Projet PF5
Interprétation Abstraite : application au déplacement d'un robot dans le plan

20 octobre 2024

1 Prérequis

Contrairement aux TPs, ce projet ne vit pas sur Learn-OCaml. Avant toute chose, il vous faut donc installer le nécessaire pour travailler (compilation, tests). Pour ce faire, faites un **fork** (avec visibilité *privée*) du dépôt `git` suivant :

https://moule.informatique.univ-paris-diderot.fr/giovanni/projet-pf5-2425

(Pour quelques rappels sur l'utilisation de `git`, reportez-vous à la Section 6). Suivez les instructions données dans le fichier `README.md`. Une fois votre dépôt `git` créé, ajoutez votre binôme, ainsi que chacun des enseignants du cours, en choisissant pour chacun le rôle *Maintainer* : `allainc`, `bernardi`, `bucciare`, `padovani`, `ledent`, `kesner`,

Si vous rencontrez un problème, faites-nous signe au plus vite. Tant que cette étape d'installation n'est pas réalisée, vous ne pourrez pas faire le projet.

Dates de rendu du projet :

- le 20 novembre 2024 jusqu'à l'exercice 4.3 inclu,
- le 19 décembre 2024 pour le reste du projet et l'extension.

L'extension du projet sera ajoutée au sujet (sur le dépôt git) au plus tard le 8 novembre.

Critères de notation :

- Le projet devra être réalisé dans un style fonctionnel pur.
- Le code doit évidemment être correct, mais aussi *soigné* : le code doit être clair, commenté, les fonctions auxiliaires bien nommées.
- Veillez également à soigner l'utilisation de `git` : des commits réguliers, des messages de commit qui ont un sens (correction d'un bug, implémentation d'une nouvelle fonction, etc.).

- Le code doit être testé : nous avons fourni quelques tests basiques, mais ils ne sont pas suffisants pour s'assurer que votre code est correct.
- Travail en *binôme* : une bonne répartition du travail est importante. Les deux participants au projet doivent faire des commits sur `git` pour qu'on puisse savoir qui a fait quelle partie.
- **L'utilisation des IA de type LLM (ChatGPT, Copilot) est interdite** et sera punie d'un 0/20, au même titre que le plagiat.

2 Introduction

L'*Analyse Statique* est une technique d'étude des propriétés d'un programme informatique. Elle vise à détecter des erreurs et vulnérabilités sans avoir à exécuter le code[1], mais en analysant sa structure.

Cette technique est principalement utilisée dans des systèmes critiques (aéronautique, nucléaire, santé ...) où il n'est même pas toujours possible de tester les programmes : les échecs peuvent être catastrophiques ! On cherche donc à *certifier* que chaque programme du système est conforme à sa spécification. Plus généralement, tester un programme peut dans le meilleur de cas dénicher les bugs cherchés, mais ne pourra jamais garantir que le programme est correct : il faudrait pour cela tester *toutes* les entrées possibles, ce qui est rarement faisable en pratique.

Lorsqu'on effectue une batterie de tests sur un programme, deux résultats sont possibles :

(a) Un ou plusieurs tests échouent : le programme comporte des erreurs qu'il faudra corriger.

(b) Le programme passe tous les tests : il comporte peut-être encore des erreurs, mais le jeu de tests ne permet pas de les détecter.

L'*interprétation abstraite*[2] est une technique d'analyse statique qui consiste à calculer une *sur-approximation* de toutes les exécutions possibles d'un programme, et vérifier que cette approximation vérifie un certain ensemble de propriétés assurant sa correction [1]. Contrairement aux tests, si l'analyse réussit, on a la garantie que le programme est correct. En contrepartie, ce type d'analyse peut donner des faux-négatifs :

(a) Si l'analyse échoue, cela ne donne pas d'information : le programme est peut-être bel et bien correct, mais l'analyseur statique n'a pas réussi à le démontrer.

1. D'où l'adjectif « statique ».
2. https://en.wikipedia.org/wiki/Abstract_interpretation

(b) Si l'analyse réussit, cela garantit que le programme est correct : toutes les exécutions possibles du programme satisfont toutes les propriétés requises.

