sont évalués, donc le nombre de $\beta$-réductions est le même pour les machines à appel par valeur et par nécessité.

Les machines GOI et questions-réponses parcourent les rédex non seulement pour effectuer les $\beta$-réductions proprement dites, mais aussi pour accéder aux variables, ce qui conduit à des valeurs beaucoup plus grandes. Pour la GOI, un parcours de rédex correspond à 2 instructions q et 2 q*; pour la machine à questions-réponses, à 1 instruction q et 1 q*; le nombre de rédex parcourus par ces deux machines est noté dans la colonne "Parcours de rédex".

On a indiqué une comparaison entre les deux versions de la géométrie de l'interaction, avec les nombres de dérélictions (D), de contractions (r et s) et de sorties de boîtes (t) (Dans la version optimisée, la sortie de plusieurs boîtes en une seule étape compte pour une sortie).

Enfin, un dernier tableau contient le nombre de cellules copiées par chacune des machines qui effectuent des copies (questions-réponses, Krivine et appel par valeur).

| Temps (ms) | GOI | GOI non-opt | Question | Krivine |
|---|---:|---:|---:|---|
| `Fact five I 1` | 383 | 1 116 | 216 | 114 183 |
| `Fact six I 1` | 2 516 | 6 666 | 1 266 | explose |
| `Fact seven I 1` | 19 050 | 52 166 | 9 650 | |
| `Fact nine I 1` | 1 680 816 | | | |
| `Fact ten I 1` | Trop long | | | |

| Temps (ms) | Valeur | Nécessité | Réd. opt. | Caml-Light |
|---|---:|---:|---:|---:|
| `Fact five I 1` | 55,1 | 7,7 | 0 | 0,8 |
| `Fact six I 1` | 273 | 42,1 | 20 | 3,9 |
| `Fact seven I 1` | 1 883 | 316 | 30 | 22,1 |
| `Fact nine I 1` | 158 000 | 19 416<br>+13 300 swap | 50 | 1 500 |
| `Fact ten I 1` | | explose | 70 | 16 350 |

| | GOI : Parcours | $\beta$-réductions | | |
|---|---:|---:|---:|---:|
| Expression | de rédex | Nom | Valeur, Nécessité | Réd. opt. |
| `Fact five I 1` | 20 620 | 1 586 | 545 | 84 |
| `Fact six I 1` | 131 050 | 8 795 | 2 720 | 100 |
| `Fact seven I 1` | 980 680 | 59 204 | 17 855 | 117 |
| `Fact nine I 1` | 80 562 340 | 4 091 222 | 1 227 485 | 154 |
| `Fact ten I 1` | Trop long | 40 379 231 | 12 113 900 | 174 |

| Expression | Dérélictions | | Contractions | |
|---|---|---|---|---|
| | GOI | GOI non-opt | GOI | GOI non-opt |
| `Fact five I 1` | 58 400 | 89 256 | 33 036 | 51 492 |
| `Fact six I 1` | 373 368 | 571 796 | 214 536 | 339 888 |
| `Fact seven I 1` | 2 809 936 | 4 310 176 | 1 634 436 | 2 756 628 |
| `Fact nine I 1` | 233 017 152 | | 138 057 276 | |

| Expression | Sorties de boîtes | |
|---|---|---|
| | GOI | GOI non-opt |
| `Fact five I 1` | 23 388 | 54 884 |
| `Fact six I 1` | 152 552 | 354 584 |
| `Fact seven I 1` | 1 166 356 | 2 690 284 |
| `Fact nine I 1` | 99 023 084 | |

| Expression | Nombre de cellules copiées | | |
|---|---|---|---|
| | Question | Krivine | Valeur |
| `Fact five I 1` | 1 | 7 601 289 | 4 233 |
| `Fact six I 1` | 1 | explose | 19 924 |
| `Fact seven I 1` | 1 | | 132 354 |
| `Fact nine I 1` | 1 | | 10 087 783 |

A l'exception de la machine de Krivine qui sature presque dès le début, et de la réduction optimale dont le nombre de $\beta$-réductions est quadratique (le nombre d'interactions augmentant cependant plus vite sans être exponentiel; voir [3]), toutes les machines ont une complexité exponentielle et, si elles ne varient pas linéairement les unes par rapport aux autres, les rapports en temps d'exécution croissent lentement mais sont toujours grands : par rapport à Caml-Light, la GOI est 500 à 1000 plus lente, l'appel par valeur 70 à 100 fois, et l'appel par nécessité 9 à 13 fois.

L'explosion de la machine à appel par nécessité est due à l'absence de GC qui conduit à une saturation très rapide de la mémoire. D'ailleurs, cette saturation se faisait déjà sentir pour `Fact nine I 1`, vu le temps passé au swap disque : la taille des données atteint dans cet exemple 58Mo ! (Plus précisément, 57 994 260 octets).

