Sous le capot du MOOC OCaml

Benjamin Canou[1] & Çağdaş Bozman[1] & Grégoire Henry[1]

1: OCamlPro SAS, France,
benjamin@ocamlpro.com, cagdas@ocamlpro.com, gregoire@ocamlpro.com

Résumé

Dans cet article, nous présentons les éléments techniques du MOOC OCaml. Dispensé sur la plate-forme France Université Numérique fin 2015, ce cours en ligne a regroupé 3553 étudiants. Nous présentons l'environnement d'exercices de programmation en ligne, qui comprend un éditeur intelligent avec intégration des messages d'erreurs et indentation forcée, un environnement interactif, ainsi qu'un correcteur automatique permettant à l'étudiant de mettre au point sa solution progressivement à l'aide de rapports détaillés. L'ensemble est exécuté dans le navigateur des étudiants; le serveur est utilisé uniquement pour la sauvegarde des solutions et des notes. Enfin, nous présentons la bibliothèque OCaml utilisée pour décrire le MOOC et le déployer sur la plate-forme de production, avec des garanties statiques de conformité. Cet article ne discute que très brièvement du contenu pédagogique, réalisé par Roberto Di Cosmo, Yann Régis Gianas et Ralf Treinen de l'Université Paris Diderot.

1 Introduction

Le MOOC *introduction à la programmation fonctionnelle en OCaml*, ou MOOC OCaml [5] pour les intimes, s'est déroulé du 19 octobre au 4 décembre 2015 sur la plate-forme France Université Numérique. Il y avait 2000 inscrits au démarrage du MOOC, et 1500 autres les ont rejoint en cours de route, dans plus de 110 pays.

Le contenu et les supports pédagogiques ont été préparés par Roberto Di Cosmo, Yann Régis Gianas et Ralf Treinen de l'Université Paris Diderot. Le support technique présenté ici a été fourni par les auteurs de l'article pour OCamlPro [1]. L'animation pédagogique, composante fondamentale du succès d'un MOOC a été faite par ces deux équipes, assistées d'Aurélien Deharbe de l'Université Pierre et Marie Curie. Quelques bénévoles ont participé à la relecture des supports, au sous-titrage des vidéos et à la traduction de ces derniers. Sans oublier un petit nombre de béta-testeurs.

Le cours s'est déroulé sur 6 semaines, dont 4 dédiées à la programmation fonctionnelle typée, une à la programmation impérative, et une dernière au langage de modules. Chaque semaine est découpée en 4 à 6 séquences thématiques. Chaque séquence comporte un court texte d'introduction, une vidéo, quelques hyperliens et ressources à télécharger, et une à deux pages d'exercices.

L'environnement que nous détaillons dans cet article fournit de façon inédite un environnement simple d'accès pour les exercices de programmation, ainsi qu'un mécanisme d'évaluation automatique qui aide l'étudiant à progresser. Contrairement au MOOC Scala [6], ou aux MOOC Python [2] [3], il fonctionne entièrement dans le navigateur, sans aucune installation de la part de l'étudiant, ni serveur de notation. Cette plate-forme d'exercices est un élément clef du succès de ce MOOC. Espérons que ce soit un pas important dans la diffusion d'une technologie à laquelle nous tenons.

La section 2 présentera rapidement la plate-forme France Université Numérique et le mode de développement d'un MOOC. La section 3 présentera la plate-forme d'exercices et son intégration

1. Travail financé pour partie par le pôle SYSTEMATIC PARIS-REGION pour le projet UCF et soutenu par l'IRILL.
2. https://www.edx.org/course/introduction-computer-science-mitx-6-00-1x-6
3. https://www.france-universite-numerique-mooc.fr/courses/inria/41001S02/session02/about

à FUN. Puis nous détaillerons dans la section 4 la bibliothèque de correction automatique. Enfin, nous expliquerons comment nous avons aussi utilisé OCaml pour automatiser le développement et le déploiement du MOOC en section 5.

2 France Université Numérique et Open edX

Le MOOC OCaml est dispensé sur la plate-forme France Université Numérique[4] (FUN), qui instancie les logiciels d'édition et de déploiement de MOOC Open edX[5].

Quelques mots sur Open edX Open edX est le moteur qui fait tourner edX.org, une des plate-formes de MOOC anglophones majeures. Développé à l'initiative du MIT et d'Harvard, il est maintenant distribué en Open Source par une organisation à but non lucratif. Aujourd'hui, edX.org propose plus de 500 cours par plus de 80 institutions. Plusieurs autres instances d'Open edX de grande ampleur sont ouvertes au public, dont France Université Numérique.

Pour l'administrateur, Open edX est un système complexe. L'application Web proprement dite est écrite en Python (Django et nombreux paquets). Pour améliorer le temps de réponse, un serveur de fichiers statiques sert de cache (Apache). Le stockage repose sur une table de hachage distribuée couplée à un moteur de recherche dédié (MondoDB, Elastic Search). Les installations recevant un grand nombre d'étudiants doivent mettre en place un équilibrage de charge (Nginx). Certains modules permettant l'exécution de code python provenant des étudiants sur le serveur, cela nécessite aussi une brique d'isolation (Docker).

Pour les étudiants et enseignants, Open edX est simple d'utilisation. Il repose d'une part sur Open edX studio, l'éditeur de contenu en ligne pour les auteurs, et d'autre part sur Open edX LMS, la visionneuse accessible aux étudiants pour suivre les cours. Les deux partagent un format commun qui décrit le contenu pédagogique et administratif (dates, notes) des cours. Ce format est volontairement contraignant et hiérarchisé, de façon à forcer l'homogénéité des cours.

Quelques mots sur FUN Fondé en 2013 à l'initiative du ministère de la recherche et de l'enseignement supérieur, FUN est issu d'efforts d'INRIA, du CINES, et de RENATER. FUN déploie deux instances du logiciel Open edX.

L'instance principale est celle ouverte public. Il est déconseillé de l'utiliser pour concevoir ou mettre à jour le MOOC, seule la visionneuse doit être utilisée. La procédure recommandée est de travailler sur l'éditeur de l'instance secondaire, le bac à sable. Ce dernier permet de concevoir le cours en collaboration et de le mettre à jour durant la période d'ouverture. Pour la mise en ligne, la procédure est d'exporter le cours sous forme d'archive Open edX, et de le réimporter sur l'éditeur de production.

La plate-forme remporte un succès certain, avec plus de 150 cours dispensés dans des matières diverses telles que l'informatique théorique, les langues étrangères ou la cuisine, par de nombreuses institutions, le plus souvent en français. Au final, l'ensemble est plutôt pratique pour un cours à la structure simple avec vidéos, exercices de type QCM, et quelques pages de texte riche.

Nous verrons par la suite que pour notre usage moins conventionnel, nous avons dû utiliser une approche alternative afin d'automatiser au maximum le déploiement et la mise à jour, sans avoir à passer par l'éditeur et le mécanisme d'export et réimport.