Dans l'industrie, plusieurs analyseurs statiques basés sur l'interprétation abstraite ont été implémentés. On peut citer par exemple `Astrée`[3], utilisé par Airbus pour ses avions, `Frama-C`[4], utilisé par EDF dans le secteur nucléaire, ou encore `Infer`[5], developpé et utilisé par Meta pour verifier le code de ses applis (Whatsapp, Instagram, ...).

Dans ce projet, nous allons étudier un langage de programmation très simple, qui permet de décrire les déplacements d'un robot dans le plan. Ce langage comporte deux déplacements de base, translation et rotation ; un opérateur permet de répéter un certain nombre de fois une suite de déplacements ; enfin, un opérateur de choix non-déterministe permet de choisir au hasard une suite de déplacements parmi deux suites possibles.

Notre objectif sera dans un premier temps de simuler l'exécution d'un tel programme. Dans un second temps, nous nous servirons de l'interprétation abstraite pour vérifier la propriété suivante : *étant donné un programme et une zone à atteindre, dans toutes les exécutions possibles du programme, le robot atteint toujours la zone cible.*

Remarques. Le niveau de difficulté des exercices est indiqué par zéro (facile) une étoile ⋆ (difficulté moyenne) ou deux étoiles ⋆⋆ (plus difficile). Pour certaines questions, il est fortement conseillé d'écrire une ou plusieurs fonctions auxiliaires ; auquel cas, donnez-leur un nom pertinent. N'hésitez pas à poster des questions (et non des réponses !) sur le sujet en écrivant à la liste de diffusion de PF5 :

`https://listes.u-paris.fr/wws/subscribe/l3.pf5.info`

3 Échauffement : un peu de géométrie

Dans cette section, il vous est demandé d'implémenter quelques fonctions génériques de géométrie du plan qui seront utiles pour la suite. Chaque fonction peut bien sûr se servir des précédentes. Vous pouvez aussi utiliser la bibliothèque `Float` d'OCaml[6], qui contient par exemple la constante `Float.pi : float` et les fonctions trigonométriques.

3. https://en.wikipedia.org/wiki/Astr%C3%A9e_(static_analysis)
4. https://en.wikipedia.org/wiki/Frama-C
5. https://fbinfer.com/
6. Voir : https://ocaml.org/manual/5.2/api/Float.html

3.1 Transformations du plan

Un point du plan est donné par deux coordonnées réelles (x, y) (abscisse et ordonnée), un vecteur du plan par deux composantes identifiables à des coordonnées. Les points et les vecteurs seront représentés par deux alias d'un type enregistrement représentant des coordonnées dans le plan, et les angles par un simple alias du type `float` :

```
type coord2D = {
    x : float;
    y : float
  }
type point  = coord2D
type vector = coord2D
type angle  = float
```

Exercice 3.0. Écrire une fonction

```
translate : vector -> point -> point
```

qui calcule la translation d'un point selon un vecteur. Par exemple,

```
let p = {x = 1.; y = 1.}
and u = {x = 2.5; y = 4.5}
in translate p u
    = {x = 3.5; y = 5.5;}
```

Exercice 3.1. Écrire une fonction

```
rad_of_deg : angle -> angle
```

convertissant en radians une valeur d'angle exprimée en degrés. Écrire également la fonction inverse

```
deg_of_rad : angle -> angle
```

Par exemple,

```
deg_of_rad (Float.pi /. 2.) = 90.
rad_of_deg 45. = 0.785398163397448279
```

Exercice 3.2. Écrire une fonction

```
rotate : point -> angle -> point -> point
```

telle que `rotate c alpha p` renvoie le point obtenu par rotation du point `p` d'angle `alpha` autour du centre `c`, l'angle `alpha` étant exprimé en degrés.
Rappelons la formule : la rotation du point $p = (x, y)$ autour du centre $c = (u, v)$ d'angle θ (en radians !) donne le point $p' = (x', y')$ où :

$$\begin{cases} x' = u + (x - u)\cos\theta - (y - v)\sin\theta \\ y' = v + (x - u)\sin\theta + (y - v)\cos\theta \end{cases}$$

Par exemple

```
rotate {x = 0.; y = 1.} 45. {x = 1.; y = 0.}
    = {x = 1.41421356237309492; y = 0.999999999999999889}
```

Les translations et les rotations peuvent être représentées uniformément à l'aide du type suivant :

```
type transformation =
       Translation of vector
     | Rotation of point * angle
```

