Université Montpellier II
Maîtrise d'Informatique
Module Langage, Evaluation, Compilation — 3 heures

R. Ducournau – M. Lafourcade

janvier 2004

Documents autorisés : notes de cours, polys, pas de livre.
Les 3 parties se suivent. Notation globale sur 26.

1 Passage des paramètres par destructuration

On se propose d'étudier une généralisation du mode de passage des paramètres habituel en LISP.

Soit une fonction `foo` définie par `(defun foo (x y z) ...)` et appelée par une expresion `(foo '1 '2 '3)`. Le mode de passage de paramètres usuel consiste à apparier les *listes* de paramètres `(x y z)` et de valeurs d'arguments `(1 2 3)` pour construire un environnement de la forme `((x . 1) (y . 2)(z . 3))` (au moins dans le méta-évaluateur).

La *destructuration* consiste à remplacer cet appariement de *listes plates* par un appariement d'*arbres binaires*. L'arbre des paramètres est un arbre de cellules LISP quelconque, dont les feuilles sont soit `NIL`, soit des paramètres (symboles non constants). **Les arguments sont évalués de façon habituelle, mais la liste de leurs valeurs est considérée comme un arbre.**

On peut par exemple définir une variante de la fonction `assoc` de recherche dans une liste d'association par :

```
(defun assoc2 (cle ((clex . valx) . al))
      (cond ((eq cle clex) valx)
            (al (assoc2 cle al))))
```

Si `cle` est égal à `clex`, la fonction retourne `valx`, sinon si le reste de la liste d'association (`al`) n'est pas vide, la fonction est appelée récursivement, et sinon elle retourne `nil`.

Lors de l'appel `(assoc2 'x '((y . 1) (x . 2)))`, l'appariement retourne `((cle . x) (clex . y) (valx . 1) (al . ((x . 2))))` (à l'ordre près) et l'appel retourne 2.

La différence avec `assoc` est que l'on retourne la valeur, et non la paire clé-valeur, et que la liste d'association passée en argument ne peut pas être vide : `(assoc2 'x ())` provoquerait une erreur car il n'est pas possible d'apparier `((clex . valx) . al)` avec `()`.

Question 1

Montrer comment le passage de paramètres de COMMON LISP avec le mot-clé `&rest` peut être remplacé par la destructuration. Donner l'exemple des fonctions `list` et `list*`.

Question 2

Définir la fonction `destruct` qui prend 3 arguments, l'arbre des paramètres, l'arbre des arguments, et l'environnement courant, et soit retourne l'environnement augmenté de l'appariement des nouveaux paramètres, soit signale une erreur.

NB. L'appariement est asymétrique car il est guidé par les paramètres : à une cellule de l'arbre des paramètres doit correspondre une cellule de l'arbre des arguments (sous peine d'une erreur "pas assez d'arguments"), à une feuille `nil` de l'arbre des paramètres doit correspondre une feuille `nil` de l'arbre des arguments (sous peine d'une erreur "trop d'arguments"). A une feuille non `nil` (c'est-à-dire un paramètre), peut correspondre n'importe quoi.

Pour le reste, il s'agit d'une récursion d'arbre normale (de même structure que `size-tree` ou `subst`, pour reprendre des exemples du cours) mais en parallèle sur les 2 arbres. Une récursion partiellement terminale est préférable car plus simple.

La méta-évaluation de la destructuration consiste juste à remplacer la fonction `make-env` par la fonction `destruct`. On va s'intéresser maintenant à la compilation de la destructuration, dans le cadre de la génération de code VM du cours. Pour simplifier en limitant les appels de fonctions, on considère que `CAR`, `CDR` et `CONSP` sont des instructions de la machine virtuelle :

- `(CAR R1 R2)` met le contenu du `car` du contenu de `R1` dans `R2` (idem pour `CDR`) ;
- `(CONSP R1)` met à vrai le flag d'égalité si le contenu de `R1` est une cellule, et à faux sinon.

Question 3

Rappelons que la destructuration ne change pas la façon dont les arguments sont évalués avant l'appel. On suppose donc que les arguments sont empilés comme d'habitude. La destructuration est donc à la charge de la fonction appelée.

1. Sur l'exemple de la fonction `assoc2`, montrer ce que doit faire la fonction appelée pour assurer la destructuration. Donner un code VM des parties clés de la fonction `assoc2`.
2. De façon plus générale, spécifier ce que devrait faire la génération de code pour le passage de paramètres par destructuration.

2 Matching

Le *matching* est une généralisation de la destructuration. Quatre points font la différence :

- l'échec du matching n'engendre pas une exception ;
- des feuilles constantes autre que `nil` sont possibles dans l'arbre des paramètres. Dans ce cas, l'appariement réussit si la valeur correspondante dans l'arbre des arguments est égale à la constante. NB La "feuille" constante pourrait bien sûr être une liste ou un symbole LISP mais on simplifiera en ne considérant que des feuilles constantes atomiques.
- le même paramètre peut apparaître plusieurs fois dans l'arbre des paramètres : pour que le matching réussisse, il faut que le paramètre soit chaque fois associé à la même valeur ;
- le symbole _ représente une position vide dont l'appariement est indifférent.

