Portage de matériels pédagogiques sur la plateforme Learn-OCaml

Loïc Sylvestre

5 juillet 2019

Learn-OCaml

▷ Environnement pour la correction automatisée d'exercices de programmation en OCaml

- logiciel libre (licence BSD)
- projet de la `Fondation OCaml`
- initialement conçu pour un MOOC

Introduction to Functional Programming in OCaml

- puis utilisé en séances de travaux pratiques à `PARIS 7`

« Les équipes enseignantes ont pu traduire leurs énoncés de travaux dirigés dans le système sans difficultés »

(Bozman, Canou, Di Cosmo, Couderc, Gesbert, Henry, Le Fessant, Mauny, Morel, Peyrot. **Learn-OCaml : un assistant à l'enseignement d'OCaml, JFLA 2019**)

Learn-OCaml

Sujet de mon stage de L3

- ▷ encadré par Emmanuel Chailloux

Point de départ : les supports de MPIL (3I008)

- ▷ séances de TD/TP + projets
- ▷ modèles fonctionnel, impératif et objet + interopérabilité
  - ▶ traits de programmation relativement avancés
  - ▶ motivent des expérimentations autour de Learn-OCaml
  - ▶ mettent à l'épreuve le système de correction automatisée

Plan

Partie I : Le fonctionnement de Learn-OCaml

- ▷ illustré par un exemple

Partie II : Les limites du système de correction

- ▷ illustrées par un contrexemple

Partie III : Des améliorations

- ▷ Faciliter l'écriture des exercices
- ▷ Introduire de nouvelles techniques de correction
- ▷ Un exemple détaillé

Partie I : Le fonctionnement de Learn-OCaml

Une boucle d'interaction nichée dans le navigateur

> *« L'évaluation de code* `OCaml` *repose sur le toplevel* [...] *pour compiler à la volée les phrases envoyées au toplevel vers* `JavaScript` *et les évaluer dynamiquement. »*
>
> (Canou, Bozman, Henry. Sous le capot du MOOC OCaml, JFLA 2016)

Des exercices, chaque exercice est décrit par 6 fichiers

- ▷ **descr.md** (l'énoncé)
- ▷ **prelude.ml**, **prepare.ml**, **template.ml**, **solution.ml** et **test.ml**

Déroulement d'un exercice

Lorsque l'étudiant commence un exercice

- ▷ **prelude.ml** et **prepare.ml** sont évalués dans le toplevel
- ▷ L'étudiant voit l'énoncé et le `prelude`
- ▷ Il rédige sa réponse à partir d'un `patron` (**template.ml**)

NB : **prepare.ml** est invisible dans l'application Learn-OCaml.
→ introduit des définitions sans en dévoiler l'implémentation.

Phase de correction

- ▷ à la demande de l'étudiant (à condition que son code compile)
- ▷ Nouveau toplevel dans lequel sont évalués successivement :
  - **prelude.ml** et **prepare.ml**
  - le code de l'étudiant encapsulé dans un module `Code`
  - **solution.ml** encapsulé dans un module `Solution`
  - des utilitaires
  - le fichier **test.ml**

La phase de correction

```
⟨prelude.ml⟩ ;;
⟨prepare.ml⟩ ;;

module Code = struct ⟨le code de l'étudiant⟩ end ;;

module Solution = struct ⟨solution.ml⟩ end ;;

module Introspection = ... ;;
let code_ast = ... ;;

let results = ref None ;;
let timeout = ... ;;

module Test_lib = ... ;;
module Report = ... ;;

⟨test.ml⟩ ;;
```

Les sources d'un exercice : *Peano*

▷ **descr.md**

```
# Peano
Définir la fonction
of_int : int → peano
```

▷ **prelude.ml**

```
type peano = Z | S of peano
```

▷ **prepare.ml**

```
exception Undefined
let ⊥ _ = raise Undefined
```

▷ **template.ml**

```
let of_int = ⊥
```

▷ **solution.ml**

```
let rec of_int = function
  | 0 → Z
  | n → S (of_int (n-1))
```

▷ **test.ml**

```
let report =
  Test_lib.ast_sanity_check
        code_ast
        make_report

Test_lib.set_result report
```

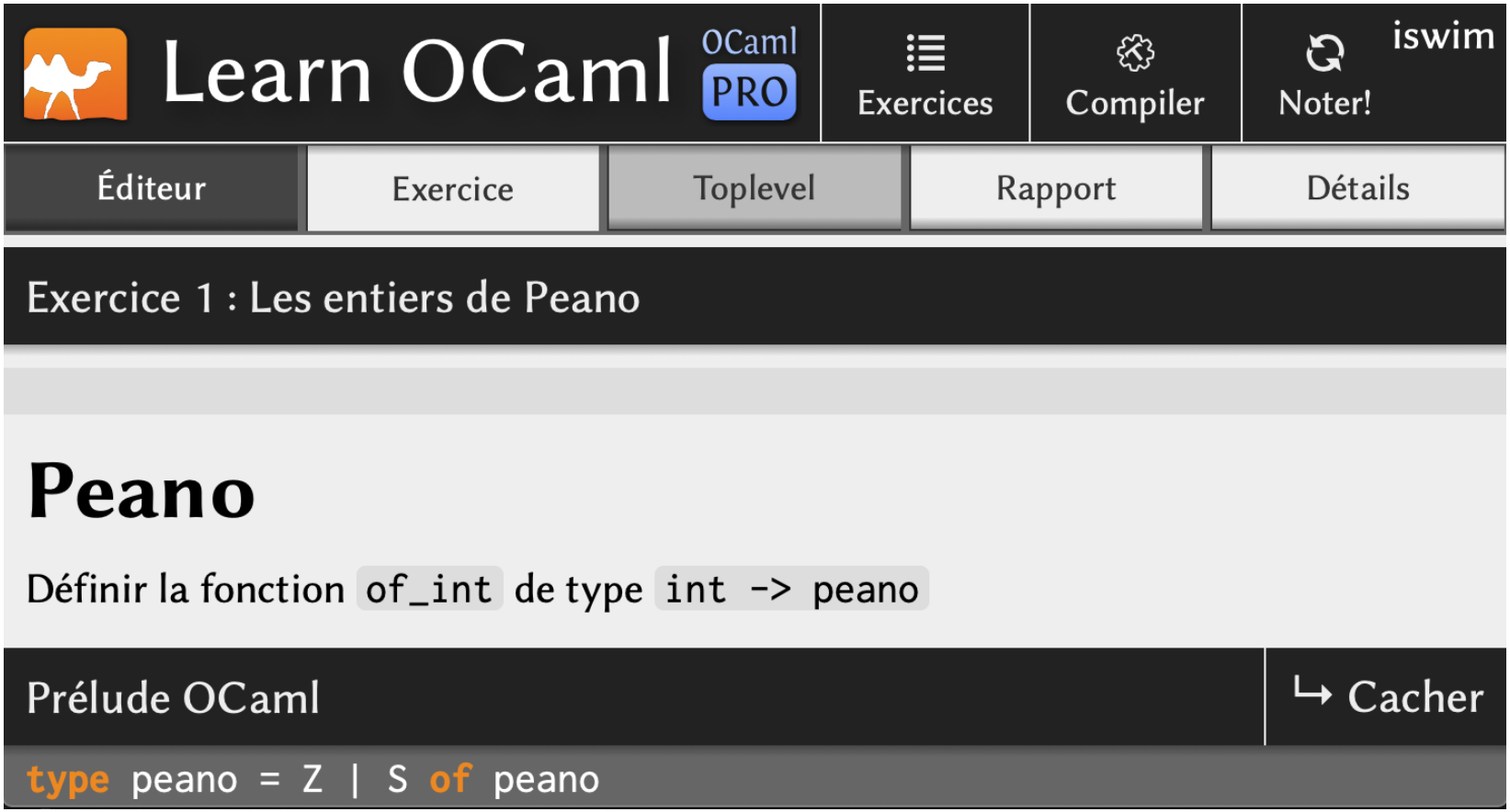
NB : `make_report` est une fonction spécifique à chaque exercice.