Exercice 3.3. Écrire une fonction

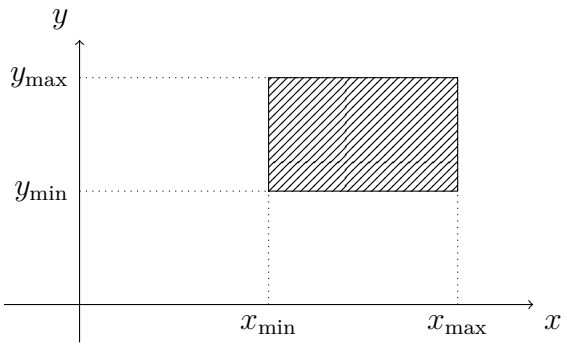
```
transform : transformation -> point -> point
```

qui prend en entrée une transformation et un point, et renvoie l'image du point par cette transformation.

3.2 Rectangles

Dans la suite, nous allons considérer des rectangles *droits*, c'est-à-dire dont les côtés sont parallèles aux axes des abscisses et des ordonnées. Dans toute la suite, par « rectangle », sans autre précision, nous entendrons toujours un rectangle droit (les termes « rectangle non nécessairement droit » étant réservés aux rectangles arbitraires).

Un rectangle est donné par la plus petite (`x_min`) et la plus grande (`x_max`) abscisse de ses sommets ainsi que par la plus petite (`y_min`) et la plus grande (`y_max`) ordonnée de ses sommets.



Nous pouvons donc représenter des rectangles par les valeurs du type suivant :

```
type rectangle = {
  x_min : float;
  x_max : float;
  y_min : float;
  y_max : float
}
```

Exercice 3.4. Écrire une fonction

```
in_rectangle : rectangle -> point -> bool
```

qui prend en entrée un rectangle et un point, et détermine si le point est intérieur au rectangle (au sens large, c'est-à-dire éventuellement sur son bord).

Exercice 3.5. Écrire une fonction

```
corners : rectangle -> point list
```

qui prend en entrée un rectangle, et renvoie la liste de ses quatre sommets.

Exercice 3.6. ($\star$) Écrire une fonction

```
rectangle_of_list : point list -> rectangle
```

qui prend en entrée une liste non vide de points (de longueur quelconque) et renvoie le plus petit rectangle contenant tous les points de la liste.

4 Un langage de déplacement d'un robot

Nous allons maintenant manipuler un petit langage de programmation qui permet de décrire les déplacements d'un robot virtuel dans le plan. Ce robot est

inspiré de la « tortue » du langage de programmation Logo[7], langage autrefois utilisé pour enseigner la programmation aux enfants.

4.1 Syntaxe du langage

Un *programme* dans notre langage consistera en une suite d'instructions de l'une ou l'autre des formes suivantes :

- *Move* : appliquer à la position courante du robot une transformation donnée (rotation ou translation) et le déplacer vers la position résultante,
- *Repeat* : répéter n fois une suite d'instructions donnée,
- *Either* : exécuter une suite d' instructions choisie au hasard parmi deux suites d'instructions données.

Les programmes seront représentés à l'aides des deux types mutuellement recursifs suivants :

```
type instruction =
  Move of transformation
| Repeat of int * program
| Either of program * program
and program = instruction list
```

Par exemple, la valeur suivante est un programme :

```
[Move (Translate {x = 1.; y =  0.});
 Repeat (5, [
    Either (
        [Move(Rotate ({x = 0.; y =  0.}, 45.));
         Move(Translate {x = 0.; y = 1.})
        ],
        [Move(Rotate ({x = 1.; y = 1.}, -45.));
         Move(Translate {x = 1.; y = 1.})
        ]
    )]
  )
]
```

4.2 Le cas déterministe

Un programme sera dit *deterministe* s'il ne contient aucune instruction *Either*. Dans ce cas, le programme décrit de manière non ambiguë une unique suite de

7. Voir https://fr.wikipedia.org/wiki/Logo_(langage)

mouvements à effectuer : son comportement sera toujours le même à chaque fois qu'on l'exécute.

Exercice 4.0. Écrire une fonction

```
is_deterministic : program -> bool
```

renvoyant `true` si et seulement si son argument est un programme déterministe.

