Université Laval
Faculté des sciences et de génie
Département d'informatique et de génie logiciel
GLO-66811

Danny Dubé
Automne 2008
Version : 6 octobre

Travail pratique #2
Implantation des fonctions

Ce travail pratique consiste à vous faire appliquer les deux techniques de défonctionnalisation, soit les transformations source à source qui emploient la représentation chaînée et la représentation plate des fermetures.

Transformations source à source

Vous devez effectuer chacune des transformations source à source sur un même programme écrit en Scheme. Les transformations devraient être effectuées manuellement, i.e. à l'aide de votre éditeur de texte préféré. Le programme obtenu suite à l'application de chacune des transformations devrait être exécutable dans un interprète Scheme. Au risque de me répéter : vous devez obtenir 2 programmes transformés différents.

Le programme original est un petit programme qui calcule un *checksum* sur une chaîne. La tâche que doit effectuer ce programme n'est pas vraiment importante pour nous mais ça peut aider de comprendre ce que le programme fait. Le calcul du *checksum* se fait en additionnant la valeur numérique de tous les caractères d'une chaîne. Le *checksum* est limité à 10 bits. Le programme est lancé en effectuant l'appel '`(run)`'.

Notez que, une fois le programme transformé suivant l'une ou l'autre des transformations, nous devrions pouvoir le faire exécuter en effectuant l'appel '`(clos-apply0 user-run)`'. Ceci implique que vous devez inclure au niveau système toutes les fonctions utilitaires nécessaires. Aussi, veuillez adopter la convention voulant que les noms introduits au niveau système gardent leur nom original et que les noms provenant du programme source original prennent le préfixe '`user-`'.

Vous devez introduire dans le programme transformé une version adaptée de chaque fonction de librairie que le programme utilise. Par exemple, le programme original utilise la fonction '`string-ref`'. Donc, vous devez introduire une version *usager* appelée '`user-string-ref`', qui est une fermeture représentée à l'aide d'une *boîte* et qui a le bon comportement lorsqu'activée en utilisant '`clos-apply2`'.

Afin de représenter les fermetures dans le programme transformé, je vous suggère d'utiliser les vecteurs de Scheme. Cette convention est particulièrement pratique car le programme original n'utilise pas de vecteurs du tout. Ainsi, après la transformation produisant les fermetures chaînées, une fermeture pourrait être représentée ainsi : '`#(<N> <code> <env>)`' où `N` est l'arité de la fonction. Après la transformation produisant les fermetures plates, une fermeture pourrait être représentée ainsi : '`#(<N> <code> <var1> ...)`' où le '`M`' habituel est encodé implicitement à l'aide de la taille du vecteur. Hormis les fermetures, les objets de tous les types ne changent pas de représentation. De plus, comme aucune fermeture n'est affichée ou retournée par le programme, alors le changement de représentation des fermetures

n'est pas censé être perceptible au niveau du comportement *observable* du programme, lors de son exécution (bien sûr, l'appel initial se fait différemment).

N'oubliez pas le fait que plusieurs des formes syntaxiques de Scheme sont du sucre syntaxique et que nombre d'entre elles cachent une ou plusieurs `lambda`-expressions (voir l'annexe). Il y a donc de nombreux points du programme qui introduisent des variables locales.

Le programme comporte des variables mutables. Ainsi, il sera nécessaire d'employer la transformation source à source qui élimine les variables mutables dans le cas des fermetures plates. Notez que vous n'avez pas à vous soucier de la question des fonctions à arité variable, lesquelles sont permises en Scheme. Toutes celles qui, en principe, sont capables d'accepter un nombre variable d'arguments sont systématiquement utilisées avec un nombre constant d'arguments (par exemple, '`+`'). Vous pouvez donc les considérer comme des fonctions à arité fixe, selon l'usage que le programme en fait.

Remise des travaux

Vous devez remettre votre programme **via l'intranet**. Vous devez aussi remettre un bref rapport qui présente les étapes que vous avez effectuées pour faire les transformations. En cas de remise de programmes défectueux, un rapport contribue à me faire comprendre que vous aviez, en fait, mieux compris que votre code ne le laisse croire.

Vous devez remettre le devoir au plus tard le **20 octobre, 18h00**. Le travail doit être fait individuellement. Des discussions verbales sont acceptables entre camarades mais pas des échanges de bouts de réponses. De plus, il va de soi que vous ne devez pas récupérer sur internet la solution partielle ou complète à ce travail pratique.

Annexe : Sucre syntaxique

Certaines formes syntaxiques cachent des `lambda`-expressions ; notamment, les '`let`' et les '`let`' nommés. Voici comment on peut récrire ces formes et exposer les `lambda`-expressions. Notez que les réécritures ne sont pas présentées dans toute leur généralité. Cependant, dans le cadre de ce travail, elles conviendront.

```
(let ((x e1))
  e2)
↦
((lambda (x)
   e2)
 e1)
```

```
(let f ((x e1))
  e2)
↦
(let ((f #f))
  (set! f (lambda (x) e2))
  (f e1))
```