Examen de Rattrapage

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 27 mai 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 3 pages sur 2 feuilles recto-verso)

1 Questions

Les questions §1.1 et §1.2 sont indépendantes entr'elles et du problème §2.

1.1 Exercice de λ-calcul

noté sur 2 points

Réduire en forme normale chacun de ces λ-termes suivants :
$(\lambda x.xx)(\lambda y.y)z$ et $(\lambda x.\lambda y.x)((\lambda x.xx)(\lambda x.\lambda y.xy))$

1.2 Exercice de *Prolog*

noté sur 3 points

Codez le prédicat `seplisnum/3` qui dans la liste de nombres fournie en premier, la separe en la seconde liste des nombres positifs et la troisième liste des nombres négatifs. Ainsi `seplisnum([1,-2,0,5],X,Y)` réussit et répond `X=[1,5]` et `Y=[-2]`

2 Problème : mini-langage T1B1

On considère le mini-langage suivant T1B1 [1] qui ressemble à certains Lisp ou Scheme pour : les valeurs (nil noté `()` qui est la seule valeur fausse [2]), la constante vraie `#t`, les symboles, les entiers, les vecteurs mutables de valeurs, les chaînes de caractères) et le typage dynamique ; la notation parenthésée ; les primitives unaires `number?`, `vector?`, `symbol?`, `make-vector`, `vector-ref` ; les primitives binaires `+`,`-`,`*`,`/`, `eq?`, `>`, `>=`, `<`, `<=`, `vector-set!`, la primitive variaire `vector` pour construire un vecteur, et les structures de contrôle **`if`**, **`begin`**, **`let`**, **`quote`** ; l'affectation `set!`. Mais T1B1 n'a pas de valeurs fonctionnelles de première classe (donc pas de `lambda` ni `letrec`, et les fonctions ne peuvent être qu'appliquées, pas passées en paramètre ou rangées dans un vecteur ou toute autre valeur mutable.). Si le cœur vous en dit, vous pouvez ajouter dans T1B1 -en l'indiquant dans votre copie- d'autres types de valeurs ou primitives de Scheme ou Lisp, à condition de toujours représenter le faux par le nil `()` et de ne pas ajouter de valeurs fonctionnelles.

1. Pour "try one by one"
2. Le faux `#f` de Scheme n'existe pas en T1B1, on utilise nil `()`.

T1B1 diffère sensiblement de Scheme ou Lisp par le traitement des erreurs. Il a une notion d'échec[3] : la primitive -sans arguments- **`fail`** échoue toujours. Une primitive échoue sur des arguments incorrectement valuées (par exemple `(+ () 1)` échoue, car nil n'et pas un nombre). Les primitives de test (dont le nom contient `?`) n'échouent jamais, mais peuvent renvoyer `#t` ou `()`. Une séquence d'expressions utilisée comme corps (d'un `begin`, `let`, etc...) échoue dès qu'une des sous-expressions évaluées échoue. Ainsi `(begin x (fail) (+ y z))` échoue sans jamais tenter de faire l'addition. L'échec se propage à l'appelant.

Un nouvel opérateur de contrôle **`t1b1`** [pour *"try one by one"*] permet de récupérer les échecs. Il prend un nombre quelconque d'operandes, qui ne sont pas forcément tous évalués. Informellement l'évaluation de $\varepsilon \equiv$ (**`t1b1`** e_1 e_2 .... e_n) se fait ainsi : on évalue d'abord e_1. Si cette évaluation réussit en donnant la valeur $v1$ cette valeur est le résultat de ε tout entier -donc de l'évaluation de l'opérateur `t1b1`. Sinon, e_1 a échoué, et on évalue alors e_2. Si l'évaluation de e_2 réussit en donnant v_2 cette valeur v_2 est le résultat de ε. Et ainsi de suite, jusqu'à l'évaluation du dernier opérande e_n. S'il réussit en renvoyant v_n cette valeur v_n est le résultat de ε renvoyé par `t1b1`. Si tous les opérandes e_i échouent à donner une valeur, l'évaluation de ε échoue elle aussi. En particulier `(t1b1)` échoue comme le fait `(fail)`. Et pour `x` valant 2, l'expression `(t1b1 (+ x 1) x)` s'évalue en 3, et s'évalue en nil i.e. `()` si `x` est nil, car alors la sous-expression `(+ x 1)` échoue. Évidemment, une division par zéro échoue, ou l'accès par `vector-ref` à un non vecteur ou en dehors des indices possibles d'un vecteur échoue aussi.

