JUNIA Grande école d'ingénieurs
HEI·ISEN·ISA

PROJET S8

# Nombre premier, Preuve formelle et Magic

Par LORIDAN Juliane, PAPEGHIN Olivier, ROSIER Enzo, SOKOLOWSKI Raphaël


Rendu le *25/04/2024*
Projet encadré par **CHÊNEVERT Gabriel** et **BAUDEL Manon**

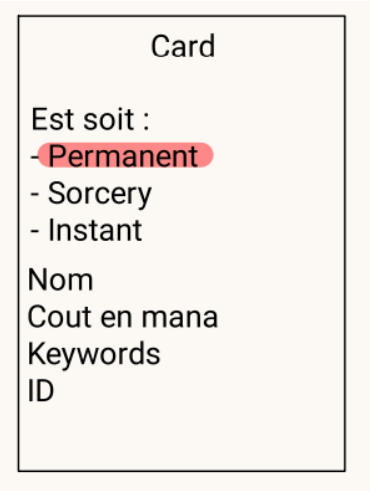
# Sommaire

# Introduction

Ce projet a pour but de prouver que si la conjecture des nombres premiers jumeaux est vraie alors il est possible d'infliger une infinité de dégâts grâce à la carte Zimone, toujours interrogatrice dans le jeu Magic : The Gathering. Pour ce faire, nous utilisons Rocq, un langage de preuve, il faut donc recréer un mécanisme capable de simuler le jeu de cartes afin de pouvoir jouer le combo permettant d'arriver à la situation où la conjecture est impliquée et prouver formellement notre théorème. Tous les fichiers présentés dans ce document sont disponibles via le lien GitHub : https://github.com/olivierpapeghin/Prime_Numbers_Formal_Proofs_Magic.git

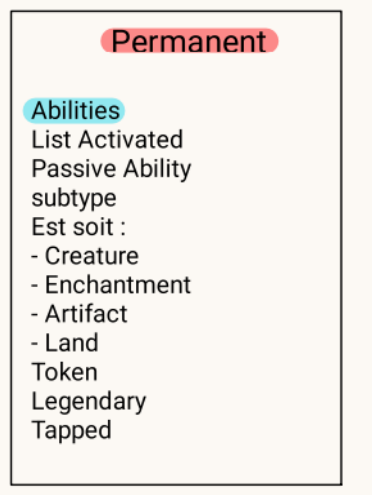
# Définition des types

Le langage Rocq fournit quelques types de bases concernant les preuves, en important des librairies, nous avons accès à plus de types tels que les entiers naturels ou encore des chaînes de caractères. Rocq n'étant pas fait initialement pour recréer un jeu, il nous faut redéfinir tous les types que nous avons besoin de manipuler.
Nous commençons donc par établir un système de cartes qui sont définies comme suit :



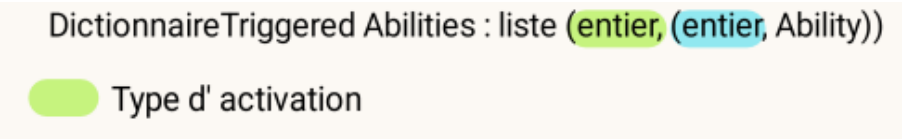

Définition globale d'une carte

Elle permet d'accéder à ses paramètres tels que son nom, son coût en mana, ses keywords ou encore son identifiant. Sans cet identifiant, il n'est pas possible de distinguer toutes les instances d'une même carte, et donc d'accéder à ses attributs sans changer les autres instances. Une carte est donc soit un Permanent, soit un Sorcery ou soit un Instant, ces types sont également définis par :




Définition globale d'un Permanent

Le premier champ de Permanent est en fait une liste d'entier, c'est grâce à ça que l'on gère le système de capacités. Cette liste d'entier fait référence aux capacités du Permanent qui sont dans un dictionnaire, ces entiers nous permettent donc de retrouver les capacités du Permanent et de les appliquer.




Définition d'un dictionnaire de capacités avec ses types d'activation

Il est nécessaire que les dictionnaires de capacités soient indépendants de l'état de jeu car une capacité est une fonction qui prend en paramètre un état de jeu et en renvoie un. On se retrouverait avec une dépendance circulaire où l'état de jeu a besoin du dictionnaire de capacités pour exister mais les capacités ont besoin de l'état de jeu pour être définies. Comme Rocq compile de façon strictement linéaire, il n'est pas possible de mettre le dictionnaire dans l'état de jeu.

