Génie Logiciel Avancé

Léo Andrès

23 Janvier 2026

Qui suis-je ?

- thèse (2024) : compilation d'OCaml vers Wasm et exécution symbolique
- ingénieur chez OCamlPro : https://github.com/ocamlpro/owi
- première fois que je donne ce cours, commentaires bienvenus

Si vous avez aimé le cours, stage possible l'année prochaine avec Saïd Zuhair et moi @ OCamlPro.

Informations pratique

- 12 séances de cours de 2h, 12 séances de projet de 2h
- Giovanni Bernardi pour les séances de projet
- j'ai écrit le sujet : c'est moi qu'il faut embêter par mail, pas lui
- je lis parfois les mails envoyés à leo+gla@kumikode.org
- pas d'examen

Merci de m'envoyer un mail aujourd'hui pour que je puisse vous contacter par la suite. J'ai rédigé un cours, je ne pourrai pas vous l'envoyer autrement.

À propos du cours

- beaucoup de notions présentées
- assez peu de formalisme mathématique
- illustrations par la programmation et les outils
- principalement OCaml, mais transposable
- projet ambitieux mais qui devrait vous apprendre beaucoup de choses
- écouter attentivement + poser questions : suffisant pour réussir

Si vous êtes complètement perdus au niveau du cours/projet, contactez-moi, je suis disponible pour vous aider.

À propos du projet

- je n'ai pas fini d'écrire le sujet, ce sera fait ce week-end™
- teaser :
  - OCaml
  - Wasm
  - jeu de la vie
  - TUI
  - interface graphique
  - exécution symbolique
  - solver-aided programming

Les modalités d'évaluation vous seront présentées une fois qu'on en aura discuté avec Giovanni.

Plan du cours (sujet à modification)

- test manuels, unitaires, d'intégration
- couverture du code, graphe d'appel et graphes de flot de contrôles
- critères de couverture formels
- mutation testing, property-testing (quickcheck), RAC
- fuzzing (boîte noire/grise/blanche) (Radamsa, AFL++, libFuzzer)
- exécution symbolique et concolique
- bounded model checking
- debug et benchmarks
- déploiement, CI, reproductibilité, packaging, NixOS
- sécurité
- langages de spécification
- interprétation abstraite, vérification déductive

Introduction

Le Génie Logiciel

Faire au mieux afin qu'un logiciel fasse ce qu'on attend de lui en ayant une consommation de ressources acceptables.

Ce qu'on attend de lui :

1. pas d'erreur à l'exécution
2. respecte sa spécification

Consommation de ressources acceptables :

1. temps et mémoire
2. parfois d'autres métriques (consommation électrique, temps de démarrage, nombre de connexions réseau, de fichiers)

Méthodes de détection d'erreurs

Plusieurs méthodes pour vérifier que le logiciel fait ce qu'on attend :

1. le test (fuzzing, exécution symbolique, model checking…)
2. l'interprétation abstraite
3. la preuve de programme (preuve interactive, vérification déductive)

Propriétés des différentes méthodes

Se différencient par ces propriétés :

1. La complétude : la méthode ne produit pas de fausses alarmes (faux positifs) dues à des sur-approximations.
2. La correction : la méthode ne rate pas de véritables erreurs (faux négatifs) du fait de sous-approximations.
3. L'automatisation : la méthode permet d'obtenir des résultats automatiquement, sans intervention humaine.