De manière évidente, tout programme déterministe contenant zéro, une ou plusieurs instructions *Repeat* peut être réécrit en un programme déterministe sans aucune instruction *Repeat* et effectuant la même suite de transformations : il suffit de "déplier" chaque boucle *Repeat* en répétant sa suite d'instructions autant de fois que spécifié.

Exercice 4.1. (⋆) Écrire une fonction

```
unfold_repeat : program -> program
```

qui, étant donné un programme déterministe, renvoie un programme déterministe effectuant la même suite de transformations que le premier mais ne contenant aucun *Repeat*. Si cette fonction rencontre un *Either* dans le programme, elle devra lever une exception.

Exercice 4.2. (⋆) Écrire une fonction

```
run_det : program -> point -> point list
```

qui, étant donnés un programme déterministe et une position de départ pour le robot, génère la liste de toutes les positions visitées par le robot durant l'exécution.

Cette fonction peut être écrite à l'aide de la précédente mais aussi, de manière un peu moins concise mais plus efficace, sans celle-ci. Là encore, si la fonction rencontre un *Either* dans le programme, elle devra lever une exception.

Exercice 4.3. Écrire une fonction

```
target_reached_det : program -> point -> rectangle -> bool
```

qui, étant donnés un programme déterministe, un point de départ pour le robot et un rectangle qui représente la cible à atteindre, renvoie `true` si le robot termine son trajet dans la cible à la fin de l'exécution du programme, et `false` sinon.

4.3 Le cas non déterministe

Exercice 4.4. Reprendre la fonction `unfold_repeat` de la Section 4.2. Au lieu de lever une exception si son argument contient un *Either*, compléter cette fonction de manière à effectuer l'élimination des *Repeat* pour tout programme, déterministe ou non.

Exercice 4.5. (⋆⋆) Écrire une fonction

```
run : program -> point -> point list
```

qui simule une exécution possible d'un programme quelconque à partir d'un point de départ donné. À chaque fois qu'un *Either* est rencontré, il faudra choisir au hasard l'une des deux branches à exécuter (on pourra utiliser `Random.bool`). Cette fonction peut donc renvoyer des résultats différents si on l'appelle plusieurs fois. Attention au cas où un *Either* se trouve dans une boucle *Repeat* : il faudra alors faire un nouveau choix à chaque itération de la boucle.

Exercice 4.6. (⋆⋆) En plus de l'élimination des *Repeat* opérée par la fonction `unfold_repeat`, on peut aussi éliminer les *Either* dans un programme non-déterministe. Cela produit alors une liste de programmes déterministes très simples : ils ne contiennent plus que des instructions *Move*. Les programmes de cette liste correspondent à chacun des deux choix possibles pour chaque *Either* rencontré dans l'exécution du code d'origine. Écrire une fonction

```
all_choices : program -> program list
```

calculant, à partir d'un programme quelconque, la liste de programmes déterministes comme expliqué ci-dessus. Par exemple, le programme

```
let p = [Either ([Move(Translate {x = 0.; y = 1.})],
                 [Move(Translate {x = 1.; y = 0.})])]
```

donnera lieu à deux exécutions possibles :

```
all_choices p =
  [[Move(Translate {x = 0.; y = 1.})];
   [Move(Translate {x = 1.; y = 0.})]]
```

Exercice 4.7. Écrire une fonction

```
target_reached : program -> point -> rectangle -> bool
```

qui prend en entrée un programme non-déterministe, un point de départ et un rectangle qui représente la zone cible à atteindre, et qui renvoie `true` si dans *toutes* des exécutions possibles, le robot termine son trajet dans la zone cible.

5 Sur-approximation par intervalles

On suppose à présent que la position initiale du robot nous est fournie par des coordonnées GPS, avec une certaine marge d'erreur. En conséquence, on suppose que cette position n'est plus spécifiée par un unique point mais par une *approximation* de ce point sous la forme d'un rectangle (droit).

5.1 Sur-approximation par des rectangles

Exercice 5.0. Écrire une fonction

```
sample : rectangle -> point
```

qui prend en entrée un rectangle, et renvoie un point choisi aléatoirement à l'intérieur de ce rectangle (on pourra utiliser `Random.float`[8]).

