Interprète Lisp en C++

Antonin Garret Rémy Sun

21 avril 2016

1 Introduction

Le Lisp est un langage fonctionnel interprété permettant des implémentations différentes. 2 tels implémentations en Ocaml ont été vues en TD de programmation. Ocaml étant un langage fonctionnel, cette implémentation est relativement naturel.

Au travers de ce projet, nous nous intéressons à l'implémentation d'un interprète Lisp en C++, et nous attachons plus particulièrement à l'implémentation d'une gestion d'environnements et de clôtures proches de celles très souples de Scheme.

2 Objectif

L'objectif que nous nous sommes fixés pour ce projet était de construire un interprète Lisp gérant les liaisons à la manière de Scheme. Plus particulièrement, nous étions intéressés par la gestion de liste chaînées de frames, et par la façon dont on pouvait représenter ces frames pour permettre la création d'un tel interpréteur.

Au delà des problèmes de définitions circulaires posés, le grand défi a été de gérer correctement la création des copies (implicites dès qu'on passe en argument de fonction) et la bonne détection des clôtures.

Nous nous somme fixés les objectifs intermédiaires suivant :

— Compléter le squelette fourni pour obtenir un interprète dynamique, ce qui constitue une première étape facilement vérifiable.
— Passer de la gestion d'environnement proposée à une gestion par listes chaînées, ce qui permettrait eu passage de commencer à réfléchir à la façon de gérer les frames.
— Créer un premier interprète à liaison statique identique à celui vu en TD.
— Implémenter une gestion des clôtures dans cet interprète. Le but original était d'explorer la possibilité de stocker tout l'environnement courant (donc en le dédoublant), possibilité non développée en TD. Après constatation des nombreux problèmes que cela pose (mais aussi de ce qu'il est possible de faire) nous avons créé une autre clôture utilisant des pointeurs d'environnement.
— Implémenter la gestion de frames, adapter la clôture Scheme.

3 Travail réalisé

3.1 Interprète dynamique

Dans un premier temps, nous avons implémenté, à partir du code fourni, un interprète qui possède les principale fonctions Lisp, avec une gestion dynamique de l'environnement.

3.1.1 Toplevel et environnement

Nous avons encapsulé la gestion des entrées dans un `toplevel`, appelé par la fonction `main`. Ce toplevel gère la directive setq, qui permet de modifier l'environnement. Elle fait appel à une fonction `add_new_binding` que nous avons implémenté, qui permet de créer un nouvel objet `Binding` (qui comporte deux éléments : un symbole et la valeur qui lui est associée) que l'on ajoute à l'environnement courant. Dans le code original, les environnement sont traités comme des vecteurs de pointeurs vers des `Binding`. Cette version dynamique de l'interprète ajoute un `Binding` à chaque fois que `setq` est appelée, sans vérifier si un `Binding` correspondant au symbole utilisé existe déjà. Comme la fonction de recherche dans l'environnement renvoie le premier `Binding` trouvé, cette nouvelle liaison masque de façon effective celles déjà enregistrées.

D'autre part nous avons aussi ré-encapsulé la gestion des entrées de l'utilisateur dans le `toplevel` dans une classe `read` qui gère les appels au parseur.

3.1.2 Subroutines

Dans le code fourni, quelques subroutines étaient traitées dans la fonction `eval` de l'interprète. Nous avons encapsulé les subroutines pour qu'elles soient traitées de manière séparée, afin de pouvoir facilement en ajouter de nouvelles. Les opérations de bases sur les nombres, ainsi que les fonctions `car`, `cdr`, `cons`, `concat`, `newline` et `read` ont été implémentées.

Pour la subroutine =, nous nous sommes inspirés de l'interprète Caml vu en TD, en comparant le type de chaque objet, avant de comparer leur valeur si le type correspond. Nous avons aussi implémenté la gestion des booléens dans la classe `Object`, afin de clarifier et de simplifier leur utilisation, en particulier dans cette fonction.

Il a par contre été plus difficile de passer des fontion en objet. En Ocaml, « functions are first-class citizens « . Pas en C++. De fait, il a été nécéssaire de prendre des moyens détournés. Pour nous rapprocher de l'interprète original, nous avons modifié la classe `Cell` pour créer un type de cellule contenant une subroutine (qui est une fonction). Pour ce faire, nous avons fais usage des pointeurs de fonctions (et non de foncteurs dont l'utilisation ne se justifiait pas vraiment ici). Une fonction `init_subr` permet d'ajouter à l'environement de départ les subroutines définies. Il convient de noter qu'on a du modifier les méthodes d'impression de liaisons pour ne pas imprimer les liaisons faisant intervenir une subroutine.