La lenteur de la machine à appel par valeur est essentiellement due au fait qu'elle effectue des copies d'environnements plutôt que des partages comme la machine à appel par nécessité.

La machine de Krivine, elle, sature beaucoup plus vite et est très lente, car elle copie beaucoup trop de données; beaucoup de cellules copiées se révéleront ensuite inutiles. Les autres machines à appel par nom sont un peu plus performantes. La machine GOI de base ne sature pratiquement jamais la mémoire car elle en nécessite très peu (920 octets seulement par exemple pour `Fact seven I 1`. Les autres consommations mémoire de cette machine sont du même ordre de grandeur : quelques kilo-octets au maximum), mais ses temps de calcul deviennent vite prohibitifs. C'est pourquoi j'ai abandonné l'exemple `Fact ten I 1`. La version optimisée de la GOI est à peu près 3 fois plus rapide que la version non optimisée, et la machine à questions-réponses est encore 2 fois plus rapide. Ceci est logique, car dans le cas du $\lambda$-calcul pur, on rencontre à chaque fois une valeur de type simple à la moitié du calcul et ensuite la machine GOI suit tout le chemin déjà parcouru en sens inverse avant de donner le résultat alors que la machine à questions réponses donne aussitôt le résultat. Il n'y a de sauvegarde d'environnement que celle de l'environnement vide initial. Ceci explique le nombre de cellules copiées de 1, et la faiblesse de la consommation mémoire : seulement 36 octets de plus que la GOI.

Le tableau de comparaison entre la GOI optimisée ou non montre clairement le progrès apporté par l'optimisation des accès aux variables : le nombre de dérélictions est réduit d'un facteur 1,5, le nombre de contractions de 1,5 à 1,7, et le nombre de sorties de boîtes d'un facteur 2,3. Les résultats sont analogues pour `Fibo` (excepté que le facteur de réduction du nombre de sorties de boîtes est plus faible, 2 environ), et pour `g`.

| Temps (ms) | GOI | GOI non-opt | Question | Krivine |
|---|---|---|---|---|
| `Fibo seven I 1` | 133 | 316 | 66 | 966 |
| `Fibo ten I 1` | 666 | 1 783 | 316 | 20 716 |
| `Fibo (Add three ten) I 1` | 7 016 | 16 350 | 3 483 | 1 140 000 |
| `Fibo (Add six ten) I 1` | 33 850 | 78 683 | 16 866 | explose |

| Temps (ms) | Valeur | Nécessité | Réd. opt. | Caml-Light |
|---|---|---|---|---|
| `Fibo seven I 1` | 46,6 | 3,1 | 30 | 0,6 |
| `Fibo ten I 1` | 236 | 10,8 | 30 | 1,2 |
| `Fibo (Add three ten) I 1` | 1 150 | 50 | 100 | 3,2 |
| `Fibo (Add six ten) I 1` | 5 450 | 150 | 400 | 14,2 |

| | GOI : Parcours | $\beta$-réductions | | |
|---|---|---|---|---|
| Expression | de rédex | Nom | Valeur, Nécessité | Réd. opt. |
| `Fibo seven I 1` | 5 806 | 704 | 239 | 106 |
| `Fibo ten I 1` | 29 440 | 3 024 | 757 | 181 |
| `Fibo (Add three ten) I 1` | 323 368 | 12 866 | 2 841 | 398 |
| `Fibo (Add six ten) I 1` | 1 525 678 | 54 530 | 11 527 | 1 185 |

| | Nombre de cellules copiées | | |
|---|---|---|---|
| Expression | Question | Krivine | Valeur |
| `Fibo seven I 1` | 1 | 78 239 | 4 027 |
| `Fibo ten I 1` | 1 | 1 564 591 | 20 063 |
| `Fibo (Add three ten) 1` | 1 | 77 169 769 | 97 623 |
| `Fibo (Add six ten) I 1` | 1 | explose | 455 941 |

Cet exemple est particulièrement favorable à Caml-Light et à l'appel par nécessité qui sont ici les deux machines les plus performantes. On voit clairement sur cet exemple la croissance exponentielle du nombre de cellules copiées par la machine de Krivine, avant l'explosion. (On ne pouvait le voir sur l'exemple précédent car la saturation intervenait trop tôt).