Son type est `unit → Report.t`

Construction du rapport de correction

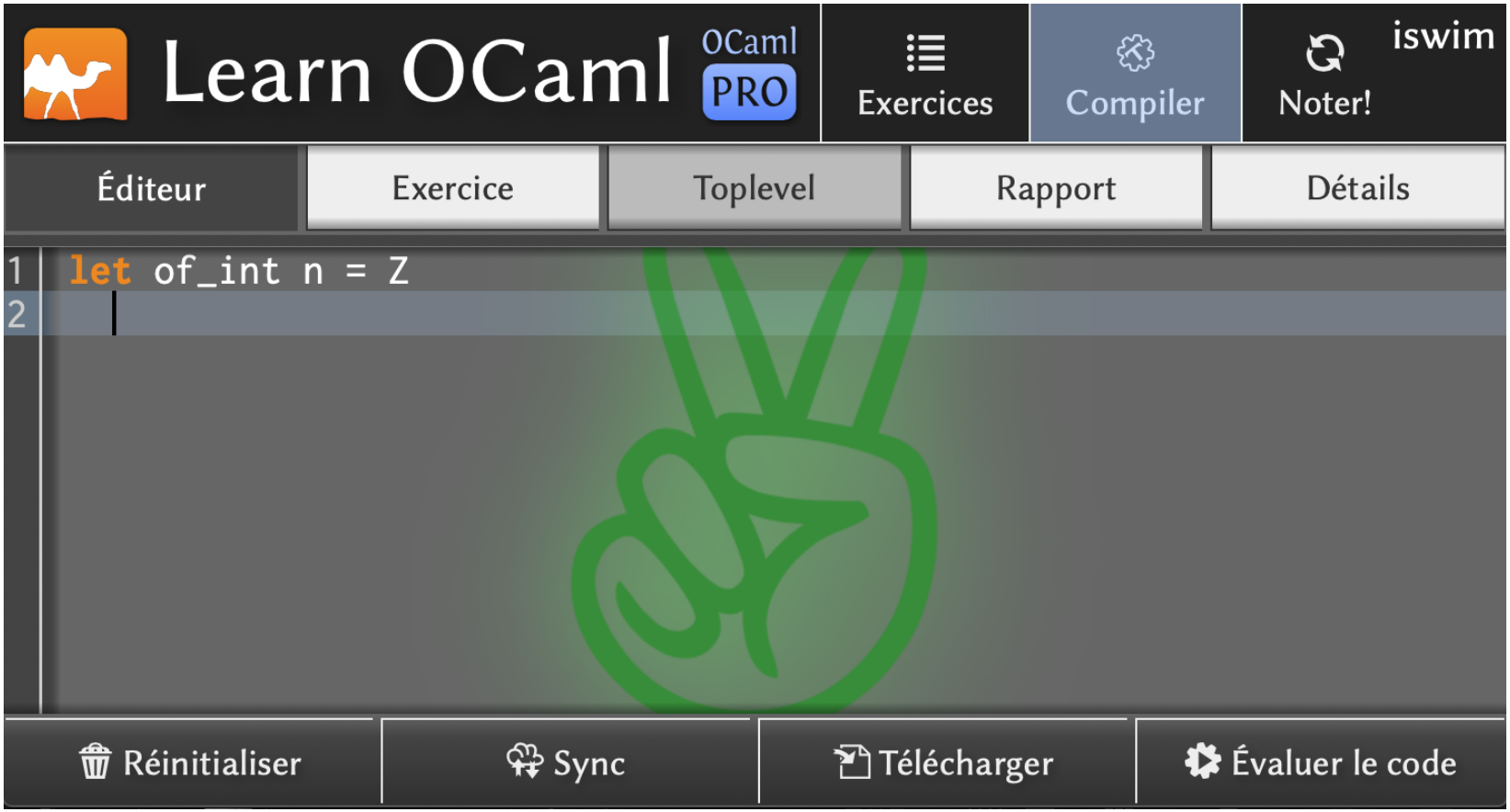
```
let make_report () =
  let results =
     Test_lib.test_function_1_against_solution
                 [%ty: int -> peano] "of_int"
                 ~sampler:Test_lib.sample_int
                 ~gen:5
                 [0;3;7;10]
  in
  [ Report.section "Q1 : of_int" results ]

let report =
  Test_lib.ast_sanity_check
        code_ast
        make_report ;;

Test_lib.set_result report
```

Peano : l'énoncé et le prélude

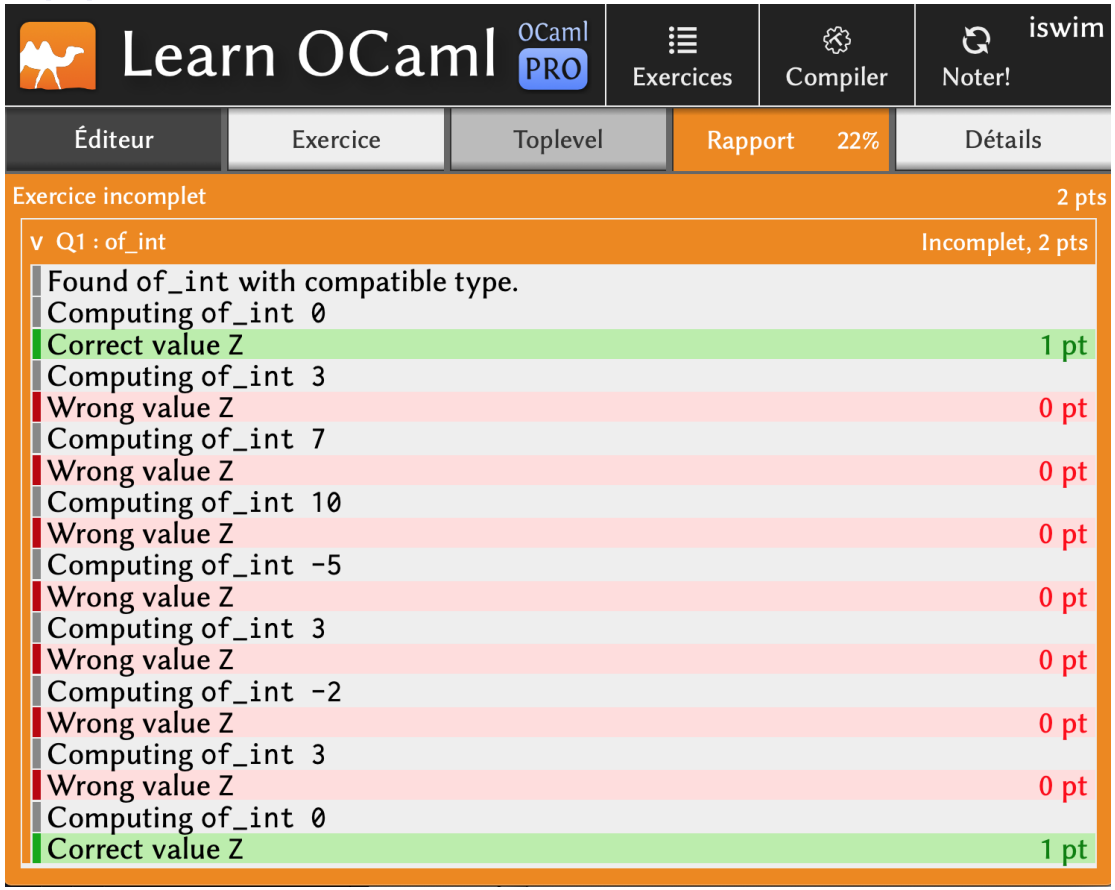


Peano : le template

Peano : la réponse de l'étudiant



Peano : le rapport de correction




```
let make_report () =
  let results =
     Test_lib.test_function_1_against_solution
                  [%ty: int -> peano] "of_int"
                  ~sampler:Test_lib.sample_int
                  ~gen:5
                  [0;3;7;10] in
  [ Report.section "Q1 : of_int" results ]
```

Partie II : Les limites du système de correction

Comment traiter les fonctions polymorphes ?

Canou, Bozman, Henry. Sous le capot du MOOC OCaml, JFLA 2016

« L'astuce est très simple. [...] *Il suffit de demander deux instances de la fonction. Par exemple, si l'étudiant doit redéfinir*

