2ème Examen Réparti
Analyse des Programmes et Sémantique

(APS - MI030 - Master 1)
Université Pierre et Marie Curie Paris 6
Cours de J.Malenfant enseigné par B.Starynkevitch
lundi 22 avril 2013

Attention : *Tous les documents sur papier sont autorisés* ; mais *tous les* ***appareils électroniques*** (téléphones, tablettes, liseuses, ordinateurs, baladeurs, écouteurs, etc...) *sont* ***interdits***.

Durée : deux heures ; (sujet de 6 pages sur 3 feuilles recto-verso)

Les questions (§1 au total notées sur 10) sont toutes indépendantes du problème §2 (noté sur 20).

N.B. Les couleurs dans cet énoncé sont là pour vous faciliter la lecture. Ne les utilisez pas sur votre copie. Si vous utilisez deux couleurs distinctes, indiquez vos conventions. N'utilisez pas le rouge. Soyez **concis et rigoureux**.

On fournit en annexe le rappel de fragments de la sémantique dénotationnelle de Mini-Scheme.

1 Questions

Les questions sont indépendantes.

1.1 Exercice de λ-calcul

Réduire en forme normale chacun de ces λ-termes suivants :

$$(\lambda x.xxx)\lambda x.y \quad \text{et} \quad (\lambda fx.(f(fx)))x$$

1.2 Sémantique dénotationnelle de l'affectation `set!`

L'affection a pour syntaxe :

expr : := ...
| **`( set!`** ***`symbol`*** *expr* **`)`** – *affectation*

Sans modifier l'état sémantique de notre Mini-Scheme (voir annexe), donner l'équation sémantique dénotationnelle correspondante à l'affection, sachant que les deux expressions
`(let ( (x 0) ) (set! x 1) x)`
et `(let ( (x 0) ) (let ( (f (lambda (n) (set! x n))) ) (f 1) x))`
s'évaluent toutes les deux en le nombre 1.

Il est explicitement demandé de suivre scrupuleusement les conventions et notations utilisées en annexe.

1.3 Exercice de programmation en Scheme

Donner le code source en Scheme (standard *R*^{5}*RS*) d'une fonction `alternapp` qui applique alternativement deux fonctions (données en arguments formels `f` et `g`) aux éléments d'une liste (donnée aussi en argument formel `l`). Votre code commence donc par la ligne : `(define (alternapp f g l)`
et une fois `alternapp` définie,
`(alternapp (lambda (n) (+ n 1)) (lambda (p) (* p 2)) (list 1 3 4 10))` s'évalue en la liste `(2 6 5 20)` tandis que
`(alternapp car cdr (list '(a b c) '(d e) '(1 2) '(3 4 5)))` s'évalue en la liste `(a (e) 1 (4 5))`

Votre fonction `alternapp` doit s'exécuter à hauteur de pile d'appel bornée (par exemple en ayant dans un `letrec` une fonction interne qui soit récursive terminale). Elle peut appeler la fonction `reverse` d'inversion de liste qui pourrait être définie par

```
(define (reverse li)
   (letrec (
      (revloop
        (lambda (ll acc)
             (if (null? ll)
                 acc
                 (revloop (cdr ll) (cons (car ll) acc)))))
     )
   (revloop li '())
  ))
```

Notez que l'appel interne à `revloop` est récursif-terminal donc `reverse` utilise la pile d'appel sur une profondeur bornée.

2 Problème : une imprimante tri-dimensionnelle

Une imprimante 3D de marque UPMIC produit un objet en ajoutant couche par couche de la matière ("additive layer manufacturing"), du plastique foudu projeté par une buse ("nozzle"). L'objet ainsi produit dépend du programme envoyé vers cette imprimante.

Le plateau de cette imprimante descend progressivement par rapport à sa buse. On note z la coordonnée (relative à sa position de départ) du plateau ($z \in [0; 2000]$) exprimée en dixièmes de millimètres. La buse se déplace dans un carré ; les coordonnées de la buse sont la longueur l et la largeur w ("width") avec $l \in [-2000, 2000]$ et $w \in [-1500, 1500]$ - en dixièmes de millimètre aussi. La machine travaille couche par couche (une couche est déterminée par sa hauteur z constante), et les coordonnées z, w, l sont entières. Il y a donc trois moteurs pas-à-pas(de vitesse constante, et lente) dans la machine . Pour ne pas trop vibrer, un seul moteur au plus peut être allumé à la fois.