Nous avons d'autre part pu remarquer l'utilité de l'encapsulation déjà effectuée, car il nous a suffit de réutiliser les fonctions de la classe `read` pour implémenter la subroutine `read`.

3.1.3 Eval

La principale modification apportée à la classe `eval` fournie est l'utilisation de l'encapsulation précédemment mentionnée pour mieux compartimenter les différentes parties de l'éva-

luation. Nous avons utilisé la programmation par exception afin de s'assurer que chacune des ces parties se déroulent correctement. Par exemple, lors de l'évaluation d'un symbole, on fait appel à une fonction qui parcours l'environnement courant afin de trouver une liaison correspondante, et renvoie une exception en cas d'échec. Cela nous permet de repérer lorsque l'utilisateur cherche à évaluer un symbole qui n'est pas attribué.

Nous avons aussi aussi implémenté un dispositif de trace via la commande `debug` et d'impression de l'environnement via `printenv`, pour permettre à l'utilisateur de suivre l'évolution de l'environnement courant.

3.2 Environnement par listes chaînées

Si l'implèmentation des environnement comme vecteurs de pointeurs vers des `Binding` a le mérite de marcher, elle n'en est pas moins problématique au sens où, pour copier un environnement, il est nécessaire de copier l'intégralité de la liste des pointeurs, ce qui présente un coût linéaire en la taille de l'environnement considéré !

La première idée pour remédier à cela est de faire de l'environnement un unique pointeur vers le vecteur de pointeurs. Cela ne fait cependant que déplacer le problème dans le cas où il faut copier aussi ce vecteur.

Aussi, il semble judicieux de transformer l'environnement en liste chaînée : chaque bloc de la liste contient un unique `Binding` et un pointeur vers le prochain bloc (éventuellement `NULL`). Il faut noter que nous ne perdons rien en efficacité, puisqu'il fallait de toute façon parcourir tout le vecteur de pointeurs pour trouver le `Binding` correspondant à une chaîne donnée.

3.2.1 Blocs d'environnement

Nous avons ainsi défini une structure `EnvBlock` qui constitue un maillon de la liste chaînée. Puisque ces maillons sont eux mêmes repérés par leur pointeur, et pour une raison que nous verrons plus tard, il n'est pas nécessaire d'utiliser des pointeurs de `Binding`.

3.2.2 Environnement = Bloc ?

La principale question qui s'est posée est la suivante : considère-t-on qu'un environnement est un bloc (le bloc de « tête ») ou considère-t-on que l'environnement est quelque chose d'autre que la somme individuelle de ses parties ?

Nous avons privilégié la deuxième approche. En effet, elle permet de gérer plus proprement l'allocation dynamique en mémoire de blocs d'environnement. De plus, cela rendait beaucoup plus facile la destruction des blocs d'environnements alloués en mémoire ainsi que la création de copies indépendantes (créant aussi de nouveaux blocs). Cette gestion des copies s'est révélée par la suite sans grand intérêt, mais le code utilisé demeure dans le dossier `Copie`

C'est cette macro-structure qui s'occupe de la gestion des ajouts et consultations de `Binding` :c'est elle qui fournit l'interface publique de l'environnement.

Une difficulté rencontrée avec l'utilisation d'instances de `Binding` au lieu de pointeurs dans les blocs d'environnements a été la création implicites de copies que nous avons du tracer à l'aide d'impressions de pointeurs. L'utilisation d'un setter à permis de remédier à ces problèmes.

3.3 Première liaison statique

Le passage en liaison statique n'a pas posé de grandes questions ou difficulté.

Nous nous sommes contentés de créer une nouvelle méthode `set_new_binding` qui parcours dans un premier temps l'environnement à la recherche d'une liaison faisant intervenir le nom de liaison qu'on veut ajouter. Si on trouve une telle liaison, on modifie physiquement la liaison ajoutée. Sinon on rajoute une nouvelle liaison.

3.4 Première clôture

3.4.1 Stockage physique de la capture

Nous étions dans un premier temps parti dans l'idée d'essayer de voir ce que le stockage complet de l'environnement courant au moment de la définition d'une clôture. La création d'une telle clôture, bien que fastidieuse ne pose pas de difficultés particulières : créer une clôture contenant assez d'informations pour reconstruire l'environnement est assez naturel, tout comme l'extraction d'informations à partir d'une clôture que nous avons nous-même construits. On vérifie bien qu'il n'est plus possible de capturer dynamiquement une variable à l'intérieur d'une fonction.