`sort : α list → α 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".
```

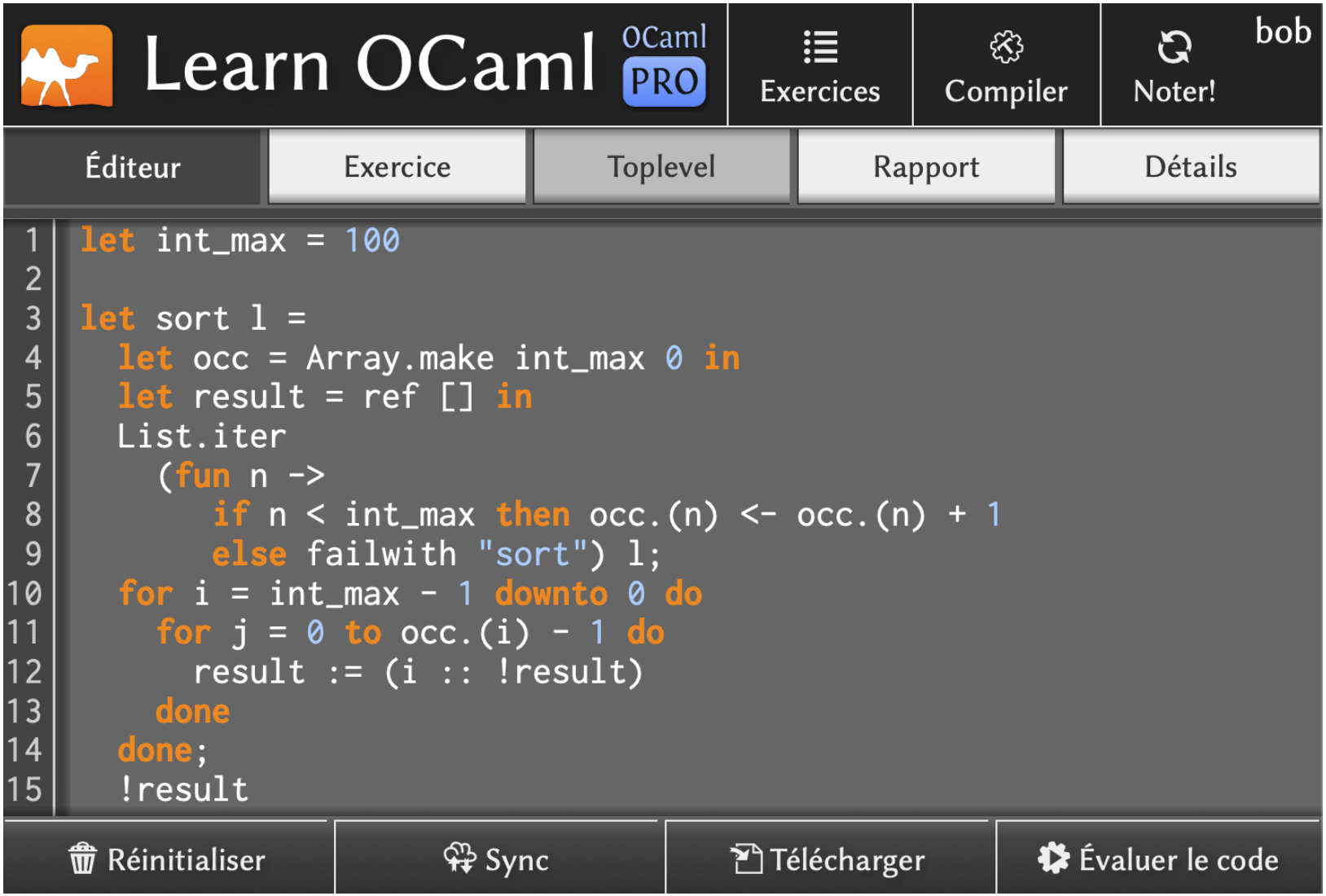
Ainsi on obtient des types monomorphes [...] *tout en testant que la fonction est effectivement polymorphe. »*

« écrire la fonction sort (5 points) »

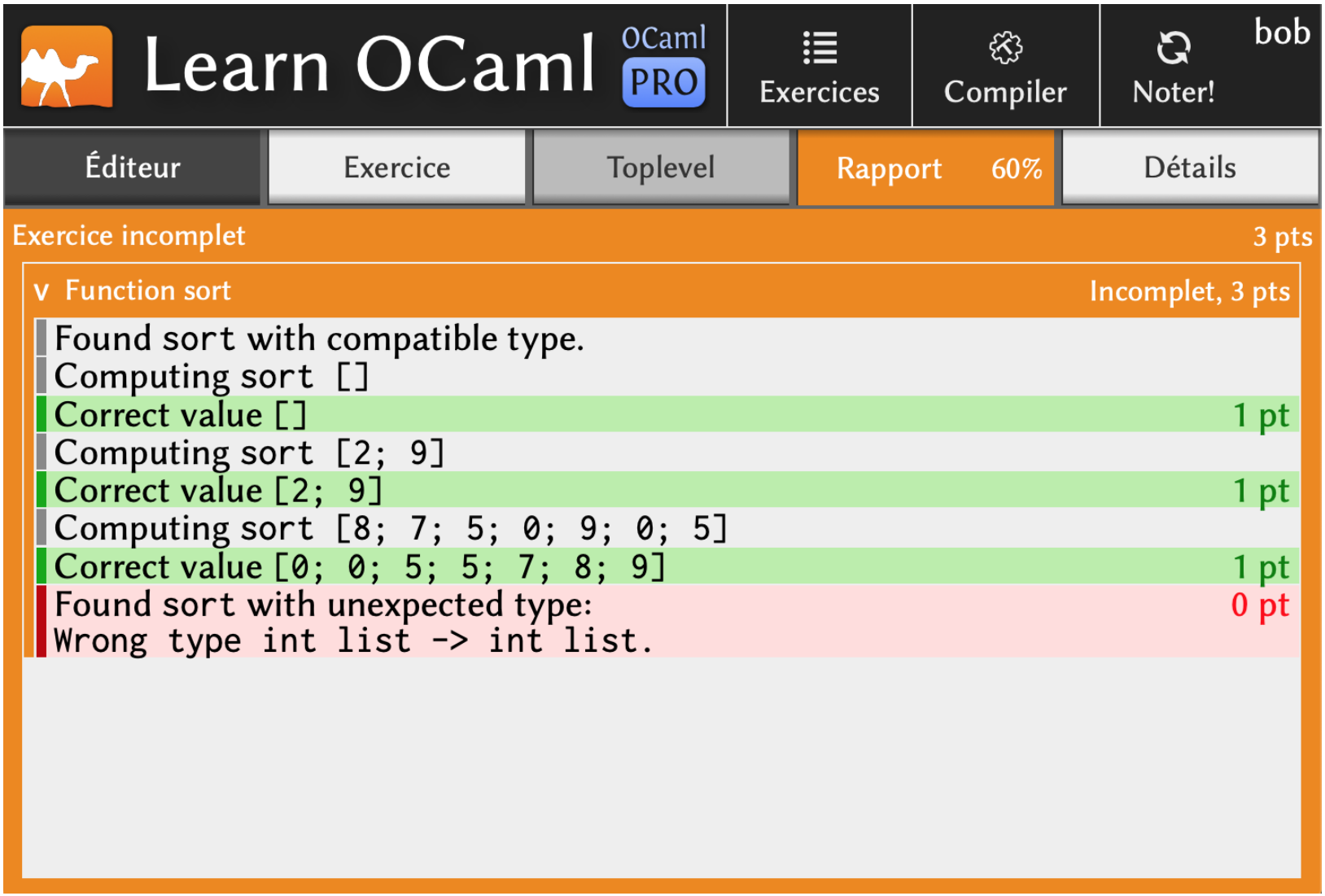
▷ le code de test (**test.ml**)

```
let make_report () =
  let report_int_list =
    Test_lib.test_function_1_against_solution
                [%ty: int list -> int list] "sort"
                ~sampler:Test_lib.(sample_list ~sorted:false sample_int)
                ~gen:3
                []

  and report_char_list =
    Test_lib.test_function_1_against_solution
                [%ty: char list -> char list] "sort"
                ~sampler:Test_lib.(sample_list ~sorted:false sample_char)
                ~gen:2
                []
  in
  let report = List.append report_int_list
                           report_char_list in
   [ (Report.section "Function sort" report) ] ;;

