Expérimentations pédagogiques en Learn-OCaml

Loic Sylvestre, Emmanuel Chailloux

Expérimentations pédagogiques en Learn-OCaml

Outils et méthodologie pour les traits avancés du langage

Loïc Sylvestre[1] Emmanuel Chailloux[2]

[1] Sorbonne Université, F-75005 Paris, France
loic.sylvestre@etu.sorbonne-universite.fr
[2] Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
emmanuel.chailloux@lip6.fr

Résumé

Learn-OCaml est un logiciel libre destiné à l'enseignement du langage OCaml : il offre un environnement d'exercices de programmation à correction automatisée. On présente dans cet article un support pour la séparation et la réutilisation du code de correction. Cette extension donne lieu, dans un second temps, à l'élaboration d'une bibliothèque de test visant à faciliter l'écriture des exercices, produire automatiquement des rapports de correction très précis et proposer de nouveaux schémas de correction pour les constructions avancées du langage. La mise en œuvre de ces outils, dans le cadre d'un cours de programmation à Sorbonne Université, est l'occasion de partager un premier retour d'expérience.

1 Introduction

Depuis la rentrée 2019, la plateforme Learn-OCaml est utilisée à Sorbonne Université dans le cadre d'un cours de remise à niveau en OCaml dispensé en première année de Master Informatique, spécialité Science et Technologie du Logiciel. Ce cours obligatoire s'adresse à un public de 58 étudiants, de niveaux très variés, sur un temps suffisamment court pour être en mesure de réaliser un projet par la suite. La préparation des séances de travaux pratiques pour ce cours a permis de tester les fonctionnalités de Learn-OCaml, et en particulier son environnement d'exercices à correction automatisée issu des technologies du MOOC OCaml [4]. Nous avons développé des outils supplémentaires [1] visant à faciliter l'écriture des exercices et rendre la correction automatisée plus précise et interactive.

Le code source d'un exercice en Learn-OCaml comprend un énoncé (`descr.md`), un environnement initial [2] (`prelude.ml` et `prepare.ml`) une esquisse de réponse destinée à l'étudiant (`template.ml`), la solution de l'enseignant (`solution.ml`) et un programme de correction (`test.ml`). La correction automatisée d'un exercice s'appuie sur le toplevel [2] dans lequel sont chargés successivement plusieurs sous-programmes, comme illustré ci-dessous.

```
⟨prelude.ml⟩ ;;
⟨prepare.ml⟩ ;;
module Code = ⟨le code de l'étudiant⟩ ;;
module Solution = ⟨solution.ml⟩ ;;
let code_ast = ⟨l'AST du module Code⟩ ;;
module Introspection = ... ;;
module Report = ... ;;
module Test_lib = ... ;;
⟨test.ml⟩ ;;
```

1. travail réalisé avec le soutien de l'IRILL (Initiative de Recherche et Innovation sur le Logiciel Libre).
2. `prelude.ml` est visible depuis le navigateur de l'étudiant, contrairement à `prepare.ml`.

Les modules `Introspection`, `Report` et `Test_lib` — fournis par Learn-OCaml — sont utilisables dans le fichier `test.ml` pour implanter le programme de correction. Le module `Introspection` offre un support pour manipuler de façon sûre le code de l'étudiant, potentiellement erroné, grâce à une extension de syntaxe permettant la représentation des types comme valeurs à l'exécution. Le module `Report` définit un type `Report.t` représentant les rapports de correction sous une forme arborescente [2]. Enfin, le module `Test_lib` propose des combinateurs de test génériques pour composer les rapports de correction.

La section 2 décrit le mécanisme d'introspection de Learn-OCaml. La section 3 présente `easy-check`, une bibliothèque que nous avons développée dans le but de faciliter l'écriture des exercices et produire automatiquement des rapports de correction très précis suivant une approche différente de celle qui est employée dans le module `Test_lib`. Cette approche nous a conduit à tester non seulement des déclarations, mais aussi des expressions quelconques, manipulant le code de l'étudiant, et pouvant donner lieu à des schémas de correction pour des traits de programmation avancés : généricité, abstraction, sous-typage, structures de données mutables et effets de bord. Enfin, la section 4 dresse un premier retour d'expérience suite à la mise en œuvre de cette bibliothèque à Sorbonne Université.

