Rapport de projet
Application web pour le lambda-calcul

Mathieu Chailloux
Vincent Botbol

24 octobre 2013

Chapitre 1

Introduction

Le but de ce projet est l'élaboration d'une application web d'évaluation de lambda-calcul fournissant une représentation graphique des lambda-termes manipulés.

Comme le titre du projet le laisse entendre, son développement s'est divisé en 2 parties : l'évaluateur de lambda-calcul et l'interface web permettant de visualiser les structures manipulées.

Chapitre 2

Interface graphique

L'interface graphique a été mise au point en OCaml vers du JavaScript utilisant les canvas introduits en HTML5. Pour réaliser cette transition, nous avons employé l'outil d'inter-opérabilité *js_of_ocaml*. Ce dernier contient un compilateur de code-octet vers JavaScript ainsi qu'une bibliothèque correspondant à l'API JavaScript. Ce "binding" nous apporte ainsi le meilleur des deux mondes : la portabilité et les possibilités d'usage de JavaScript et, du côté d'OCaml, la sécurité du programme par le typage et l'expressivité du langage.

2.1 Représentation de l'arbre

Pour la visualisation des lambda-termes, nous avons choisi une représentation arborescente. Il a donc été nécessaire de pouvoir dessiner ces arbres dans les canvas HTML. Par un soucis du détail, nous avons décidé d'implémenter un algorithme d'agencement de noeuds afin d'optimiser l'espace et donc améliorer la lisibilité des termes affichés.

L'étape du desin de l'arbre dans un canvas, une fois les calcul des positions des noeuds effectué, est quant à elle, triviale. Il nous suffit de parcourir l'arbre et de dessiner les noeuds aux coordonnées associées et de relier visuellement les différents noeuds. Une fonction récursive s'adapte parfaitement à ce type de manipulation.

2.2 Interface utilisateur

L'interface de l'application se compose :

- d'un champ texte, attendant un lambda-terme écrit avec la syntaxe spécifiée dans l'annexe de la page html,
- d'une boite de séléction pour choisir la stratégie de réduction à employer (*Appel par nom* ou *Appel par valeur*),
- d'un bouton lancant l'évaluation lors du click,
- et d'un canvas qui va nous servir à afficher l'arbre lors de l'évaluation.

Pour parcourir les différentes réductions effectuées, nous avons inclu un paneau latéral contenant des boutons "précédent" et "suivant" ainsi que d'une liste contenant les différents termes à chaque étape de réduction. Les termes de cette liste (ainsi que les boutons) sont cliquables et afficheront l'arbre associé. Enfin, ce panneau est dynamique et ne sera affiché qu'après l'évaluation d'un terme.

Comme évoqué plus haut, nous avons également fourni une annexe contenant la syntaxe à employer, quelques combinateurs prédéfinis et des exemples évaluables.

2.3 Implémentation

Il a été nécessaire de lier le modèle de l'application à l'interface graphique. L'affichage de l'arbre, comme décrit plus haut, se fait assez naturellement. Pour rendre l'interface réactive, il a également fallu lier les composants graphique à des fonctions associées (handlers). Par exemple, le handler du bouton d'évaluation sur un événément de clic va effectuer les actions suivantes :

1. Parser le terme,
2. Générer la liste des réductions,
3. Récupérer le dernier terme [1],
4. Opérer l'algorithme d'agencement des noeuds sur l'arbre de syntaxe abstraire de ce lambda-terme,
5. Afficher dans le canvas ce dernier,
6. Générer le paneau latéral : boutons "next", "prec" et la liste des réductions de ce terme. Il faut également, à ce moment-ci, sensibiliser ces derniers.

Les actions des "handlers" étant triviales, nous ne les détaillerons pas ici.

Une fois, la sensibilisation des composants achevées l'interface est prête à être utilisée.

1. On a préféré afficher en priorité la forme normale (si existante) plutôt que le terme passé en argument. Cela nous semblait plus judicieux.

Chapitre 3

Lambda-calcul

La deuxième partie de ce projet a été d'évaluer les lambda-termes rentrés par l'utilisateur. Il nous a donc fallu modéliser le lambda-calcul, reconnaître les termes rentrés de manière textuelle, et enfin les évaluer.

3.1 Grammaire

Pour analyser le texte décrivant le lambda-terme, nous dû tout d'abord établir une grammaire.

<**expr**> : := <*name*> | <*abstr*> | <*expr*> <*expr*> | **(** <*expr*> **)**
<**abstr**> : := **l** <*string*> **.** <*expr*>
<**string**> : := <*name*>+
<**name**> : := <*name*> '* | caractère classique sauf un caractère réservé.

Ainsi, il n'y a que 5 caractères réservés "**(**", "**)**", "**l**", "**.**", "**'**" : . Tous les autres caractères sont considérés comme des noms (variables ou constantes). Il est possible de parenthéser un terme, de construire une abstraction en lui liant plusieurs variables d'un coup (on peut écrire *lxyz.y(xz)*), ou encore d'appliquer un terme à un autre.