let report = ast_sanity_check code_ast make_report in
  Test_lib.(set_result report)
```

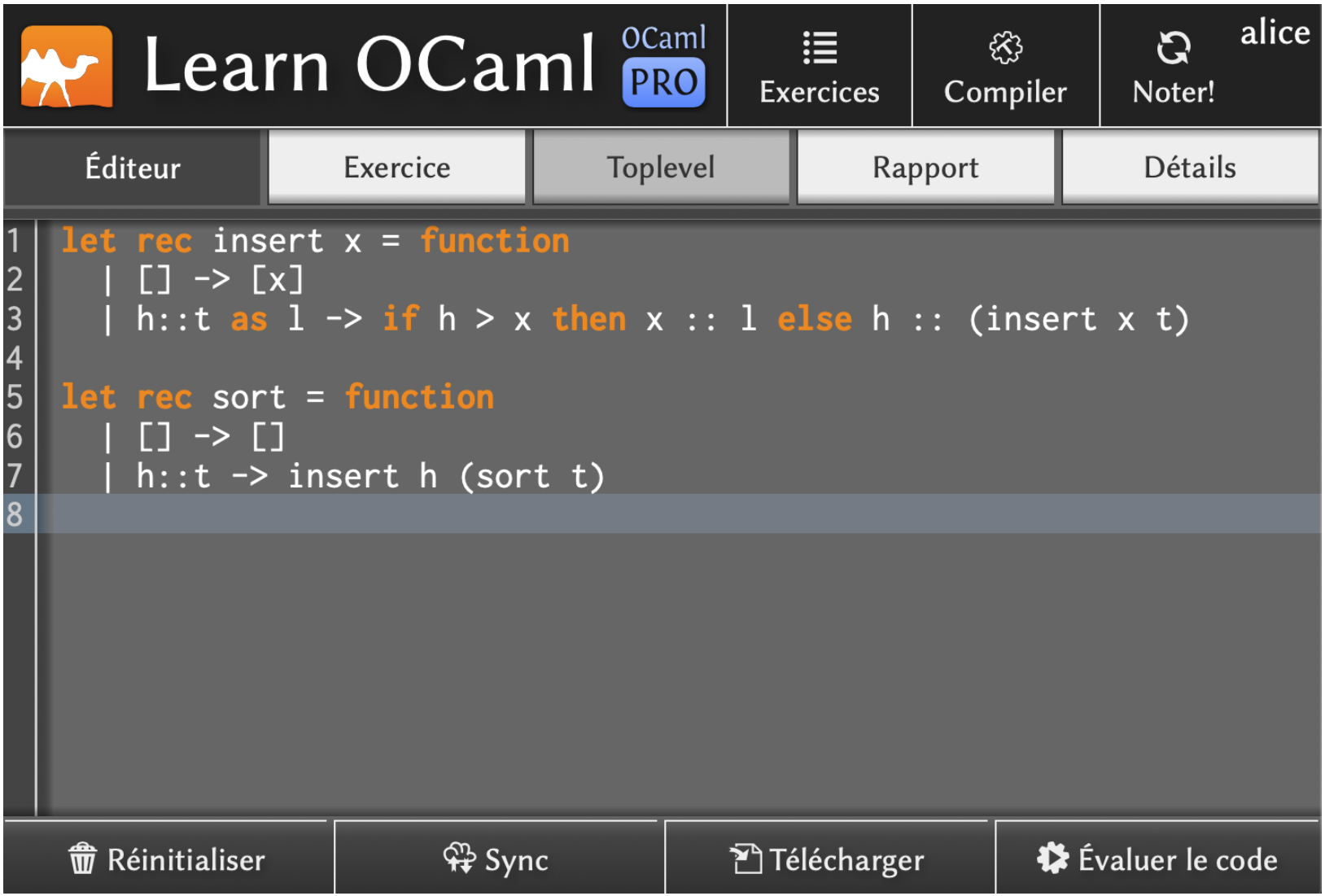
Scénario 1 : un tri d'entiers en O(n)



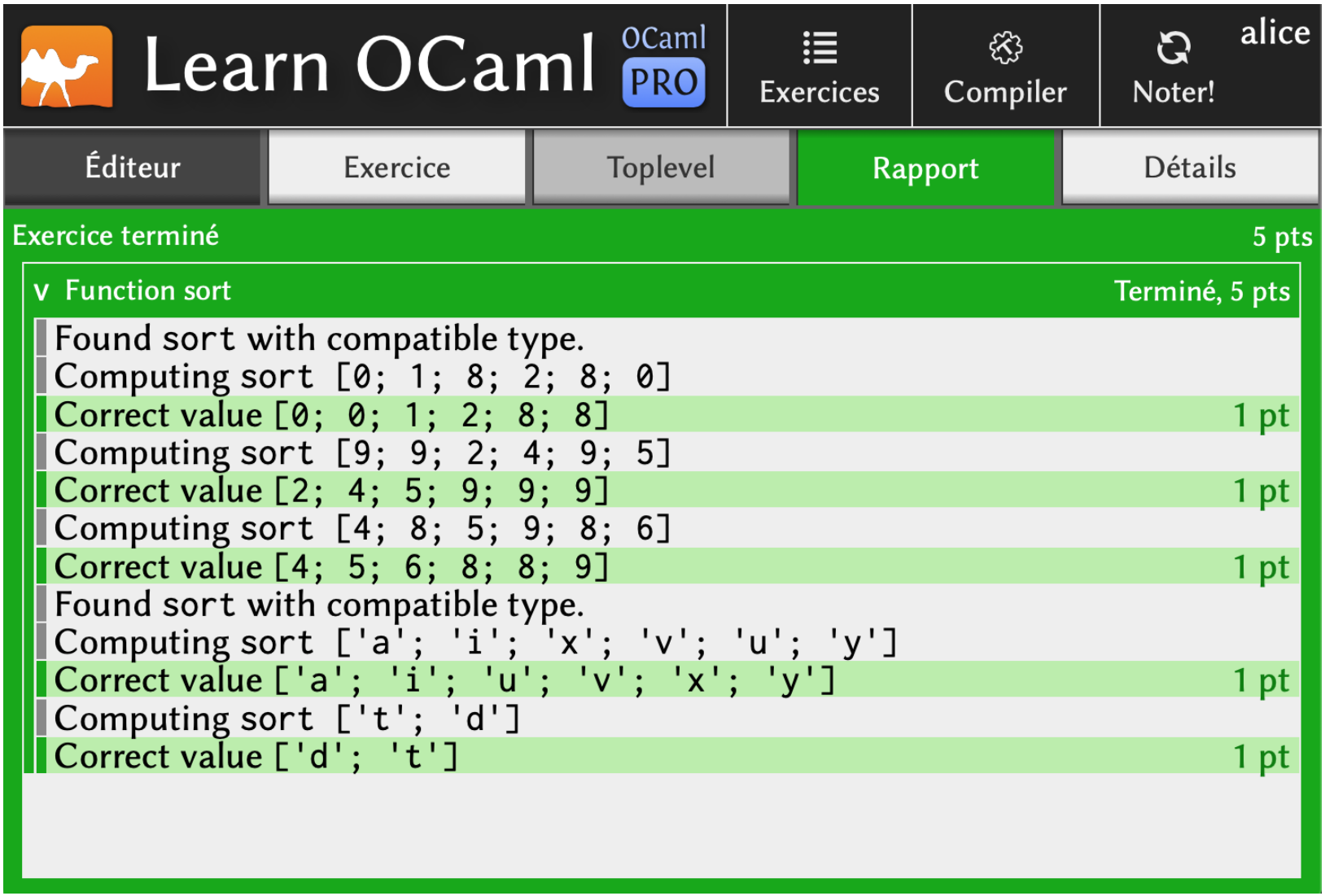
Scénario 1 : un tri d'entiers en O(n)



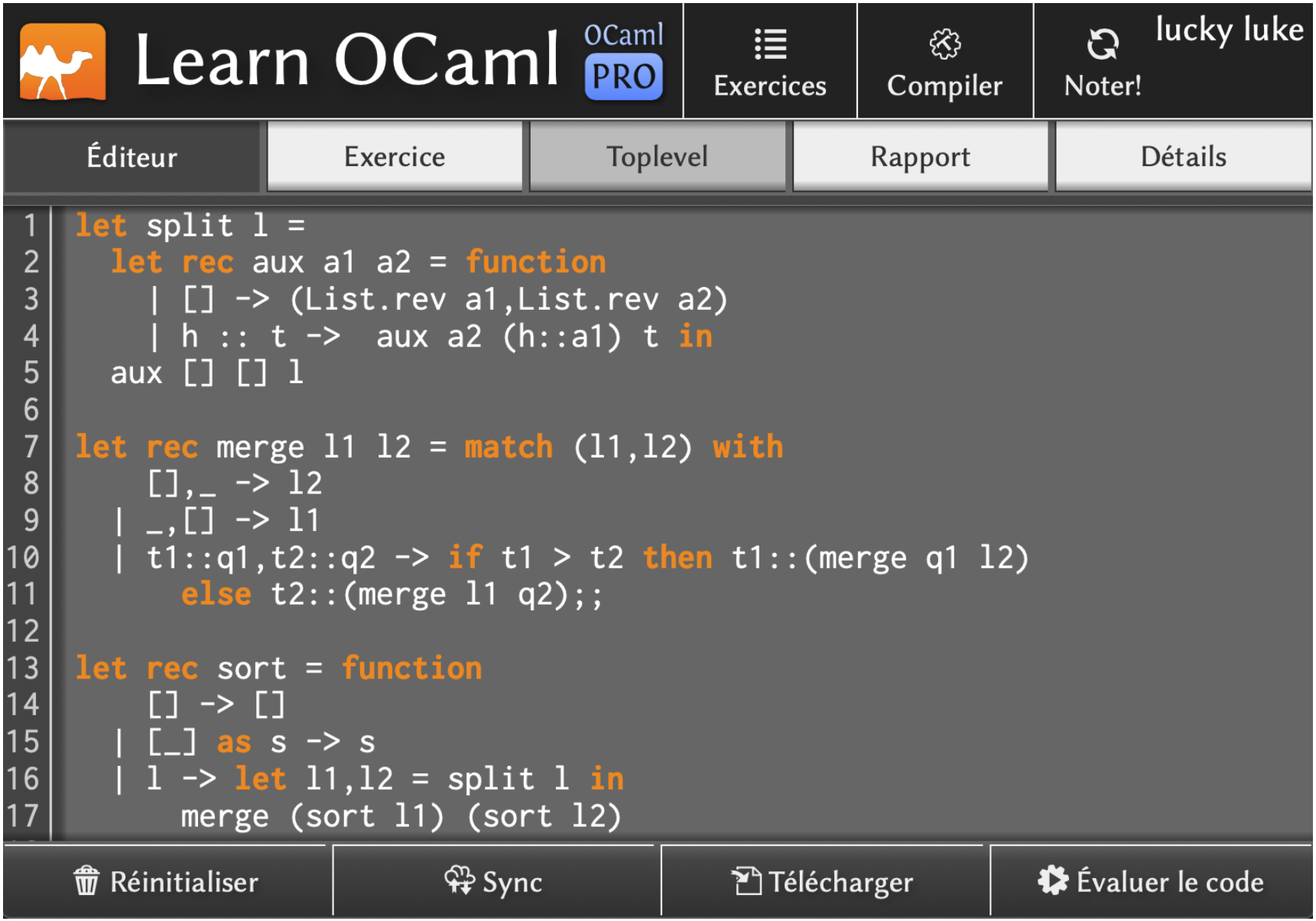
Scénario 2 : un tri par insertion



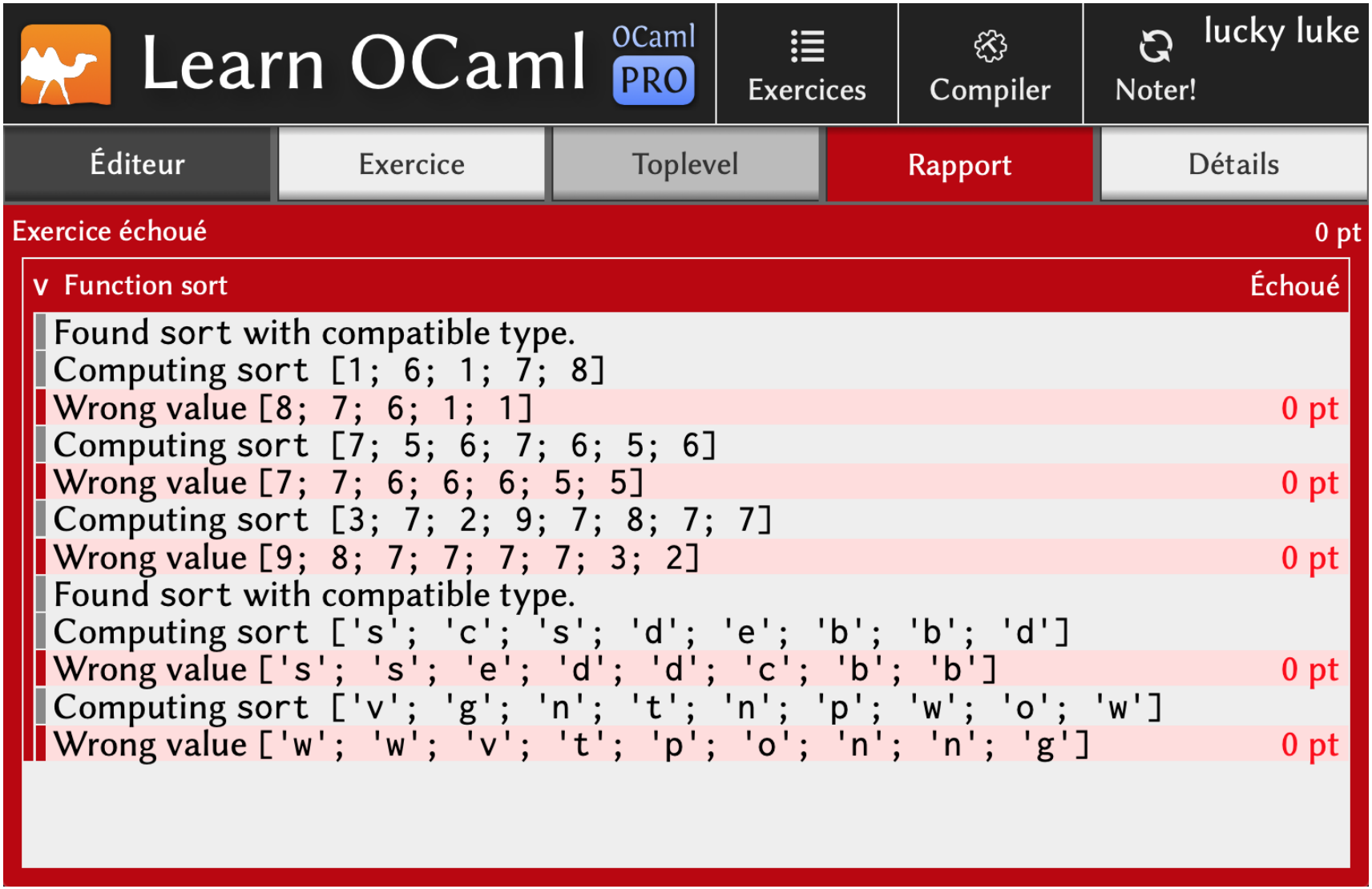
Scénario 2 : un tri par insertion



Scénario 3 : un tri fusion



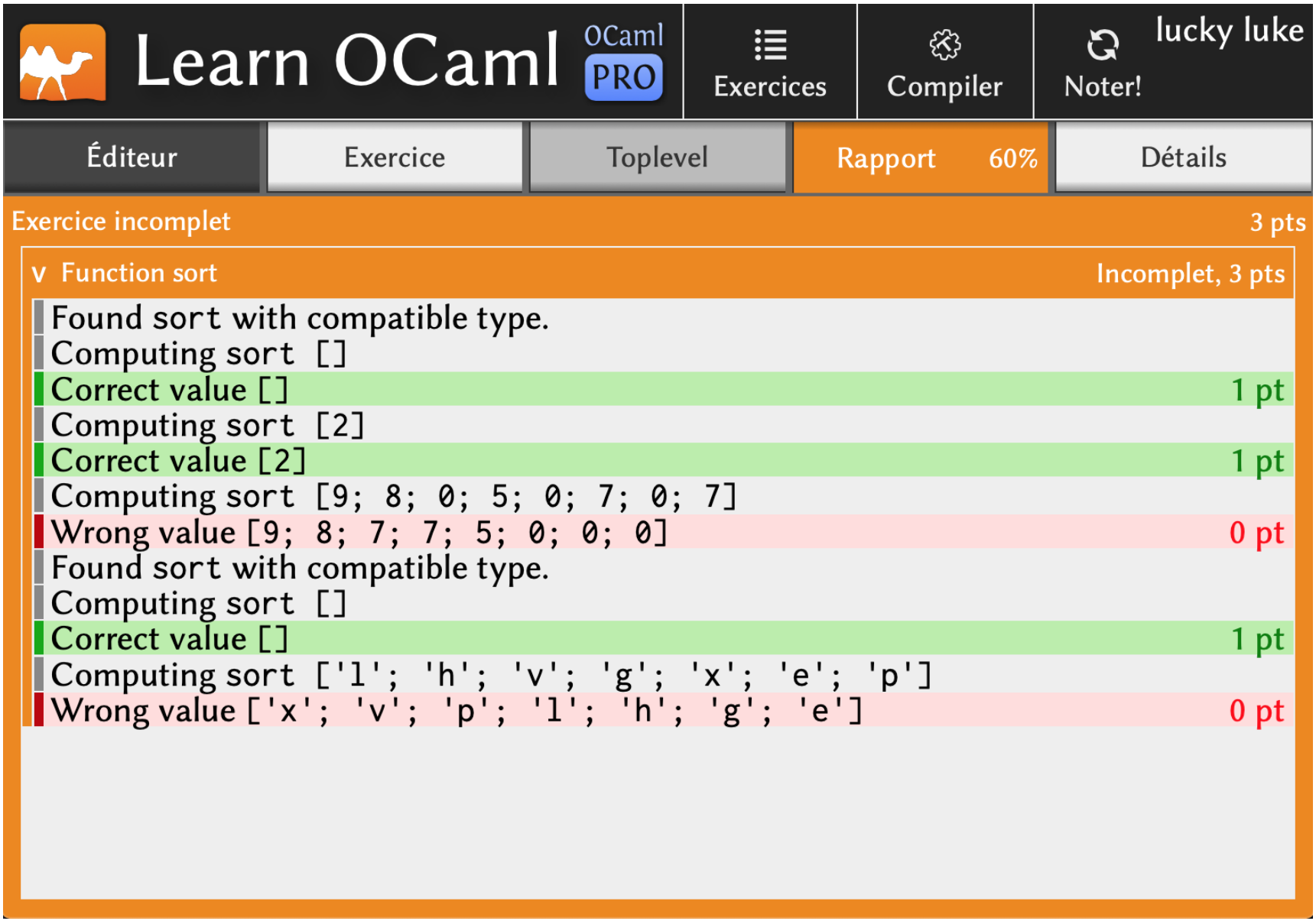
Scénario 3 : un tri fusion



- **Question : Où est l'erreur de Lucky Luke ?**

Scénario 3 : un tri fusion

- **Lucky Luke relance la correction :**



Partie III : Des améliorations

- séparation et réutilisabilité du code de test
- gestion du polymorphisme
- mise en place d'un nouveau schémas de correction

Séparation du code de test : *Peano revisité*

▷ **descr.md**