Cette imprimante est pilotée par un interprète de Mini-Scheme convenablement étendu et modifié. Voici les primitives spécifiques :

- **`(start-3d-printer)`** pour démarrer mécaniquement l'imprimante ; le plateau commence alors à descendre.
- **`(on-z-level`** z f **`)`** enregistre la fonction f qui sera appelée avec comme argument le nombre z quand le plateau atteint.
- **`(start-spit)`** pour que la buse crache son jet de plastique fondu.
- **`(stop-spit)`** pour qu'elle cesse de projeter du plastique.
- **`(move-w`** dw**`)`** pour déplacer en largeur (à l constant) la buse sur un deplacement de dw
- **`(move-l`** dl**`)`** pour déplacer en longueur (à w constant) la buse sur un deplacement de dl
- **`(park-nozzle)`** pour ranger la buse à sa position d'origine ($l = 0, w = 0$)
- **`(down-plate)`** pour mouvoir le plateau vers le bas (donc avec z croissant) d'un cran (de $0,1$mm). Si un niveau intéressant est atteint, on exécute la fonction qui a été préalablement enregistrée avec `on-z-level`.
- **`(stop-3d-printer)`** pour arrêter mécaniquement l'imprimante, qui s'ouvre alors pour permettre le retrait de la pièce.

2.1 exemple

Coder le programme fabriquant un pavé plastique plein de $32 \times 32 \times 20$ millimètres [1], donc remplissant les points $(w, l, z) \in [-160; 160] \times [-160; 160] \times [0; 200]$

1. C'est la taille d'une brique Duplo élémentaire

2.2 État et domaines sémantiques

Donnez formellement l'état sémantique de l'interprète MINI-SCHEME-UPMIC de cette imprimante. Cet état tient compte de l'état physique de l'imprimante !

Donnez les domaines sémantiques.

Vous respecterez, compléterez et modifierez les notations données en annexe.

2.3 Équations sémantiques

Décrivez formellement et rigoureusement, par des équations sémantiques, le comportement des primitives de l'imprimante 3D.

Expliquez brièvement en quelques phrases s'il faut modifier la sémantique des constructions `lambda` et `let` de Mini-Scheme.

2.4 Le prototype du futur ; la 3D en couleurs

Les ingénieurs d' UPMIC travaillent avec acharnement sur le prochain modèle d'imprimante 3D. Celui-ci dispose de trois buses fournissant des plastiques de couleur différente, et l'imprimante peut changer de buse active. De plus, les problèmes de vibration on été surmontés : les deux moteurs de deplacement de la buse peuvent tourner simultanément, à la même vitesse (donc la buse active se déplace aussi en diagonale dans le plan horizontal).

Inventez les primitives nouvelles de Mini-Scheme-Upmic pour les nouvelles possibilités de l'imprimante.

Formalisez (en vous basant sur la sémantique précédente) la nouvelle sémantique.

Les anciens clients de UPMIC veulent réutiliser leurs vieux codes. Formalisez la transformation de programme source qui permettra de transformer les vieux codes en du code profitant du mouvement en diagonale de cette nouvelle imprimante.

Pourquoi le nouveau modèle d'imprimante fabriquera plus rapidement des briques Duplo^{TM}(dont la base est le pavé de votre exemple 2.1) ?

FIN DES QUESTIONS DE L'EXAMEN RÉPARTI 2 DU 22 AVRIL 2013 - APS MI030

L'annexe rappelant en partie la sémantique de Mini-Scheme suit sur 3 pages suivantes.

Annexe : fragments de sémantique dénotationnelle de Mini-Scheme

Voici en rappel des fragments d'une sémantique dénotationnelle de Mini-Scheme, semblable à ce qui a été vu en TD et TME. Il s'agit d'un rappel pour fixer les notations et la terminologie. J'ai tâché de choisir une notation facile à écrire au stylo.

Nous utilisons des "mots-clefs" français **si**, **alors**, **sinon**, **soit**, **dans**, **avec**, pour la méta-langue "mathématique" exprimant la sémantique. On y note **fixe** l'opérateur de point fixe et **erreur** les cas d'erreur. **dom**(f) désigne le domaine de la fonction f c.à.d. l' "ensemble" des x tels que $f(x)$ soit défini.