3 L'environnement des exercices

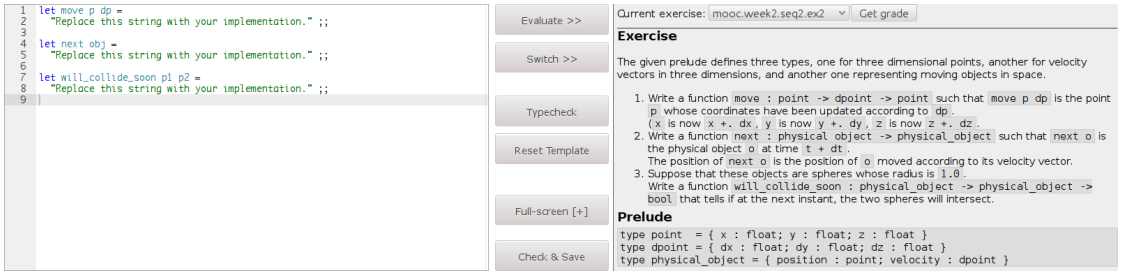
Notre rôle initial dans la préparation de ce MOOC était de concevoir et intégrer dans Open edX un composant permettant aux étudiants de faire leurs exercices de programmation OCaml, et de concevoir les correcteurs automatiques.

4. https://www.france-universite-numerique.fr/
5. https://www.fun-mooc.fr/

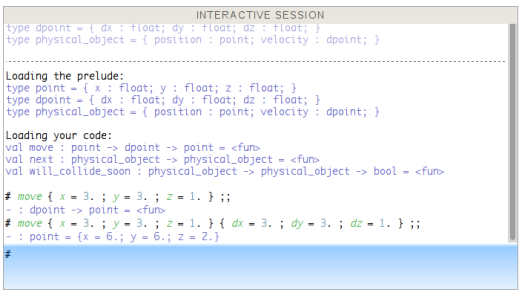
Application autonome Nous avons commencé par concevoir une plate-forme d'exercices autonome, avant de l'intégrer à Open edX. Un aperçu des différentes vues de cette application est donné figure 1.

Tirant parti de notre expérience dans le domaine, dont un des résultats les plus connus est TryOCaml[6], nous avons choisi de réaliser une solution fonctionnant entièrement dans le navigateur. Cela signifie rien à installer pour l'étudiant, mais aussi pas de serveur actif, ou en option uniquement pour le stockage.

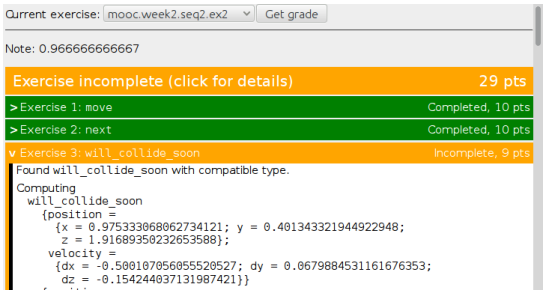
L'application fournit une liste déroulante des exercices. Lorsqu'un exercice est choisi, il montre un panneau avec son énoncé et initialise un éditeur avec un patron de solution. L'étudiant peut à tout moment passer à une session de toplevel pour tester sa solution manuellement. Il peut aussi soumettre sa solution à un correcteur, qui lance un jeu de tests et produit un rapport complet.



(a) L'étudiant est accueilli avec un énoncé et un éditeur contenant un patron de solution.



(b) Il peut expérimenter dans un toplevel OCaml. Celui-ci vient masquer temporairement l'éditeur.



(c) Une fois satisfait, il lance le test, et obtient un rapport de correction interactif.

FIGURE 1 – Les différents états de la plate-forme d'exercices autonome

3.1 Architecture

Une fois construite, l'application peut se déployer facilement. Un simple serveur HTTP peut servir les quelques fichiers qui la composent : un squelette HTML et quelques images ; un programme principal et quelques bibliothèques auxiliaires JavaScript ; et des fichiers de description d'exercices comme ils seront présentés à la section 4. Les fichiers des exercices sont (faiblement) encodés, afin de ne pas tomber sur la solution par erreur, et pour dissuader les tricheurs peu courageux.

Bien entendu, le programme JavaScript est issu de la compilation d'un programme principal en OCaml. Celui-ci implante trois fonctionnalités principales : l'interface graphique, l'évaluation de code OCaml et la construction de rapports de tests.

Le test et la fabrication de rapports sont des développements nouveaux, qui seront décrits dans cet article. L'évaluation de code OCaml repose sur le toplevel, maintenant inclus dans la distribution de `js_of_ocaml` [9], qui embarque le compilateur OCaml et `js_of_ocaml`, tous deux compilés en JavaScript, pour compiler à la volée les phrases envoyées au toplevel vers JavaScript et les évaluer dynamiquement. L'interface utilisateur améliorée du toplevel est issue de TryOCaml.

6. `https://try.ocamlpro.com/`

Nous avons développé pour l'occasion une version où l'évaluateur et l'interface sont dans des *Workers*[7]. Cela permet de ne pas bloquer le navigateur en cas de boucle infinie ou d'affichage fou. L'interface affiche un message demandant à l'étudiant s'il est d'accord pour tuer l'interprète ou masquer sa sortie. Ainsi, les étudiants peuvent lancer leurs tests en confiance sans craindre que leur navigateur ne s'écroule. Il serait intéressant d'étudier d'autres approches pour obtenir les mêmes garanties, comme l'utilisation d'une machine virtuelle [1] ou la récriture en forme CPS. Nous avons simplement choisi l'approche la plus simple à mettre en œuvre dans le temps imparti.

L'éditeur est basé sur la bibliothèque JavaScript ACE[8]. C'est un éditeur de code générique, que nous avons configuré pour avoir la coloration syntaxique, et dans lequel nous avons branché l'outil `ocp-indent`. Le code de l'étudiant est immédiatement et automatiquement indenté, sans possibilité de reprise manuelle. Ainsi, l'étudiant intègre le style de mise en forme standard sans y penser, et peut voir tout de suite certaines erreurs, comme les conditionnelles mal imbriquées.

3.2 Intégration avec Open edX

La figure 2 donne un aperçu de l'application quand elle est intégrée à Open edX. La différence principale est que les parties de description et de rapport sont intégrées comme des composants edX, et que la sauvegarde et le stockage se font en utilisant l'API fournie pour les composants edX d'exercices JavaScript personnalisés.

4 La correction automatique

Cette section décrit notre principale contribution, le cœur technique de cet article. Tout d'abord, nous décrivons le format d'exercices et son interprétation par la plate-forme. Ensuite, nous présentons le mécanisme d'introspection utilisé pour exécuter le code de l'étudiant sans compromettre la sûreté du typage du code de test. Nous décrivons enfin le langage intermédiaire des rapports de correction, et notre bibliothèque de test. Pour finir, Nous donnons un aperçu des correcteurs d'exercices.

4.1 Format de description d'un exercice

Chaque exercice est décrit par six fichiers, dont trois sont visibles par l'étudiant et trois sont cachés.