```
# Peano
Définir la fonction
of_int : int → peano
```

▷ **prelude.ml**

```
type peano = Z | S of peano
```

▷ **prepare.ml**

```
exception Undefined
let ⊥ _ = raise Undefined
```

▷ **template.ml**

```
let of_int = ⊥
```

▷ **solution.ml**

```
let rec of_int = function
  | 0 → Z
  | n → S (of_int (n-1))
```

▷ **test.ml**

```
let make_report () =
     Make_report.test_arity1
        "of_int"
        [%ty : int -> peano]

Set_result.set make_report
```

Séparation du code de test : *Peano revisité*

▷ **descr.md**

```
# Peano
Définir la fonction
of_int : int → peano
```

▷ **prelude.ml**

```
type peano = Z | S of peano
```

▷ **prepare.ml**

```
exception Undefined
let ⊥ _ = raise Undefined
```

▷ **template.ml**

```
let of_int = ⊥
```

▷ **solution.ml**

```
let rec of_int = function
  | 0 → Z
  | n → S (of_int (n-1))
```

▷ **depend.txt** [optionnel]

```
../../../libs/set_result.mli,
../../../libs/set_result.ml,
../../../libs/make_report.ml
```

▷ **test.ml**

```
let make_report () =
     Make_report.test_arity1
        "of_int"
        [%ty : int -> peano]

Set_result.set make_report
```

Séparation du code de test

Effet sur l'environnement de correction

```
⟨prelude.ml⟩ ;;
⟨prepare.ml⟩ ;;

module Code = struct ⟨le code de l'étudiant⟩ end ;;

module Solution = struct ⟨solution.ml⟩ end ;;

module Introspection = ... ;;
let code_ast = ... ;;

let results = ref None ;;
let timeout = ... ;;

module Test_lib = ... ;;
module Report = ... ;;

module Set_result : sig ⟨set_result.mli⟩ end = struct ⟨set_result.ml⟩ end
module Make_report = struct ⟨make_report.ml⟩ end ;;
                    (* Make_report peut dépendre de Set_result *)

⟨test.ml⟩ ;;
```

Fonctions polymorphes : sur la piste des types abstraits

▷ fichiers séparés **set_results.ml** et **variables.ml** réutilisables

```
1  let set make_report =
2    let report =
3      Test_lib.ast_sanity_check
4                  code_ast
5                  make_report in
6   Test_lib.set_result report
```

```
1  module type PARAM = sig
2     type t
3     val sample : unit → t
4  end
5
6  module Int = struct
7     type t = int
8     let sample = Random.bits
9  end
```

▷ le code de test (**test.ml**)

```
1  module A = (Variables.Int : Variables.PARAM)
2
3  let make_report () =
4    [ (Report.section "Function sort"
5      Test_lib.test_function_1_against_solution
6                [%ty: A.t list -> A.t list] "sort"
7                ~sampler:(Test_lib.sample_list ~sorted:false A.sample)
8                ~gen:5
9                [] ]
10
11 let () = Set_result.set make_report
```

J'ouvre une parenthèse : l'introspection en Learn-OCaml

L'accès aux valeurs