État sémantique

Domaines sémantiques

On note **Val** le domaine des valeurs dénotables, donc :

les adresses	**Ad**	=	un ensemble dénombrable d'adresses, par exemple les naturels **N**
les booléens	**T**	=	$\{true, false\}$
les entiers	**Z**	=	$\{\ldots,-2,-1,0,1,2,\ldots\}$
les symboles	**Sym**	=	un ensemble dénombrable de symboles
les environnements	$\rho \in$ **Env**	=	**Sym** $\rightarrow$ **Ad** à support fini
les stores	$\sigma \in$ **Sto**	=	**Ad** $\rightarrow$ **Val** à support fini
les paires d'addresses	**Paire**	=	**Ad** $\times$ **Ad** $\cup$ $\{nil\}$
les valeurs fonctionnelles	**VFun**	=	**Env** $\rightarrow$ **Sto** $\rightarrow$ **Val*** $\rightarrow$ **Val** $\times$ **Sto**
les valeurs dénotables	**Val**	=	**T** $\oplus$ **Z** $\oplus$ **Paire** $\oplus$ **VFun** $\cup$ $\{\bot, unspecified\}$

Fonctions sémantiques

La sémantique de l'évaluation d'une expression e dans un environnement ρ et un store σ donnés produit une valeur dénotable v et un store σ' , ce qu'on note

$$Eval[\![e]\!]\rho\sigma = (v, \sigma')$$

avec

$$Eval : Expr \rightarrow \textbf{Env} \rightarrow \textbf{Sto} \rightarrow \textbf{Val} \times \textbf{Sto}$$

L'évaluation d'une séquence d'expressions e^* dans un environnement ρ et un store σ donnés produit une séquence de valeurs v^* et un store final σ_f, ce qu'on note :

$$EvalSeq[\![e^*]\!]\rho\sigma = (v^*, \sigma_f)$$

avec

$$EvalSeq : Expr^* \rightarrow \textbf{Env} \rightarrow \textbf{Sto} \rightarrow \textbf{Val}^* \times \textbf{Sto}$$

L'évaluation d'un corps prend une séquence d'expressions e^* dans un environnement ρ et un store σ donnés produit la dernière valeur v_f et un store final σ_f, ce qu'on note :

$$EvalCorps[\![e^*]\!]\rho\sigma = (v_f, \sigma_f)$$

avec

$$EvalCorps : Expr^* \rightarrow \textbf{Env} \rightarrow \textbf{Sto} \rightarrow \textbf{Val} \times \textbf{Sto}$$

L'évaluation d'un corps vide produira une erreur.

La transcription d'un symbole syntaxique en le symbole "sémantique" correspondant est

$$Syms : Symbol \rightarrow \textbf{Sym}$$

La transcription des formels en la séquence de symboles sous-jacents est

$$FormalSyms : Formals \rightarrow \textbf{Sym}^*$$

La transcription des liaisons en une séquence de couples symbole, expression est

$$BindingsSeq : Bindings \rightarrow (\textbf{Sym} \times Expr)^*$$

On a besoin d'un allocateur qui, étant donné un store fournit une adresse inutilisée donc libre dans le store :

$$\mathit{Allocate} : \mathbf{Sto} \to \mathbf{Ad}$$

Si $\mathbf{Ad} = \mathbf{N}$ on pourrait prendre $\mathit{Allocate}(\sigma) = 1 + \max(\mathbf{dom}\,\sigma)$ par exemple.

Equations sémantiques

L'évaluation d'un symbole utilise le store et l'environnement :

$$\begin{array}{rl} \mathit{Eval}[\![\,\texttt{symbol}\,]\!] = & \textbf{soit } s = \mathit{Syms}(\texttt{symbol}) \textbf{ dans} \\ & \lambda\rho\sigma.\ \textbf{si } s \in \mathbf{dom}(\rho) \textbf{ alors } ((\sigma(\rho(s)), \sigma) \\ & \qquad \textbf{sinon erreur } \text{“symbole indéfini”} \end{array} \tag{1}$$