Ici, même la réduction optimale a une complexité exponentielle. La non-linéarité de la GOI et de ses dérivés par rapport à Caml-Light est plus marquée ici : la GOI est environ 200 à 2000 fois plus lente que Caml-Light, l'appel par valeur 80 à 400 fois et l'appel par nécessité 5 à 10 fois.

| Temps (ms) | GOI | GOI non-opt | Question | Krivine |
|---|---|---|---|---|
| `g ten` | 216 | 633 | 116 | 1 416 |
| `g (Add three ten)` | 4 983 | 13 616 | 2 283 | 50 166 |
| `g (Add six ten)` | 38 716 | 96 866 | 19 183 | 430 000 |

| Temps (ms) | Valeur | Nécessité | Réd. opt. | Caml-Light |
|---|---|---|---|---|
| `g ten` | 72 | 28,3 | 10 | 3,5 |
| `g (Add three ten)` | 583 | 200 | 20 | 25,8 |
| `g (Add six ten)` | 4 700 | 2 066 | 20 | 210,4 |

| Expression | GOI : Parcours de rédex | $\beta$-réductions Nom | Valeur, Nécessité | Réd. opt. |
|---|---|---|---|---|
| `g ten` | 6 146 | 3 073 | 2 060 | 34 |
| `g (Add three ten)` | 262 158 | 24 597 | 16 405 | 49 |
| `g (Add six ten)` | 2 097 278 | 196 741 | 131 096 | 58 |

Cet exemple est très favorable à la réduction optimale, qui est linéaire. La complexité de toutes les autres machines croît exponentiellement car aucune ne sait profiter au mieux des partages de rédex.

| Temps (ms) | GOI | GOI non-opt | Question | Krivine |
|---|---|---|---|---|
| `I I ... I I 3` (12 I) | 116 | 300 | 50 | 0,1 |
| `I I ... I I 3` (18 I) | 6 716 | 14 833 | 3 316 | 0,2 |
| `I I ... I I 3` (22 I) | 107 583 | 262 583 | 53 216 | 0,2 |

| Temps (ms) | Valeur | Nécessité | Réd. opt. | Caml-Light |
|---|---|---|---|---|
| `I I ... I I 3` (12 I) | 0,1 | 0,2 | 0 | 0,04 |
| `I I ... I I 3` (18 I) | 0,2 | 0,3 | 0 | 0,05 |
| `I I ... I I 3` (22 I) | 0,3 | 0,4 | 0 | 0,06 |

| Expression | GOI : parcours de rédex | $\beta$-réductions |
|---|---|---|
| `I I ... I I 3` (12 I) | 4 095 | 12 |
| `I I ... I I 3` (18 I) | 262 153 | 18 |
| `I I ... I I 3` (22 I) | 4 194 303 | 22 |

| Expression | Nombre de cellules copiées Question | Krivine, Valeur |
|---|---|---|
| `I I ... I I 3` | 1 | 0 |

Sur cet exemple la machine de Krivine est très efficace, car son environnement reste vide, ce qui évite toute copie de données (le dernier tableau le montre), alors que pour la GOI, le nombre d'étapes de recherche d'une variable est très important (ceci se voit sur le nombre de parcours de rédex qui croît exponentiellement, alors que le nombre de $\beta$-réductions proprement dites est égal au nombre d'identités dans le terme). La GOI et la machine à pile de questions ont ici une complexité exponentielle alors que les autres ont une complexité linéaire.

Sur ce type de terme très simple, la principale optimisation de la deuxième version de la GOI est la diminution du nombre de contractions, car `I` est partagé grâce à des contractions. Les sorties de boites et les dérélictions restent inchangées. Le résultat sur le temps d'exécution est quand même appréciable.

## 2 Listes

Les exemples sur les listes utilisent le listing suivant :

```
let rec length = fn a =>
   if isnil a
   then 0
   else succ (length (tl a));;

let rec append = fn a => fn b =>
   if isnil a
   then b
   else (hd a) :: append (tl a) b;;

let rec rev2 = fn x =>
   if isnil x
   then x
   else (append (rev2 (tl x)) [hd x]);;

(* listIt *)
let rec listIt = fn f => fn l => fn e =>
   if isnil l
   then e
   else (f (hd l) (listIt f (tl l) e));;

(* partition prend une fonction de test booléenne t et une liste l
et la partage en deux sous-listes, suivant le résultat du test *)
let partition = fn t => fn l =>
   let switch = fn e => fn l2 =>
      if (t e) then [hd l2; e :: hd (tl l2)]
               else [e :: hd l2; hd (tl l2)]
```

```
   in (listIt switch l [[];[]]);;

(* quicksort *)
let rec quicksort = fn l =>
   if isnil l
   then []
   else if (length l) = 1
        then l
        else let l1 = hd(partition (fn x => (hd l) <= x) (tl l)) in
             let l2 = hd(tl(partition (fn y => (hd l) < y) (tl l)))
             in (append (quicksort l1) ((hd l)::(quicksort l2)));;

(* genlist prend un entier n et crée la liste des entiers de n à 0
dans l'ordre décroissant *)
let rec genlist = fn n =>
   if iszero n
   then []
   else n :: (genlist n-1);;

let parti = fn a => fn l =>
   let switch = fn e => fn l2 =>
      if e < a then [e :: hd l2; hd (tl l2)]
                else if e > a then [hd l2; e :: hd (tl l2)]
                          else l2
   in listIt switch l [[];[]];;

let rec quicksort2 = fn [] => []
                      | [a] => [a]
                      | (a::l) => let part = parti a l in
                                  append (quicksort2 (hd part))
                                   (a::quicksort2 (hd (tl part)));;

let rec sigma = fn [] => 0
                 | a::l => a + (sigma l);;
```

L'algorithme de retournement de liste est volontairement peu efficace, pour voir facilement les différences de vitesse sans avoir à utiliser des listes trop longues.