2 Introspection en Learn-OCaml

Lors de la phase de correction, le code de l'étudiant est disponible dans un module `Code`. Toutefois, si celui-ci est incompatible avec la signature attendue, l'accès à son contenu via la notation qualifiée `Code.f` provoquerait des erreurs de compilation. C'est pourquoi Learn-OCaml introduit un mécanisme d'introspection permettant d'explorer le code d'étudiant et l'exécuter partiellement sans compromettre la sûreté du typage du code de test [2].

2.1 Garder la trace des types à l'exécution

Le module `Ty` de Learn-OCaml définit un type `'a ty` représentant un type OCaml [3] sous forme d'arbre syntaxique, AST par la suite. De plus, une extension de syntaxe `[%ty: τ]` permet d'engendrer une valeur de type τ `ty` transportant précisément l'AST du type τ. Cette valeur `[%ty: τ]` appelée témoin du type τ offre alors une représentation de celui-ci à l'exécution [2]. Les phrases ci-dessous sont des exemples d'utilisation de cette extension de syntaxe.

```
# [%ty: int] ;;
- : int Ty.ty = Ty.Ty <abstr>
# [%ty: ?acc:int -> k:int -> int] ;;
- : (?acc:int -> k:int -> int) Ty.ty = Ty.Ty <abstr>

# [%ty: ([> `Cons of ('a * 'b) | `Nil] as 'a)] ;;
- : ([> `Cons of 'a * 'b | `Nil ] as 'a) Ty.ty = Ty.Ty <abstr>
# [%ty: (< me : 'a ; ..> as 'a)] ;;
- : (< me : 'a; .. > as 'a) Ty.ty = Ty.Ty <abstr>

# module type S = sig val id : 'a -> 'a end ;;
module type S = sig val id : 'a -> 'a end
# [%ty: (module S)] ;;
- : (module S) Ty.ty = Ty.Ty <abstr>

# [%ty: 'a] ;;
'a Ty.ty = Ty.Ty <abstr>
# [%ty: 'a -> 'b] ;;
- : ('_a -> 'b) Ty.ty = Ty.Ty <abstr>
# [%ty: '_a];;
Error: The type variable name '_a is not allowed in programs
```

3. Concrètement, `type 'a ty = Ty of repr and repr = Parsetree.core_type`.

2.2 Accès aux déclarations

Le module `Introspection` de Learn-OCaml fournit un support permettant d'accéder de façon sûre aux déclarations de l'environnement du toplevel.

```
type 'a value = Absent
              | Present of 'a
              | Incompatible of string

val get_value : string -> 'a Ty.ty -> 'a value
```

Accès à une variable La primitive `get_value` reçoit un identificateur `x` et un témoin de type `[%ty:` τ`]`. Si `x` est présent dans l'environnement du toplevel et lié à une valeur `(`v`:`τ'`)`, un test d'instantiation est réalisé entre τ et τ'. Si τ' est plus général que τ, la valeur retournée est `(Present` v'`)` avec v' physiquement égale à v ; dans le cas contraire, la valeur retournée est `(Incompatible` s`)` avec s un message d'erreur engendré par le compilateur. Enfin, si `x` est absent de l'environnement du toplevel, la valeur retournée est `Absent`. Les phrases ci-dessous illustrent ce principe.

```
# let x = 0 ;;
val x : int = 0
# get_value "x" [%ty: int] ;;
- : int value = Present 0
# get_value "x" [%ty: float] ;;
- : float value = Incompatible "Wrong type int."
# get_value "z" [%ty: int] ;;
- : int value = Absent

# let v = `N 0 ;;
val v : [> `N of int ] = `N 0
# get_value "v" [%ty: [> `B of bool | `N of int]] ;;
- : [> `B of bool | `N of int ] value = Present (`N 0)

# let id x = x;;
val id : 'a -> 'a = <fun>
# let f x = (match get_value "id" [%ty: 'a -> 'a] with
             | Present v -> assert (v == id); v
             | _ -> raise Not_found) x ;;
val f : 'a -> 'a = <fun>
```