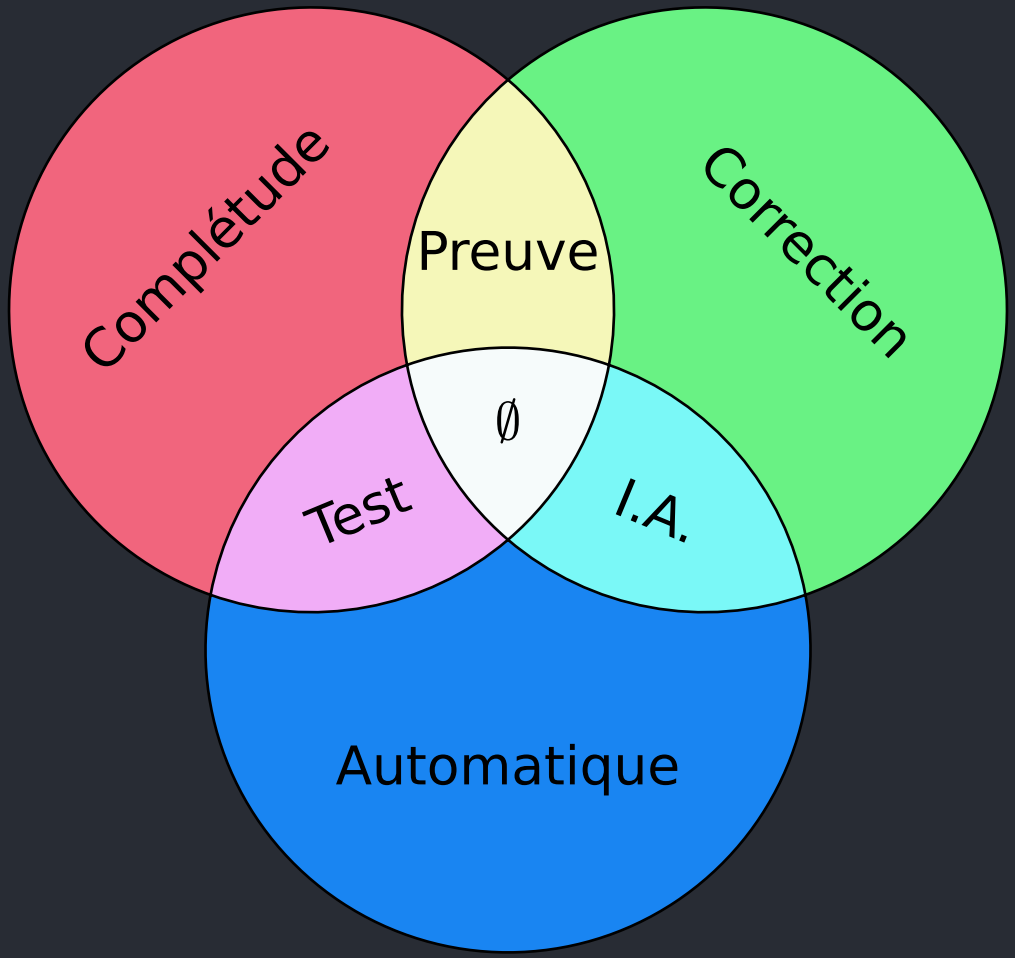
Théorèmes de Rice et Gödel

Impossible pour une méthode d'analyse d'avoir les trois propriétés à la fois. On peut au mieux en avoir deux. C'est de là que viennent ces trois familles :

1. le test est automatique et complet, pas correct
2. l'interprétation abstraite est automatique et correcte, pas complète
3. la preuve de programme est complète et correcte, pas automatique

Chacune essaie d'être la moins mauvaise possible sur la propriété qui lui manque.

Illustration des méthodes et propriétés



Preuve de programme : vérification déductive

- correcte et complète, repose sur l'intervention humaine
- on écrit le programme, la spécification des invariants (boucles, types) et des variants (terminaison)
- formule générée est envoyée à un prouveur automatique
- si le prouveur échoue, soit :
  - le programme ne respecte pas sa spécification (contre-exemple parfois)
  - la spécification n'est pas la bonne
  - il manque des variants/invariants pour aider le solveur

Exemple d'outils : Why3, Boogie, Viper.

Preuve de programme : preuve interactive

S’effectue au travers d’un assistant de preuve qui guide l’utilisateur dans la preuve manuelle d’un théorème qu’il a lui-même écrit.

- la preuve se fait via des tactiques qu’on doit combiner
- exemple : appliquer un autre théorème, utilisation d’une hypothèse
- possible d’automatiser une partie via des solveurs

Exemple d’outils : Rocq, HOL, Isabelle, Lean.

Interprétation abstraite

La méthode construit automatiquement une sur-approximation des états possibles du programmes en utilisant des domaines abstraits. Cela lui permet de générer des invariants qui garantissent certaines propriétés.

- si une propriété ne peut pas être garantie par les invariants générés, une alarme est levée
- ce n'est peut-être pas une vraie erreur, l'utilisateur doit faire le tri

Exemple d'outils : Astrée, Mopsa.

Test

Trop de techniques différentes pour être décrites en une diapositive. Mais c'est tout l'objet de ce cours.

- méthodes automatiques et complètes
- pas correctes en général : par exemple, peuvent ne pas terminer
- possible de garantir la correction sous-condition (e.g. l'analyse termine)
- adapté pour faire du bug-finding

Toutes les entrées ?