Le *matching* de `((x . 1)(y . x))` avec `((a . 1)(b . a))` réussit et retourne `((x . a)(y . b))`. Par contre, avec `((a . 2)(b . a))` ou avec `((a . 1)(b . c))`, il échoue.

Question 4

Définir la fonction `match`, similaire à `destruct` sauf qu'elle retourne le symbole `:fail` en cas d'échec et qu'elle applique les 2 contraintes précitées d'égalité avec les constantes ou entre plusieurs occurrences d'un même paramètre, ainsi que le traitement de _.

3 Filtrage

Le filtrage est une structure de contrôle basée sur le *matching* : la forme qui l'implémente est nommée `case-match`. Elle est similaire à un `case` (variante du `cond` dans laquelle la valeur d'une expression est comparée à des constantes), mais la comparaison s'effectue par *matching* et, lorsqu'elle réussit, les paramètres correspondants sont considérés comme liés, comme dans un `let` ou une lambda-expression. La syntaxe générale est :

```
(case-match expr
     (filtre-1 . progn-1)
     (filtre-2 . progn-2)
     ...
     (filtre-k . progn-k))
```

Chaque clause est constituée d'une expression (le filtre), qui n'est pas évaluée, et d'un `progn` implicite. La valeur de `expr` est appariée successivement avec les `filtre-i` : au premier succès, le `progn-i` correspondant est évalué, dans l'environnement résultant de l'appariement : l'usage des paramètres présents dans `filtre-i` (à l'exception de _) est donc possible dans `progn-i`. Le filtre _ (placé en dernier) récupère tous les cas, comme un `T` dans un `cond`. Si aucun filtre ne réussit, `case-match` retourne `nil`.

On pourra ainsi définir une nouvelle variante de `assoc` de la façon suivante :

```
(defun assoc3 tree
      (case-match tree
            ((_ ()) ())
            ((cle ((cle . val) . _)) val)
            ((_ ((_ . _) . rest)) (assoc3 cle rest))
            (_ (error "assoc3, erreur de syntaxe ~s" tree))))
```

On compare la valeur de `tree`, liée à la totalité de l'arbre des arguments, aux filtres successifs formés :

- d'une clé quelconque et d'une liste d'association vide,
- d'une clé et d'une liste d'association dont la première clé est la clé précitée,
- d'une clé quelconque et d'une liste d'association dont la première clé est quelconque,
- le filtre quelconque traduit enfin une erreur d'utilisation de la fonction.

NB `assoc3` a encore un comportement légèrement différent de `assoc` et `assoc2` : comme la première il accepte une liste d'association vide, mais comme la seconde, il retourne la valeur associée et non pas la paire clé-valeur.

Question 5

Dans le cadre du méta-évaluateur, implémenter `case-match`. On définira une fonction `meval-case-match` avec les paramètres adéquats.

NB C'est la façon la plus simple d'implémenter le filtrage.

La compilation du filtrage est plus compliquée.

Question 6

Une première façon de faire relève de la transformation source à source, donc d'une macro. Une clause de filtrage doit alors se traduire par des tests, pour vérifier l'appariement de la valeur avec le filtre, et des `let`, pour introduire les nouvelles variables (l'environnement n'est plus créé explicitement, ce sont les `let` qui le feront, à l'évaluation).

1. En prenant l'exemple de la fonction `assoc3`, montrer comment devrait s'expanser une macro `case-match`.

2. Définir la macro `case-match` (on pourra se restreindre à une esquisse et se contenter de spécifier des fonctions auxiliaires).
3. Montrer quelles optimisations seraient nécessaires pour que l'appariement sur un filtre ne refasse pas ce qui a déjà été fait sur les filtres précédents (prendre l'exemple de `assoc3`).

Question 7

Pour générer le code VM correspondant, on peut se contenter de compiler le résultat de l'expansion de la macro précédente. On peut aussi générer directement le code VM. Comme pour la question précédente, montrer quel code VM générer sur l'exemple de `assoc3`, comment le générer, et quelles optimisations envisager.

En savoir plus

Le filtrage est à la base de plusieurs langages de programmation, fonctionnels (HASKELL) ou logiques (PROLOG). Christian Queinnec lui a consacré un joli petit livre, malheureusement épuisé (chez InterEditions). Le *matching* est à la base de tous les systèmes de règles (logiques, réécriture, systèmes experts, etc.) Enfin, la destructuration est utilisée dans de nombreux dialectes LISP : en COMMON LISP, elle n'existe que dans la forme spéciale `destructuring-let`.