Programmation avancée

TD5 : Programmation orientée Objet

Jacques-Henri Jourdan Armaël Guéneau Arnaud Golfouse Paul Robert

Pensez bien à vérifier que votre code compile. Si vous n'avez pas terminé, remplacez les parties manquantes par `failwith "TODO"`.

Note Téléchargez le code de démarrage de ce TP à l'adresse suivante : `https://jhjourdan.gitlabpages.inria.fr/prog3-l3-ensps/tds/05-oop/td05.zip`

1 Échauffement

(fichier `lib/partie1.ml`)

Il y aura à présent des tests écrits directement dans le code Ocaml, de la forme

`let%test "nom_du_test" = ...` ou `let%test_unit "nom_du_test" = ...`

Pour les lancer, on utilise `dune` :

```
dune test
```

Bien penser à décommenter ces tests au fur et à mesure !

Exercice 1. *Créez une classe `counter`, qui contient une valeur `x : int` et une méthode `incr : () -> ()`, qui augmente de `1` l'entier `x`.*

On a alors un problème lorsqu'on veut utiliser cette classe : le corriger en ajoutant une méthode.

Exercice 2. *Implémentez la classe suivante :*

```
class int_range (* arguments *) = object
  (* ... *)
  method next () : int option = (* ... *)
end
```

De façon à ce qu'appeler `next` sur `int_range 2 5` donne `Some 2`, `Some 3`, `Some 4`, `None`, `None`, ...

Écrire alors la fonction `foreach range (f : int -> unit)`, qui appelle `f` sur le résultat de `next`.

Exercice 3. *Écrire explicitement le type des arguments de* `foreach`*, tel qu'inférés par le compilateur. On voit alors que* `range` *n'est pas nécessairement un objet de type* `int_range` *!*

Implémentez une classe `iter_int_list`*, qui reçoit une liste d'entier en argument et dont la méthode* `next` *donne successivement tous les entiers de la liste.*

Généralisez ensuite cette classe à `iter_list`*, de manière à pouvoir itérer sur des listes de type* `'a list` *(changez également le type des arguments de* `foreach` *si besoin).*

Exercice 4. *On donne une structure d'arbres binaires suivante :*

```
type 'a binary_tree = Node of ('a binary_tree * 'a binary_tree) | Leaf of 'a
```

Écrire une classe `iter_binary_tree` *qui permette d'itérer sur les feuilles de ce type.*

Exercice 5. *Implémentez une classe* virtuelle `['a] iterator`*, telle que* `foreach` *soit à présent une méthode (non virtuelle) de cette classe.*

En d'autres termes, on veut pouvoir implémenter la fonction `foreach` *ainsi :*

```
let foreach (it : 'a iterator) (f : 'a -> unit) : unit = it#foreach f
```

Faire ensuite en sorte que les itérateurs définis dans les exercices 2, 3 *et* 4 *héritent de cette classe.*

2 Visiteurs et transformations de programmes

(fichier `lib/partie2.ml`)

Les visiteurs sont un *design pattern* très puissant et expressif, fréquemment utilisé dans les langages impératifs (tel que Rust). En OCaml la librairie `visitors` de Francois Pottier développe le concept. Nous allons maintenant explorer d'autres formes de visiteurs en faisant des transformations de programmes sur un langage arithmétique.

On définit le type de nos expressions comme ceci:

```
type 'var expr =
  | Var of 'var
  | Int of int
  | Plus of 'var expr * 'var expr
  | Times of 'var expr * 'var expr
```

Ainsi que la signature polymorphe de nos visiteurs:

```
class virtual ['var, 'res] arith_visitor =
  object (self)
    method virtual visit_var : 'var -> 'res
    method virtual visit_int : int -> 'res
    method virtual visit_plus : 'var expr -> 'var expr -> 'res
    method virtual visit_times : 'var expr -> 'var expr -> 'res
```

```
  method visit_expr (e : 'var expr) : 'res = (* ... *)
end
```

Les objets de la class `arith_visitor` sont paramétré par le type des variables et le type de sortie de la visite. Cela nous permettra d'écrire un visiteur qui produit un résultat particulier en plus des effets de bord qui sont effectués pendant la visite.

Exercice 6. *Écrivez un visiteur concret* `['var] eval` *comme une sous-classe de* `arith_visitor`*, qui prend en argument un* `('var * int) list` *représentant les valeurs de toutes les variables, et évalue une expression arithmétique.*

Exercice 7. *Écrivez un visiteur* `const_eval`*, qui évalue les sous-expressions constantes. C'est à dire, on transforme* `x + 4 * 5` *en* `x + 20`*.*

Validez votre transformation en utilisant `eval` *pour vérifier que l'évaluation produit le même résultat dans les deux cas.*

Exercice 8. *Écrivez un visiteur* `permutation_of`*, qui prend une expression* `e1` *en argument, et dont le visiteur détermine si une expression* `e2` *est une* permutation *de* `e1` *; c'est-à-dire, si* `e1` *est égal à* `e2` *où certaines additions/multiplications ont leurs opérandes inversées.*

3 Interface « Graphique »

(fichier `lib/partie3.ml`)

Dans cet exercice nous allons simuler une « interface graphique » dans le terminal, en utilisant la libraire `notty`. La documentation de cette librairie est disponible ici.

On fournit le module `Onotty`, qui définit une classe `widget`, et divers classes qui héritent de `widget`, comme `button`, `textfield`… Voir le fichier `common/onotty.mli` pour plus d'informations.

Exercice 9. *Commencez par créer une fenêtre contenant trois widgets:*

1. *un* `button` *qui affiche* `Num. clics: 0`
2. *un* `textfield`
3. *un* `textarea`

Exercice 10. *En utilisant la méthode* `register_mouse_listener`*, faire en sorte que le numéro affiché sur le bouton soit incrémenté à chaque clic sur le bouton.*

Exercice 11. *Créer une* sous-classe *de* `button` *qui rajoute une fonction* `set_color` *qui rajoute une bordure colorée au bouton. Modifier le bouton de votre interface pour qu'il change de couleur de bordure au premier clic.*

Exercice 12. *Utiliser un event listener pour recopier le texte dans le* `textfield` *jusqu'au* `textarea` *quand l'utilisateur frappe la touche retour.*

Exercice 13 (Bonus). *En créant une nouvelle classe, faire en sorte que le* `textfield` *soit également un bouton, qui efface la zone de texte lorsqu'il est cliqué.*