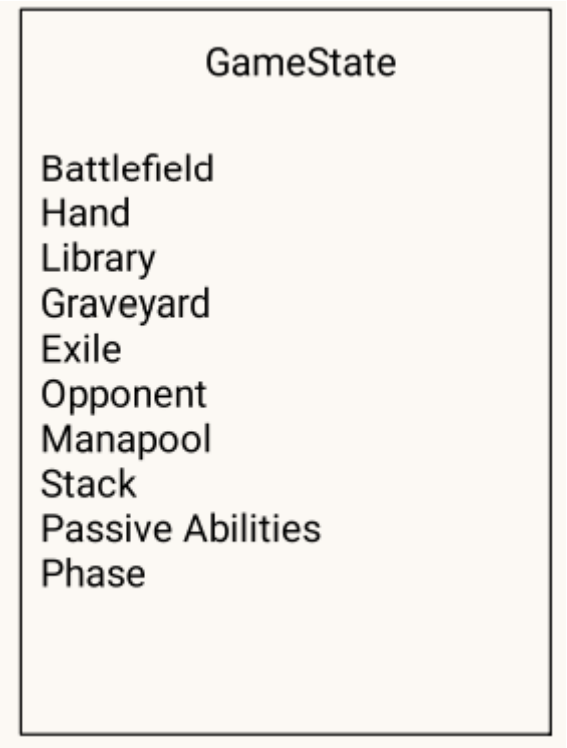
Le dictionnaire de capacités déclenchées est ainsi défini en Rocq :

```
(* Définition des sous-dictionnaires *)
Definition OnCast : Dict := [(1,birgi_ability)].
Definition OnPhase : Dict := [(1,sacrifice_end_step);(2,zimone_ability)].
Definition OnDeath : Dict := [(1,myrkul_ability)].
Definition OnEnter : Dict := [(1,isochron_scepter_enter)].

(* Définition du dictionnaire principal avec des clés de type string *)
Definition Triggered_Abilities : list (nat * Dict) :=
  ( 1 , OnCast) :: (2 , OnPhase) :: (3 , OnDeath) :: (4 , OnEnter) :: nil.
```

La clef 1 correspond donc à toutes les capacités qui sont déclenchées lorsque l'on joue une carte, `OnCast` correspond à une liste de paire d'entier et de capacité. Par exemple, si un permanent a `(2,3)` dans `Abilities`, cela signifie qu'il a la capacité avec la clef 3 de type `OnPhase`.

Afin de modéliser le jeu, nous avons créé un objet appelé `GameState`, qui contient toutes les variables nécessaires à la vision globale de l'état du jeu :




Définition globale de l'état de jeu

Le champ de bataille, la main, la bibliothèque, le cimetière et l'exil sont des listes de cartes. L'opposant est un entier, la mana pool est une liste de Mana, qui est un type que l'on a défini comme : une couleur et une quantité. Le stack est également définie avec un type à part entière : `CardOrPair`, qui est soit une carte soit une paire d'entiers car le stack peut contenir à la fois des cartes et à la fois des capacités, représentées par ces paires.
Passive Abilities est un dictionnaire de capacités passives et Phase permet de connaître la phase actuelle du jeu.

Par exemple, le permanent est ainsi retranscrit en Rocq :

```
Record Permanent := mkPermanent {
  Abilities : list (nat * nat);
  ListActivated : list nat;
  PassiveAbility : option PassiveKey;
  subtype : list string;
  creature : option Creature;
  enchantement : option Enchantement;
  land : option Land;
  artifact : option Artifact;
  token : bool;
  legendary : bool;
  tapped : bool;
}.
```

Définition d'un objet en Rocq

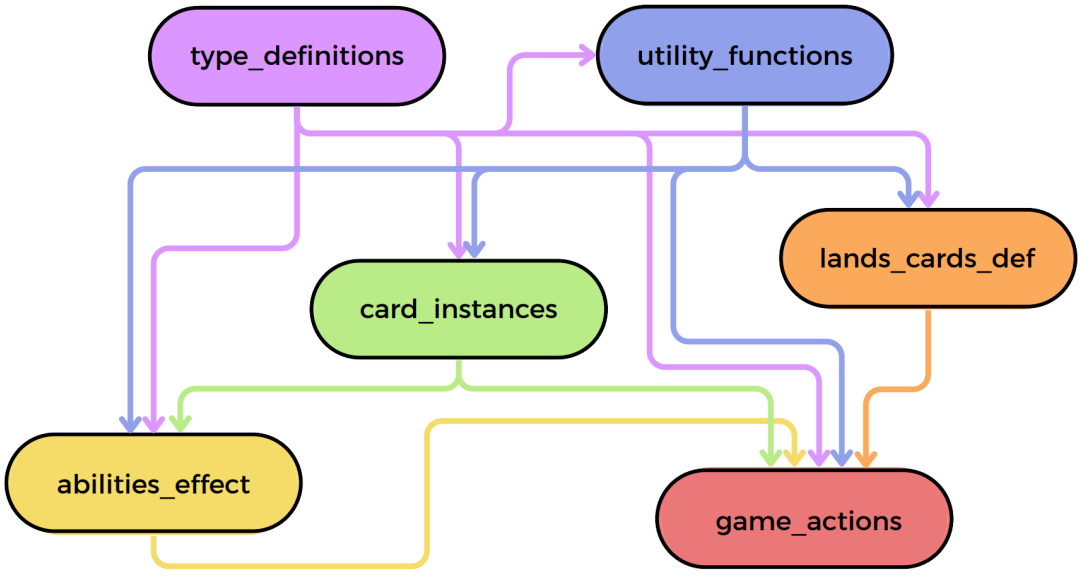
Le mot clé `option` permet d'indiquer que lors de la déclaration, il peut y en avoir ou pas avec les mots clefs `Some` et `None`. Ensuite nous retrouvons des notions basiques avec `nat` qui est un entier naturel et `bool` qui est un booléen. `mkPermanent` est un constructeur permettant de faciliter l'instanciation de Permanent par la suite.

