Sous le capot du MOOC OCaml

Benjamin Canou, OCaml PRO

30 janvier 2016

Journées Francophones des Langages Applicatifs

Le MOOC OCaml

Quelques données :

- ouvert de septembre 2015 à janvier 2016 ;
- hébergé sur France Université Numérique ;
- 3705 inscrits, dont 1200 actifs ;
- 317 certifiés dont la majorité obtient un score proche de 100% ;
- étudiants dans plus de 100 pays ;
- cours et supports en anglais (sous titres français).

Pédagogie

Contenu des cours :

- programmation fonctionnelle (5 semaines) ;
- traits impératifs et langage de modules (1 semaine chaque) ;
- semaines découpées en séquences (42 séquences en tout) ;
- vidéo (8 à 15 minutes) et exercices pour chaque séquence ;

Trois types d'exercices :

- 7 questionnaires à choix multiples ;
- 48 exercices de programmation ;
- 1 projet final (au choix parmi 2).

L'équipe

Équipe Université Paris Diderot : pédagogie, cours et exercices

- Roberto Di Cosmo
- Yann Régis Gianas
- Ralf Treinen

Équipe OCaml PRO : plate-forme, exercices et correcteurs, animation

- Benjamin Canou
- Grégoire Henry
- Çağdaş Bozman
- Pierrick Couderc

Volontaires et vacataires :

- Aurélien Deharbe et la bande du LIP6
- Jean-Francois Monin pour les sous titres en français
- Béta testeurs et étudiants actifs sur le forum

Plate-forme d'exercices

Plate-forme d'exercices

Tout le MOOC peut se faire sans rien installer !

Démo en kit, à monter soi-même :

`http://try.ocamlpro.com/fun-demo/`

Plate-forme d'exercices intégrée :

- éditeur avancé avec intégration des messages ;
- coloration syntaxique et indentation forcée ;
- toplevel OCaml dans le navigateur ;
- correction automatique avec rapports détaillés.

Deux autres variantes :

- application autonome pour la conception des exercices ;
- simulateur pour valider les mises à jour ou corriger par lot.

```
1  let plus x y =
2    x + y ;;
3  let minus x y =
4    y - x ;;
5  let times x y =
6    x *  y ;;
```

Evaluate >>

Switch >>

Typecheck

Reset Template

Full-screen [+]