La première version de quicksort est prise dans les exemples du logiciel de réduction optimale, tandis que la deuxième est une version un peu plus optimisée et en même temps un exemple d'utilisation du filtrage.

Le compilateur de la réduction optimale ne peut pas afficher les listes. Pour pouvoir l'inclure dans les comparaisons, il a donc fallu rajouter une application de fonction (`sigma`) pour obtenir un entier et non une liste.

| Temps (ms) | GOI | GOI non-opt | Question | Krivine |
|---|---|---|---|---|
| `sigma(rev2 [1;2;3;4;5;6;7])` | 3 516 | 6 633 | 17 116 | 7 783 |
| `sigma(rev2 [1;2;...;9;10])` | 199 033 | 373 400 | | |
| `sigma(quicksort (genlist 4))` | 19 716 | 41 800 | 192 483 | 113 983 |
| `sigma(quicksort (genlist 5))` | 493 183 | 1 000 416 | | |
| `sigma(quicksort2 (genlist 4))` | 10 616 | 21 700 | 121 666 | 50 233 |
| `sigma(quicksort2 (genlist 5))` | 258 416 | 519 033 | | |

| Temps (ms) | Valeur | Nécessité | Réd. opt. | Caml-Light |
|---|---|---|---|---|
| `sigma(rev2 [1;2;3;4;5;6;7])` | 11,7 | 2,8 | 30 | 0,6 |
| `sigma(rev2 [1;2;...;9;10])` | 27,7 | 3,8 | 50 | 1,0 |
| `sigma(quicksort (genlist 4))` | 516 | 5,8 | 40 | 0,9 |
| `sigma(quicksort (genlist 5))` | 5 883 | 9,1 | 90 | 1,3 |
| `sigma(quicksort2 (genlist 4))` | 183 | 2,8 | 40 | 0,6 |
| `sigma(quicksort2 (genlist 5))` | 1 050 | 4,8 | 70 | 0,7 |

| Expression | GOI : Parcours de rédex | $\beta$-réductions Nom | Valeur, Nécessité | Réd. opt. |
|---|---|---|---|---|
| `sigma(rev2 [1;2;3;4;5;6;7])` | 34 982 | 29 078 | 72 | 30 |
| `sigma(rev2 [1;2;...;9;10])` | 1 480 041 | 1 247 912 | 132 | 42 |
| `sigma(quicksort (genlist 4))` | 374 258 | 247 165 | 159 | 59 |
| `sigma(quicksort (genlist 5))` | 8 740 858 | 5 723 300 | 239 | 71 |
| `sigma(quicksort2 (genlist 4))` | 190 925 | 122 663 | 91 | 49 |
| `sigma(quicksort2 (genlist 5))` | 4 425 295 | 2 820 365 | 133 | 60 |

| | Dérélictions | | Contractions | |
|---|---|---|---|---|
| Expression | GOI | GOI non-opt | GOI | GOI non-opt |
| `sigma(rev2 [1;2;3;4;5;6;7])` | 56 386 | 282 356 | 504 926 | 727 480 |
| `sigma(rev2 [1;2;...;9;10])` | 2 363 418 | 15 017 768 | 27 538 182 | 40 058 574 |
| `sigma(quicksort (genlist 4))` | 901 126 | 2 014 382 | 3 140 964 | 3 880 070 |
| `sigma(quicksort (genlist 5))` | 21 083 312 | 48 410 974 | 76 192 254 | 94 319 696 |
| `sigma(quicksort2 (genlist 4))` | 485 564 | 1 019 968 | 1 681 958 | 2 001 442 |
| `sigma(quicksort2 (genlist 5))` | 11 219 444 | 24 394 754 | 40 521 022 | 49 801 990 |

| Expression | Sorties de boîtes GOI | GOI non-opt |
|---|---|---|
| `sigma(rev2 [1;2;3;4;5;6;7])` | 44 718 | 270 668 |
| `sigma(rev2 [1;2;...;9;10])` | 1 860 560 | 14 514 910 |
| `sigma(quicksort (genlist 4))` | 814 118 | 2 198 006 |
| `sigma(quicksort (genlist 5))` | 20 153 164 | 54 875 202 |
| `sigma(quicksort2 (genlist 4))` | 397 668 | 1 078 974 |
| `sigma(quicksort2 (genlist 5))` | 9 817 388 | 26 918 910 |

| | Nombre de cellules copiées | | |
|---|---|---|---|
| Expression | Question | Krivine | Valeur |
| `sigma(rev2 [1;2;3;4;5;6;7])` | 916 672 | 498 638 | 856 |
| `sigma(rev2 [1;2;...;9;10])` | | | 2 167 |
| `sigma(quicksort (genlist 4))` | 11 821 795 | 9 467 512 | 34 317 |
| `sigma(quicksort (genlist 5))` | | | 336 174 |
| `sigma(quicksort2 (genlist 4))` | 7 228 753 | 4 085 084 | 11 736 |
| `sigma(quicksort2 (genlist 5))` | | | 59 636 |