- `descr.html` : énoncé de l'exercice au format HTML ; usuellement, quelques paragraphes de présentation suivis d'une liste numérotée de questions ;
- `prelude.ml` : définitions de types et fonctions données à l'étudiant qu'il ne peut pas modifier ;
- `template.ml` : patron que l'étudiant doit remplir, compléter ou corriger, suivant l'exercice ;
- `prepare.ml` : prélude caché à l'étudiant, il sert à donner des fonctions à l'étudiant sans en montrer l'implantation, à initialiser des variables cachées ou encore à redéfinir des fonctions standard ;
- `solution.ml` : copie parfaite utilisée comme référence pour les tests et calculer le score maximum ;
- `test.ml` : le jeu de tests qui doit construire et retourner le rapport de correction.

Ces fichiers sont traités de la façon suivante.

1. On évalue le contenu de `prelude.ml` puis de `prepare.ml` dans une session fraîche de toplevel ;
2. la copie est évaluée dans un module `Code`, son arbre de syntaxe lié dans une variable `code_ast` ;
3. la copie de référence est évaluée, dans un module `Solution` ;
4. l'environnement est enrichi avec des primitives d'introspection (décrites plus tard, section 4.2) ;
5. une référence `result: (bool * float * string) ref` est ajoutée, pour stocker les informations demandées par Open edX (exercice terminé, pourcentage réalisé, commentaire en HTML) ;
6. le jeu de tests `test.ml` est évalué, et affecte la référence susmentionnée ;
7. enfin, la référence `result` est examinée et convertie pour être renvoyée au serveur.

7. Processus JavaScript isolés communicant par passage de messages.
8. `https://ace.c9.io/`

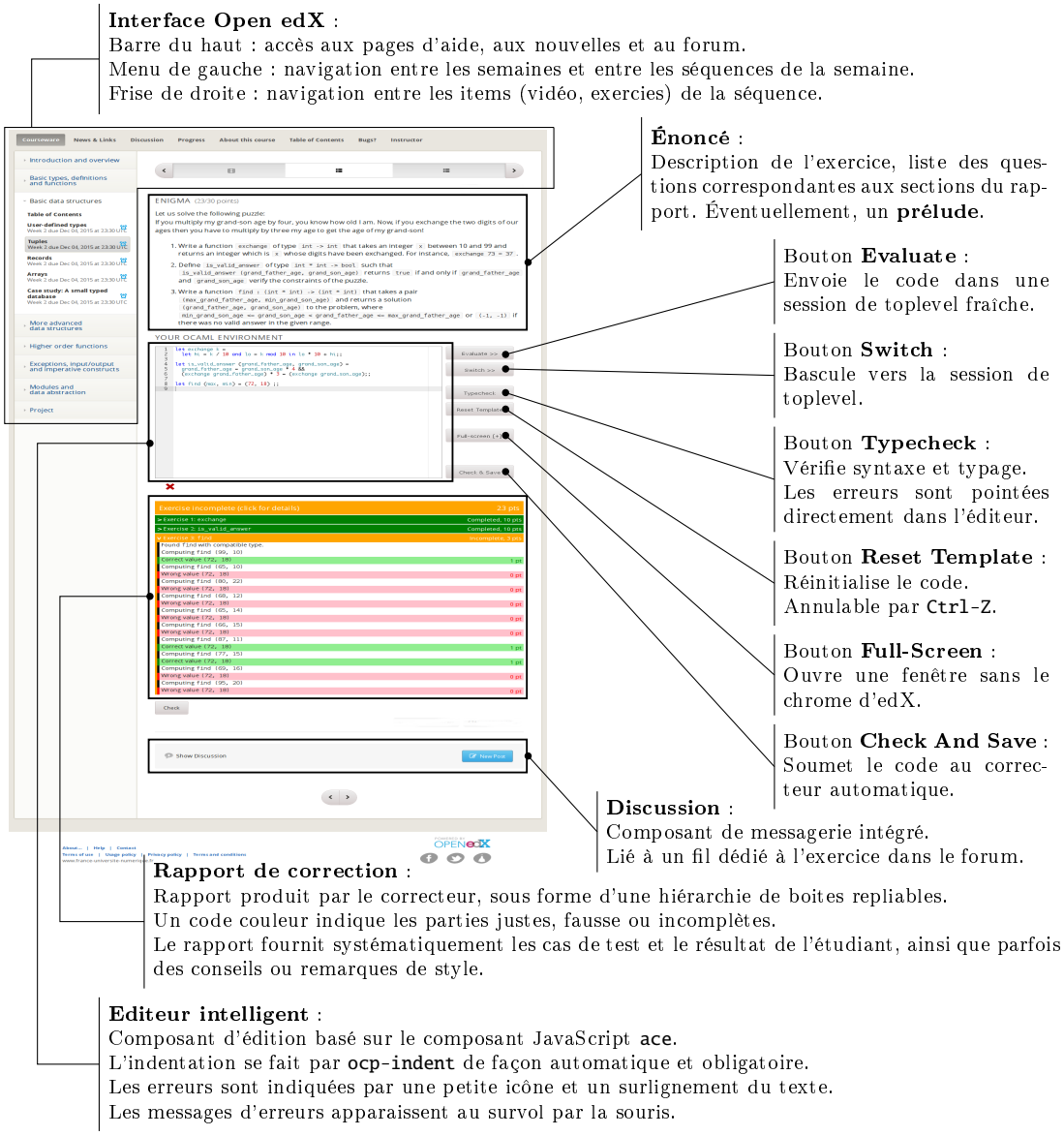


FIGURE 2 – La plate-forme d'exercices intégrée à Open edX

Un script OCaml simule cette séquence hors navigateur. Cela permet de valider tous les exercices du MOOC avant chaque mise à jour. Il extrait les sorties standard et d'erreurs, ainsi que le rapport au format texte, pour examiner les problèmes éventuels. Le script considère que la copie de l'étudiant est `solution.ml`, ce qui permet de calculer le score maximal réclamé par Open edX.

4.2 Primitives d'introspection typées

Le code de l'étudiant est chargé dans le même environnement que le code de test. Les primitives suivantes nous permettent de faire cohabiter ces deux parties de façon sûre, sans opérations (`Obj.`)magiques dans le code de test, qui reste strictement et statiquement typé.

Nous avons étendu le langage à l'aide d'un pré-processeur d'AST dérivé de `ppx_metaquot`[9] d'Alain

9. `https://github.com/alainfrisch/ppx_tools`

Frisch. Celui-ci ajoute une structure `[%ty: τ ]`, qui est compilée vers l'AST de ce littéral de type. L'astuce réside dans le fait que ce terme du type abstrait τ `Ty.ty`. À moins de faire volontairement sortir ces témoins de leur portée, on a alors une représentation à l'exécution du type statique τ.

