Programmer des relations en Common Lisp

Robert Strandh

2009-03-13

1 Introduction

Dans cet article, nous expliquons comment on peut se servir du langage Common Lisp pour définir des relations, et plus particulièrement comment se servir des spécificités de la partie "orientée objets" de ce langage, à savoir CLOS (Common Lisp Object System) pour représenter des relations.

Alors que nos définitions permettent à la fois des relations infinies et des relations finies de manière transparente, c'est surtout les relations finies qui permettent une représentation intéressante du point de vue de la programmation. Pour cette raison, l'essence du code présenté traite les relations finies.

2 Classes et de fonctions génériques pour les relations

La classe de base aura comme seul attribut l'*arité* de la relation. C'est un attribut qui est indiqué lors de la création de la relation et qui ne peut pas être modifié par la suite :

```
(defclass relation ()
  ((arity :initarg :arity :reader arity)))
```

La seule opération commune possible pour toutes relations est de vérifier si un uplet est élément de la relation. Nous appelons cette opération `element` :

```
(defgeneric element (tuple relation))
```

Il est judicieux de vérifier avant tout application de la fonction `element` que l'uplet fourni a un nombre de composants correspondant à l'arité de la relation. Ceci est valable pour toutes relations. On se sert alors d'une méthode `:before` qui sera exécutée *avant* les implémentations plus spécifiques de la fonction :

```
(defmethod element :before (tuple (relation relation))
  (assert (= (arity relation) (length tuple))))
```

Certaines relations sont définies par une fonction caractéristique qui, étant donné un uplet, renvoie vrai ou faux selon si l'uplet est élément de la relation. Nous appelons ce type de relation `functional-relation`. Pour une telle relation, il faut stocker la fonction caractéristique dans un créneau. Comme pour l'arité, on ne peut pas modifier la fonction une fois la relation créée :

```
(defclass functional-relation (relation)
  ((fun :initarg :fun :reader fun)))
```

L'implémentation de l'opération `element` pour une relation de ce type est simple. Il suffit d'appliquer la fonction caractéristique à l'uplet donné :

```
(defmethod element (tuple (relation functional-relation))
  (apply (fun relation) tuple))
```

Une relation de type `functional-relation` est utile, mais on ne peut pas avoir d'opérations plus intéressantes que `element`, car la définition de la relation est opaque. Par contre, pour une relation *finie*, on peut trouver des manières intéressantes d'implémentation. Pour cette raison, nous définissions une relation `finite-relation` sous classe de `relation`. Pour une relation finie, on introduit un créneau contenant une list d'uplets éléments de la relation. Ce créneau peut être modifié après création afin de permettre la modification a posteriori de la relation :

```
(defclass finite-relation (relation)
  ((tuples :initarg :tuples :initform '() :accessor tuples)))
```

L'implémentation de l'opération `element` pour une relation finie consiste à vérifier si l'uplet figure dans la liste. Pour cela, on utilise une fonction `mem` qui n'existe pas en Common Lisp, mais qui est facile à programmer à partir de la fonction existante `member`, en utilisant comme `test` la fonction `equal`. Ceci est nécessaire car la fonction `member` utilise par défaut comme `test` la fonction `eql` qui n'est pas capable de comparer deux listes non identique mais dont la structure est la même :

```
(defmethod element (tuple (relation finite-relation))
  (mem tuple (tuples relation)))
```

Certaines relations sont *réflexives*. Afin de capturer ce qui caractérise une telle relation, et afin de pouvoir définir des relations plus spécifique à partir de relations réflexives, on peut se servir de la technique des classes *mixins* qui sont destinées à être héritées par d'autres classes, mais non à être instanciées en tant que telles :

```
(defclass reflexive-mixin () ())
```

Pour une relation réflexive, afin de vérifier si un uplet est élément de la relation, on peut commencer par vérifier si tous les composant de l'uplet sont identique. Ici, nous allons nous restreindre à des relation binaires. On se sert d'une méthode `:around` sur la fonction `element` dont le but est de vérifier cela avant que méthode principale sur la relation ne soit invoquée. Si les deux composants sont identique, alors on renvoi vrai, sinon, on appelle la méthode suivante dans la chaîne de méthodes applicables. Cela évite d'avoir besoin de stocker explicitement les uplets dont les composants sont identique.

```
(defmethod element :around (tuple (relation reflexive-mixin))
  (or (destructuring-bind (a b) tuple
        (equal a b))
      (call-next-method)))
```

On va avoir besoin d'instancier une relation ayant comme seule caractéristique la réflexivité. Pour cela, on définie une classe pour cet effet, et qui hérite de la classe `reflexive-mixin` :

```
(defclass reflexive-relation (finite-relation reflexive-mixin) ())
```

Certaines relations sont *symétriques*. On utilise alors la même technique que pour les relations réflexive afin de capturer ce qui caractérise de telles relations :

```
(defclass symmetric-mixin () ())
```

Pour une relation symétrique, soit l'uplet donnée à la fonction `element` est élément de la relation, et dans le cas contraire, on vérifie si c'est le cas de l'uplet renversé :

```
(defmethod element :around (tuple (relation symmetric-mixin))
  (or (call-next-method)
      (call-next-method (reverse tuple) relation)))
```

Comme pour la relation réflexive, on définie une classe destiné à l'instanciation :

```
(defclass symmetric-relation (finite-relation symmetric-mixin) ())
```

Certaines relations sont *transitives*. On utilise alors la même technique que pour les relations réflexive afin de capturer ce qui caractérise de telles relations :

```
(defclass transitive-mixin () ())
```

Pour une relation transitive, un uplét `(x y)` est élément de la relation s'il existe un uplet `(x b)` dans la relation et l'uplet `(b y)` est élément de la relation :

```
(defun mem-aux (tuple list)
  (or (mem tuple list)
      (destructuring-bind (x y) tuple
        (some (lambda (tuple)
                (destructuring-bind (a b) tuple
                  (and (equal x a)
                       (mem-aux (list b y) (rmm tuple list)))))
              list))))

(defmethod element :around (tuple (relation transitive-mixin))
  (or (call-next-method)
      (mem-aux tuple (tuples relation))))
```

Comme pour la relation réflexive, on définie une classe destiné à l'instanciation :

```
(defclass transitive-relation (finite-relation transitive-mixin) ())
```

Une relation ayant plusieurs des caractéristiques définis ci-dessus peut être créée par héritage de plusieurs des classes mixins. Ici nous avons créé une relation d'équivalence par héritage des trois classes mixins ci-dessus :

```
(defclass equivalence-relation
    (finite-relation reflexive-mixin symmetric-mixin transitive-mixin)
  ())
```

3 Conclusions

Le langage Common Lisp contient un système de programmation par objets (CLOS) très performant. Nous nous servons des spécificités de ce système (méthodes :before et :around) afin de factoriser au maximum le code. Cela nous permet d'éviter de stocker tous les uplets d'une relation dont certains aspect peuvent être décrits par un petit programme à la place.

Ainsi, pour une relation symétrique, ce petit programme nous permet d'économiser la moité des uplets, et pour une relation réflexive, nous n'avons pas besoin de représenter explicitement les uplets dont les composants sont égaux.