Un environnement réfutable pour les tests en OCaml

Xavier Van de Woestyne

Décembre 2014

Cet article présente l'usage d'une nouveauté dans OCaml 4.02 permettant une extension de syntaxe entre des quotations. Cette nouveauté à été l'occasion d'implémenter un petit outil pour favoriser la création de "tests" *inline*.

Cet article ne présente pas `ppx` en règle général mais uniquement l'outil `ppx_test`.

Avant-propos

Lorsque j'ai eu l'occasion de contribuer à Batteries, mon principal apport aura été assez annecdotique. En effet, j'ai mis en oeuvre une série de tests pour réussir à extraire le module Substring de l'incubateur. La méthode de tests utilisée (QTest/ItemML, qui évalue des commentaires sous forme de tests s'ils sont préfixés par T) s'est révélée assez utile pour plusieurs raisons :

- Test en dessous de la fonction testée
- Plus digeste (de mon point de vue) que OUnit

Mais à ces avantages s'ajoutent quelques désavantages (de mon point de vue, une fois de plus) :

- Perte de la colorisation syntaxique (facilement réfutable en écrivant le test avant de le commenter)
- Impossibilité de décrire "facilement" une variable locale.
- Obligation d'utiliser un "\" en fin de ligne pour créer un test sur plusieurs lignes.

Utilisation de -ppx Depuis sa version 4.02, OCaml possède un préprocesseur moins lourd que CamlP4 qui mimétise le mécanisme de quotation. L'idée de `ppx_test` est d'ajouter des portions de codes qui ne sont pas intèrprétée à la compilation. Sauf en cas d'usage de l'annotation `-ppx ppx_test`.

Présentation

Dans les grandes lignes, en utilisant `ppx_test`, des séquences de codes seront exécutées. Par exemple, le code :

```
[@@@test_process
  let x = 9
  in assert(x = 9)
]
```

Sera ignoré à la compilation. Par contre, si le préprocesseur de test est utilisé, la portion présentée sera substituée par :

```
let _ =
  let x = 9
  in assert(x = 9)
```

A l'exécution de code compilé, le test sera donc exécuté.

[@@@test_process]

Cette annotation permet de définir une portion de code qui sera exécutable en cas de test. Il est donc possible d'utiliser n'importe quelle expression OCaml valide. Comme une ouverture, ou une inclusion.

Par exemple :

```
[@@@test_process open OUnit]
```

N'ouvrira le module OUnit qu'en cas de test.

[@@@test_register]

Il est aussi possible de convertir une portion de code en fonction qui sera sauvegardée dans une liste et utilisable par après. Par exemple :

```
(* Example of ppx_test usage*)

module Essay =
struct
  let succ x = x + 1
  let pred x = x - 1
  let is_null x = (x = 0)
  let divisible_by2 x = (x mod 2) = 1
end

[@@@test_register
  assert((Essay.pred 9) == 8)
]
[@@@test_register
  assert(Essay.is_null 0);
  assert(not (Essay.is_null 1))
]
(* Test with value binding*)
[@@@test_register
  let x = 9 in
    assert((Essay.succ x) = 12) (* this test fail *)
]

(* Run all tests *)
[@@@test_process
  execute_tests ()
]
```

La primitive `execute_tests ()` va exécuter chacun des tests au moment de son appel.

Mini_unit

Mini_unit n'a pas d'autre intérêt que de présenter un usage de `ppx_test`. C'est une collection de fonctions d'assertions et en voici sa signature :

```
module Assert :
sig
  val count : int ref
  val success_count : int ref
  val output : string -> bool -> unit
  val execute : ?name:string -> bool -> unit
  val isTrue : ?name:string -> bool -> unit
  val isFalse : ?name:string -> bool -> unit
  val isEquals : ?name:string -> 'a -> 'a -> unit
```

```
  val isNotEquals : ?name:string -> 'a -> 'a -> unit
  val isSame : ?name:string -> 'a -> 'a -> unit
  val isNotSame : ?name:string -> 'a -> 'a -> unit
end
val report : unit -> unit
```

Voici un exemple de son utilisation :

```
(* Small example with mini_unit usage *)

(* Open Mini_unit in test context*)
[@@@test_process open Mini_unit]

let incr_list l = List.map (succ) l
let decr_list l = List.map (pred) l
let sum_list  l = List.fold_left ( + ) 0 l

(* Test definition *)
[@@@test_register
  let raw_list = [1;2;3] in
  let out_list = incr_list raw_list
  in Assert.isEquals
    ~name:"Test of incr_list"
    out_list [2;3;4]
]

[@@@test_register
  let li = [1;2;3]
  in Assert.isEquals
    ~name:"Test of sum_list"
    (sum_list li) 6
]

(* Test with failure !! *)
[@@@test_register
  Assert.isTrue
    ~name:"Test with fail !!"
    false
]

(* Test execution *)
[@@@test_process
  let _ = execute_tests () in report ()
]
```

La fonction `Mini_unit.report ()` affiche un compte rendu du jeu de test exécuté.

ppx_no_test

Dans le cas où les annotations non évaluée renverraient des warnings, le projet propose un autre préprocesseur qui éradique les annotations de tests. `ppx_no_test`.

Conclusion

C'est un projet amusant pour s'initier aux -ppx's. Dans un futur très proche, je me concentrerai sur son utilisation avec OUnit en pensant, notamment, à une manière de nommer un test enregistrer, autrement qu'au moyen de la fonction d'assertion.