Check & Save

Plate-forme d'exercices

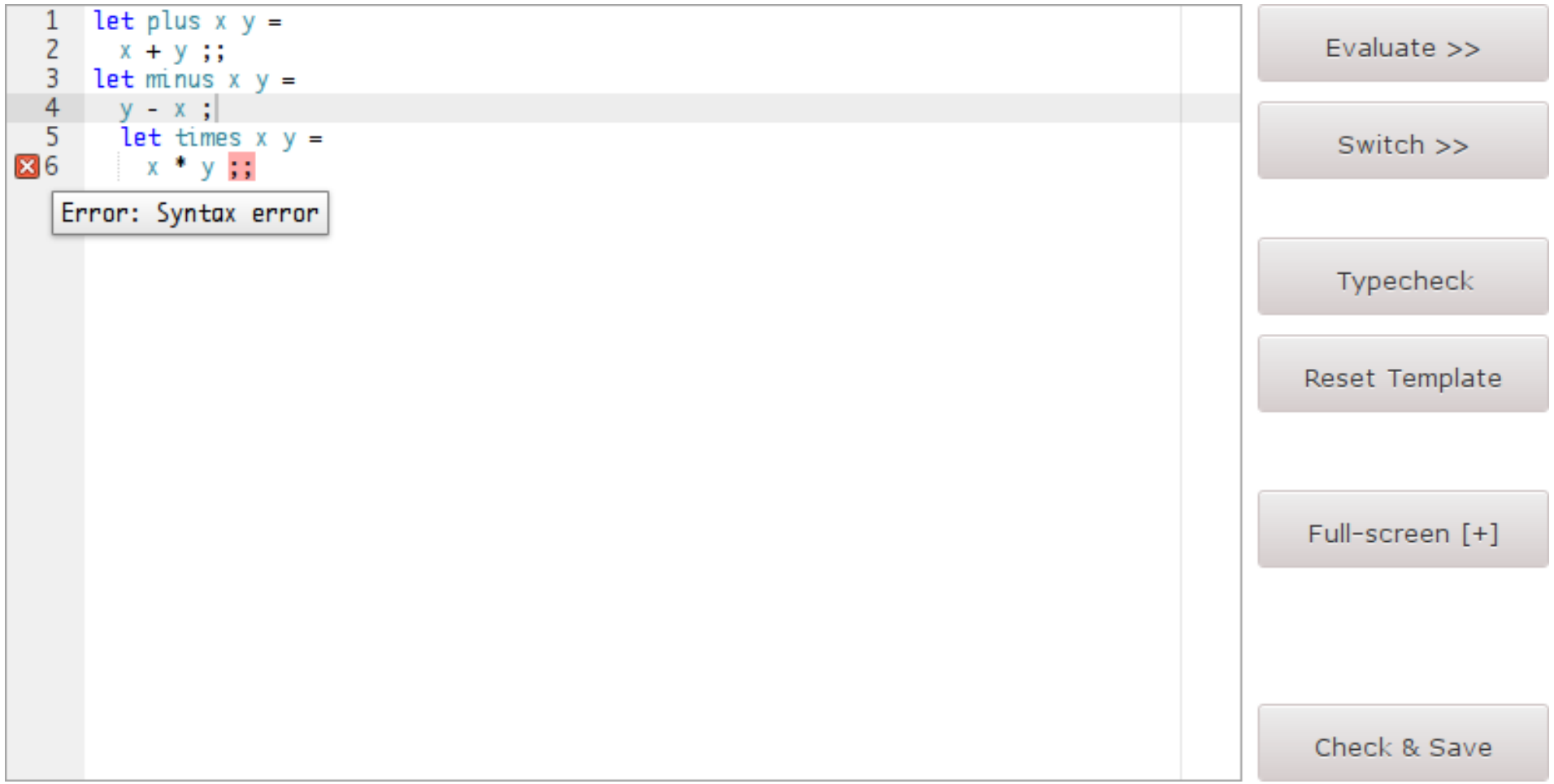

Plate-forme d'exercices

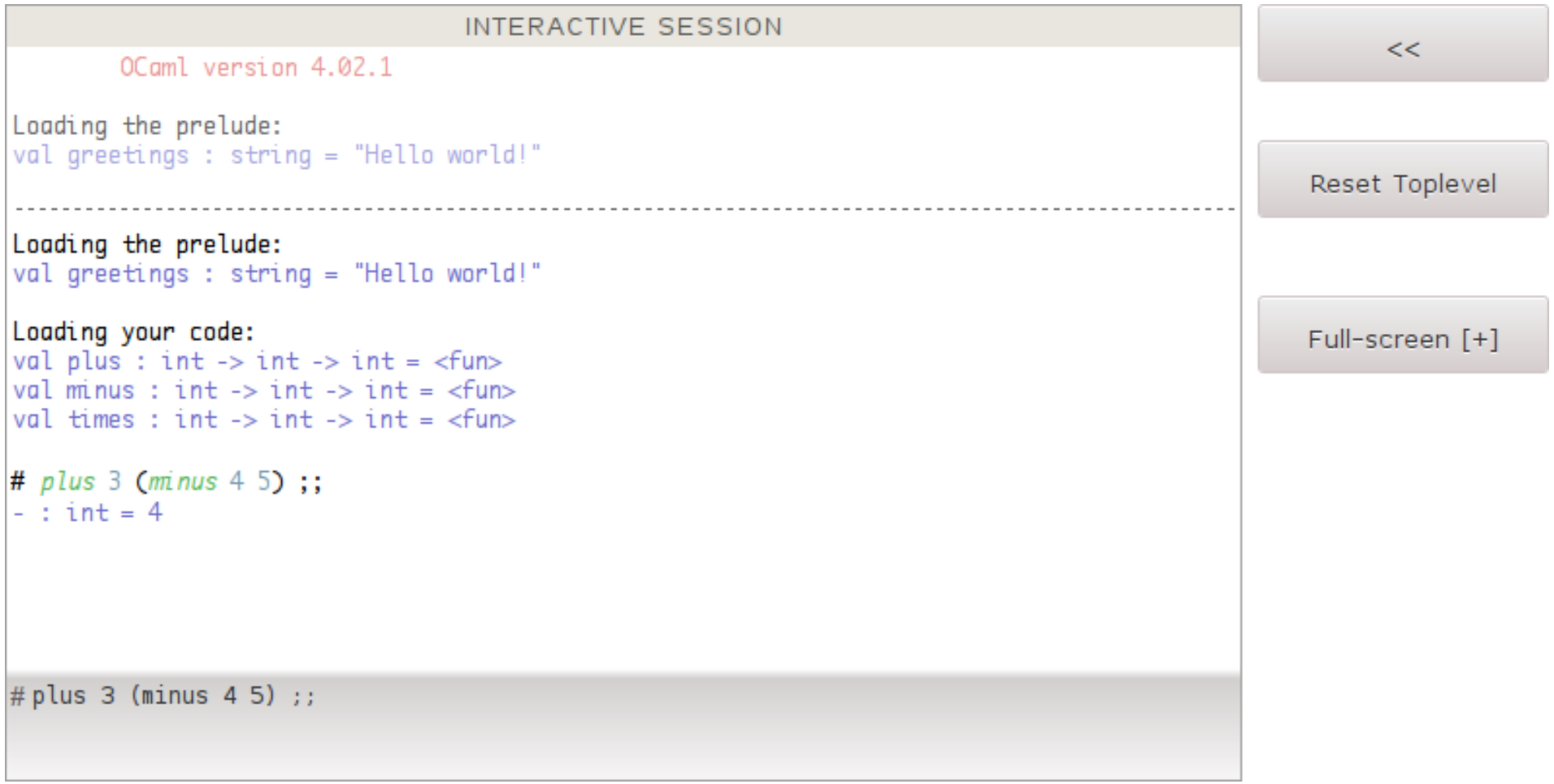

Plate-forme d'exercices

5 / 5

Exercise incomplete (click for details) 22 pts

> Function: plus — Completed, 10 pts

v Function: minus — Incomplete, 2 pts

Found minus with compatible type.

Computing minus 1 1

Correct value 0 — 1 pt

Computing minus 1 -2

Wrong value -3 — 0 pt

Computing minus 3 1

Wrong value -2 — 0 pt

Computing minus 4 4

Correct value 0 — 1 pt

Computing minus 3 -1

Wrong value -4 — 0 pt

Computing minus 2 1

Wrong value -1 — 0 pt

> Function: times — Completed, 10 pts

v Function: divide — Failed

Cannot find divide — 0 pt

Format d'un exercice

Chaque exercice est décrit par 6 fichiers :

- 3 publics :
  - descr.html : énoncé de l'exercice ;
  - prelude.ml : définitions de types et fonctions données à l'étudiant ;
  - template.ml : patron à remplir, compléter ou corriger.
- 3 cachés :
  - prepare.ml : prélude caché ;
  - solution.ml : copie parfaite utilisée comme référence ;
  - test.ml : jeu de tests qui construit le rapport.

Correction automatique

Structure de la bibliothèque de correction

Le correcteur (test.ml) :

- Reçoit une session de toplevel avec le code de l'étudiant évalué ;
- doit produire un score et un rapport HTML.