Cette représentation est utilisée pour remonter une valeur depuis l'environnement du toplevel, et pour générer et imprimer des valeurs d'un type donné. Seuls les types monomorphes sont autorisés.

```
type 'a value = Absent | Present of 'a | Incompatible of string
val get_value: string -> 'a Ty.ty -> 'a value
val get_sampler: 'a Ty.ty -> (unit -> 'a)
val get_printer: 'a Ty.ty -> (Format.formatter -> 'a -> unit)
```

La fonction `get_value` cherche la valeur associée à un demandé l'environnement du toplevel. S'il ne la trouve pas, il renvoie `Absent`. S'il la trouve, il interprète le témoin de type dans le contexte courant du toplevel, et lance un test d'instanciation avec le type de la valeur conservé dans l'environnement du toplevel. Il renvoie `Present` avec la valeur si son type est plus général, ou `Incompatible` avec un message d'erreur destiné au rapport. Le code de test doit alors analyser le résultat pour obtenir la valeur du bon type recherchée, si elle existe. L'unification statique du `'a` dans le paramètre `'a Ty.ty` avec celui dans le retour `'a value` est prouvée correcte par le test d'instantiation à l'exécution.

La fonction `get_printer` appelle de façon sûre l'imprimeur du toplevel pour un type donné.

La fonction `get_sampler` parcourt la représentation de types pour générer un échantillonneur aléatoire, à la manière de Quickcheck [2, 3]. L'auteur du jeu de tests peut aussi personnaliser le générateur de valeurs pour un type nommé donné. Pour ceci, lorsque `get_sampler` rencontre un constructeur de type `t`, elle inspecte l'environnement courant du toplevel à la recherche d'une fonction `sample_t` de type `unit -> t`. Une variante existe pour les constructeurs paramétrés. Pour faciliter l'écriture de ces fonctions, un ensemble de combinateurs de génération aléatoire pour les principaux types prédéfinis est fourni, avec différents paramètres permettant de biaiser la distribution.

4.3 Bibliothèque de Rapports

Le résultat attendu par Open edX comporte un petit message en HTML. Il comporte aussi un pourcentage de réussite. Une autre partie d'Open edX réclame aussi un score maximal en points (le pourcentage de réussite est multiplié par le nombre de points avant affichage).

Nous avons fait le choix de définir un langage intermédiaire permettant de produire un rapport par parties, et d'examiner la composition finale pour à la fois calculer le score total en points et imprimer le HTML (ou le texte brut pour le script de vérification des tests). Étant donné les scores maximaux calculés par le script de vérification en examinant le rapport d'une copie de référence, il est aussi possible de déduire le pourcentage de réussite.

Définition du langage Ce langage intermédiaire est une simple structure d'arbre, ce qui permet aussi au jeu de test d'observer le rapport d'une question pour, par exemple, ne pas faire un jeu de tests qui n'a aucune chance de réussir, ou attribuer des points bonus. Le format mixe la sémantique (échec ou succès et score) avec les messages destinés à l'étudiant pour apprendre de ses erreurs.

```
type report = item list (* Un rapport est une liste d'éléments de rapport *)
and item = (* Un élément de rapport: soit une ligne, soit un bloc *)
  | Section of inline list * report (* Bloc avec titre *)
  | Message of inline list * status (* Ligne de rapport avec score *)
and status = (* Le score associé à une ligne de rapport *)
  | Success of int (* Résultat correct avec pondération *)
  | Failure | Incomplete (* Échec complet ou partiel *)
  | Informative | Important (* Une ligne de message sans résultat *)
and inline = (* Contenu les lignes et titres de rapport *)
  | Text of string (* Du texte *) | Code of string (* Une expression OCaml *)
  | IO of string (* Un bloc de texte lu ou imprimé *)
```

Calcul de score La fonction `result_of_report` extrait les informations sémantiques et calcule le score total du rapport pour la copie. Elle renvoie un booléen indiquant si au moins une ligne de rapport avait un score d'échec, et la somme de toutes les lignes de rapport ayant rapporté des points.

```
val result_of_report : report -> int * bool
val html_of_report : report -> string
val output_text_report : Format.formatter -> report -> unit
val output_html_report : Format.formatter -> report -> unit
```

Sorties La vue HTML est produite par `html_of_report`. Celle-ci infère un score pour chaque bloc en fonction des lignes et sous-bois qu'il contient. Le HTML généré permet de déplier et replier les blocs. Le score d'un bloc est `Success n` s'il contient au moins un succès et aucun échec. Le nombre de points `n` est alors le cumul de tous les succès. Dans ce cas, cette partie est considérée comme terminée, le bloc est affiché en vert et replié. Dans les autres cas, cette partie n'est pas terminée, elle est donc dépliée par défaut, pour que l'étudiant puisse voir quels points corriger. Elle peut valoir `Incomplete` (en orange) ou `Failure` (en rouge). La figure 2 comporte un exemple de rapport avec deux exercices terminés et un exercice incomplet.

4.4 Bibliothèque de tests

La bibliothèque de tests fait le lien entre les primitives d'introspection et la bibliothèque de rapports. C'est une collection de combinateurs permettant d'exprimer le protocole de test d'un exercice, en testant le code de l'étudiant pour produire et combiner des parties de rapport.

Exécution du code de l'étudiant Le principe central de la bibliothèque est l'exécution du code de la copie de l'étudiant sur des valeurs forgées par le code de test, suivi de la comparaison des résultats avec ceux d'une copie de référence. On utilise pour cela le type `'a result` suivant, qui permet de gérer les retours normaux et exceptionnels.

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

Deux combinateurs sont fournis, l'un produisant le résultat complet de l'exécution, y compris les sorties standard et d'erreur, l'autre seulement la valeur résultat. Il revient aux combinateurs de plus haut niveau d'encapsuler la fonction de l'étudiant et les paramètres de test dans une fermeture dont le seul paramètre est le `()` attendu par `exec` et `result`.

La comparaison peut se faire avec des combinateurs de comparaison fournis, du type `'a tester`.

```
type 'a tester = 'a Ty.ty -> 'a result -> 'a result -> Report.report
val test : 'a tester
val test_eq_ok : ('a -> 'a -> bool) -> 'a tester
val test_ignore : 'a tester
```

Le plus automatique est `test`. Il utilise l'égalité structurelle pour comparer les résultats, et produit un rapport de succès ou d'erreur. Il utilise le témoin de type fourni pour afficher le résultat. Des combinateurs permettent d'adapter les tests. Par exemple, `test_eq_ok` permet de préciser la fonction d'égalité, et `test_ignore` est utile pour tester les fonctions dont seuls les effets sont utiles.

Pour vérifier les sorties, une autre série de testeurs du type `io_tester` est fournie.

```
type io_tester = string -> string -> Report.report
val io_test_equals : ?trim: char list -> ?drop: char list -> io_tester
val io_test_lines : ?trim: char list -> ?drop: char list ->
                    ?skip_empty: bool -> ?test_line: io_tester -> io_tester
```

Un tel testeur prend en argument la sortie obtenue, celle attendue, et produit un rapport. Comme il est souvent inutile d'être trop strict sur les sorties, les testeurs permettent d'ignorer des caractères, tronquer les parties inutiles, ou découper le texte pour l'examiner par parties.

