Université Montpellier II — Master d'Informatique
UMINM 121 Evaluation des Langages Applicatifs — 2 heures

R. Ducournau – M. Lafourcade

10 janvier 2006

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 habituel de passage des paramètres.

Soit une fonction `foo` définie par `(defun foo (x y z) ...)` et appelée par une expression `(foo 'a 'b (+ 1 2))`. Le mode usuel de passage de paramètres 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))` — c'est du moins la structure des environnements 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 binaire quelconque de cellules LISP, dont les feuilles sont soit `NIL`, soit des paramètres (symboles non constants) ;
- l'arbre des valeurs d'arguments est en fait la liste des valeurs d'arguments — évalués de façon habituelle — mais qui est interprétée comme un arbre binaire.

L'appariement n'est possible que si chaque cellule de l'arbre des paramètres correspond à une cellule de l'arbre des arguments.

La destructuration va donc permettre de définir la fonction `foo` qui précède, avec les résultats habituels, mais aussi la fonction `bar` suivante :

```
(defun bar (cle ((clex . valx) . al))
       ...)
```

qu'il est possible d'appeler avec l'expression `(bar a b)` à condition que la valeur de `b` puisse s'apparier à `((clex . valx) . al)`.

Lors de l'appel `(bar 'x '((y . 1) (x . 2)))`, l'appariement entre l'arbre de paramètres `(cle ((clex . valx) . al))` et la liste de valeurs `(x ((y . 1) (x . 2)))` retourne `((cle . x) (clex . y) (valx . 1) (al . ((x . 2))))` — à l'ordre près.

Les expressions `(bar 'x ())`, `(bar 'x 1)`, `(bar 'x '(1))` provoqueraient toutes une erreur car il n'est pas possible d'apparier `((clex . valx) . al)` avec la valeur de leur deuxième argument — `()`, `1` ou `(1)`.

Question 1

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, à une feuille `nil` de l'arbre des paramètres doit correspondre une feuille `nil` de l'arbre des arguments. A une feuille non `nil` (c'est-à-dire un paramètre), peut correspondre n'importe quoi.

1. Montrer comment exprimer avec la destructuration le même passage de paramètres qu'avec le mot-clé `&rest`. Donner l'exemple des fonctions `list` et `list*`.

2. montrer comment se traduit, dans la destructuration, l'erreur habituelle `"mauvais nombre d'arguments"`, par exemple lors des appels `(foo '1 '2)` ou `(foo '1 '2 '3 '4)`.
3. 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.

   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.
4. Quelle modification faut-il faire au méta-évaluateur pour remplacer le passage de paramètres habituel par la 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 ;
– lorsqu'une feuille de l'arbre des paramètres est une constante autre que `nil`, l'appariement réussit si la valeur correspondante dans l'arbre des arguments est égale à la constante ;
– 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 2

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`.

Question 3

Le filtrage permet de se passer de la plupart des conditionnelles.

1. Écrire en utilisant `case-match` la fonction `assoc` qui prend 2 arguments, une clé et une liste d'association, et recherche la clé dans la liste, en retournant la paire dont le `car` est la clé, ou `nil`.

   NB. Le code résultant ne doit pas contenir d'autres structures de contrôle que `case-match`.

2. Dans le cadre du méta-évaluateur, implémenter `case-match` comme une forme spéciale, en l'écrivant en LISP de base (avec `if` et `cond`). On définira une fonction `meval-case-match` avec les paramètres adéquats.

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

3. Redéfinir `meval-case-match` en l'écrivant avec `case-match`, sans autre structure de contrôle si c'est possible.

   Le code résultant est-il évaluable ?

Question 4

Une alternative consiste en une transformation source à source, donc 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. Montrer sur l'exemple de la fonction `assoc` 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 les tests qui ont déjà été faits pour les filtres précédents (prendre l'exemple de `assoc`).

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), mais disponible sur sa page Web. Le *matching* est à la base de tous les systèmes de règles (logique, 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`.