Pour l'exemple ci-dessus, la machine à questions réponses est moins rapide que les GOI optimisée et non optimisée, ce qui s'explique par les nombreuses questions posées (à chaque expression conditionnelle, test `isnil` ou `iszero` ou opération arithmétique) qui conduisent à beaucoup de copies. La machine de Krivine est ici plus rapide que la machine de questions-réponses, ce qui surprend quelque peu étant donné ses performances décevantes dans les premiers exemples. Ceci est nettement corrélé avec le nombre de copies, qui est lui aussi plus faible pour la machine de Krivine que pour les questions-réponses.

Du fait que chaque variable est appelée plusieurs fois, les machines à appel par nom ont une complexité qui croît nettement plus vite que celle des autres machines, et ceci se retrouve sur le nombre de $\beta$-réductions, beaucoup plus élevé pour l'appel par nom que pour les appels par valeur et par nécessité.

L'optimisation de la GOI réduit de façon très importante le nombre de dérélictions et de sorties de boîtes dans l'exemple du retournement de listes (le rapport est de 5 à 8). Par contre, l'effet sur les contractions est peu important, et les conséquences sur le temps d'exécution sont sensiblement les mêmes que pour le $\lambda$-calcul pur (facteur 2 environ).

En temps d'exécution, l'appel par nécessité et Caml Light sont plus performantes que la réduction optimale. Ceci s'explique car l'avantage en nombre de $\beta$-réductions est faible en faveur de la réduction optimale (facteur 2 à 3) et ne suffit pas à compenser la lourdeur du travail sur le graphe du programme.

## 3 Calculs arithmétiques

Les exemples de calculs arithmétiques sont pris dans le programme suivant :

```
(* Fonctions arithmétiques *)
```

```
let rec add = fn x => fn y =>
   if iszero x
   then y
   else (succ (add (pred x) y));;

let rec mult = fn a => fn b =>
   if iszero a
   then 0
   else (add b (mult (pred a) b));;

let rec power = fn x => fn y =>
   if iszero y
   then 1
   else (mult x (power x (pred y)));;
```

| Temps (ms) | GOI | GOI non-opt | Question | Krivine |
|---|---|---|---|---|
| `add 50 3` | 33 | 83 | 66 | explose |
| `add 250 3` | 483 | 1 550 | 1 866 | explose |
| `mult 10 12` | 50 | 100 | 266 | explose |
| `mult 50 50` | 3 216 | 7 833 | 92 300 | explose |
| `power 4 2` | 66 | 133 | 266 | 1216 |
| `power 4 3` | 4 633 | 11 716 | 47 633 | explose |

| Temps (ms) | Valeur | Nécessité | Réd. opt. | Caml-Light |
|---|---|---|---|---|
| `add 50 3` | 4,8 | 4,0 | 30 | 0,4 |
| `add 250 3` | 24,6 | 20,8 | 130 | 1,8 |
| `mult 10 12` | 13,6 | 10,3 | 50 | 1,1 |
| `mult 50 50` | 248 | 200 | 750 | 14,3 |
| `power 4 2` | 3,6 | 1,3 | 20 | 0,3 |
| `power 4 3` | 11,0 | 7,6 | 50 | 0,7 |

| | GOI : Parcours | $\beta$-réductions | | |
|---|---|---|---|---|
| Expression | de rédex | Nom | Valeur, Nécessité | Réd. opt. |
| `add 50 3` | 153 | 102 | 102 | 102 |
| `add 250 3` | 753 | 502 | 502 | 502 |
| `mult 10 12` | 1 127 | 282 | 282 | 48 |
| `mult 50 50` | 72 777 | 5 202 | 5 202 | 204 |
| `power 4 2` | 2 083 | 932 | 82 | 26 |
| `power 4 3` | 154 339 | 63 524 | 230 | 52 |

| Expression | Dérélictions | | Contractions | |
|---|---|---|---|---|
| | GOI | GOI non-opt | GOI | GOI non-opt |
| `add 50 3` | 310 | 3 060 | 5 716 | 5 716 |
| `add 250 3` | 1 510 | 65 260 | 128 516 | 128 516 |
| `mult 10 12` | 2 256 | 5 896 | 8 150 | 8 150 |
| `mult 50 50` | 145 556 | 415 956 | 556 510 | 556 510 |
| `power 4 2` | 3 808 | 8 108 | 10 124 | 10 124 |
| `power 4 3` | 284 064 | 617 372 | 770 156 | 770 156 |