Accès à un module De façon analogue, la primitive `get_value` permet d'extraire de l'environnement du toplevel un module à partir d'un identificateur de module et d'un témoin de type `[%ty: (module` S`)]`. Le module est retourné comme valeur et peut être désempaqueté.

```
# module type LIST = module type of List ;;
module type LIST = ...
# let map f l = (match get_value "List" [%ty: (module LIST)] with
                 | Present m -> let module L = (val m : LIST) in L.map
                 | _ -> raise Not_found) f l ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Un support pour l'écriture des tests L'introspection est directement utilisable dans les correcteurs automatiques pour extraire le code de l'étudiant depuis l'environnement de toplevel. Comme vu précédemment, ce mécanisme s'adapte aux constructions avancées du langage — polymorphisme paramétrique, modules, objets, sous-typage — et peut naturellement servir de support à des outils de plus haut niveau, à l'instar du module `Test_lib` de Learn-OCaml.

3 Outils et méthodologie pour l'écriture des exercices

On présente dans cette section une extension de Learn-OCaml permettant la séparation du code de correction. On s'appuie ensuite sur ce mécanisme pour developper une bibliothèque de test — qui manipule implicitement les modules `Introspection`, `Report` et `Test_lib` de Learn-OCaml — dans le but de simplifier l'écriture des exercices et annoter précisément les réponses d'étudiants. À noter que cette bibliothèque s'inspire des correcteurs automatiques de François Pottier [4] provenant du corpus d'exercices libres pour Learn-OCaml [1].

3.1 Extension du code source

Actuellement, le code de correction d'un exercice pour Learn-OCaml doit être défini dans un unique fichier `test.ml`. On ne peut pas partager de code entre les exercices, et cela les rend difficiles à écrire et à modifier. C'est pourquoi nous proposons d'introduire une courte extension dans le code source de Learn-OCaml — de l'ordre de 80 lignes de code — permettant la séparation et la réutilisation du code de correction. Cette extension offre la possibilité d'ajouter, dans le dossier d'exercices, un fichier optionnel `depend.txt`. Ce fichier indique à Learn-OCaml des noms de dépendances `.ml` et `.mli` à charger dynamiquement dans le toplevel pendant la phase de correction, avant que le fichier `test.ml` ne soit évalué. Ainsi, le code de correction dans `test.ml` a non seulement accès aux modules de Learn-OCaml, mais aussi à des modules de dépendances déclarés par l'enseignant.

3.2 Une nouvelle bibliothèque de test

L'extension présentée ci-dessus a motivé le développement d'`easy-check`, une bibliothèque de test utilisable comme dépendance pour implanter les exercices. Cette bibliothèque comprend environ 1000 lignes de code réparties en 10 modules parmi lesquels `Get`, `Result`, `Assume`, `Check` et `Autotest`, succinctement décris par la suite.

Des annotations concises La stratégie mise en œuvre dans `easy-check` pour annoter les programmes d'étudiants repose sur trois modèles de rapports de correction :

- $\mathbf{I}_x$ indique que la déclaration x est absente ou incompatible avec la signature attendue ;
- $\mathbf{F}_e$ donne un contre-exemple justifiant que l'expression `Code.(`e`)` est incorrecte ;
- $\mathbf{T}_e$ indique que l'expression `Code.(`e`)` semble correcte.

Structure d'un fichier `test.ml` Un fichier `test.ml` basé sur `easy-check` contient en général un code de correction de la forme `(Result.set [`q_1`;`q_2`;...])` où chaque q_i est une fonction de type `unit -> Report.t` qui peut signaler une exception `Fail of Report.t`. Le combinateur `Result.set` calcule chaque expression `(try` q_i `() with Fail` r `->` r`)`, fusionne les rapports de correction obtenus, puis renvoie le résultat à l'étudiant.

Accès au code de l'étudiant L'expression `(Get.value "x" [%ty:` τ`])` extrait la variable `(Code.x :` τ`)` de l'environnement du toplevel, grâce à `Introspection.get_value`. Si `Code.x` est absente ou incompatible avec le type τ, alors l'exception `(Fail` $\mathbf{I}_x$`)` est signalée. De manière analogue, `(Get.mod_value "M" [%ty: (module` S`)])` extrait le module `M` de l'environnement du toplevel et le restitue sous une forme empaquetée. Enfin, `(Get.code [%ty: (module` S`)])` est une abréviation pour `(Get.mod_value "Code" [%ty: (module` S`)])`.

4. https://github.com/ocaml-sf/learn-ocaml-corpus/blob/master/exercises/merge_sort/test.ml

Vérifier le type d'une déclaration L'expression `(Assume.compatible "x" [%ty: τ])` rend `()` si `Code.x` est bien présente dans le toplevel et si son type est plus général que τ. Sinon, l'exception `(Fail I`$_x$`)` est signalée.

Tester une déclaration L'expression `(Check.name`$_n$ `"f" [%ty:` τ_1 `->` ... `->` τ_n `->` τ_{res}`] ℓ)` accède via l'introspection à $f_{\texttt{code}}$ = `(Code.f :` τ_1 `->` ... `->` τ_n `->` τ_{res}`)`, puis évalue l'expression `(`$f_{\texttt{code}}$ a_1 ... a_n`)` pour chaque n-uplet `(`a_1`,...,`a_n`)` pris dans la liste ℓ. Chaque valeur retournée est comparée vis-à-vis de `(Solution.f` a_1 ... a_n`)`. Si un contre-exemple est trouvé, alors l'exception `(Fail F`$_f$`)` est signalée. Sinon, le rapport `T`$_f$ est retourné. La comparaison entre `Code.f` et `Solution.f` est personnalisable via différents arguments optionnels de `Check.name`$_n$. La valeur de l'argument `~equal` (respectivement `~equal_exn`) compare [5] les valeurs retournées (respectivement les exceptions signalées) par `Code.f` et `Solution.f`. Par ailleurs, la valeur de l'argument `~equal_stdout` (respectivement `~equal_stderr`) compare [6] les affichages produits par `Code.f` et `Solution.f` sur la sortie standard (respectivement la sortie d'erreur). Ainsi, le fichier `test.ml` suivant teste une fonction d'affichage (`q1`), une fonction d'arité 2 (`q2`) et une fonction polymorphe (`q3`). On constate que, pour tester une fonction polymorphe, il suffit de vérifier son type, puis tester une instance de la fonction.