Il fait appel à :

- des primitives d'introspection sûre du code de l'étudiant ;
- un format intermédiaire de rapports ;
- des combinateurs de correction prêts à l'emploi reliant ces deux parties.

Bibliothèque de rapports

```
1 : type report = item list (* Liste d'éléments de rapport *)
2 : and item = (* Soit une ligne, soit un bloc *)
3 :   | Section of inline list * report (* Bloc avec titre *)
4 :   | Message of inline list * status (* Ligne avec score *)
5 : and status = (* Score associé à une ligne *)
6 :   | Success of int (* Résultat correct pondéré *)
7 :   | Failure | Incomplete (* Échec complet ou partiel *)
8 :   | Informative | Important (* Message *)
9 : and inline = (* Contenu texte *)
10 :   | Text of string (* Texte *)
11 :   | Code of string (* Expression OCaml *)
12 :   | IO of string (* Bloc de texte lu ou imprimé *)
13 :
14 : val result_of_report  : report -> int * bool
15 : val html_of_report  : report -> string
16 : val output_text_report  : Format.formatter -> report -> unit
17 : val output_html_report  : Format.formatter -> report -> unit
```

Primitives d'introspection

Une nouvelle forme d'expression `[%ty: τ]` est évaluée vers un témoin du type `τ ty`.

Ces témoins sont utilisés pour imprimer et générer des valeurs :

```
1 : val get_printer : 'a ty -> (formatter -> 'a -> unit)
2 : val get_sampler : 'a ty -> (unit -> 'a)
```

Mais surtout introspecter le code de l'étudiant :

```
1 : type 'a value =
2 :   | Absent
3 :   | Present of 'a
4 :   | Incompatible of string
5 : val get_value : string -> 'a ty -> 'a value
```

Lorsque le correcteur exécute `let x = get_value "x" [%ty: τ_r]` :

- on recherche `x` dans l'environement du toplevel ;
- on recherche son type effectif τ_s dans l'environnement du typeur ;
- on tente d'unifier τ_r et τ_s ;
- si tout fonctionne, on a remonté le `x` de l'étudiant dans le correcteur.

Principe général : exécuter le code de l'étudiant depuis le code de test.

```
1 : type 'a result = Ok of 'a | Error of exn
2 : val result  : (unit -> 'a) -> 'a result
3 : val exec  : (unit -> 'a) -> ('a * string * string) result
```

Exemple de calcul de rapport :

```
1 : [Message ([Text "Computing succ 3"], Informative)] @
2 : match get_value "succ" [%ty : int -> int] with
3 : | Absent -> [Message ([Text "Cannot find succ"], Failure)]
4 : | Incompatible _ -> [Message ([Text "Bad type"], Falure)]
5 : | Present succ ->
6 :   match exec (fun () -> succ 3) with
7 :   | Error _ -> [Message ([Text "Exception"], Failure)]
8 :   | Ok 4 -> [Message ([Text "Correct result"], Success 5)]
9 :   | Ok _ -> [Message ([Text "Wrong result"], Failure)]
```

On veut factoriser le code trivial :

- collection de combinateurs de test prédéfinis
  (test de fonction, variable, trait de syntaxe, etc.) ;
- (très) personnalisables via paramètres fonctionnels optionnels ;
- le comportement par défaut est celui attendu ;
- second niveau de combinateurs pour faciliter la personnalisation
  (test de valeurs, de sorties, génération des cas, etc.).

Pour se concentrer sur :

- la génération intelligente de cas de test ;
- les tests spécifiques s'il y en a.

