Un environnement réfutable pour les tests en OCaml

Xavier Van de Woestyne

Décembre 2014

Cet article présente l'extension de syntaxe que j'ai développé pour permettre de créer des tests "inlines" en bénéficiant des avantages proposés par QuickTest. Notamment le fait de rédiger les tests dans des commentaires.

Vous pourrez trouver cette extension ici

Avant-propos

Lors de mes contributions (bien maigre) à Batteries, mon apport principal aura été de tester (principalement des fonctions du modules `Substring`). Les tests s'effectuaient au moyen de **QuickTest**. Ils apportaient bien du confort, notamment le fait que les tests soient rédigés directement dans le fichier. De cette manière :

```
(*$T foo
foo  0 ( + ) [1;2;3] = 6
foo  0 ( * ) [1;2;3] = 0
foo  1 ( * ) [4;5]   = 20
foo 12 ( + ) []      = 12
*)
```

Le principal avantage (de mon point de vue) est que les tests sont concis et que l'on peut vérifier facilement l'exhaustivité des tests. Cependant, pour la construction de tests multi-lignes, il faut impérativement terminer les lignes par un backslash et les commentaires ne préservent pas la colorisation syntaxique de notre editeur.

Depuis OCaml 4.02, il existe un préprocesseur "local", n'étant pas un constituant de CamlP4. Il propose deux type d'extensions. Les point d'extensions, qui génèrent une erreur s'ils ne sont pas évalués et les attributs, qui sont ignorés s'ils ne sont pas évalués.

L'idée général de `ppx_test` est d'utiliser les attributs pour proposer un environnement réfutable de tests. Soit, lors d'une compilation sans l'utilisation du

préprocesseur `ppx_test`, les tests sont ignorés, et l'utilisation du préprocesseur `ppx_test`, produit un exécutable ayant rendu les tests efficient.

Code exécuté dans le contexte des tests

Le préprocesseur propose l'évaluation de deux annotations. La première: `@@@test_process` rend le code qu'elle embarque exécuté dans l'exécutable produit avec avec les tests. Par exemple, le code :

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

Produira, à la compilation avec le préprocesseur, le code suivant :

```
let _ =
  let x = 9 in
    assert(x = 0)
```

En général, cette annotation sert à ouvrir, uniquement dans le contexte de tests, des modules destiné aux tests, ou alors d'exécuter une routine d'exécution des tests enregistrés, que nous verrons à la section suivante.

> Il est évidemment possible de mettre autant d'annotation de procédure de tests que l'on désire !

Enregistrement de tests

Il existe une autre annotation qui sauvegarde un tests comme une fonction de type `unit -> unit` dans la liste des tests créés. Cette liste est accessible au moyen de a fonction `registered_tests` dont le type est `unit -> (string * (unit -> unit)) list`. Cette fonction est construite par le préprocesseur. La chaine de caractère correspond à l'identifiant choisi pour le test et la fonction `unit -> unit` a la fonction de test. Par exemple :

```
[@@@test_register "un_test",
   let x = 9 in assert(x = 9)
]
(* et en plus *)
[@@@test_register "un_autre_test",
   let x = 10 in assert(x = 11) (* Ce test échouera *)
]
```

Ces deux tests seront sauvegardés dans la liste des tests enregistrés (`registered_tests ()`). Il est évidemment possible, comme pour les annotations de procédure, de définir autant d'annotation d'enregistrement de tests que l'on désire.

Exécution des tests enregistrés

Le préprocesseur génère, en plus, une fonction qui exécute la liste de tests. (Bien qu'elle soit très facile à coder, un simple `List.map` suffit). Par exemple :

```
let du_code_ocaml_normal = 10
[@@@test_register "un_test",
   let x = 9 in assert(x = 9)
]
let encore_du_code_ocaml_normal = 11
(* et en plus *)
[@@@test_register "un_autre_test",
   let x = 10 in assert(x = 11) (* Ce test échouera *)
]
(* Exécution des tests *)
[@@@test_process
   execute_test ()
]
```

Au lancement de ce programme, compilé avec `-ppx ppx_test.native`, la liste tests sera exécutée. Il affiche par défaut, sur la sortie standard, l'identifiant du test en cours d'exécution. Comme les tests enregistrés utilisent `assert`. Un echec de test interrompra le programme et affichera l'exception sur la sortie standard.