| Expression | Sorties de boîtes | |
|---|---|---|
| | GOI | GOI non-opt |
| `add 50 3` | 5 402 | 8 152 |
| `add 250 3` | 127 002 | 190 752 |
| `mult 10 12` | 5 592 | 9 232 |
| `mult 50 50` | 405 652 | 676 052 |
| `power 4 2` | 5 940 | 10 240 |
| `power 4 3` | 460 372 | 793 680 |

| Expression | Nombre de cellules copiées | | |
|---|---|---|---|
| | Question | Krivine | Valeur |
| `add 50 3` | 1 530 | | 302 |
| `add 250 3` | 64 006 | | 1 502 |
| `mult 10 12` | 12 256 | | 842 |
| `mult 50 50` | 3 526 906 | | 15 602 |
| `power 4 2` | 15 140 | 95 468 | 242 |
| `power 4 3` | 2 554 412 | | 686 |

Le calcul de puissance, `power x y`, est très défavorable à l'appel par nom, car les paramètres sont évalués plusieurs fois, et ceci récursivement, d'où un rapport exponentiel en $y$ : à $y$ fixé, le nombre de $\beta$-réductions est un polynôme en $x$ de degré $\frac{y(y+1)}{2}$ pour l'appel par nom, et seulement $y$ pour les appels par valeur et par nécessité.

L'optimisation de la GOI réduit de façon importante le nombre de dérélictions (Le rapport, qui croît avec le premier nombre à ajouter, varie de 10 à 40 dans les exemples cités). Par contre, le nombre de contractions est inchangé car chaque variable est référencée au plus deux fois. Les sorties de boîtes sont aussi moins nombreuses mais le facteur avoisine seulement 1,5.

L'évaluation par valeur ou par nécessité n'effectue pas plus de $\beta$-réductions que la réduction optimale dans le cas de l'addition. La complexité des opérations sur le graphe du programme fait que cette dernière est moins rapide que les machines à appel par valeur ou par nécessité et que Caml Light dans cet exemple. Elle est moins défavorisée pour la multiplication et l'élévation à la puissance mais reste plus lente.

Contrairement au $\lambda$-calcul pur, dans les cas de calculs arithmétiques ou sur les listes, la machine à questions-réponses est moins performante car il y a beaucoup d'arrêts sur le chemin du retour pour modifier la valeur calculée. Alors le temps pris par les copies d'environnements devient important, et la machine est nettement moins rapide que la GOI. Dans les programmes habituels, cette optimisation n'est donc pas très intéressante (l'évaluation de termes du $\lambda$-calcul pur n'est pas ce qu'on rencontre le plus souvent dans les applications pratiques!).

Ce problème de copies trop nombreuses se pose aussi pour la machine de Krivine : dans l'exemple de l'addition, on a la succession d'états suivante (si `x1` = `pred x` $\neq 0$):

| Terme à évaluer | Env. | Pile |
|---|---|---|
| `add (pred x) y` | e | s |
| `add = fn x1 => fn y1 => if ...` | e | (`pred x`,e):(y,e):s |
| `fn y1 => if iszero x1 then y1 else (succ ...)` | (`pred x`,e):e | (y,e):s |
| `if iszero x1 then y1 else (succ ...)` | (y,e):(`pred x`,e):e | s |
| `succ (add (pred x1) y1)` | (y,e):(`pred x`,e):e | s |
| `add (pred x1) y1` | (y,e):(`pred x`,e):e | (`succ`, NIL):s |

La taille de l'environnement triple donc à chaque appel récursif, et il y a autant d'appels que la valeur du premier nombre à ajouter, `x`, donc la taille des données croît exponentiellement en `x`. Ceci explique l'explosion rapide de la machine de Krivine, et comme un phénomène analogue se produit dans les autres exemples, cela fait d'elle la moins performante des implémentations réalisées ici, aussi bien en temps d'exécution qu'en occupation mémoire. Une implémentation qui partagerait les environnements au lieu de les copier permettrait à la taille de l'environnement de croître seulement linéairement, et serait donc beaucoup plus performante. Cependant, on ne sait alors plus quand libérer la mémoire et il faut utiliser un GC (glaneur de cellules) [1].

Une démonstration analogue permet de prouver que dans cet exemple la taille des structures manipulées par la machine à appel par valeur est linéaire, bien qu'elle effectue des copies d'environnements. En effet, les copies sont éliminées juste après l'évaluation des paramètres (car ce sont ici des valeurs de types simples, donc indépendantes de l'environnement), et donc avant d'entrer dans le tour de boucle suivant. Ainsi l'appel par valeur est nettement plus performant que la machine de Krivine, non seulement à cause d'un nombre de $\beta$-réductions plus réduit en général (ce qui est logique, l'appel par nom répétant plusieurs fois un même calcul si une variable est appelée plusieurs fois) mais aussi à cause de sa meilleure gestion des copies d'environnements.