Exemple : test d'une fonction à un paramètre.

```
1 : val test_function_1_against_solution :
2 :   ?test : 'b tester ->
3 :   ?test_stdout : io_tester -> ?test_stderr : io_tester ->
4 :   ?sampler : (unit -> 'a) ->
5 :   ?samples : int ->
6 :   ?before_reference : ('a -> unit) ->
7 :   ?before_user : ('a -> unit) ->
8 :   ?after : ('a ->
9 :             ('b * string * string) ->
10 :            ('b * string * string) -> report) ->
11 :  ~ty : ('a -> 'b) ty ->
12 :  ~name : string ->
13 :  ~tests : 'a list ->
14 :  report
```

Personnalisation des tests :

Test du résultat :

```
1 : type 'a tester = 'a ty -> 'a result -> 'a result -> report
2 : val test  : 'a tester
3 : val test_eq_ok  : ('a -> 'a -> bool) -> 'a tester
4 : val test_ignore  : 'a tester
```

Test des sorties :

```
1 : type io_tester = string -> string -> report
2 : val io_test_lines  :
3 :   ?trim : char list -> ?drop : char list ->
4 :   ?skip_empty : bool -> ?test_line : io_tester -> io_tester
```

Exemple de correcteur

Un petit exemple d'exercice complet

L'énoncé (`descr.html`) :

```
1 : <p>
2 :   Définissez deux fonctions utilisant des chaînes :
3 : </p>
4 : <ol>
5 :   <li>
6 :     <code>last</code>
7 :     donne le dernier caractère d'une chaîne ;
8 :   </li>
9 :   <li>
10 :    <code>string_of_bool</code>
11 :    convertit un booléen en chaîne à la OCaml.
12 :  </li>
13 : </ol>
```

Copie de référence (`solution.ml`) :

```
1 : let last s =
2 :   String.get s ((String.length s) - 1);;
3 : let string_of_bool b =
4 :   if b then "true" else "false";;
```

Patron (`template.ml`) :

```
1 : let last str =
2 :   "Replace by your implementation." ;;
3 : let string_of_bool truth =
4 :   "Replace by your implementation." ;;
```

Jeu de tests (`test.ml`) :

```
let sample_string () =
    [|"Ca";"Boz";"Hen"|].(Random.int 3)
  ^ [|"nou";"man";"ry"|].(Random.int 3) ;;

set_result @@ ast_sanity_check code_ast @@ fun () ->
  [ Section ([ Text "Exercise 1 :" ; Code "last" ],
             test_function_1_against_solution
               [%ty : string -> char] "last" []) ;
    Section ([ Text "Exercise 2 :" ; Code "string_of_bool" ],
             test_function_1
               [%ty : bool -> string] "string_of_bool"
               [ true, "true", "", "" ;
                 false, "false", "", ""]) ] ;;
```

Conclusion

Conclusion

Bonne nouvelle : beaucoup d'aspects testables grâce au fort lien à OCaml :

- traits impératifs : E/S, effets sur les arguments ;
- vérifications de complexité (ex. comptage d'accès tableaux) ;
- vérification de l'AST
  - pour l'apprentissage de nouveaux traits,
  - pour restreindre l'environnement ;
- modules et signatures ; etc.

Ratio $\frac{\text{taille correcteur}}{\text{taille solution}}$ raisonnable grâce aux combinateurs.

Difficulté principale : générer de bons cas de tests.

Limites de l'approche :

- définition de types concrets par l'étudiant ;
- commentaires de style (manque de retour humain).

Environ 22 000 lignes de code

Environnement des exercices.

Composant	Lignes	Type
Bibliothèque de rapports	500	`.ml`
Bibliothèque de tests	1500	`.ml`
Script de tests hors navigateur	500	`.ml` + Makefile
Problèmes	4700 1500 800 4000	`.ml` (tests) `.ml` (copies de référence) `.ml` (patrons de solution) `.html` (descriptions)
TryOCaml	2000	`.ml`
Interface graphique	1500	`.ml`

Script de description et déploiement

Composant	Lignes	Type
Bibliothèque générique	2000	`.ml` + `.mli`
Combinateurs spécifiques au MOOC OCaml	1000	`.ml`
Description du MOOC OCaml	1500	`.ml`

Sous le capot du MOOC OCaml

Merci !

Journées Francophones des Langages Applicatifs