Tests de variables La bibliothèque fournit plusieurs ensembles de testeurs de plus haut niveau. Un premier jeu permet de tester les variables globales définies par l'étudiant.

```
val test_variable : 'a Ty.ty -> string -> ('a -> Report.report) -> Report.report
val test_variable_against : 'a Ty.ty -> string -> 'a -> Report.report
val test_variable_against_solution : 'a Ty.ty -> string -> Report.report
```

Les testeurs prennent le type d'une variable et son nom. Ils produisent un rapport d'échec lorsque la valeur n'est pas trouvée dans le code de l'étudiant ou n'a pas le bon type. Si la valeur est bien trouvée, `test_variable` passe sa valeur à une fonction de rapport personnalisée. Deux testeurs simplifiés sont fournis, `test_variable_against` qui vérifie légalité structurelle avec une valeur de référence donnée, et `test_variable_against_solution` qui vérifie l'égalité avec la valeur de la copie de référence.

Tests de fonctions Des combinateurs sont prédéfinis pour tester les fonctions de l'étudiant sur le même modèle. La fonction ci-dessous est un exemple pour les fonctions à un seul argument. Il en existe pour quelques arités fixées, ainsi qu'un générique utilisant un témoin de type flèche en GADT.

```
val test_function_1_against_solution :
  ?test: 'b tester -> ?test_stdout: io_tester -> ?test_stderr: io_tester ->
  ?sampler : (unit -> 'a) -> ?samples: int ->
  ?before_reference : ('a -> unit) -> ?before_user : ('a -> unit) ->
  ?after : ('a -> ('b * string * string) -> ('b * string * string) -> Report.report) ->
  ~ty: ('a -> 'b) Ty.ty -> ~name: string -> ~tests: 'a list -> Report.report
```

Ce testeur recherche la fonction `name` de l'étudiant et celle de référence qui doivent être du type `ty`. Il complète les `tests` donnés pour obtenir au moins `samples` cas en utilisant le générateur prédéfini ou `sampler` s'il est donné. Puis il vérifie que les résultats et sorties sont égales en utilisant les testeurs fournis, et enfin produit un rapport avec une ligne de succès ou d'échec par cas de test. Trois fonctions de rappel permettent de tester certains traits impératifs, comme décrit un peu plus loin.

Tests de langage Il est parfois utile de vérifier des propriétés sur le code plutôt que sur les valeurs. Pour ceci sont définis des testeurs d'AST, permettant de simplifier le parcours et la détection de motifs. Ils parcourent l'AST OCaml de façon générique, en appelant des fonctions de rappel.

```
type 'a ast_checker =
  ?on_expression: (Parsetree.expression -> Report.report) ->
  ?on_pattern: (Parsetree.pattern -> Report.report) ->
  ?on_external: (Parsetree.value_description -> Report.report) ->
  ?on_open: (Parsetree.open_description -> Report.report) ->
  ?on_module_occurence: (string -> Report.report) ->
  ?on_variable_occurence: (string -> Report.report) ->
  (* autres fonctions optionnelles *) -> 'a -> Report.report
val ast_check_expr : Parsetree.expression ast_checker
val ast_check_structure : Parsetree.structure ast_checker
```

Le résultat est la concaténation des résultats des fonctions de rappel. Si le vérificateur est appelé sans passer aucune des fonctions de rappel optionnelles, le résultat est simplement le rapport vide.

D'autres combinateurs de tests de l'AST sont fournis, soit pour utilisation directe, soit pour instancier les fonction de rappel d'un `ast_checker`. Une fonction de vérification de l'AST `ast_sanity_check` est aussi fournie, qui ne lance pas les tests si elle détecte des motifs douteux, comme `external` ou `Obj`.

Ces combinateurs sont utiles non seulement pour interdire globalement l'utilisation de certaines fonctions ou structures, mais aussi pour délimiter un sous ensemble précis du langage et de la bibliothèque pour un exercice donné, ou même pour donner des conseils de style à l'étudiant en repérant des motifs améliorables classiques.

4.5 Quelques exemples

Les combinateurs présentés couvrent un éventail de tests assez large pour ne rien avoir de plus à définir pour tester la plupart des exercices. On peut alors se concentrer sur la structure du rapport et la génération de cas de tests appropriés.

Cas simple La figure 3 est un des premiers exercices du MOOC. La seule spécificité est la génération des chaînes, dont on veut s'assurer qu'elle ne fabrique pas de chaîne vide. La plupart des exercices du MOOC se ramènent à cet exemple.

```
<p>Définissez deux fonctions utilisant des chaînes:</p>
<ol><li><code>last</code> donne le dernier caractère d'une chaîne;</li>
    <li><code>string_of_bool</code> convertit un booléen en chaîne à la OCaml.</li></ol>
```

(a) L'énoncé (`descr.html`)

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

(b) La copie de référence (`solution.ml`)

```
let last str =
  "Replace by your implementation." ;;
let string_of_bool truth =
  "Replace by your implementation." ;;
```

(c) Le patron (`template.ml`)

```
let sample_string () = [|"Ca";"Boz";"Hen"|].(Random.int 2)
                     ^ [|"nou";"man";"ry"|].(Random.int 2)
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", "", ""]) ] ;;
```

(d) Le jeu de tests (`test.ml`)

FIGURE 3 – Exemple de définition d'un exercice

Cas des expressions à toplevel Il est facile de tester des fonctions sur un jeu de tests. On utilise simplement l'application. Mais comment tester du code dont l'évaluation ne se fait qu'une fois ?

1. On définit dans `prepare.ml` une variable `x` que l'on choisit aléatoirement ;
2. on demande de définir une opération en supposant que `x` existe ;
3. on vérifie la valeur résultat de l'opération ;
4. sans oublier de donner la valeur de `x` pour cette exécution dans le rapport.

Cas du polymorphisme Si l'introspection typée présentée section 4.2 permet d'accéder à des valeurs en vérifiant que leur type à l'exécution correspond bien à celui attendu statiquement, ils n'ont pas pour but de transformer OCaml en langage dynamiquement typé. Ainsi, si une variable reste libre dans un témoin de type, aucune introspection ne pourra avoir lieu à l'exécution. Comment alors tester les fonctions polymorphes de l'étudiant ?

L'astuce est très simple. Puisque `get_value` effectue une un test d'instanciation entre les termes de types attendu et effectif et non un simple tests d'égalité syntaxique, il suffit de demander deux instances de la fonction. Par exemple, si l'étudiant doit redéfinir `sort : 'a list -> 'a list`, on effectuera une moitié des cas de tests avec `test_function_1 [%ty: int list -> int list] "sort"` et une autre avec `test_function_1 [%ty: char list -> char list] "sort"`. Ainsi on obtient des types monomorphes que l'on peut afficher et échantillonner, tout en testant que la fonction est effectivement polymorphe.