```
let q1 () = Check.name1 "print_int_list" [%ty: int list -> unit]
                        ~equal_stdout:(=) [[];[0];[1;2];...] ;;
let q2 () = Check.name2 "pow" [%ty: int -> int -> int] [(2,8);(4,3);...] ;;
let q3 () = Assume.compatible "sort" [%ty: 'a list -> 'a list];
            Check.name1 "sort" [%ty: int list -> int list] [[2;3;1];...] ;;
let () = Result.set [q1;q2;q3] ;;
```

Tester une expression En Learn-OCaml, on est souvent amené à vouloir tester, non seulement des déclarations, mais aussi des expressions manipulant le code de l'étudiant. Supposons par exemple que l'on souhaite tester un opérateur `(|>) : 'a -> ('a -> 'b) -> 'b` défini par l'étudiant. Une solution naturelle consiste à tester une expression de la forme `Code.(`e `|>` f`)`. Toutefois, à notre connaissance, il est difficile d'écrire un tel code de correction avec le module `Test_lib` de Learn-OCaml. C'est pourquoi `easy-check` propose des combinateurs supplémentaires, qui généralisent `Check.name`$_n$ dans le but de tester des expressions quelconques.

Pour cela, nous introduisons d'abord l'extension de syntaxe [7] `[%code` e`]` qui expanse le triplet `(Code.(`e`), Solution.(`e`),` e_p`)` avec e_p une chaîne de caractères qui représente l'expression e en syntaxe OCaml [8] en vue de l'afficher dans le rapport de correction.

L'expression `(Check.expr`$_n$ `[%code` e`] [%ty:` τ_1 `->` ... `->` τ_n `->` τ_{res}`] ℓ)` évalue l'expression `((Code.(`e`)` a_1 ... a_n`) :` τ_{res}`)` pour chaque n-uplet `((`a_1`,...,`a_n`) : (`τ_1 `*` ... `*` τ_n`))` pris dans la liste ℓ. Chaque valeur retournée est comparée vis-à-vis de `(Solution.(`e`)` a_1 ... a_n`)`. Si un contre-exemple est trouvé, alors l'exception `(Fail F`$_e$`)` est signalée. Sinon, le rapport `T`$_e$ est retourné. Le fichier `test.ml` suivant teste l'opérateur `Code.(|>)` comme évoqué plus haut.