Exercice 5.1. (⋆) Écrire une fonction

```
transform_rect : transformation -> rectangle -> rectangle
```

qui renvoie l'image d'un rectangle par la transformation donnée en argument :

- Si la transformation est une translation, le rectangle résultant est la translation du rectangle initial.
- S'il s'agit d'une rotation, l'image du rectangle initial n'est plus nécessairement un rectangle droit. Dans ce cas, le rectangle résultant sera la meilleure sur-approximation de l'image. Il s'agit du plus petit rectangle droit contenant les images des quatre sommets du rectangle initial. La Figure 1 montre un exemple, où le rectangle resultant de la rotation est dessiné en bleau.

Exercice 5.2. (⋆) Écrire une fonction

```
run_rect : program -> rectangle -> rectangle list
```

qui, à partir d'un programme et d'un rectangle représentant une approximation de la position initiale du robot, simule une exécution possible du programme et renvoie la liste des approximations successives des positions visitées par le robot. Le résultat de la transformation d'une approximation donnée sera bien sûr calculé à l'aide de la fonction précédente.

8. Voir https://ocaml.org/manual/5.2/api/Random.html

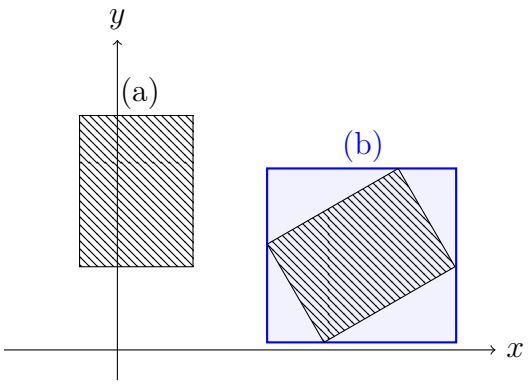


FIGURE 1 – Le rectangle de départ (a) est transformé par rotation ; le rectangle obtenu est le rectangle (b), en bleu.

Exercice 5.3. Écrire une fonction

```
inclusion : rectangle -> rectangle -> bool
```

qui prend en entrée deux rectangles, et renvoie `true` si le premier rectangle est contenu entièrement dans le second, `false` sinon.

Exercice 5.4. Écrire une fonction

```
target_reached_rect : program -> rectangle -> rectangle -> bool
```

qui prend en entrée un programme non déterministe, un rectangle représentant une approximation de la position initiale du robot et un deuxième rectangle représentant une zone cible à atteindre, et qui renvoie `true` si et seulement si dans *toutes* les exécutions du programme, le robot termine son trajet dans la zone cible.

5.2 Abstraction des états du robot

Exercice 5.5. On notera que les fonctions `run` et `run_rect` effectuent quasiment le même traitement :

- la première construit une suite de points par application d'une suite de transformations sur les points à l'aide de la fonction `transform`,
- la seconde construit une suite de rectangles par application d'une suite de transformations sur les rectangles à l'aide de `transform_rect`.

En déduire une fonction

```
run_polymorphe
    : (transformation -> 'a -> 'a) -> program -> 'a -> 'a list
```

effectuant le traitement suivant :

- la variable de type `'a` peut être spécialisée à tout type décrivant un état courant du robot (position, approximation de position, etc.),
- le premier argument est une fonction spécifiant la manière dont une transformation, appliquée à un état quelconque, produit un nouvel état,
- le second argument est un programme,
- le troisième argument est un état initial du robot,
- la fonction doit renvoyer la liste des états atteints en appliquant à l'état initial la suite des transformations spécifiée par le programme pour l'une des exécutions possibles.

En particulier :

- `(run_polymorphe transform)` devrait avoir le même comportement que `run`,
- `(run_polymorphe transform_rect)` devrait avoir le même comportement que `run_rect`,

5.3 Sur-approximation du Either

Le nombre d'exécutions possibles d'un programme augmente très rapidement, notamment lorsqu'on a des choix « Either » dans des boucles « Repeat ». Il devient vite impossible de lister toutes les exécutions pour les vérifier une par une. Par exemple, le programme de la Figure 2 dénombre pas moins de 4^{100} exécutions possibles !

```
[Repeat (100, [
    Either (
      [Move(Translate {x = 1.; y = 0.})],
      [Move(Translate {x = -1.; y = 0.})]
    );
    Either (
      [Move(Translate {x = 0.; y = 1.})],
      [Move(Translate {x = 0.; y = -1.})]
    )]
  )
]
```

FIGURE 2 – Un programme avec 4^{100} exécutions possibles

Pour remédier à ce problème, nous allons changer la façon d'interpréter un programme de la forme `Either(p,q)`. Supposons que l'exécution du programme `p`

envoie le robot dans un rectangle R_p, et que l'exécution du programme `q` envoie le robot dans un rectangle R_q. Alors nous allons considérer que l'exécution du programme `Either(p,q)` envoie le robot dans un rectangle $R_{\texttt{Either(p,q)}}$, qui est le plus petit rectangle qui contient à la fois R_p et R_q. Un exemple est montré dans le Figure (3).