Les différentes techniques sont souvent des variations sur l'idée :
« essayer le programme avec toutes les entrées possibles ».

- compliqué : il y en a énormément, voire une infinité
- on va essayer de dénombrer les entrées possibles
- chercher des ordres de grandeur sur le temps d'énumération
- objectif : développer des intuitions sur ce qui est réaliste ou non

On va s'intéresser au cas particulier des fonctions, un programme pouvant être vu comme une fonction.

Type et habitants

Soit t un type. On note $|t|$ le nombre d'habitants du type t. Il s'agit du nombre de valeurs distinctes de type t. Quelques cas simples :

t	Valeurs	$\|t\|$
`type t = |`	$\emptyset$	0
`unit`	`()`	1
`bool`	`false`, `true`	2
`char`	..., `'a'`, `'b'`, `'c'`, ...	256
`int`	..., `-2`, `-1`, `0`, `1`, `2`, ...	$2^{63} - 1$

On peut construire des types plus complexes en composant les types simples.

Types produits

On peut composer deux types en formant leur produit, noté : $t_1 \times t_2$.

On a alors $|t_1 \times t_2| = |t_1| \times |t_2|$.

En OCaml, on note le produit de deux types `t1` et `t2` ainsi : `t1 * t2`.

t	Valeurs	$\lvert t \rvert$
`bool * char`	`(true, 'a'), (false, '\n'), ...`	$2 \times 256 = 512$
`int * int`	`(0, 1), (3, -2), ...`	$(2^{63} - 1) \times (2^{63} - 1)$

Types produits : enregistrements

Les enregistrements sont aussi des types produits, par exemple :

```
type t = {
  x : int;
  y : int;
}
```

Le type `t` est équivalent à `int * int`. Ses champs sont nommés, mais il permet de représenter le même ensemble de valeurs.

Types sommes

On peut composer deux types en formant leur somme, notée : $t_1 + t_2$.

On a alors $|t_1 + t_2| = |t_1| + |t_2|$.

En OCaml, on peut définir `int` $+$ `float` ainsi :

```
type t =
  | A of int
  | B of float
```

Il a comme valeurs possibles :

```
...; A ~-2; A ~-1; A 0; A 1; A 2; ...
...; B nan; B 0.42; B 12.34; B 100.42; ...
```

Type sommes : constructeurs constants

Un constructeur constant n'a qu'un seul habitant, par exemple :

```
type t =
  | A
  | B of int
```

Ici, `A` n'a qu'une seule valeur possible : `A`. La définition précédente est donc équivalente à :

```
type t =
  | A of unit
  | B of int
```

Et on peut donc dénoter ce type par `unit` + `int`.

Type sommes : exemples

t	Valeurs	$\|t\|$
`bool Option.t`	`None`, `Some false`, `Some true`	3
`bool List.t`	`[]`, `[false]`, `[true]`, `[false; false]`, `[false; true]`, `[true; false]`, `[true; true]`, ...	∞

Types de fonctions

On ne considère ici que les fonctions pures, i.e. sans effet de bord et qui n'échouent pas.

Soient t_1 et t_2 deux types. On note $|t_1 \longrightarrow t_2|$ le nombre de fonctions de t_1 vers t_2.

On considère les fonctions d'un point de vue sémantique et des valeurs observables. On se fiche de la façon dont elles sont implémentées. Par exemple, on ne considère pas ces fonctions comme différentes :

```
let and_1 x y = if x then y else false
let and_2 x y = x && y
let and_3 x y = if y then x else false
let and_4 x y = if x then if y then true else false else false
```

Elles calculent toutes « la même chose ».

Types de fonctions : premier exemple

Par exemple `bool -> bool` peut prendre les valeurs suivantes :

```
(* Constante `false` *)
let f1 = function
  | false -> false
  | true  -> false

(* Identité *)
let f2 = function
  | false -> false
  | true  -> true
```

```
(* Constante `true` *)
let f4 = function
  | false -> true
  | true  -> true

(* Négation *)
let f3 = function
  | false -> true
  | true  -> false
```

Types fonctions : second exemple

Regardons le type `unit -> bool`. Il peut prendre les valeurs :

```
let f1 () = false
```

```
let f2 () = true
```

Question : comment calculer $|t_1 \longrightarrow t_2|$?

Types fonctions : formule magique

- erreur courante : $|t_1 \longrightarrow t_2| = t_1 \times t_2$

Ne faites pas ça !