```
module type CODE = sig val (|>) : 'a -> ('a -> 'b) -> 'b end ;;
let q1 () =
  let m = Get.code [%ty: (module CODE)] in
  let module Code = (val m : CODE) in
  Check.expr1 [%code (fun n -> n + 1 |> string_of_int)]
              [%ty: int -> string] [1;2;3;4] ;;
let () = Result.set [q1] ;;
```

5. Par défaut, c'est l'égalité structurelle d'OCaml qui est utilisée.
6. Si non précisé, les affichages sont ignorés.
7. Cette extension ppx nécessite un ajout, d'environ 20 lignes, dans le code source de Learn-OCaml.
8. Cette chaîne de caractères est produite directement à partir de l'AST de l'expression e, au moyen de la fonction `Pprintast.string_of_expression` de la distribution OCaml.

À noter que l'extension de syntaxe `[%code e]` manipule le module `Code` de façon non sûre. En particulier ci-dessus, la valeur `[%code (fun n -> n + 1 |> string_of_int)]` n'est correcte que si `Code.(|>)` est compatible avec le type `int -> (int -> string) -> string`. En effet, si la réponse de l'étudiant n'est pas compatible avec la signature attendue, une erreur de compilation surviendrait lors du chargement de fichier `test.ml` dans le toplevel. Il est donc indispensable de redéfinir localement un module `Code` sûr grâce à `Get.code`. Ce recours à l'introspection peut sembler laborieux au premier abord. Pour autant, les rapports de correction engendrés par `Check.expr`$_n$ sont alors très précis, concis et plus expressifs :

```
The following expression:                                    1 pt
  (fun n  -> (n + 1) |> string_of_int)
seems correct.
```

Engendrer les valeurs pour les tests L'argument optionnel `~testers` de `Check.name`$_n$ et `Check.expr`$_n$ attend une liste de valeurs de type `(`τ_1`,`$\tau_2`) tester`. Le rôle d'un testeur est d'engendrer des valeurs x_i de type τ_1 puis leur appliquer la fonction candidate f `:` τ_1 `->` τ_2 et une fonction solution f'. Les deux résultats sont alors comparés à l'aide d'une fonction eq `:` τ_2 `->` τ_2 `-> bool` afin de mettre en évidence un éventuel contre-exemple $\langle x_0, v, v' \rangle$ tel que `(`f x_0`)` s'évalue en v, `(`f' x_0`)` en v' et `(`eq v v'`)` en $false$. Le module `Autotest` d'`easy-check` fournit un testeur paramétré `Autotest.tester` qui, étant donnée une valeur de type τ_1 `sampler = unit ->` τ_1, rend un testeur de type `(`τ_1`,'a) tester`. Ce module propose également des utilitaires [9] pour composer des sampleurs. `(fun` x `->` `Autotest.choose` f g x`)`, par exemple, engendre soit `(`f x`)`, soit `(`g x`)`. `(Autotest.oneof [`x_1`;...;`x_n`])` engendre une valeur prise parmi x_1, ..., x_n. `(Autotest.tree ~depth:`d f `(fun t ->` g `t))` engendre un arbre de profondeur `(`d `())`, de feuilles `(`f `())` et de nœuds `(`g `t)` avec `(t ())` un sous-arbre.

Un exemple d'exercice : réduction de λ-termes On propose en exercice la définition d'une fonction `reduce` qui réduit un terme du λ-calcul pur suivant une stratégie d'appel par valeur. On représente les termes par le type `t = Var of var | Lam of (var * t) | App of (t * t) and var = string`. Voici le code de correction (fichier `test.ml`) :