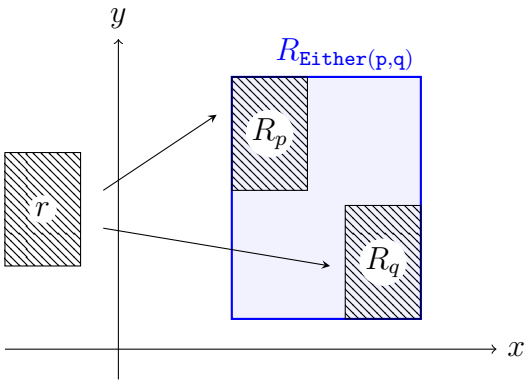


FIGURE 3 – La position initiale du robot est le rectangle r. Les rectangles R_p et R_q représentent les positions finales du robot après l'exécution du programme `p` ou `q`. La sur-approxiomation de `Either(p,q)` est le rectangle bleu $R_{\texttt{Either(p,q)}}$.

Ainsi, nous pouvons *sur-approximer* l'ensemble des positions atteignables par le robot. En faisant cela, nous perdons évidemment de l'information, mais nous avons éliminé les branchements. Nous n'avons plus besoin de vérifier séparément que chaque exécution possible vérifie bien notre propriété (le robot a atteint la cible) : il nous suffira de calculer la sur-approximation de tous états finaux atteignables par le robot, et de faire une seule vérification.

Exercice 5.6. (⋆⋆) Écrire une fonction

```
over_approximate : program -> rectangle -> rectangle
```

qui prend en entrée un programme non-déterministe p et un rectangle r représentant l'ensemble des positions initiales possibles du robot. Cette fonction doit renvoyer un rectangle représentant la sur-approximation des états atteignables par le robot, comme expliqué ci-dessus.

Exercice 5.7. Écrire une fonction

```
feasible_target_reached
    : program -> rectangle -> rectangle -> bool
```

qui prend en entrée un programme, un premier rectangle qui représente la position initiale du robot, et un deuxième rectangle qui représente la cible à atteindre. Cette fonction doit renvoyer `true` si et seulement si le rectangle obtenu par sur-approximation des états finaux du robot est inclus dans la cible.

Pour tester la fonction de l'exercice précédent, vérifiez que pour le programme de la Figure 2, si le robot part initialement du point de coodonnées $(0, 0)$, il ne sort jamais du rectangle défini par $x_{\min} = -101, x_{\max} = 101, y_{\min} = -101, y_{\max} = 101$.

Références

[1] X. Rival and K. Yi. *Introduction to Static Analysis : An Abstract Interpretation Perspective*. MIT Press, 2020.

6 Usage de GitLab

La création d'un depôt git sur le GitLab de l'UFR permettra à votre groupe de disposer d'un dépôt commun de fichiers sur ce serveur. Chaque membre du groupe pourra ensuite disposer d'une copie locale de ces fichiers sur sa machine, les faire évoluer, puis sauvegarder les changements jugés intéressants et les synchroniser sur le serveur.

Si vous ne vous êtes pas déjà servi de GitLab dans d'autres matières, cette section décrit son usage le plus élémentaire. Pour plus d'informations sur git et GitLab, il existe de multiples tutoriels en ligne. Nous contacter rapidement en cas de problèmes.

Départ. La toute première étape est d'installer `git`[9] et `ssh`[10] sur votre machine.

Création d'une clé SSH. Si vous n'avez pas encore de clé SSH, la commande à utiliser pour en générer une sous linux est :

```
ssh-keygen -t rsa
```

Acceptez le choix de répertoire par défaut et choisissez un mot de passe pour votre clé. Ceci créera deux fichiers dans un nouveau répertoire `.ssh`. La partie publique de la clé est le contenu du fichier `id_rsa.pub`. Lancez ensuite la commande :

```
ssh-add
```

et entrez le mot de passe de la clé.