```
module I = Introspection

match I.get_value "Code.v" [%ty : τ] with
| Absent → e1
| Incompatible message → e2
| Present v → e3
```

Remarque : τ ne peut pas contenir de variable de types.

J'ouvre une parenthèse : l'introspection en Learn-OCaml

L'accès aux modules

- ▷ par une contrainte de signature
- ▷ fournit un module empaqueté dans une valeur

```
match I.get_value "Code.M" [%ty : (module S)] with
| Absent → e1
| Incompatible message → e2
| Present m → let module M = (val m : S) in e3
```

Remarque : `S` peut contenir des variables de types.

Je ferme la parenthèse.

La bonne manière de traiter les fonctions polymorphes

- ▷ définir une signature S = **sig val** f **:** τ **end**
- ▷ accéder au module `Code` (le code de l'étudiant)
  - ▶ par la contrainte de signature S
  - ▶ fournit un module empaqueté dans une valeur
- ▷ désempacter la valeur pour obtenir un module `Code_safe`
- ▷ accéder à la valeur f avec la notation qualifiée `Code_safe.f`
- ▷ tester une instance de `Code_safe.f`

La bonne manière de traiter les fonctions polymorphes

▷ Ne pas récupérer la valeur **sort** qui nous intéresse,
mais le module **Code** qui l'englobe !

```
1  module type S = sig
2     val sort : α list → α list
3  end
4
5  match I.get_value "Code" [%ty : (module S)] with
6  | Absent → e1
7  | Incompatible message → e2
8  | Present code → let module Code = (val code : S) in
9                   e3
```

Guider l'étudiant vers la solution

d'après François Pottier

`test_arity1 "f" [%ty: τ] Solution.f [x1; x2; ...; xn]`

- ▷ chercher une variable `Code.f` liée à une valeur compatible avec τ
- ▷ vérifier pour chaque sample x_i de $[x_1; x_2; \ldots; x_n]$:

```
(let v_i = Code.f x_i
 and w_i = Solution.f x_i in equal v_i w_i)
```

- ▷ trouver un contrexemple (x_i, v_i, w_i) et produire le rapport :

```
Found f with compatible type.
f is incorrect.
The following expression:
f x_i
produces the following result:
v_i
This is invalid. Producing the following result is valid:
w_i
```

Je propose un mécanisme de correction plus général

▷ tester des expressions fonctionnelles :

```
test_arity1  ~test:    (fun x → e1)
             ~expect:  (fun x → e2)
             ~samples: [x1;x2;...;xn]
```

▷ vérifier pour chaque sample x_i :

```
(let v_i = test x_i
 and w_i = expect x_i in equal v_i w_i)
```

▷ trouver un contrexemple (x_i, v_i, w_i) et produire le rapport :

```
The following expression:
(fun x → e1) x_i
produces the following result:
v_i
This is invalid. Producing the following result is valid:
w_i
```

Tester des expressions vis à vis d'autres expressions :

- ▷ permet d'exprimer des propriétés que le code doit vérifier.
  - ▶ montre clairement à l'étudiant ce qu'il doit corriger, par exemple

    `(fun n → odd (pred n)) ∼ even`

- ▷ permet de corriger des fonctions manipulant des types abstraits

```
module Code = sig
    type α t
    val map : (α → β) → α t → β t
end
```

  - ▶ `Solution.map` est incompatible avec `Code.map` !
  - ▶ Mais nul besoin de `Solution.map` :

    `(fun ℓ → map identity ℓ) ∼ (fun ℓ → ℓ)`
    `(fun ℓ → map f⁻¹ (map f ℓ)) ∼ (fun ℓ → ℓ)`

Préconditions/postconditions

▷ Idée : abstraire l'application

```
test_arity1  ~test:    (fun x → e1)
             ~expect:  (fun x → e2)
             ~apply:   (fun f x → f x)
             ~samples: [x1; x2; ...; xn]
```

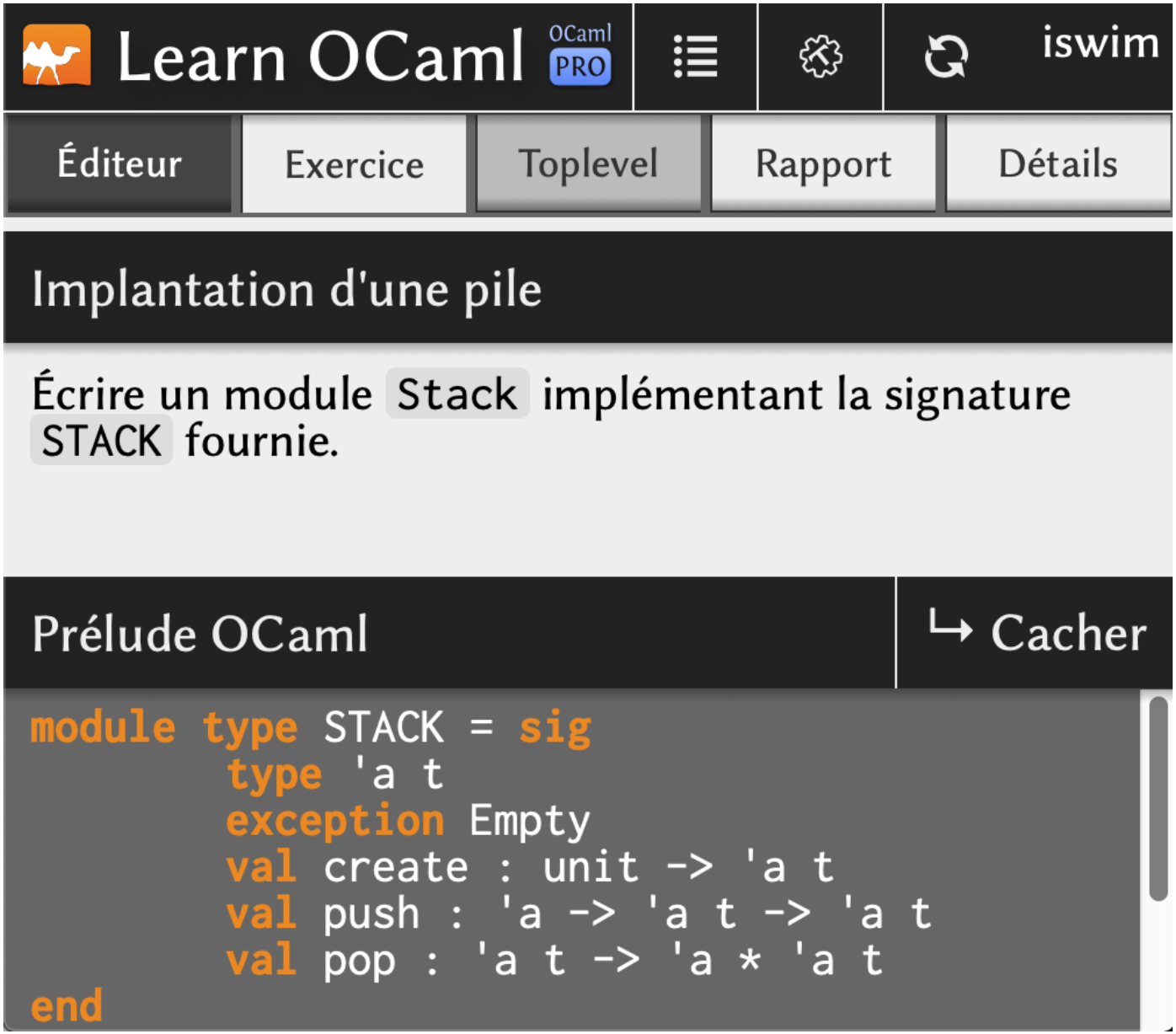
calcule pour chaque x_i :