- réponse correcte : $|t_1 \longrightarrow t_2| = |t_2|^{|t_1|}$

Exercice, combien d'habitants pour :

```
val f1 : bool -> bool
val f2 : unit -> bool Option.t
val f3 : bool -> int
val f4 : bool Option.t Option.t Option.t -> bool
val f5 : bool -> bool -> bool
```

Génération et énumération : entiers 32 bits

Combien de temps faut-il pour énumérer tous les entiers 32 bits ?
Considérons le programme `enum.ml` suivant :

```
let () =
  let start = Int32.to_int Int32.min_int in
  let stop = Int32.to_int Int32.max_int in
  for i = start to stop do
    Sys.opaque_identity ignore i;
  done
```

Génération et énumération : entiers 32 bits

On compiler et mesurer le temps d'exécution avec :

```
$ ocamlopt -O3 enum.ml
$ time ./a.out
./a.out  5.60s user 0.00s system 99% cpu 5.620 total
```

Combien de temps pour le même test sur des entiers 64 bits ?

Génération et énumération : entiers 64 bits

- avec un entier 64 bits, temps estimé à $2^{32} \times 5.6\text{s} \approx 760$ ans !
- la fonction testée ici est simplement un appel à `ignore`…

On comprend alors pourquoi espérer tester n'importe quel programme avec toutes ses entrées possibles est… irréaliste.

Essayer quand même ?

Et pourtant, il est possible en pratique de tester ses programmes d'une façon telle que l'ont peut espérer, sans essayer toutes les entrées possibles, détecter la grande majorité des erreurs et arriver à un résultat similaire à celui obtenu par énumération brutale et naïve.

Production manuelle de tests

Production manuelle de tests

- tests manuels
- tests unitaires
- tests d'intégration

Le minimum que vous devrez faire peu importe où vous allez travailler ensuite.

Test manuel

Le test manuel consiste à faire tester le logiciel par un humain afin que celui-ci vérifie que tout se passe comme prévu.

- utile en phase de prototypage pour le développeur (vérifier rapidement que tout a l'air OK)
- pertinent dans d'autres contexte : interaction humaine importante (jeu vidéo) ou environnement physique complexe à simuler (robot)
- très coûteux (en temps humain)
- potentiellement difficile à reproduire (bug dans un jeu vidéo après 4h de jeu)

La notion de boîte noire

Le test manuel est qualifié de test en boîte noire.

- se fait au travers d'interactions avec le logiciel sans « regarder à l'intérieur » de celui-ci
- on interagit avec le binaire exécutable, pas le code source de celui-ci
- on observe les résultats sans savoir comment le logiciel est implémenté

Test unitaire

À l'opposé du test manuel, se trouve le test unitaire.

- on écrit de nombreux tests (sous forme de programmes)
- chacun se concentre sur une petite partie du programme à tester
- peu coûteux en : exécution rapide et sans intervention humaine
- reproductible (on sait quelles entrées ont été utilisées)

La notion de boîte blanche

Le test unitaire est qualifié de test en boîte blanche.

- l'opposé des tests en boîte noire
- on a accès à « l'intérieur » du programme testé

Exemple de test unitaire

Bibliothèque OCaml `lib`. Contient un module `Math`. Contient une fonction `square`. Module dans le fichier `math.ml` :

```
(** The Math module. *)
let square x = x * x
```

La bibliothèque est construite avec le fichier `dune` suivant :

```
(library
  (public_name lib)
  (modules math))
```

Exemple de test unitaire

On appelle le programme qui appelle la fonction à tester un harnais de test. Voilà un fichier `test_math.ml` qui teste notre fonction sur une entrée donnée :

```
(** Tests for the Math module. *)

(* Test for the square function *)
let test_square () =
  let x = 3 in
  let result = Lib.Math.square x in
  let expected = 9 in
  assert (Int.equal result expected)

let () =
  test_square ();
  Format.printf "All tests are OK!"
```

Exécution des tests unitaires

On peut alors écrire le fichier `dune` suivant :

```
(test
  (name test_math)
  (modules test_math)
  (libraries lib))
```

On peut exécuter notre test au moyen de la commande `dune runtest` :

```
$ dune runtest
All tests are OK!
[0]
```

Exécution des tests uniaires

Comportement spécifique à `dune` :

- lancer la commande `dune runtest` plusieurs fois de suite ne va pas forcément relancer l'ensemble des tests
- afin d'économiser des ressources, `dune` est capable de se rappeler des tests qui ont réussi
- les relance seulement si l'une de leur dépendance a changé depuis la dernière exécution

Ici, l'unique dépendance est la bibliothèque `lib`. Possible de forcer `dune` à relancer les tests avec l'option `--force`.