Accès au serveur et configuration. Connectez-vous via l'interface web :

`https://moule.informatique.univ-paris-diderot.fr`

avec les mêmes nom et mots de passe que sur les machines de l'UFR, (et non ceux de votre compte ENT).

Pour ajouter votre clé SSH au serveur, cliquez sur l'icône la plus en haut à droite sur le panneau gauche (votre avatar) et selectionnez *Préférences*. Dans le panneau gauche, aller dans la section *Clés SSH*. Cliquez sur *Ajouter une nouvelle clé*.

Copiez-collez le contenu de `id_rsa.pub` dans le cadre *Clé* de la nouvelle page. Après l'ajout, testez sur votre machine la connexion au serveur (répondez "yes" à la première connexion) :

9. Sous Ubuntu, package `git`.
10. Sous Ubuntu, package `openssh-client`, en principe déjà installé dans une distribution standard.

```
ssh -T git@moule.informatique.univ-paris-diderot.fr
```

La réponse devrait être “Welcome to gitlab !”.

Création du dépôt. Pour ce projet, nous vous fournissons quelques fichiers initiaux. Votre dépôt git sera donc un dérivé (un “fork”, une bifurcation) du dépôt publique du cours. En pratique :

1. L'un des membres de votre groupe se rend sur la page du cours :

   https://moule.informatique.univ-paris-diderot.fr/giovanni/projet-pf5-2425

   s'identifie si ce n'est pas déjà fait, et appuie sur le bouton *Créer une bifurcation* (en haut à droite). Attention, une seule bifurcation par groupe suffit.
2. Dans la page suivante, sélectionnez le niveau de visibilité **privé** pour votre dépôt, et confirmez la création de la bifurcation.
3. Une fois la bifurcation créée, dans le panneau gauche de sa fenêtre principale, sélectionnez dans *Gestion* l'item *Membres*.
4. Ajoutez votre binôme, ainsi que chacun des enseignants de ce cours (*Inviter des membres*, recherche des noms dans le premier cadre de la fenêtre pop-up), en choisissant pour chacun le rôle *Maintainer* : `bucciare`, `allainc`, `bernardi`, `padovani`, `ledent`, `kesner`,
5. Voilà, votre dépôt sur le GitLab est prêt !

Création et synchronisation de vos copies locales de travail. Chaque membre du projet “clone” le dépôt du projet sur sa propre machine, c'est-à-dire en télécharge une copie locale : `git clone` suivi de l'addresse du projet tel qu'il apparaît dans l'onglet *Code* sur la page du projet, champ en *Cloner avec SSH*. Pour cela, il faut avoir installé `git` et `ssh` et configuré au moins une clé SSH dans GitLab, voir ci-dessus.

Une fois le dépôt cloné et en se plaçant dans le répertoire du dépôt local, chaque membre pourra à tout moment :

- Télécharger en local la version la plus récente du dépôt distant sur Gitlab : `git pull`
- Téleverser sa copie locale modifiée sur GitLab : `git push`

Avant toute synchronisation, vérifiez que vous avez une copie locale “propre” (où toutes les modifications sont enregistrées dans des “commits”).

Modifications du dépôt : les commits. Un dépôt Git est un répertoire dont on peut sauvegarder l'historique des modifications. Chaque action de sauvegarde est appeléee une *révision* ou "commit". L'*index* du dépôt est l'ensemble des modifications qui seront sauvegardées à la prochaine révision. La commande

`git add` suivi du nom d'un ou plusieurs fichiers

permet d'ajouter à l'index toutes les modifications faites sur ces fichiers. Si l'un d'eux vient d'être créé, on ajoute dans ce cas à l'index l'opération d'ajout de ce fichier au dépôt. La même commande suivie d'un nom de répertoire ajoute à l'index l'opération d'ajout du répertoire et de son contenu au dépôt.

La révision effective du dépôt se fait par la commande

`git commit -m` suivi d'un message entre guillemets doubles

en suite la commande

`git push`

Les commandes `git mv` et `git rm` se comportent comme `mv` et `rm`, mais ajoutent immédiatement les modifications associées du répertoire à l'index. Ne vous servez pas de la commande `mv` sans la faire précéder de `git` : ceci serait interprété comme l'effacement du fichier et la création d'un nouveau fichier.