3.2 Modélisation

Pour modéliser notre lambda-calcul, nous nous sommes basés sur les *indices de Bruijn* pour représenter les variables liées. L'indice d'une telle variable correspond donc au nombre d'abstractions entre une occurence et l'abstraction qui déclare cette variable, en remontant l'arbre. Ainsi, dans *lx.ly.xy*, l'occurence de y est d'indice 0 car liée à la dernière abstraction, tandis que celle de x est d'indice 1. Pour manipuler les lambda-termes, nous avons utilisés un type somme *OCaml* qui correspond bien à leur structure arborescente :

```
type term =
| Const of string          (* Constants and free variables *)
| App of term * term       (* Applications *)
| Abstr of string * term   (* Abstractions *)
| Var of int               (* Bound variables *)
```

3.3 Parser

Une fois la grammaire et le type *OCaml* établis, il nous a fallu être capable de convertir un mot de cette grammaire en un lambda-terme en accord avec le type décrit juste au-dessus. Cette étape de *parsing* repose

sur la manipulation des chaînes de caractères en *OCaml* et surtout sur l'idée de considérer une chaîne comme une liste de caractères, afin de pouvoir facilement itérer dessus. C'est le rôle de la fonction *explode*. Une fois les espaces supprimés (fonction *lex*), on peut alors passer au parsing proprement dit (fonction *parse*). Ainsi, on va itérer sur les caractères en respectant la logique suivante :

- Si le caractère est un l, on construit une abstraction qui comprend une chaîne décrivant une liste de symboles et un corps qui est un autre terme. Ces 2 éléments sont séparés par le caractère réservé ".", ainsi ($lxyz.T$) décrit l'abstraction de paramètres **x**, **y**, et **z** et de corps **T**. De plus, il faut mettre à jour les indices de *Bruijn* dans l'évaluation du terme T. Toujours dans cet exemple, on évalue T avec z d'indice 0, y d'indice 1, x d'indice 2, et les variables liées à des abstractions plus proches de la racine dans la branche augmentent leur indice de 3 (puisqu'il y a 3 nouvelles variables liées).
- Si le caractère est une parenthèse ouvrante, on évalue la suite comme un terme jusqu'à trouver une parenthèse fermante.
- S'il reste au moins 2 termes juxtaposés, on applique le premier au second, et on réévalue le nouveau terme ainsi obtenu.
- Si le caractère représente un nom, 2 cas sont possibles : le nom correspond à une variable liée à une abstraction dont on est sous la portée, sinon on le considère comme une constante. Pour faire cette différence, on garde un environnement contenant les variables actuellement liées lors du parsing d'un terme (ajout lors d'une abstraction, retrait lors d'une application).

3.4 Alpha-conversion

Nous allons maintenant nous intéresser à la mise en place du lambda-calcul et, dans un premier temprs, à l'alpha-conversion. C'est l'opération qui permet de renommer des variables liées pour obtenir un terme alpha-équivalent. Par exemple *lx.x* peut s'alpha-convertir en *ly.y*, ou encore *lx.lx.x* en *ly.lx.x*. Les nouveaux termes ainsi obtenus sont considérés comme équivalents.

La règle retenue a été de renommer une variable liée si elle était sous la portée d'une abstraction liant une variable de même nom. Ainsi *lx.lx.x* se convertira en *lx.lx'.x'*. La règle de renommage consiste à concaténer le caractère ' jusqu'à obtenir un nom inédit.

Pour mettre en place ces règles, l'idée est de parcourir le terme de la racine vers les feuilles en conservant un environnement contenant les noms de variables déjà liées dans la branche courante, ainsi il n'y a pas de doublon. Enfin, et surtout, le changement de nom d'une variable liée n'affecte en rien sa représentation (cad son indice de *Bruijn*) car elle reste à la même place, il faut juste changer le nom du paramètre lors de la création de l'abstraction à laquelle elle est liée. Le code effectuant cette opération correspond à la fonction *alpha*

3.5 Bêta-réduction

A présent, il ne nous reste plus qu'à évaluer les terme du lambda-calcul. Pour cela on va utiliser la bêta-réduction afin d'obtenir des termes en forme normale (plus de réduction possible + sous-terme normaux). Cette règle fondamentale du lambda-calcul consiste en l'application d'une abstraction à un argument, pour retourner donc le corps de l'abstraction dans lequel le paramètre formel est remplacé par l'argument. plus simplement, cette règle s'écrit ainsi :

(App (Abstr (x,body)) arg) -> body[arg/x]

La difficulté réside donc dans la substitution du paramètre formel par l'argument. Pour cela, il faut parcourir le corps du terme ainsi modifié pour identifier quelles sont les variables liées correspondant au paramètre que l'on veut substituer. Pour cela, on va encore utiliser les indices de *Bruijn*. En effet, lorsqu'on

commence à parcourir le corps de l'abstraction, on cherche les variables liées d'indice 0 pour les remplacer par l'argument. De plus, il faut encore maintenir les indices à jour. Ainsi, lorsqu'on rentre dans une nouvelle abstraction, on augmente l'indice recherché (initialement 0) de 1, mais aussi on acutalise les indices liés à des abstractions au-dessus de celle qu'on transforme. Par exemple dans *lx.(lyz.yz)x*, l'occurence de x est d'indice 0, alors qu'àpres bêta-réduction, dans *lx.lz.xz*, x est d'indice 1.

Enfin, il nous faut établir une stratégie d'application. Nous allons en mettre en place 2 : l'appel par valeur (*call by name*) et l'appel par nom (*call by value*). La différence réside dans l'ordre de réduction des termes lors d'une application : doit-on d'abord évaluer la fonction et l'argument, ou d'abord le substituer au paramètre ? La première proposition correspond à l'appel par valeur, et la deuxième à l'appel par nom. En terme d'implémentation, il s'agit juste de changer l'ordre entre les appels récursifs sur les sous-termes et l'application explicite de la réduction. Si les étapes de réduction peuvent différer entre les 2 stratégies, elles aboutissent au même résultat, si c'est un terme réductible (pas de boucle infinie lors de la réduction). Nous avons de plus regroupé toutes les étapes dans une liste pour pouvoir naviguer d'une à l'autre, mais aussi afin de détecter les termes irréductibles : si, lors de la réduction, on obtient un terme égal au terme d'une étape précédente, c'est qu'il est irréductible, et on s'arrête donc.