Exécution des tests unitaires

Si l'on introduit volontairement une erreur dans le programme ou dans le harnais de test (par exemple en remplaçant `9` par `6`), on aura une erreur :

```
$ dune runtest
File "dune", line 2, characters 8-17:
6 |   (name test_math)
           ^^^^^^^^^
Fatal error: exception Assert_failure("test_math.ml", 8, 2)
```

Ce n'est pas terrible…

Bibliothèque pour les tests unitaires

- aident à l'écriture de tests en fournissant des primitives
- pour comparer les résultats obtenus à ceux attendus
- pour gérer des fonctions particulières (e.g. qui lèvent des exceptions)
- afficher le résultat de manière plus lisible

Alcotest

En OCaml, la bibliothèque la plus répandue est `alcotest`. Voilà ce qu'on obtient quand on lance `dune runtest` :

```
Testing `Synchronizer'.
This run has ID `RLSUYDMJ'.

  [OK]          Basic operations            0   Empty queue no pledges.
  [OK]          Basic operations            1   Get single item.
  [OK]          Basic operations            2   Get multiple items.
  [OK]          Pledge mechanics            0   Manual pledge.
  [OK]          Pledge mechanics            1   Get creates pledge.
  [OK]          Pledge mechanics            2   Blocking on pledge.
  [OK]          work_while                  0   work_while simple.
  [OK]          work_while                  1   work_while empty.
  [OK]          Close functionality         0   Close returns None.
  [OK]          Close functionality         1   Close with items.
  [OK]          Concurrent operations       0   Producer/consumer.
  [OK]          Concurrent operations       1   Multiple workers.
  [OK]          Concurrent operations       2   Concurrent write/get.
  [OK]          Concurrent operations       3   Graph traversal.
  [OK]          Stress tests                0   Stress test.

Full test results in `synchronizer/_build/default/test/_build/_tests/Synchronizer'.
Test Successful in 0.018s. 15 tests run.
```

Alcotest

Plusieurs avantages :

- tests documentés via le groupe et la description
- si un test échoue, les autres sont lancés
- on peut sélectionner un sous-ensemble à lancer
- lorsqu'un test échoue, la différence valeur obtenue/attendue est affichée
- enregistre les sorties standard et d'erreur dans un fichier pour permettre d'investiguer

Alcotest : en cas d'erreur

```
Testing `Synchronizer'.
This run has ID `5MJ9FN0B'.

  [OK]          Basic operations              0   Empty queue no pledges.
> [FAIL]        Basic operations              1   Get single item.
  [OK]          Basic operations              2   Get multiple items.
  [OK]          Pledge mechanics              0   Manual pledge.
  [OK]          Pledge mechanics              1   Get creates pledge.
  [OK]          Pledge mechanics              2   Blocking on pledge.
  [OK]          work_while                    0   work_while simple.
  [OK]          work_while                    1   work_while empty.
  [OK]          Close functionality           0   Close returns None.
  [OK]          Close functionality           1   Close with items.
  [OK]          Concurrent operations         0   Producer/consumer.
  [OK]          Concurrent operations         1   Multiple workers.
  [OK]          Concurrent operations         2   Concurrent write/get.
  [OK]          Concurrent operations         3   Graph traversal.
  [OK]          Stress tests                  0   Stress test.

 [FAIL]         Basic operations              1   Get single item.

ASSERT get returns Some 42
FAIL get returns Some 42

   Expected: `Some 666'
   Received: `Some 42'

                synchronizer/_build/default/test/_build/_tests/Synchronizer/
Logs saved to `Basic operations.001.output                                  '.

Full test results in `synchronizer/_build/default/test/_build/_tests/Synchronizer'.
1 failure! in 0.018s. 15 tests run.
```

Alcotest : écriture d'un test

L'écriture d'un test se fait ainsi :

```
let test_get_single_item () =
  let result = (* call to the function being tested *) in
  Alcotest.(check (option int)) "get returns Some 42" (Some 42) result
```

Alcotest : groupe de tests

La déclaration d'un groupe de tests est une simple liste où l'on décrit chaque test :

```
let basic_tests =
  [ Alcotest.test_case "Empty queue no pledges" `Quick test_empty_queue_no_pledges
  ; Alcotest.test_case "Get single item"        `Quick test_get_single_item
  ; (* ... *) ]
```

Alcotest : run

Finalement, on peut lancer l'ensemble de nos groupes de tests de cette manière :

```
let () =
  Alcotest.run "Synchronizer"
    [ ("Basic operations", basic_tests)
    ; ("Pledge mechanics", pledge_tests)
    (* ... *)
    ]
```

Bibliothèque de tests

En résumé, rien de très sophistiqué, mais rendent le travail quotidien avec les tests unitaires plus agréable.

