Licence STS, semestre 4 — 2009–2010
Langages et Génie Logiciel (Info 223) — 19 février 2010
`http://www.lri.fr/~paulin/LGL`

Cours 5 : Tests automatiques de fonctions

1 Motivations

Il est important de tester son code *juste après* l'avoir écrit. Il est ainsi plus facile de se souvenir des détails de son implémentation et donc de résoudre les problèmes. De plus, il est important de tester son code *le plus souvent possible*. Ceci facilite la localisation du problème.

Il faut faire particulièrement attention aux fonctions qui vont être réutilisées plusieurs fois, par exemple celles qui sont exportées par une librairie. Il est utile d'écrire un petit programme qui vérifie que la fonction a bien le comportement attendu sur un certain nombre d'entrées bien choisies. On peut alors avoir confiance en la fonction testée et passer à la suite. De plus, si l'implémentation de la fonction change, le programme de test peut être réutilisé pour vérifier que la fonction n'a pas perdu des fonctionnalités.

2 Tester sur quelques entrées choisies à la main

Voici un programme vérifie que la fonction `max` a le comportement attendu.

```
(* verifier un seul cas *)
let test f x y r =
  if f x y <> r then
    prerr_string "Erreur"

(* verifier une liste de cas *)
let tests f l =
  List.iter (fun (x, y, r) -> test f x y r) l

(* verifier quelques cas choisis a la main *)
let () =
  test max [
    5, 9, 9; (* cas x < y *)
    4, 2, 4; (* cas x > y *)
    5, 5, 5; (* cas x = y *)
  ]
```

Les fonctions `test` et `tests` peuvent être réutilisées pour tester d'autres fonctions à deux arguments.

Il est important de bien choisir les cas à tester. Pour la fonction `max`, il y en a essentiellement trois : $x < y$, $x > y$ et $x = y$.

3 Tester des entrées aléatoires

3.1 Prédicats

Pour tester une fonction sur un grand nombre d'entrées, on peut générer celles-ci aléatoirement. Mais dans ce cas, on ne connaît pas le résultat attendu. On connaît cependant certaines *propriétés* du résultat. Dans le cas de la fonction `max`, on sait que :

- le résultat est plus grand que le premier argument `x` ;
- le résultat est plus grand que le second argument `y` ;
- le résultat est égal soit à `x`, soit à `y`.

Ces propriétés forment une *spécification* de la fonction `max`. De plus, cette spécification est *totale*, car toute fonction vérifiant ces propriétés est équivalente à la fonction `max` vu de l'extérieur.

On peut écrire ces propriétés sous forme de *prédicats*. Il s'agit de fonctions prenant en entrée : les entrées `x` et `y` de la fonction `max`, une valeur de sortie `r`, et qui renvoie un booléen. Voici les prédicats respectifs des propriétés ci-dessus :

```
let p1 x y r = r >= x
let p2 x y r = r >= y
let p3 x y r = r = x || r = y
```

On peut combiner ces trois prédicats en un seul :

```
(* combinaison de deux predicats *)
let pred_and p1 p2 x y r = p1 x y r && p2 x y r

(* specification de la fonction max *)
let max_spec = pred_and (pred_and p1 p2) p3
```

3.2 Test

```
(* test_predicat f p n :
     teste si la fonction f verifie le predicat p sur n entrees aleatoires *)
let rec test_predicat f p n =
  if n >= 0 then begin
    (* generer les entrees x et y aleatoirement *)
    let x = Random.int 1000 and y = Random.int 1000 in

    (* verifier f sur ces entrees *)
    let r = f x y in
    if not (p x y r) then
      Printf.eprintf "Erreur: pour x = %d et y = %d, le resultat est %d\n%!"
        x y r;

    (* passer au cas suivant *)
    test_predicat f p (n - 1)
  end
```

Par exemple, pour tester la fonction `max` sur 1000 cas, vous pouvez utiliser :
`test_predicat max max_spec 1000`
Pour d'autres fonctions, il vous faudra adapter `test_predicat` en changeant le nombre, le type et l'ensemble des valeurs que les arguments peuvent prendre.