```
(let vi = apply test xi
 and wi = expect xi in equal vi wi)
```

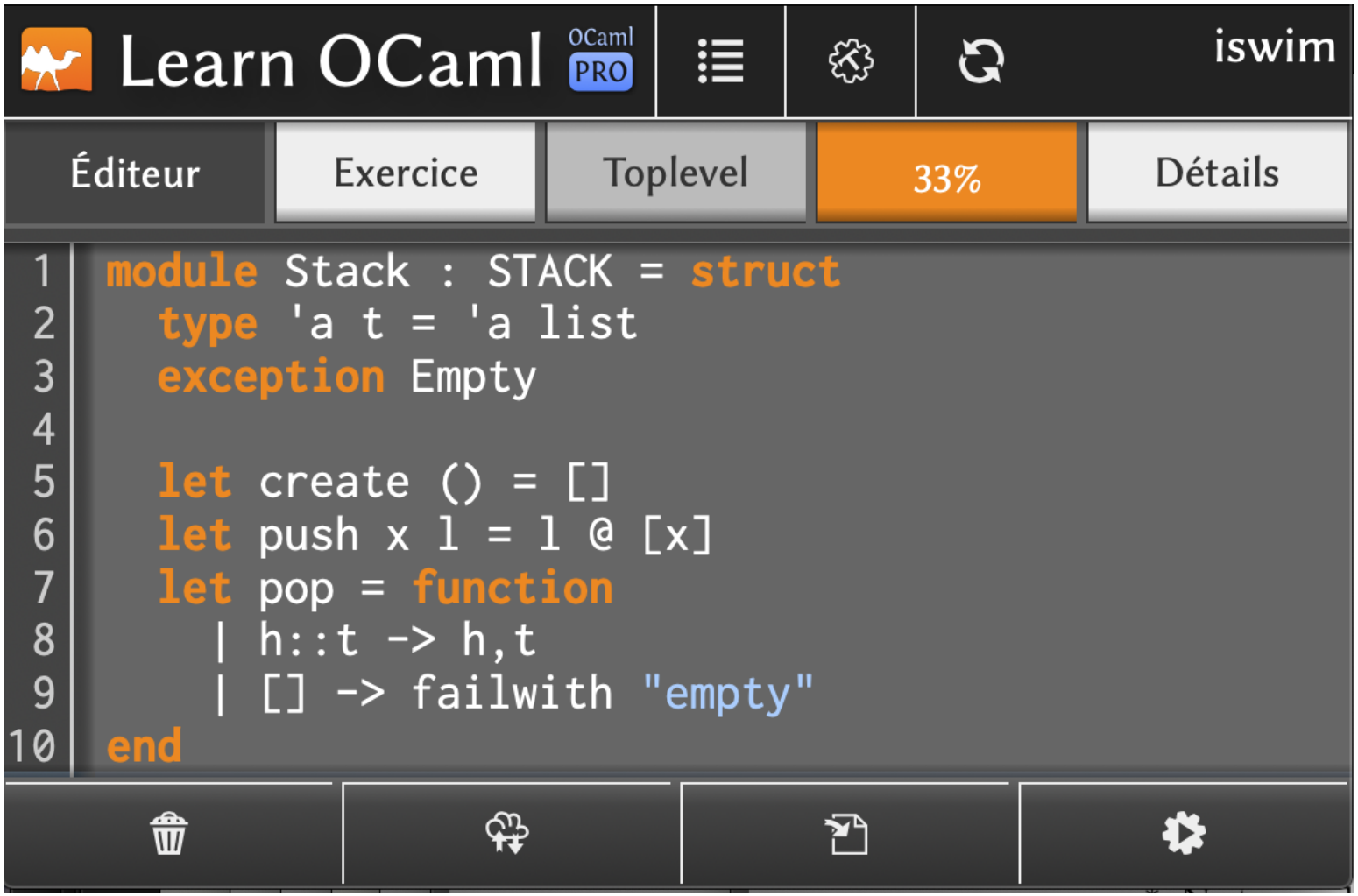
▷ Un exemple : les descripteurs de fichiers

```
let apply_ic f ic =
   let p = pos_in ic in
   let r = f ic in
   if pos_in ic <> p then (seek_in ic p ; e)
   else report r
```

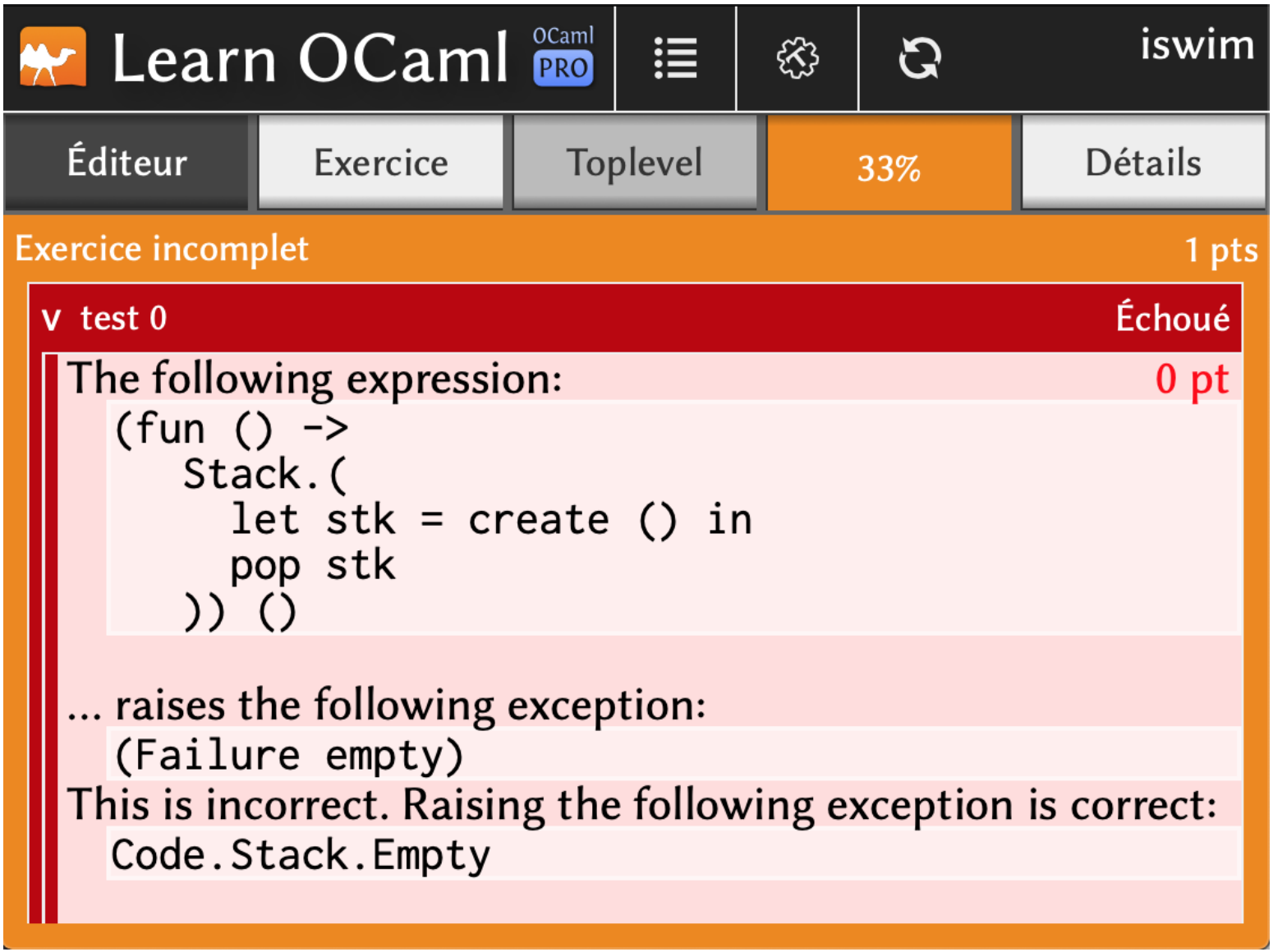
Exemple synthétique



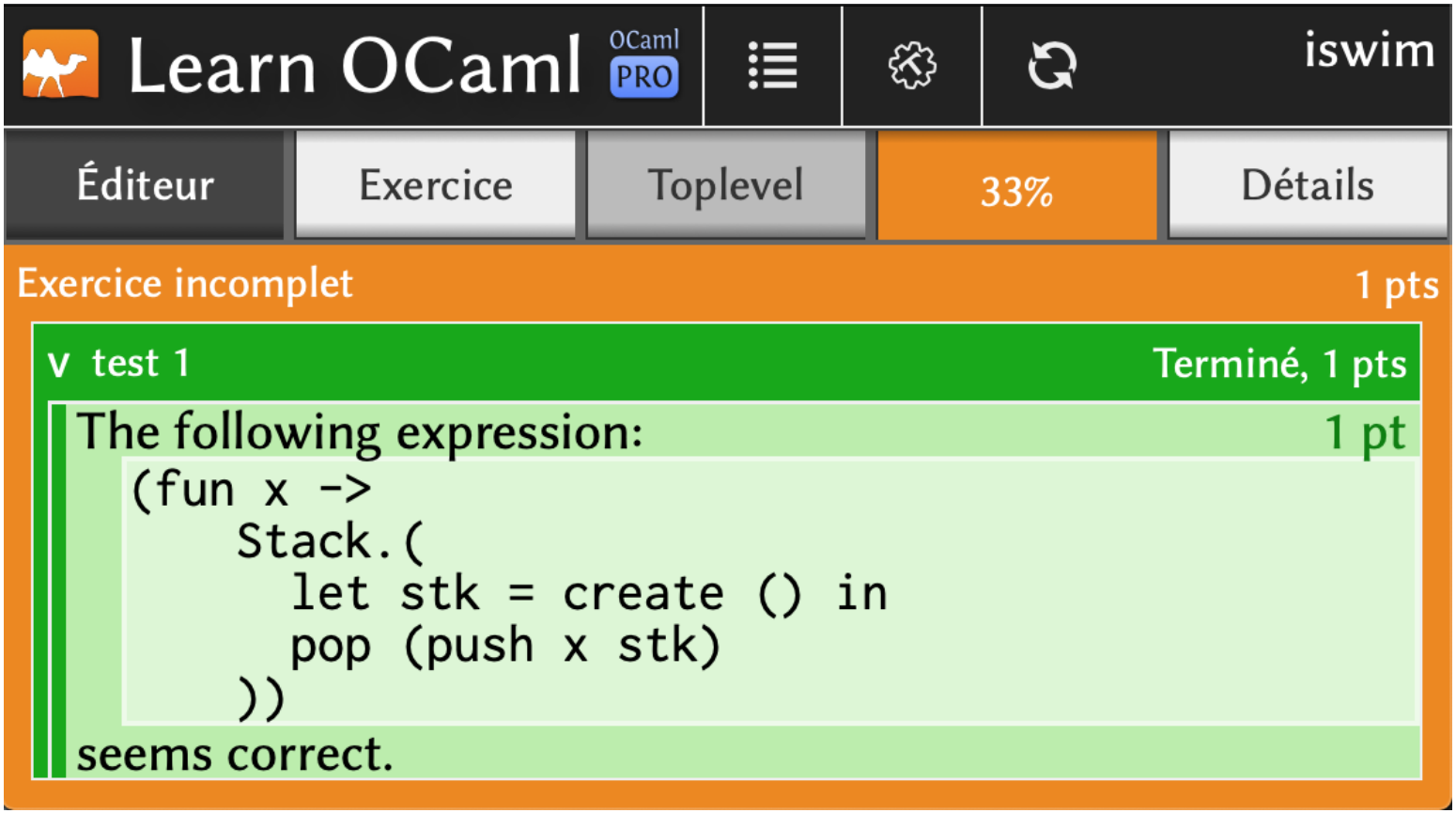
Une copie d'étudiant



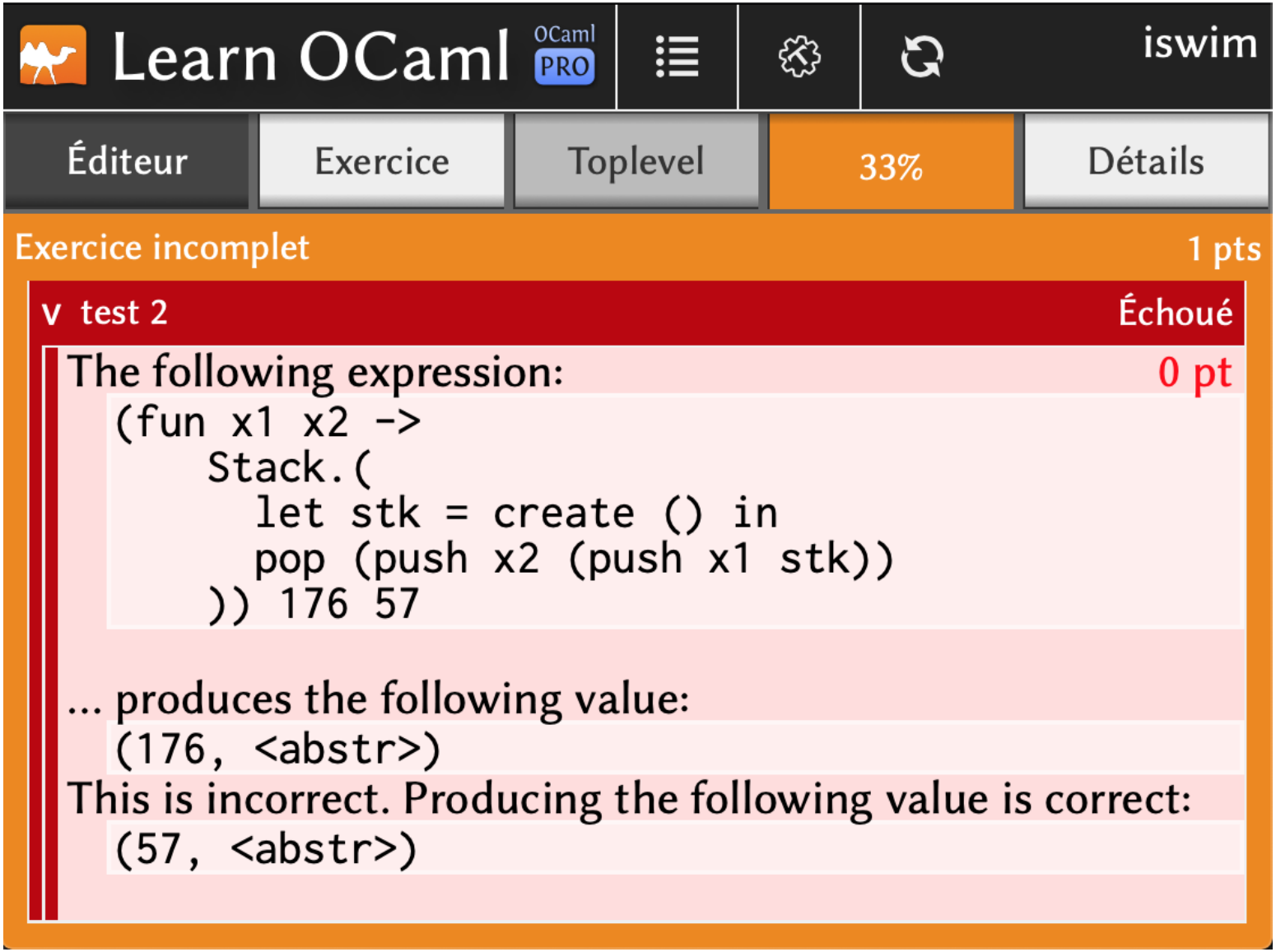
Le rapport de correction (0)



Le rapport de correction (1)



Le rapport de correction (2)



Le programme de test . . . d'abord, des dépendances !