```
1  let lambda_term v =
2    let lam t = Lam (v (),t ())
3    and app t = App (t (),t ()) in
4    Autotest.(tree ~depth:(nat 5) (fun () -> Var (v ())) (choose lam app));;
5  let q1 () =
6    Check.name1 "reduce" [%ty: t -> t]
7                ~testers: [Autotest.(tester (lambda_term (oneof ["x";"y";"z"])))]
8                [] ;;
9  let () = Result.set [q1];;
```

On teste la fonction `reduce` avec `Check.name1` qui accède de façon sûre à la variable `Code.reduce` et l'applique à des termes engendrés automatiquement par `Autotest.tree` pour mettre en évidence un contre-exemple. Si l'étudiant propose une solution incorrecte, un rapport annoté lui indiquera clairement l'origine de son erreur ; ici une capture de variable :

```
The following expression:                                    0 pt
  reduce (App (Lam ("z", App (Lam ("z", Var "z"), Var "x")), Var "y"))

... produces the following value:
  (Var "y")
This is incorrect. Producing the following value is correct:
  (Var "x")
```

9. Les sampleurs d'`Autotest` suivent le modèle du module `Test_lib` de Learn-OCaml, qui fournit « des combinateurs de génération aléatoire pour les principaux types prédéfinis [...] avec différents paramètres permettant de biaiser la distribution » [2]. À l'avenir, on pourrait reprendre les travaux de génération aléatoire de structures de données [3] et ainsi s'assurer que la génération est uniforme.

Modules et types abstraits Une problématique récurrente en Learn-OCaml est la correction des fonctions manipulant des types définis par l'étudiant : « Soit un module avec un type abstrait `t` et deux fonctions `to_string : t -> string` et `from_string : string -> t`. Tester ces deux fonctions n'est pas possible puisqu'elles utilisent des valeurs d'un type `t` qui peut être implémentés différemment par l'étudiant et par l'enseignant. Une solution consiste à simplement tester `(fun s -> to_string (from_string s))` » [4]. Le fichier `test.ml` ci-dessous illustre la mise en œuvre de cette technique avec `easy-check`. On redéfinit un module `Code` sûr grâce à `Get.code`, puis à l'aide de notre extension de syntaxe `[%code` e`]`, on teste l'expression souhaitée en l'appliquant à des chaînes de caractères engendrées par `Autotest.string`.