1. Je termine actuellement la mise au point d'une telle machine. Elle occupe bien moins de mémoire que la machine de Krivine, et elle m'a permis de donner les nombres de $\beta$-réductions de l'appel par nom quand la machine de Krivine explose. Le comparatif en temps de cette machine n'a pu être réalisé car je ne disposais que d'un PC et non d'une Sparc 20 comme pour les autres tests.

## Quatrième partie
# Conclusion

Les performances obtenues sont en général assez décevantes. Ceci s'explique en partie par l'implémentation de l'appel par nom qui est intrinsèquement nettement moins performant que les appels par valeur et nécessité. En matière de vitesse, la machine à appel par nécessité a des performances raisonnables, nettement meilleures que celles de toutes les autres machines implémentées ici, mais en l'absence de "garbage collection". Par rapport à Caml-Light, les rapports en temps d'exécution sont grands, mais heureusement ils varient peu: le quotient croît lentement dans la plupart des cas; cependant on a vu un exemple où la GOI était exponentiellement plus lente que Caml-Light. Il serait peut-être souhaitable de faire des expériences sur de plus grandes valeurs pour mieux vérifier ces résultats mais ceci pose problème étant donné la croissance trop rapide des temps d'exécution.

Contrairement à ce qu'on pourrait croire, la machine de Krivine est très lente dans beaucoup d'exemples. Bien qu'elle exécute beaucoup moins d'étapes, chaque étape est en moyenne beaucoup plus longue, car il y a de nombreuses copies d'environnements : la taille des structures de données peut avoir une croissance exponentielle.

Il serait intéressant de voir, pour la machine de Krivine et pour l'appel par valeur qui en dérive, l'efficacité d'une implémentation avec GC, qui éviterait les copies multiples. En même temps, elle conduirait à utiliser deux structures simplement chaînées, une pour l'environnement et une pour la pile, plutôt qu'un arbre binaire ce qui contribuerait sans doute aussi à améliorer les performances. Cependant, cette implémentation est un peu plus compliquée à mettre en œuvre (et un des intérêts de la géométrie de l'interaction est d'éviter le recours à un GC par une gestion économique de la mémoire).

La machine à pile de questions souffre aussi de ces problèmes de copies trop nombreuses. De ce fait, si elle est plus efficace dans le cas du $\lambda$-calcul pur comme on pouvait le prévoir, elle perd de son intérêt dès qu'on sort de ce cadre, intéressant du point de vue théorique mais peu utilisé dans les applications pratiques.

Les quelques optimisations effectuées sur la GOI apportent une amélioration mais hélas seulement d'un facteur constant (2 ou 3 en général). On pourrait encore ajouter quelques optimisations : comme on n'utilise pas de nœud `split` (qui sépare en une seule opération tête et queue d'une liste), il est inutile d'effectuer les opérations sur les listes lors du chemin de retour. On diviserait ainsi par deux le nombre d'opérations sur les listes. Cependant, ceci ne porterait pas remède au handicap principal de la GOI, qui est la nécessité de parcourir une partie parfois très importante du graphe du programme pour trouver la valeur d'une variable.

Le fait que la machine par nom la plus rapide soit la géométrie de l'interaction de base laisse présager de bonnes performances quand on aura une machine par valeur ou encore mieux la réduction optimale dans un modèle analogue. La réduction optimale telle qu'elle est implémentée pour l'instant, c'est-à-dire par réduction directe du graphe du programme, est handicapée par la complexité des structures de données manipulées, mais on peut espérer des progrès dans ce domaine.