Nous avons besoin de savoir évaluer une séquence, donc de *EvalSeq* :

$$\begin{array}{l} \mathit{EvalSeq}[\![e^*_{seq}]\!] = \\ \quad \textbf{soit } \mathit{boucle} = \lambda b.\ \lambda\rho\sigma e^*. \\ \quad\quad \textbf{si } \#(e^*) = 0 \textbf{ alors } \varnothing_{Val^*}, \sigma \textbf{ sinon} \\ \quad\quad\quad \textbf{soit } e_{tete} = \uparrow e^* \textbf{ dans} \\ \quad\quad\quad\quad \textbf{soit } (v_{tete}, \sigma_{tete}) = \mathit{Eval}[\![\ e_{tete}\ ]\!]\ \rho\sigma \textbf{ dans} \\ \quad\quad\quad\quad \textbf{soit } e^*_{reste} = \downarrow e^* \textbf{ dans} \\ \quad\quad\quad\quad\quad \textbf{soit } (v^*_{reste}, \sigma_{reste}) = b\,\rho\,\sigma_{tete}\,e^*_{reste} \textbf{ dans} \\ \quad\quad\quad\quad\quad\quad (\,\text{prefix}\,v_{tete}\,v^*_{reste}, \sigma_{reste}\,) \\ \quad \textbf{dans fixe } \mathit{boucle}\ e^*_{seq} \end{array} \tag{2}$$

L'évaluation d'un corps *EvalCorps* est similaire :

$$\begin{array}{l} \mathit{EvalCorps}[\![e^*_{seq}]\!] = \\ \quad \textbf{soit } \mathit{boucle} = \lambda b.\ \lambda\rho\sigma e^*. \\ \quad\quad \textbf{si } \#(e^*) = 0 \textbf{ alors erreur } \text{“corps vide”} \\ \quad\quad \textbf{sinon soit } (v_1, \sigma_1) = \mathit{Eval}[\![\uparrow e^*]\!]\ \rho\,\sigma \textbf{ dans} \\ \quad\quad\quad \textbf{si } \#(e^*) = 1 \textbf{ alors } (\,v_1, \sigma_1\,) \textbf{ sinon } b\,\rho\,\sigma_1 \downarrow e^* \\ \quad \textbf{dans fixe } \mathit{boucle}\ e^*_{seq} \end{array} \tag{3}$$

L'application d'un opérateur (fonction ou primitive) teste d'abord [2] la valeur de cet opérateur puis l'application à l'évaluation de la séquence de ses arguments :

$$\begin{array}{l} \mathit{Eval}[\![\ (\ \mathit{expr}_{op}\ \ \mathit{expr}_{arg}{}^*\ )\ ]\!] = \lambda\rho\sigma. \\ \quad \textbf{soit}(v_{op}, \sigma_{op}) = \mathit{Eval}[\![\mathit{expr}_{op}]\!]\ \rho\sigma \textbf{ dans} \\ \quad\ \textbf{si } \text{is}_{\mathbf{VFun}}\,v_{op} \textbf{ alors} \\ \quad\quad \textbf{soit } (v^*_{arg}, \sigma_{arg}) = \mathit{EvalSeq}[\![\mathit{expr}_{arg}{}^*]\!]\,\rho\,\sigma_{op} \\ \quad\quad \textbf{dans } (\text{out}_{\mathbf{VFun}}v_{op})\ (\,\rho\,\sigma_{arg}\,v^*_{arg}\,) \\ \quad\quad \textbf{sinon erreur } \text{“mauvais opérateur à appliquer”} \end{array} \tag{4}$$

Notez que le test de concordance de l'arité de l'opérateur avec le nombre d'arguments actuels se fait ailleurs.

2. Ce n'est pas obligé de faire ce test en premier, et pas le cas dans tous les *Scheme* ; ainsi `guile-2.0` échoue par division par 0 quand il évalue l'expression `(let ( (f 1) ) (f (/ 3 0)))` au lieu d'échouer sur un opérateur non-fonctionnel.