# Organisation et structure

Le projet est décomposé en plusieurs fichiers permettant une bonne structuration :

- `type_definitions` : contient toutes les définitions des types comme carte, permanent etc
- `utility_functions` : contient toutes les fonctions utiles mais qui n'est pas directement appelée par l'utilisateur
- `card_instances` : contient toutes les instances des cartes
- `abilities_effects` : contient tous les effets des capacités des cartes
- `game_actions` : contient toutes les fonctions utiles au jeu comme Cast, Resolve etc
- `Land_cards_def` : contient les définitions des terrains

Voici un schéma illustrant l'organisation et les liens entre les fichiers :




Un exemple de fonction souvent implémentée dans `utility_functions` est une fonction d'équivalence, étant donné que la plupart des objets que l'on manipule utilisent des types créés de zéro, il n'existe pas de fonction pour vérifier que 2 objets sont égaux. Il faut donc recréer ces fonctions, par exemple pour vérifier que 2 couleurs de mana sont égales, on utilise :

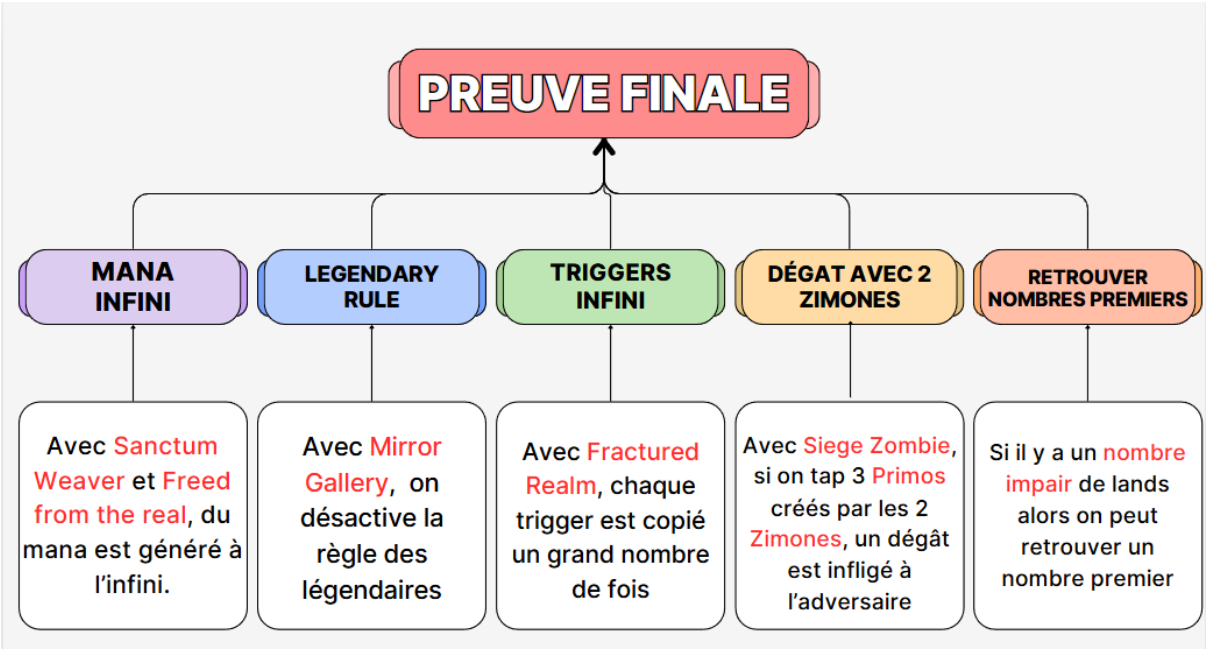
```
Definition eq_mana_color (c1 c2 : ManaColor) : bool :=
  match c1, c2 with
  | White, White => true
  | Blue, Blue => true
  | Black, Black => true
  | Red, Red => true
  | Green, Green => true
  | Generic, Generic => true
  | _, _ => false
  end.
```