Tests d'intégration

Les tests d'intégration sont des tests où l'on considère le comportement d'une grosse portion du programme. C'est l'opposé des tests unitaires.

Instance particulièrement intéressante : les cram tests.

Cram tests

Les cram tests sont des tests d'intégration pour les outils en ligne de commande.

- à l'origine un programme particulier
- plusieurs variantes implémentées depuis
- désigne aujourd'hui la méthode qui en est issue plutôt que cet outil

En particulier, `dune` intègre nativement un système de cram tests.

Cram tests dans dune

- fichiers ayant l'extension `.t`.
- toutes les lignes qui ne sont pas indentées sont des commentaires
- lignes indentées de deux espaces sont des lignes significatives

L'idée consiste à écrire une série de commandes :

```
On peut appeler l'outil que l'on souhaite tester et sauvegarder sa sortie :
  $ ./my_amazing_tool.exe > output.txt
```

Les lignes significatives commençent par `$` sont des commandes qui vont être exécutées.

Cram tests dans dune

Les lignes significatives qui commencent par un autre caractère représentent en effet la sortie attendue des commandes exécutées. Par exemple, n peut vérifier que le fichier a bien été créé et que son contenu est le bon :

```
  $ ./my_amazing_tool.exe > output.txt
Vérifions le contenu du nouveau fichier :
  $ cat output.txt
  Hello from my_amazing_tool!
```

Cram tests dans dune

Si au cours du temps, la sortie produite par le programme est modifiée, voilà ce qu'on obtient avec `dune runtest` :

```
File "amazing.t", line 1, characters 0-0:
------ amazing.t
++++++ amazing.t.corrected
File "amazing.t", line 3, characters 0-1:
  $ ./my_amazing_tool.exe > output.txt
  $ cat output.txt
- Hello from my_amazing_tool!
+ Hello, world!
```

Cram tests dans dune

- si la différence semble conforme au nouveau comportement attendu : `dune promote`
- le fichier `amazing.t` sera automatiquement mis à jour
- extrêmement pratique pour écrire des nouveaux tests rapidement
- il suffit d'écrire les commandes que l'on veut, sans se préoccuper de la sortie

Cram tests dans dune

Par exemple, pour tester la sortie de la commande d'aide de `git`, on peut simplement écrire le fichier `git.t` suivant :

```
  $ git --help
```

On lance alors `dune runtest` directement.

Cram tests dans dune

```
File "git_help.t", line 1, characters 0-0:
------ git_help.t
++++++ git_help.t.corrected
File "git_help.t", line 2, characters 0-1:
  $ git --help
+ usage: git [-v | --version] [-h | --help] [-C <path>] [-c <name>=<value>]
+             [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
+
           [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--no-lazy-fetch]
+            [--no-optional-locks] [--no-advice] [--bare] [--git-dir=<path>]
+
             [--work-tree=<path>] [--namespace=<name>] [--config-env=<name>=<envvar>]
+             <command> [<args>]
+
+ These are common Git commands used in various situations:
+
+ start a working area (see also: git help tutorial)
+    clone      Clone a repository into a new directory
+    init       Create an empty Git repository or reinitialize an existing one
+
+ work on the current change (see also: git help everyday)
+    add        Add file contents to the index
+    mv         Move or rename a file, a directory, or a symlink
+    restore    Restore working tree files
+    rm         Remove files from the working tree and from the index
```

Cram tests dans dune

On lance alors `dune promote` et on obtient notre fichier de test complet.

Ce comportement est très utile lorsque la sortie d'un programme évolue au cours du temps :

- bien plus agréable de taper ces quelques commandes
- on inspecte et on confirme le résultat
- évite de devoir modifier à la main la sortie attendue par de nombreux tests unitaires manipulant des chaînes de caractères…

Bonus

Blablabla

blablabla