L'évaluation d'un `lambda` crée une valeur fonctionnelle :

$$
\begin{array}{l}
\mathit{Eval}[\![\ \texttt{(lambda}\ \mathit{formals}\ \mathit{body}\texttt{)}\ ]\!] = \lambda\rho_0\sigma_0. \\
\quad \textbf{soit}\ s^* = \mathit{FormalSyms}(\mathit{formals})\ \textbf{dans} \\
\quad\quad \textbf{soit}\ \phi = \lambda\rho_f\sigma_f v_f^*. \\
\quad\quad\quad\quad \textbf{si}\ \#v_f^* \neq \#s^*\ \textbf{alors erreur}\ \text{“arité incorrecte pour application”} \\
\quad\quad\quad\quad \textbf{sinon soit}\ \mathit{boucleformels} = \lambda b.\lambda s^* v^* \rho\sigma. \\
\quad\quad\quad\quad\quad \textbf{si}\ s^* = \varnothing_{\textbf{Sym}}\ \textbf{et}\ v^* = \varnothing_{\textbf{Val}}\ \textbf{alors}\ (\rho,\sigma) \\
\quad\quad\quad\quad\quad \textbf{sinon soit}\ s_1 =\uparrow s^*\ \textbf{et}\ v_1 =\uparrow v^*\ \textbf{dans} \\
\quad\quad\quad\quad\quad\quad \textbf{soit}\ \alpha_1 = \mathit{Allocate}(\sigma)\ \textbf{dans} \\
\quad\quad\quad\quad\quad\quad\quad\quad \textbf{soit}\ \rho_1 = \rho[s_1 \mapsto \alpha_1]\ \textbf{et}\ \sigma_1 = \sigma[\alpha_1 \mapsto v_1]\ \textbf{dans} \\
\quad\quad\quad\quad\quad\quad\quad\quad b \downarrow s^* \downarrow v^* \rho_1\sigma_1 \\
\quad\quad\quad\quad \textbf{dans soit}(\rho_b,\sigma_b) = \textbf{fixe}\ \mathit{boucleformels}\,\rho_0\,\sigma_f \\
\quad\quad\quad\quad \textbf{dans}\ \mathit{EvalCorps}[\![\mathit{body}]\!]\,\rho_b\sigma_b \\
\quad \textbf{dans Inj}_{\textbf{Val}}\ \phi,\sigma_0
\end{array}
\tag{5}
$$

L'évaluation d'un `begin` utilise *EvalCorps* tout naturellement :

$$
\mathit{Eval}[\![\ \texttt{(begin}\ \mathit{body}\texttt{)}\ ]\!] = \mathit{EvalCorps}[\![\mathit{body}]\!] \tag{6}
$$

L'évaluation d'un `let` étend localement les liaisons et le store :

$$
\begin{array}{l}
\mathit{Eval}[\![\ \texttt{(let}\ \mathit{bindings}\ \mathit{body}\texttt{)}\ ]\!] = \\
\quad \textbf{soit}\ bs = \mathit{BindingsSeq}(\mathit{bindings})\ \textbf{dans} \\
\quad\quad \lambda\rho\sigma. \\
\quad\quad\quad \textbf{soit}\ \mathit{boucleliaisons} = \lambda b.\lambda l^*.\lambda\rho_{li}\sigma_{li}. \\
\quad\quad\quad\quad \textbf{si}\ \#l^* = 0\ \textbf{alors}\ (\rho_{li},\sigma_{li}) \\
\quad\quad\quad\quad\quad \textbf{sinon soit}\ (s_{tete},e_{tete}) =\uparrow l^*\ \textbf{dans} \\
\quad\quad\quad\quad\quad\quad \textbf{soit}(v_{tete},\sigma_{tete}) = \mathit{Eval}[\![\ e_{tete}\ ]\!]\ \rho_{li}\,\sigma_{li}\ \textbf{dans} \\
\quad\quad\quad\quad\quad\quad\quad \textbf{soit}\,\alpha_{ap} = \mathit{Allocate}(\sigma_{tete})\ \textbf{dans} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{soit}\ \rho_{ap} = \rho_{li}[s_{tete} \mapsto \alpha_{ap}] \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{et}\ \sigma_{ap} = \sigma_{tete}[\alpha_{ap} \mapsto v_{tete}] \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{dans}\ b\ \downarrow l^*\ \rho_{ap}\ \sigma_{ap} \\
\quad\quad \textbf{dans soit}(\rho_{in},\sigma_{in}) = \textbf{fixe}\ \mathit{boucleliaisons}\,\rho\,\sigma \\
\quad\quad\quad \textbf{dans}\ \mathit{EvalCorps}[\![\ \mathit{body}\ ]\!]\,\rho_{in}\ \sigma_{in}
\end{array}
\tag{7}
$$
