Interprète Lisp en C++

Antonin Garret Rémy Sun

21 avril 2016

1 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'environement 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.

2 Travail réalisé

2.1 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

2.1.1 Toplevel

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.

2.2 Environement par listes chaînées

Les environements dans le code original étaient de simples vecteurs de pointeurs vers des `Binding`. Si cet solution 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'environement considéré !

La première idée pour remédier à cela est de faire de l'environement 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'environement 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.

2.2.1 Blocs d'environement

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`.

2.2.2 Environemnent = 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'environement 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'environement.

Une difficulté rencontrée avec l'utilisation d'instances de `Binding` au lieu de pointeurs dans les blocs d'environnement 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.

2.3 Première liaison statique

2.4 Première cloture

2.5 Passage en frames

2.6 Clotures scheme

3 Vérification

4 Contribution

— http ://stackoverflow.com
— https ://openclassrooms.com
— http ://www.cs.rpi.edu