Invocable à tout instant, la commande

`git status`

affiche l'état courant du dépôt depuis sa dernière révision : quels fichiers ont été modifiés, renommés, effacés, créés, etc., et lesquelles de ces modifications sont dans l'index. Elle indique également comment rétablir l'état d'un fichier à celui de la dernière révision, ce qui est utile en cas de fausse manœuvre.

Il est conseillé d'installer et d'utiliser les interfaces graphiques `gitk` (visualisation de l'arbre des commits) et `git gui` (aide à la création de commits).

Une dernière chose : git est là pour vous aider à organiser et archiver vos divers fichiers sources. Par contre il vaut mieux ne *pas* y enregistrer les fichiers issues de compilations (binaires, répertoire temporaire tels que `_build` pour `dune`, fichiers objets OCaml `*.cm{o,x,a}`, etc).

Les fusions (merge) et les conflits. Si vous êtes plusieurs à modifier vos dépôts locaux chacun de votre côté, celui qui se synchronisera en second avec votre dépôt GitLab commun aura une manoeuvre nommé "merge" à effectuer. Tant que vos modifications respectives concernent des fichiers ou des zones de code différentes, ce "merge" est aisé, il suffit d'accepter ce que git propose, en personnalisant éventuellement le message de merge. Si par contre les modifications se chevauchent et sont incompatibles, il y a alors un conflit, et git vous demande d'aller décider quelle version est à garder. Divers outils peuvent aider lors de cette

opération, mais au plus basique il s'agit d'aller éditer les zones entre <<<<< et >>>>> puis faire `git add` et `git commit` de nouveau.

Intégrer les modifications venant du dépôt du cours. Si le dépôt du cours reçoit ultérieurement des correctifs ou des évolution des fichiers fournis pour le projet, ces modifications peuvent être intégrés à vos dépôts.

- La première fois, allez dans votre répertoire de travail sur votre machine, et tapez :

  ```
  git remote add prof \
  https://moule.informatique.univ-paris-diderot.fr/\
  giovanni/projet-pf5-2425
  ```

  Remarque. Dans les versions précédentes de l'énoncé, la commande proposée était erronée. Si vous l'avez exécutée, entrez d'abord la commande suivante avant la précédente :

  ```
  git remote rm prof
  ```

- Ensuite, à chaque fois que vous souhaiterez récupérer des commits du dépôt du cours :

  ```
  git pull prof main
  ```

- Selon les modifications récupérées et les vôtres entre-temps, cela peut nécessiter une opération de "merge" comme décrit auparavant.
- Enfin, ces modifications étant intégrés à votre copie locale de travail, il ne reste plus qu'à les transmettre également à votre dépôt sur GitLab :

  ```
  git push
  ```

Les branches. Il est parfois pratique de pouvoir essayer différentes choses, même incompatibles. Pour cela, Git permet de travailler sur plusieurs exemplaires d'un même dépôt, des *branches*. Un dépôt contient toujours une branche principale, la branche "master", dont le rôle est en principe de contenir sa dernière version stable. Les autres branches peuvent servir à développer des variantes de la branche master, par exemple pour tenter de corriger un bug sans altérer cette version de référence. La création d'une nouvelle branche, copie conforme de la branche courante – initialement, master – dans son état courant, se fait par :

`git branch` suivi du nom choisi pour la branche.

Sans arguments, cette commande indique la liste des branches existantes, ainsi que celle dans laquelle se trouve l'utilisateur. Le passage à une branche se fait par

`git checkout` suivi du nom de la branche.

Pour ajouter au dépôt distant une branche qui n'est pas encore sur celui-ci, après s'être placé dans la branche :

`git push -set-upstream origin` suivi du nom de la branche

Un `push` depuis une branche déjà sur le serveur se fait de la manière habituelle. Enfin, on peut "réunifier" deux branches avec `git merge`, voir la documentation pour plus de détails.

Noter que GitLab propose également un mécanisme de "Merge Request" : il permet de proposer des modifications, soit à son propre projet, soit au projet qui a été "forké" à l'origine, les membres du projet en question pouvant alors accepter ou non ces suggestions après discussion.