Cas de la complexité On veut parfois que l'étudiant implante une version spécifique d'un algorithme pour résoudre un problème dans un temps attendu. Par exemple, une recherche dans un tableau par

dichotomie. Pour cela, il faut vérifier que le bon nombre d'opérations est effectué en fonction de la taille du cas de test. La façon de procéder est la suivante.

1. Spécifier les opérations autorisées pour l'exercice, par exemple `Array.get` ;
2. redéfinir les opérateurs et fonctions dans le `prepare.ml` pour incrémenter une référence globale ;
3. avant le test, utiliser la fonction de vérification d'AST, en fournissant une fonction de rappel sur les variables libres qui vérifie que seules les opérations autorisées apparaissent [10] ;
4. lors du test, réinitialiser la référence avant chaque appel de la fonction utilisateur, et vérifier la complexité après.

Pour les exercices du MOOC, nous avons choisi de formuler les problèmes très précisément, en demandant à l'étudiant d'effectuer le nombre minimum d'opérations, que nous testons de façon stricte. Il serait envisageable de tester le comportement de la fonction sur l'ensemble des cas, et de vérifier que la fonction a le bon comportement à une constante près. Mais nous avons remarqué que les étudiants préféraient en général que les résultats attendus soient les plus précis possibles.

Cas de l'application partielle Afin de travailler sur le concept d'application partielle, un des exercices du MOOC demande à l'étudiant de prendre une formule mathématique à 4 variables, et de l'encoder en une fonction OCaml dont l'application de chaque variable effectue un maximum de calculs. Du point de vue mathématique, il n'y a aucune différence observable entre une bonne et une mauvaise solution. On utilise alors une technique brutalement impérative pour tester le comportement d'un code pur ! La technique est tout à fait similaire à la précédente, par redéfinition d'opérateurs. Simplement, en plus de tester que le nombre d'opérations total est bien inférieur ou égal à celui attendu, on applique les arguments un par un, en vérifiant après chaque le compteur de chaque opération.

Cas des entrées Nous avons montré les combinateurs de tests pour les sorties. Tester du code utilisant les fonctions de lecture sur l'entrée standard est un peu plus complexe. En effet, s'il est assez simple de manipuler les sorties pour rester flexible sur les espaces, sauts de lignes et autres séparateurs, on peut difficilement modifier les entrées pour s'adapter au code. Pour cela, la solution est de restreindre les fonctions d'entrées autorisées. On autorisera par exemple `read_line ()`, et on la redéfinira pour retourner successivement les éléments d'une liste (les mêmes pour le code de référence et celui de l'étudiant) puis lever `End_of_file`. On est alors certain que tous les étudiants utiliseront les mêmes fonctions, et on pourra afficher dans le rapport les entrées pour chaque test.

Cas des effets Afin de tester les effets de fonctions telles que le tri en place d'un tableau de type `'a array -> unit`, on a besoin d'observer l'état des valeurs passées en paramètres à différents moments durant le test. Pour cela, on utilise les fonctions de rappel des testeurs de fonctions. À chaque étape, on effectue une copie du paramètre à observer et on le réinitialise. On compare à la fin du test les copies entre elles. Des combinateurs spécifiques sont prédéfinis pour générer les fonctions de rappel destinées à observer un paramètre impératif en particulier, comme dans l'exemple ci-dessous.

```
1 Section ([ Text "Exercise 1: " ; Code "rotate" ],
2          let before_reference, before_user, after, test =
3            array_arg_mutation_test_callbacks [%ty: int array] in
4          test_function_1_against_solution ~before_reference ~before_user ~after ~test
5            [%ty: int array -> unit] "rotate" []) |> set_result ;;
```

Cas des modules Les problématiques des exercices sur les modules sont généralement sur les thèmes de l'abstraction et du typage. Le test d'un tel exercice est en général le suivant.

1. Tout d'abord, on récupère le module sous la forme d'un module de première classe. Un cas particulier dans la fonction `get_value` recherche les modules plutôt que les valeurs si le nom commence par une majuscule et le témoin de type est un module empaqueté. Une test d'inclusion de signatures est effectué, fournissant en cas d'échec un message pour le rapport.
2. On effectue les tests fonctionnels sur le code de ce module.

3. Éventuellement on vérifie manuellement des propriétés sur la signature du module en observant l'environnement du toplevel, pour savoir si tel type est bien abstrait, un alias attendu, etc.

5 Le script de description et déploiement

Comme décrit à la section 2, le mode de développement classique d'un cours Open edX passe par l'utilisation de l'éditeur visuel Open edX studio.

Ce mode de développement ne nous a pas paru adapté à notre usage pour deux raisons. La première est que nous voulions pérenniser le contenu une fois le MOOC dispensé, et donc maîtriser le format. De plus, nous préférions utiliser nos outils de développement et contrôle de versions habituels. La seconde est que l'édition visuelle est très coûteuse en clics (et donc en temps) et très répétitive. Une troisième raison est apparue en cours de route. L'éditeur ne fait pas assez de vérifications, et il est très facile de créer un lien mort ou de changer un identifiant, ce qui peut mener dans certains cas à des problèmes aussi gênants que la perte des copies des étudiants pour un exercice.

Finalement, il était vraiment rentable en temps et en prévention du stress de disposer du script fiable que nous décrivons dans cette section. Il est même clair que le contenu n'aurait pas pu être le même sans ce système qui a permis d'ajouter des fonctionnalités comme la génération automatique d'archives contenant les supports de cours pour chaque semaine (avec leur mise à jour automatique après correction des erreurs détectées par les étudiants et testeurs), et l'ajout de tables des matières plus complètes et faciles à examiner que celles d'Open edX.

Le format des archives Open edX Nous avons donc choisi de générer une archive à partir d'une description du cours, que nous importons via l'éditeur de l'instance de production de FUN.

Pour comprendre ce format, il a fallu glaner dans la documentation officielle du format et dans le code d'implantation d'Open edX, mais surtout étudier les archives exportées par l'éditeur, puis essayer d'en produire par essais et erreurs. Cette étape cruciale a pris beaucoup plus de temps que prévu car les différentes documentations ne concordent pas toujours, que la visionneuse et l'éditeur ne sont pas d'accord sur le format, et que le code Python qui lit les archives ne détecte les erreurs que trop tard (ou parfois pas du tout). À force de tâtonnements, nous avons abouti à un sous langage pour lequel nous somme confiants sur l'interprétation par Open edX, et suffisamment spécifié pour résister aux mises à jour en production.

Nous passons sur les détails du format, qui est un fichier `.tar.gz` contenant les fichiers statiques, ainsi que la structure et le contenu du cours éparpillés en une multitude de fichiers XML et JSON.

Notre format intermédiaire de description Une fois le format Open edX compris et étudié, une solution aurait été d'écrire un validateur, et de produire l'archive à la main. Les raisons de pérennité déjà évoquées, ainsi que le simple fait que le format n'a pas été prévu pour des humains à la base, ont fait que nous avons opté pour une description dans un format un peu plus abstrait.