# A Annexe : Options

| Compilateur | Toplevel | Description |
|---|---|---|
| `-o` $< Fichier >$ | `#out "`$< Fichier >$`";;` | Fixe le nom du fichier de sortie (par défaut a.c) |
| `-e` $< Fichier >$ | `#exe "`$< Fichier >$`";;` | Fixe le nom du fichier exécutable. Par défaut, pour le compilateur pas de sortie exécutable ; pour le toplevel, ./a.out |
| `-t` | `#typing;;/#notyping;;` | Active/désactive le typage (actif par défaut) |
| `-l` | `#latex;;/#nolatex;;` | Active/désactive la trace latex (inactif par défaut) |
| `-c` | `#count;;/#nocount;;` | Active/désactive le comptage des opérations (actif par défaut) |
| `-a`$N$ | `#auto` $N$`;;` | Fixe les copies automatiques de variables<br>$N = 0$ Toutes les contractions sont manuelles<br>$N = 1$ Le programmeur doit contracter lui-même les paramètres des fonctions, mais les déclarations `let` le sont automatiquement<br>$N = 2$ Le compilateur place seul les contractions<br>L'état par défaut est $N = 2$ |
| `-h` | `#help;;/#h;;` | Affiche l'aide |
| Choix de la machine | | |
| `-goi` | `#goi;;` | Géométrie de l'interaction (par défaut) |
| `-goinopt` | `#goi nopt;;` | Version non optimisée de la géométrie de l'interaction |
| `-quest` | `#quest;;` | Machine à pile de questions |
| `-irrevkriv` | `#irrev kriv;;` | Machine de Krivine |
| `-irrevneed` | `#irrev need;;` | Machine irréversible à appel par nécessité |
| `-irrevvalue` | `#irrev value;;` | Machine irréversible à appel par valeur |
| Commandes spécifiques du toplevel | | |
| `-top` | | Active le toplevel |
| | `#load "`$< Fichier >$`";;` | Charge un fichier ne contenant pas d'expression à évaluer. Les identificateurs définis dans ce fichier sont mémorisés par le toplevel pour un usage futur par des programmes |
| | `#run "`$< Fichier >$`";;` | Exécute un programme sur la machine actuelle |
| | `#runall "`$< Fichier >$`";;` | Exécute un programme sur toutes les machines |
| | `#compile "`$< Fichier >$`";;` | Compile un programme en C |
| | `#tex;;` | Compile et affiche une trace d'exécution |
| | `#quit;;/#exit;;` | Quitte le toplevel |

Les options `-l`, `-t`, `-c` du compilateur sont des bascules.

Dans le toplevel, toutes les commandes qui ne commencent pas par un `#` sont considérées comme des expressions du langage. Les déclarations `let` sont enregistrées pour tout usage futur, et les expressions sont évaluées dans la machine en cours.

# B Annexe : Grammaire du langage

Voici un résumé rapide de la grammaire du langage utilisé, selon la notation BNF. Je n'ai pas ici tenu compte des priorités pour simplifier.

```
< progelem > ::= < expr >;;
               | let < var > = < expr >;;
               | let rec < var > = < expr(var) >;;

< prog > ::= (< progelem >)*

< expr > ::= (< expr >)
           | < expr > < expr >
           | fn < pattern > => < expr >
               |< pattern > => < expr >
               | ...
           | let < var > = < expr > in < expr >
           | let rec < var > = < expr(var) > in < expr >
           | < var >
           | < const >
           | if < expr > then < expr > else < expr >
           | match < expr > with < pattern > => < expr >
                                |< pattern > => < expr >
                                | ...
           | < expr > < op > < expr >
           | [< expr >; ... ; < expr >]
           | [< var >=_] < expr >
           | [< var >=< < var >, < var > >] < expr >
           | < expr > : < type >

< const > ::= < constante entière >
           | cond | tt | ff | not | Y | succ
           | pred | iszero | isnil | hd | tl | []

< op > ::= + | - | * | / | %
         | < | > | <= | >= | =
         | <> |:: | and | or | xor

< pattern > ::= < var > | _
              | < constante entière >
              | < pattern > :: < pattern >
              | [< pattern >; ... ; < pattern >] | []
              | (< pattern >)

< type > ::= int | bool | '< ident >
           | < type > list
           | < type > => < type >
           | (< type >)
```

## Références


[1] ABRAMSKY (Samson). – Semantics of PCF, 4 novembre 1991.

[2] ABRAMSKY (Samson). – Computational interpretations of linear logic. *Journal of Theoretical Computer Science*, 1993, pp. 111:3–57. – Version précédente parue comme Imperial College Technical Report DOC 90/20, octobre 1990.

[3] ASPERTI (Andrea). – $\delta \circ ! \epsilon = 1$. *In : Proceedings of the 6th Rewriting Techniques and Applications Conference (RTA'95)*, éd. par Hsiang (Jieh). pp. 102 – 116. – Springer Verlag.

[4] DANOS (Vincent) et REGNIER (Laurent). – Local and asynchronous beta-reduction (an analysis of Girard's execution formula). *In : Proceedings, Eighth Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, pp. 296–306.

[5] DANOS (Vincent) et REGNIER (Laurent). – Reversible & Irreversible Computations, GOI and $\lambda$-machines, 1994.

[6] GIRARD (Jean-Yves). – Geometry of interaction 1: Interpretation of System F. *In : Logic Colloquium 88*, éd. par Ferro (R.), Bonotto (C.), Valentini (S.) et Zanardo (A.). – Amsterdam, 1989.

[7] MACKIE (Ian). – *The Geometry of Implementation.* – Thèse de PhD, Imperial College of Science & Technology, septembre 1994.

[8] MACKIE (Ian). – Lilac : a functional programming language based on linear logic. *Journal of Functional Programming*, octobre 1994, pp. 395–433.

[9] MACKIE (Ian). – The geometry of interaction machine. *In : Proceedings, 22nd ACM Symposium on Principles of Programming Languages*, pp. 198–208.

# Table des matières