Là où la situation devient plus épineuse est quand on veut pouvoir modifier physiquement une variable déjà définie. Si il peut paraître en première approche qu'il « suffirait » de parcourir toutes les clôtures pour modifier les informations à l'intérieur, cette idée n'est pas concluante. En effet, si on veut redéfinir une fonction dans la clôture lui correspondant, il faudrait effectuer une infinité de modification de clôture puisqu'une telle clôture rendrait l'environnement circulaire. Nous n'avons pas trouvé de solution satisfaisante à ce problème.

En l'état, une telle construction ne permet pas l'utilisation de fonctions mutuellement dépendantes (on ne peut pas aller modifier le placeholder) ou récursives.

3.4.2 Stockage d'un pointeur de capture

Nous avons donc du reprogrammer la clôture en stockant un pointeur vers le bloc environnement courant au moment de la clôture. Il suffira alors de créer un environnement pointant vers le bloc environnement au moment de la reconstruction. Une légère difficulté est que simplement ajouter une valeur pointant vers un environnement peut créer une situation où il y a des inclusions circulaires de fichiers puisque la classe environnement dépend de la classe cellule. Heureusement, une forward declaration de la classe environnement dans la classe `Cell` permet de remédier à cela.

Avec cette méthode, il est possible de créer des fonctions mutuellement récursives et des fonctions récursives sans problèmes apparents. Par contre il faut encore gérer un problème posé par la définition circulaire des clôtures. En effet, cela pose de gros problèmes si on veut afficher l'environnement. La solution adoptée a été le remplacement d'objets environnement par une chaîne <env>.

3.5 Passage en frames

Notre structure en liste chaînée a déjà grandement abordée les difficultés de la gestion d'un environnement par frames. Deux possibilités d'adaptation étaient envisageables : on

pouvait laisser la structure grandement inchangée et simplement remplacer le contenu des blocs d'environnements par un vecteur de pointeurs vers des `Binding`, ou traiter les anciens `Environment` comme des frames et créer un nouvel environnement pointant vers une frame. C'est la deuxième possibilité que nous avons privilégié.

Procéder de la sorte nous semble présenter certains avantages : de cette manière il est complètement naturel de créer des frames intermédiaires, et il y a séparation claire des structures de stockage de liaisons et de frames.

Un aspect particulièrement intéressant de la gestion par frame est la limitation de l'action de certaines méthodes (comme `set_new_binding`) à la frame locale tandis que d'autres s'appliquent à l'ensemble des frames en cascade défini par l'enchaînement des frames.

3.6 Clôtures scheme

La gestion des clôture en scheme est comparativement plus simple que dans l'interprète à liaison statique précédente. Il suffit de stocker le pointeur vers la frame courante au moment de la clôture et il ne sera plus possible de capturer une variable au cours de l'exécution de fonctions imbriquées.

Lors du passage par un `lambda`, une nouvelle frame est créée (ce qui protège les variables de l'environnement courant), et à la lecture d'une clôture on se replace dans la frame où la fonction a été définie ce qui évite d'être gène par une définition interne à la frame où la fonction est appelée.

Grâce à la particularité des frames, il est parfaitement naturel de définir des fonctions mutuellement récursives. En effet, puisque la clôture pointe vers une frame et non un bloc d'environnement fixé, il n'y a pas besoin d'utiliser un placeholder pour réserver une place dans la clôture : les fonctions définies dans la frame courante après la définition e la clôture seront quand même capturées.

4 Vérification

4.1 Interprète dynamique

Les fonctions de l'interprète dynamique fonctionnent correctement et permettent une utilisation classique de Lisp. Le seul problème est la gestion des erreurs de l'utilisateur via les exceptions. La programmation défensive utilisée dans le code original permet de s'assurer que tout fonctionne comme prévu, mais en cas d'erreur, la fonction `assert` qui a été utilisée stoppe le processus de l'interprète et empêche de rattraper les exception comme nous voulions le faire. Il serait possible de surmonter ce problème en modifiant la manière dont ces tests sont effectués afin de générer des exceptions spécifiques, mais nous avons choisi de consacrer notre temps aux autres aspects de l'interprète.

4.2 Clotûre Scheme

5 Contribution

— http ://stackoverflow.com
— https ://openclassrooms.com

— http ://www.cs.rpi.edu