Cependant, étant donné que le MOOC OCaml exploite la quasi totalité des fonctionnalités, nous ne pouvions pas nous contenter d'une description trop éloignée du format (comme le fait par exemple le script GaNyMede, écrit par Fabrice Kordon[11], qui génère une archive à partir d'une description très simplifiée au format CSV et d'une hiérarchie de fichiers respectant des conventions de nommage).

Nous avons fait le choix d'implanter notre langage de description sous la forme d'une bibliothèque OCaml, permettant d'utiliser le langage OCaml comme méta-langage de macro-génération. La bibliothèque est très proche de la structure du format Open edX. Nous avons ensuite écrit une surcouche spécifique au MOOC OCaml, qui fait automatiquement le lien entre nos dépôts d'exercices et de supports pédagogiques, et qui compile les archives de supports téléchargeables et les tables des matières.

11. `https://github.com/fkordon/GaNyMEDE`

Garanties statiques Le faible niveau des vérifications effectuées lors de l'import est très gênant car il empêche de faire des modifications mineures sans tout tester à nouveau. Tous les problèmes mentionnés ci-après peuvent vraiment être rencontrés (et non détectés) durant un import d'archive. Nous avons donc décidé non seulement de vérifier la bonne structure du document, mais aussi d'apporter des garanties plus avancées sur les archives crées.

— Sur la forme : les parties redondantes du format sont générées de façon à s'assurer qu'elles ne se contredisent pas ; les identificateurs uniques sont vérifiés en utilisant la sémantique de l'éditeur afin qu'ils ne puissent pas causer de conflits d'URL ou d'écrasements de fichiers ; les parties en HTML brut sont récrites pour êtres valides dans le dialecte d'XML compris par Open edX ; les caractères UTF-8 sont échappés.
— Sur les méta-données : la cohérence des différentes parties de la politique de notation est vérifiée (note finale ne dépassant pas 100%, pas d'exercices n'appartenant à aucun ou plusieurs modules de notation, etc.) ; les dates sont vérifiées (les dates de début sont bien avant les dates de fin, les dates ne sont pas dans le passé, etc.).
— Sur les liens : les liens entre pages sont vérifiés ; les fichiers externes sont ajoutés automatiquement à l'archive, les liens vers ces fichiers sont vérifiés, et les collisions détectées ; les identifiants des pages et exercices sont générés de façon à pouvoir écraser le contenu par une archive de mise à jour sans perdre les copies des étudiants ou les liens du forum vers les exercices.

Et enfin, spécifiquement pour ce MOOC, le script vérifie localement que les exercices passent les tests, afin d'éviter de mettre en ligne une mise à jour d'énoncé ou de correcteur qui ne fonctionne pas.

Aperçu de la bibliothèque La bibliothèque définit des types abstraits pour les différents niveaux de la structure. Ils sont construits via des fonctions de construction. La bibliothèque utilise les arguments étiquetés et optionnels pour représenter les parties variables du format.

```
type chapter and sequence and page and item
val chapter :
  name: string -> ?start: date -> ?path: string -> ?uid: string -> sequence list -> chapter
val sequence :
  name: string -> ?start: date -> ?path: string -> ?uid: string -> ?grade: grade ->
  page list -> sequence
val page : name: string -> ?path: string -> ?uid: string -> item list -> page
val video : name: string -> ?path: string -> ?uid: string -> string -> item
```

Un type abstrait est aussi défini pour les notes. Ici celui-ci ne représente pas directement une entité d'Open edX, mais regroupe en une valeur des informations disséminées dans le format.

```
type grade
val grade : ?due: date -> label: string -> drop: int -> weight: int -> string -> grade
```

La bibliothèque définit aussi des combinateurs XML de plus bas niveau pour construire le contenu des pages. Le type des éléments XML est paramétré par une variable fantôme, utilisée pour assurer la grammaire de l'imbrication des balises, là où cela fait sens. Par exemple, des combinateurs pour construire du HTML bien formé (à la manière de [8]) sont fournis. Pour accepter les fichiers externes, un petit lecteur d'XML est fourni, qui sait interpréter les liens et les vérifier.

```
type 'a node
val text : string -> [> `TEXT ] node
val div : [< `INLINE | `BLOCK | `TEXT] node list -> [> `BLOCK ] node
```

Les URL sont des valeurs OCaml et non des chaînes, supprimant les liens morts par construction.

```
type 'a url
val to_uid : string -> [> `INTERNAL ] url
val to_url : string -> [> `HTTP ] url
val to_file : ?rename: string -> ?download: string -> string -> [> `STATIC ] url
val img : alt: string -> [< `STATIC | `HTTP ] url -> [> `TEXT ] node
```

Finalement, la fonction principale `course` assemble les différentes entités principales du cours et un certain nombre de méta-données. Plutôt que donner sa signature, voici comment cette fonction est appelée pour le MOOC OCaml. On peut voir que la structure du cours est écrite dans plusieurs modules OCaml. On utilise donc OCaml comme langage hôte non seulement pour écrire la description du MOOC, mais aussi pour la structurer et faciliter la collaboration.

```
write @@ course ~is_public: true ~is_new: true
    ~tabs: [ Static ("About this course", Ocamooc_2015_overview.overview_tab ()) ;
             Static ("Table of Contents", toc contents) ;
             Static ("Bugs?", Ocamooc_2015_bugs.bugs_tab ()) ]
    ~enrollment: (date "2015-07-20T00:00:00Z", date "2015-11-29T23:30:00Z")
    ~period: (date "2015-10-19T08:00:00Z", date "2015-12-04T23:30:00Z")
    ~days_early_for_beta: 7 ~grace_days: 14
    ~overview: (Ocamooc_2015_overview.overview_page ())
    ~description:
      [ text  "In this course you will discover the power of Functional Programming, \
               using the OCaml language to write efficient and elegant programs." ]
    ~organization_name: "Université Paris Diderot"
    ~organization_path: "parisdiderot"
    ~image: (to_file ~rename: "images/course_small.png" "../images/course_small.png")
    ~name: "Introduction to Functional Programming in OCaml"
    ~id: "56002"
    ~updates: (Ocamooc_2015_updates.updates ())
    ~discussions: [ discussion ~path: "ocaml" "About OCaml" ;
                    discussion ~path: "course" "About this course" ;
                    discussion ~path: "coffee" "Coffee room" ]
    [ (* les chapitres *) ] ;;
```

Tous ces combinateurs produisent un arbre intermédiaire, qui est vérifié, puis compilé par la fonction `write` vers le fichier `/course.xml` et les différents fichiers JSON. Durant cette traversée, les fichiers cibles de liens sont recopiés dans le répertoire des fichiers statiques. Une erreur est produite si un fichier manque. Finalement, la commande `tar` est appelée pour produire une archive, que nous pouvons téléverser avec confiance.