```
        ../../../libs/printer.mli,
        ../../../libs/sampler.mli,
        ../../../libs/check_fun/protect.mli,
        ../../../libs/check_fun/fail_report.mli,
        ../../../libs/check_fun/lookup_value.mli,
        ../../../libs/check_fun/check_fun.mli,
        ../../../libs/check_fun/set_result.mli,

        ../../../libs/printer.ml,
        ../../../libs/sampler.ml,
        ../../../libs/check_fun/protect.ml,
        ../../../libs/check_fun/fail_report.ml,
        ../../../libs/check_fun/lookup_value.ml,
        ../../../libs/check_fun/check_fun.ml,
        ../../../libs/check_fun/set_result.ml
```

- ▷ **L'algorithmique sous-jacente est structurée en modules séparées réutilisables**

Le programme de test (*test.ml*)

```
1  let test_stack () =
2    let stk = Lookup_value.get "Code.Stack" [%ty: (module STACK)]
3    in
4    let module Stack = (val stk : STACK) in
5
6    let test0 () =
7      Check_fun.arity1
8        ~source:  "(fun () ->                    \n\
9                   \   Stack.(                   \n\
10                  \     let stk = create () in \n\
11                  \     pop stk                 \n\
12                  \   ))"
13       ~test:    (fun () → Stack.(pop (create ())))
14       ~expect:  (fun () → raise Stack.Empty)
15       ~samples: [();()]
16       ~view:    Printer.(unit, (pair int abstract))
17       () in
```

- ▷ la chaîne aux lignes 7-11 n'est pas interprétée : c'est un simple affichage.
  - ▶ L'affichage des valeurs fonctionnelles est une question à développer.

Le programme de test (*test.ml*)

```
17  let test1 () =
18    Check_fun.arity1
19      ~source:  "(fun x ->                        \n\
20                 \    Stack.(                     \n\
21                 \      let stk = create () in \n\
22                 \      pop (push x stk)         \n\
23                 \    ))"
24      ~test:    (fun x → Stack.(pop (push x (create ()))))
25      ~expect:  (fun x → x,Stack.create ())
26      ~seed:    Sampler.int
27      ~view:    Printer.(int,abstract)
28      ()
```

Le programme de test (*test.ml*)

```
29  let test2 () =
30   Check_fun.arity2
31     ~source:  "(fun x1 x2 ->                    \n\
32                \    Stack.(                     \n\
33                \      let stk = create () in    \n\
34                \      pop (push x2 (push x1 stk))  \n\
35                \    ))"
36     ~test:    (fun x1 x2 → let stk = create () in
37                            Stack.(pop (push x2 (push x1 stk))))
38     ~expect:  (fun x1 x2 → (x2,Stack.(push x1 (create ()))))
39     ~seed:    Sampler.(int, int)
40     ~view:    Printer.(int, int, pair int abstract)
41     ()
42
43  let make_report () =
44     [test0; test1; test2]
45
46  let () =
47    Set_result.set make_report
```

Conclusion

Learn-OCaml

- ▷ Facile à utiliser, modulo des améliorations

Améliorations proposées

- ▷ séparation du code de test, pour faciliter le travail de l'enseignant.
- ▷ nouvelles techniques de correction, avec un double objectif :
  - apporter à l'étudiant une correction interactive et de qualité ;
  - permettre à l'enseignant de porter des contenus pédagogiques avancés en OCaml.
    → support pour la correction de programmes modulaires avec types abstraits (projets de L3 / M1)

Quelques pistes

- ▷ support pour l'interopérabilité avec `JavaScript`
- ▷ écriture de tests par l'étudiant (cf. `Mr Scheme` et `Mr Python`)


Benjamin Canou, Roberto Di Cosmo, Grégoire Henry.
Sous le capot du MOOC OCaml.
Journées Francophones des Langages Applicatifs (JFLA 2016), Jan 2016, Saint-Malo, France. hal-01333599


Benjamin Canou, Grégoire Henry, Çagdas Bozman, Fabrice Le Fessant.
Learn OCaml, An Online Learning Center for OCaml.
OCaml Users and Developers Workshop 2016, Sep 2016, Nara, Japan. hal-01352015


Benjamin Canou, Cagdas Bozman, Grégoire Henry.
Scaling up functional programming education : under the hood of the OCaml MOOC.
Proc. ACM Program. Lang. 1, ICFP, Article 4 (September 2017)


Bozman, Canou, Di Cosmo, Couderc, Gesbert, Henry, Le Fessant, Mauny, Morel, Peyrot, Régis-Gianas.
Learn-OCaml : un assistant à l'enseignement d'OCaml
Journées Francophones des Langages Applicatifs (JFLA), Jan 2019, Les Rousses, France. hal-01962838