> L'exécution des tests est assez anecdotique, typiquement, nous verrons plus loin dans ce document qu'il faut parfois implémenter notre propre routine d'exécution.

Il est possible d'utiliser la fonction `execute_test` de cette manière : `execute_test ~verbose:false ()` pour ne pas afficher sur la sortie standard le test en cours de vérification.

Mini_unit

`Mini_unit` est une toute petite collection de fonctions pour émettre des assertions. Il n'a aucune vocation particulière au delà de pouvoir tester facilement `ppx_test`. Ce n'est absolument pas un outil qu'il est conseillé d'utiliser. Sa présence n'a de sens que dans la présentation de `ppx_test`. Voici sa signature :

```
module Assert :
  sig
    val isTrue : ?verbose:bool -> bool -> unit
    val isFalse : ?verbose:bool -> bool -> unit
    val isEquals : ?verbose:bool -> 'a -> 'a -> unit
    val isNotEquals : ?verbose:bool -> 'a -> 'a -> unit
  end
val report : unit -> unit
```

La fonction `report` affiche un petit état des lieux des tests exécutés. Voici par exemple un usage du framework (uhu, c'est peut être un peu trop overkill comme nom) testant deux fonctions :

```
[@@@test_process open Mini_unit ]

let succ x = x + 1
let pred x = x - 1

[@@@test_register "test_succ",
    let x = 9 in
       Assert.isEquals (succ x) 10
]

[@@@test_register "test_pred",
    let x = 9 in
       Assert.isEquals (pred x) 8
]

(* Test qui va échouer *)
[@@@test_register "test_qui_echoue",
    let x = 9 in
       Assert.isNotEquals (succ x) 10
]

(* un dernier tests pour la route *)
[@@@test_register "un_dernier_test",
    Assert.isNotEquals 10 11
]

(* Lancement des tests *)
[@@@test_process
  let _ = execute_tests () in report ()
]
```

Comme l'indique le code, on commence par ouvrir le module `Mini_unit`. Ensuite on enregistre quelques tests. Et a la fin, on exécute les tests et on affiche le rapport.

Utilisation avec OUnit

La fonction `registered_tests ()` renvoyant l'intégralité des tests enregistré sous forme de liste de couple `("label", function)`. Il est tout à fait possible (et simple) de lier `ppx_test` à `OUnit`. En voici un exemple :

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

```
(* To test *)
let succ x = x + 1
let pred x = x - 1

(* Tests register*)
[@@@test_register "testSucc", assert_equal 10 (succ 9)]
[@@@test_register "testPred", assert_equal 8 (pred 9)]
(* Test failure *)
[@@@test_register "testFail", assert_equal 9 (succ 9)]

(* Execute *)
[@@@test_process
  let uTests =
    List.map (fun (l,t) -> l>::t ) (registered_tests ())
  in
    run_test_tt
     ~verbose:true
    ("TestSuccAndPred">:::uTests)
]
```

Le préprocesseur n'implémente pas de création de fonction adapté à OUnit, pour rester agnostique de l'outil de tests, c'est pour ça que je dois me servir d'un `List.map`.

Conclusion et travaux futurs

Cet extension aura été avant tout une expérience avec les extensions de syntaxes, qui sont très agréables à utiliser (malgré un premier contact difficile :)). Je remercie Gasche, Companion_cube et Whitequark (et Apolly obviouzli) pour leurs conseils et leurs aides.

Dans le futur, si `ppx_test` trouve des utilisateurs, je continuerai de le maintenir.