Un programme en T1B1 est une suite de définitions de variables globales implicitement initialisées à nil par **`defvar`** - syntaxe `(defvar` *nomvariable*`)` - ou de fonctions par **`defun`** - syntaxe `(defun` *nomfonction formels corps*...`)`. L'une des fonctions doit s'appeller **`main`**. L'exécution du programme se fait avec des valeurs supposées déjà lues (arguments actuels de `main`), et appliquées à la fonction `main`. Si cette application de `main` produit une valeur, celle-ci est affichée. Si cette application échoue, un bip surgit du haut-parleur.

Voici un petit exemple de programme complet en T1B1.

```
1  (defun checkp2 (x)
2    (if
3      (number? x)
4      (+ x 2)
5      (fail)))
6  (defun main (a)
7    (t1b1
8      (check a)
9      (vector a a)))
```

Si on lance ce programme avec 1 comme argument, `checkp2` est appelé avec `x` lié à 1 en ligne 8. Puis la ligne 3 donne `#t` donc `x` valant 1, c'est 3 qui est renvoyé en ligne 4 par `check` à `main` fin de ligne 8, et qui est le résultat de l'expression `t1b1` lignes 7 à 9 donc du programme. Si on lance ce programme avec le symbole `s` comme argument, `checkp2` est appelé avec `s` en ligne 8. Le test de `number?` en ligne 3 donne nil, donc on continue pour échouer en ligne 5. L'échec est propagé dans l'appelant en fin de ligne 8, donc `t1b1` évalue la primitive `vector` ligne 9 qui donne le vecteur à 2 composants `#(s s)` qui est le résultat de l'expression `t1b1` lignes 7 à 9 donc du programme.

3. L'échec de T1B1 ressemble au lancer d'une exception en Java ou en Ocaml.

En réalité, le traitement des échecs est dynamiquement imbriqué.
Tout comme le traitement des exceptions en Java, C++, Common Lisp, Ocaml est imbriqué dynamiquement.

2.1 question de sémantique opérationnelle

Donnez **brièvement** l'état sémantique puis la sémantique opérationnelle de `fail` et de `/` (en tenant compte de l'échec par division par 0 ou par opérande non numérique).

noté sur 4 points

NB. Il s'agit de donner quelques règles de transitions, pas toute la sémantique opérationnelle du langage !
Cette question est indépendante du reste !

2.2 questions de sémantique dénotationnelle

Traitez les questions 2.2.1 et 2.2.2 dans l'ordre : elles sont dépendantes, la 2.2.2 dépend de la précédente 2.2.1.

2.2.1 état et domaines sémantiques dénotationnels

Donnez l'état sémantique dénotationnel en tenant explicitement compte de l'**imbrication dynamique du traitement des échecs** par les `t1b1` du programme. Donnez les domaines, la signature et le type des fonctions sémantiques.

noté sur 4 points

2.2.2 sémantique dénotationnelle

Donnez par des fonctions sémantiques précises la sémantique dénotationnelle de T1B1. Prenez particulièrement soin de **bien expliquer le traitement des échecs et leur récupération** par `t1b1`.

noté sur 15 points

Insistez surtout sur les **traits originaux** de T1B1 par rapport à ceux déjà vus en cours ou en TD (de Scheme ou de BOPL), et soyez très **rigoureux** et précis.

FIN DE L'ÉNONCÉ DE L'EXAMEN DE RATTRAPAGE DU 27 MAI 2013 - APS MI030