```
module type CODE = sig
  module M : sig
    type t
    val from_string : string -> t
    val to_string : t -> string
  end
end ;;
let q1 () = let m = Get.code [%ty: (module CODE)] in
            let module Code = (val m : CODE) in
            Check.expr1 [%code (fun s -> M.to_string (M.from_string s))]
                        [%ty: string -> string]
                        ~testers: [ Autotest.(tester string) ]
                        [""] ;;
let () = Result.set [q1] ;;
```

3.3 Méthodologie pour l'écriture des exercices

Le support que nous avons proposé pour la séparation du code de correction, au moyen d'une courte extension dans le code source de Learn-OCaml, rend possible une certaine discipline de programmation. Un point clé est le fait de pouvoir s'appuyer sur des fichiers de dépendances qui, puisqu'ils sont chargés dynamiquement dans le toplevel, peuvent être modifiés sans devoir recompiler Learn-OCaml. Des barrières d'abstraction peuvent alors être mises en place, masquant les primitives d'introspection et la représentation concrète des rapports de correction pour se focaliser sur les seuls aspects pédagogiques. L'expérimentation est alors facilitée et encouragée, à l'instar de notre bibliothèque de test, qui annote avec précision la réponse de l'étudiant, de manière automatique, par la mise en évidence d'un éventuel contre-exemple indiquant clairement ce qui est incorrect et ce qui est attendu. La continuation de cette approche consiste à tester des fragments de programmes, autrement dit des expressions, au moyen d'une extension de syntaxe `[%code` e`]`. On en déduit de nouveaux schémas de correction pour les constructions avancées du langage, telles que les modules paramétrés ou les objets.

trait du langage	signature du module `Code`	expression à tester
module	`sig module M : S end`	`[%code (let open M in` e`)]`

foncteur	`sig module F :`
	`  functor (X:S1) -> S2`
	`end`
		`[%code`
		`  (let module M = F(V) in`
		`  let open M in` e`)]`

opérateur	`sig val (!) =` τ_1 `->` τ_2 `end`	`[%code (!)]`
appel de méthode	`sig val o : < m :` τ `> end`	`[%code (o#m)]`

classe	`sig class ['a] c =`
	`  object method m :` τ `end`
	`end`
		`[%code ((new c)#m)]`

classe abstraite	`sig class virtual c = object`
	`  method virtual m1 :` τ_1 `end`
	`  method m2 :` τ_2 `end`
	`end`
		`[%code`
		`(let o = object inherit c`
		`  method m1 =` v `end in`
		`  e)]`

4 Conclusion

Cet article présente `easy-check`, la bibliothèque que nous avons développée pour faciliter l'écriture d'exercices à correction automatisée en Learn-OCaml. Cet outil a été mis en œuvre dans l'UE « Ouverture » dispensée à l'ensemble des étudiants de M1 Informatique spécialité Science et Technologie du Logiciel à Sorbonne Université. Au regard de la diversité des profils d'étudiants, il apparaît que Learn-OCaml permet à chacun de progresser à son propre rythme, en choisissant les exercices adaptés parmi des séries d'exercices de difficultés variées. On a de plus constaté que Learn-OCaml pousse les étudiants à travailler de manière plus autonome, tout en interagissant individuellement avec l'enseignant, ce qui est propice à un apprentissage efficace en séances de travaux pratiques comme cela avait été formulé aux JFLA l'an passé [1]. Enfin, l'accès direct à l'environnement via un navigateur Web est un gain de temps précieux, car cela évite l'installation d'un environnement OCaml classique pendant la séance de travaux pratiques [5]. C'est pourquoi on envisage d'adopter Learn-OCaml dès le semestre prochain. On pense en particulier au nouveau cours de L2 « Programmation fonctionnelle », et à celui de L3 dont sont tirés la plupart des exercices de cette expérimentation.

Des correcteurs automatiques ont déjà été utilisés à Sorbonne Université ; en particulier l'environnement polyglotte CodeGradX [6] développé par Christian Queinnec, pour des examens de Licence en OCaml et Java. On remarquait que, plus les annotations automatiques sur les copies étaient précises, plus la correction était acceptée, car les étudiants lisaient vraiment ces annotations et posaient alors des questions pertinentes. L'environnement MrPython — utilisé pour le cours d'initiation à la programmation en première année de Licence à Sorbonne Université — propose par ailleurs un cadre pour évaluer les tests effectués par les étudiants. Une démarche similaire a été employée dans un cours de programmation en OCaml à l'Université McGill : demander aux étudiants d'écrire leurs propres tests et utiliser l'infrastructure de correction automatique de Learn-OCaml pour évaluer leur pertinence [5].

Plusieurs projets collaboratifs s'appuient sur Learn-OCaml [1] pour promouvoir l'enseignement du langage OCaml. Il s'agit maintenant de pouvoir les combiner.

Références

[1] Çağdaş Bozman, Benjamin Canou, Roberto Di Cosmo, Pierrick Couderc, Louis Gesbert, Grégoire Henry, Fabrice le Fessant, Michel Mauny, Carine Morel et Loïc Peyrot : Learn-OCaml : un assistant à l'enseignement d'OCaml. *In Trentièmes Journées Francophones des Langages Applicatifs (JFLA'19)*, janvier 2019.

[2] Benjamin Canou, Çağdaş Bozman et Grégoire Henry : Sous le capot du MOOC OCaml. *In Vingt-septièmes Journées Francophones des Langages Applicatifs (JFLA'16)*, janvier 2016.

[3] Benjamin Canou et Alexis Darrasse : Fast and sound random generation for automated testing and benchmarking in Objective Caml. *In ACM SIGPLAN Workshop on ML*, pages 61–70, 2009.

[4] Benjamin Canou, Roberto Di Cosmo et Grégoire Henry : Scaling up functional programming education : under the hood of the OCaml MOOC. *In ACM SIGPLAN International Conference on Functional Programming (ICFP'17)*, pages 1–25, août 2017.

[5] Aliya Hameer et Brigitte Pientka : Teaching the art of functional programming using automated grading (experience report). *In ACM SIGPLAN International Conference on Functional Programming (ICFP'19)*, août 2019.

[6] Christian Queinnec : An Infrastructure for Mechanised Grading. *In 2nd International Conference on Computer Supported Education (CSEDU'10)*, pages 37–45, avril 2010.