Aperçu des combinateurs spécifiques Nous avons pris des convention simples de nommage dans nos dépôts pour le matériel de cours et pour les correcteurs d'exercices, qui nous ont permis de définir des fonctions à l'appel très concis. Voici par exemple la définition de la séquence 4 de la semaine 6.

```
let arrays = ocamooc_sequence 4 ~name: "Mutable arrays"
    ~files: [ "../slides/week5/4arrays/code/squarecubes.ml";
              "../slides/week5/4arrays/code/arraymutables.ml"; ]
    ~description:
      (p @@ xml "Now we meet data structures that are modifiable <em>in place</em>. \
                 We will start from the arrays, as we find out that thay con actually \
                 be modified, like arrays in most other programming languages." ] ]
    [ ocamooc_problem
        "Rotating the contents of an array"
        "mooc.week5.seq4.ex1" ;
      ocamooc_problem
        "Implementing a stack with an array"
        "mooc.week5.seq4.ex2" ]
```

Ce code va : créer une page contenant le titre, la vidéo, la petite description, des liens vers les transparents, les sous-titres (dans les différentes langues qu'il trouve dans le dépôt), et vers les fichiers de code donnés ; copier tous les fichiers mentionnés dans l'archive Open edX ; ajouter ces fichiers à l'archive ZIP de la semaine pour les étudiants ; inscrire tous ces éléments dans les différentes tables des matières ; vérifier que les exercices mentionnés existent dans le dépôt des exercices ; vérifier que leurs tests passent ; et enfin inclure les fichiers des exercices et créer une page par exercice avec les composants de description et de discussion, ainsi que le composant d'exercice dont il génère le code HTML + JavaScript + Python nécessaire.

6 Conclusion

L'effort déployé à la fois par l'équipe pédagogique pour concevoir le matériel de cours et par nous pour réaliser les solutions présentées dans cet article a été conséquent. La figure 4 donne un aperçu de la répartition du code écrit pour l'occasion, pour un volume total d'environ 22 000 lignes de code.

Ce travail est salué par les étudiants, apprécient la possibilité de travailler dans leurs navigateurs, avec un environnement qui n'a presque rien à envier à une installation sur leur machine, ainsi que pour la quantité et la qualité des exercices et des correcteurs. C'est pour nous une vraie récompense pour ces efforts.

Tout cela n'aurait pas été envisageable sans la possibilité de travailler hors ligne, en collaboration via des dépôts versionnés, et donc sans le script de déploiement. L'écriture d'autant d'exercices a aussi été possible grâce à ce script, mais aussi à la bibliothèque de tests permettant de tester la plupart des programmes de façon très simple.

Par effet de bord, ce travail a de plus permis de faire en sorte que le MOOC OCaml se démarque de la plupart des autres MOOC disponibles sur plate-forme edX par l'ajout de fioritures comme le téléchargement des supports sous forme d'archives, ou les tables des matières.

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

(a) Environnement des exercices.

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

(b) Script de description et déploiement

(*i*) Une partie a été contribuée par l'équipe du cours, Roberto Di Cosmo, Yann Régis Gianas et Ralf Treinen.
(*ii*) Contient aussi des exercices QCM pour les premières séquences du cours.

FIGURE 4 – Taille du code (hors interfaces triviales, fichiers de construction, etc.)

D'autres plate-formes de MOOC ? Comme nous l'avons vu, la plate-forme d'exercices possède déjà un mode autonome. Il n'y a donc pas de verrou à la porter vers d'autres environnements de MOOC. C'est principalement le choix d'architecture entièrement dans le navigateur qui donne cette possibilité.

D'autre part, la description du MOOC OCaml est concise, et ne laisse que peu transparaître le format propriétaire Open edX. L'architecture choisie est donc plutôt satisfaisante, puisqu'elle permet à la fois d'exploiter toutes les capacités d'edX, de façon plus rapide qu'avec son éditeur, avec un déploiement et des mises à jour sûres, tout en gardant une description du cours en tant que telle concise et abstraite de la cible.

Nous avons l'intention, une fois le MOOC OCaml terminé, de mettre une partie du contenu à disposition sur un site Web autonome, généré statiquement à partir de la description.

D'autres plate-formes de correction automatique ? Une des particularités de notre bibliothèque de tests est d'être très liée à OCaml. Cependant, il ne serait pas difficile de l'intégrer à des solutions de correction automatique généralistes (citons CodeGradX [7]).

Tout d'abord, ces plate-formes ne testent en général pas le code de l'étudiant directement dans le navigateur comme nous le faisons, mais dans des environnements confinés côté serveur. Hors notre système de correction fonctionne déjà hors du navigateur pour nous permettre de vérifier les correcteurs avant mise en ligne. Il est donc tout à fait faisable de produire des exécutables de test autonomes à lancer sur le code de l'étudiant par le serveur de test.

Ensuite, ces plate-formes fonctionnent souvent en examinant la sortie des programmes de l'étudiant. Une astuce pour tester des propriétés de haut niveau étant alors de lancer des tests unitaires sur le code de l'étudiant. Ceux-ci produisent une sortie dans un format standard compris par la plate-forme, qui les interprète pour produire un rapport et un score. Le fait d'avoir défini un arbre de syntaxe abstraite pour les rapports devrait nous permettre d'adapter facilement la sortie de notre testeur.

D'autres MOOC ? Le script de déploiement pourrait tout à fait être réutilisé pour produire d'autres MOOC sur plate-forme Open edX. Bien sûr, le coût d'accès est le langage OCaml, il ne se place donc pas du tout en concurrence avec le mode habituel de développement. Mais pour des enseignants connaissant déjà OCaml, le jeu en vaut probablement la chandelle. Nous avons prévu de libérer le code source de ce script.

Il est aussi probable qu'une partie de l'environnement des exercices pourrait être réutilisée. On pourrait par exemple imaginer un MOOC d'algorithmique en OCaml, ou une initiation à un assistant de preuve implanté en OCaml.

Références

[1] B. Canou, V. Balat, and E. Chailloux. O'Browser : Objective Caml on browsers. In *ACM Workshop on ML*, 2008.

[2] K. Claessen and J. Hughes. Quickcheck : A lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 46(4) :53–64, May 2011.

[3] A. Darrasse and B. Canou. Fast and sound random generation for automated testing and benchmarking in objective caml. In *ACM Workshop on ML*, 2009.

[4] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme : a programming environment for scheme. *Journal of Functional Programming*, 12 :159–182, 2002.

[5] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. The OCaml system release 4.02 : Documentation and user's manual. Technical report, 2014.

[6] H. Miller, P. Haller, L. Rytz, and M. Odersky. Functional programming for all! scaling a MOOC for students and professionals alike. In *ICSE 2014*, 2014.

[7] C. Queinnec. Exposé invité : De la correction automatisée. In *Journées francophones des langages applicatifs (JFLA)*, 2010. `http://www.paracamplus.com/`.

[8] P. Thiemann. A typed representation for html and xml documents in haskell. *Journal of Functional Programming*, 12 :2002, 2001.

[9] J. Vouillon and V. Balat. From bytecode to javascript : the js_of_ocaml compiler. *Software : Practice and Experience*, 44(8) :951–972, 2014.