Définition d'une fonction en Rocq

La fonction prend en paramètre 2 objets du même type `ManaColor` et renvoie un booléen. On utilise le mot clef `match` afin de comparer les 2, ensuite on regarde si les 2 sont égaux auquel cas on renvoie `true`, sinon `false`.

# Preuves

Afin de vérifier notre hypothèse de base, nous avons démontré plusieurs théorèmes à l'aide de preuves formelles, telles que la preuve du mana infini ou encore retrouver un nombre premier. Toutes ses petites preuves participent à la preuve finale, on définit un axiome tel que la conjecture des nombres premiers jumeaux est vraie et si toutes les sous preuves sont vérifiées et que toutes les cartes nécessaires sont présentes sur le champ de bataille alors nous pouvons regarder si le théorème final est vrai.




Vous pouvez trouver toutes les preuves dans les fichiers dont le nom débute par *`proof.`*

# Comment utiliser le projet ?

## Compiler tous les fichiers

En Rocq, il faut compiler complètement le fichier avant de pouvoir l'exporter en tant que module et ainsi utiliser ses éléments dans d'autres fichiers via l'import. L'ordre de compilation est donc à respecter, le voici :

- type_definitions
- utility_functions
- land_cards_def
- card_instances
- abilities_effects
- game_actions
- setup_combo
- proof_4_primos
- proof_ability
- proof_infinite_mana
- proof_legendary_rule_bypass
- proof_mirror_rooms
- proof__infinite_situation.v
- main

Pour compiler un fichier, il est possible d'utiliser la commande `coqc` *`nomdufichier.v`* dans l'invite de commande Rocq.

Voici les commandes à faire :

```
cd C:\Chemin\vers\projet
coqc type_definitions.v
coqc utility_functions.v
coqc Land_cards_def.v
coqc abilities_effects.v
coqc game_actions.v
coqc card_instances.v
coqc setup_combo.v
coqc proof_4_primos.v
coqc proof_ability.v
coqc proof_infinite_mana.v
coqc proof_legendary_rule_bypass.v
coqc proof_mirror_rooms.v
coqc proof__infinite_situation.v
coqc main.v
```

(Certaines étapes peuvent prendre plusieurs minutes)

Vous pouvez maintenant ouvrir main.v pour exécuter la preuve principale

## Instancier tous les éléments

Pour pouvoir simuler une partie, il faut instancier les différents éléments qui la composent. C'est-à -dire qu'il est nécessaire de créer toutes les cartes du deck du joueur.
Prenons l'exemple de Birgi :

```
Definition Initial_GS : GameState := mkGameState
nil [cartes_créées] nil nil nil (* aucune cartes sauf dans la main *)
20 (* l'opposant a 20 pv*)
[mkMana Green 0; mkMana Red 0; mkMana Blue 0 ;mkMana White 0 ; mkMana Black 0 ;mkMana Generic 0] (*la manapool est vide *)
nil (* le stack est vide *)
DefaultListPassiveAbility (* Aucune capacité passive *)
BeginningPhase. (* Phase du début *)
```

```
Definition birgi (id : nat) : Card :=
  mkCard
  (Some (mkPermanent (* Est un permanent *)
    [(1,1)] (* La liste des capacités déclenchées *)
    nil (* N'a pas de capacités à activer *)
    None (* N'a pas de capacités passives *)
    ["God"] (* le sous-type *)
    (Some (mkCreature 3 3)) (* Est une créature 3/3*)
    None (* N'est pas un enchantement *)
    None (* N'est pas un artifact *)
    None (* N'est pas une land *)
    false (* N'est pas un token *)
    true (* Est légendaire *)
    false)) (* N'est pas tapped *)
  None (* N'est pas un instant *)
  None (* N'est pas un sorcery *)
  [mkMana Red 1; mkMana Generic 2] (*Cout en mana*)
  "Birgi, God of Storytelling" (* nom *)
  id
  nil. (* pas de keywords *)
```

On utilise `mkCard` afin de créer une carte, puis Some `mkPermanent` afin d'indiquer qu'il s'agit d'un permanent. `(1,1)` signifie que Birgi à la capacité avec la clef 1 de type `OnCast`.
On indique ensuite ses autres caractéristiques à l'aide de listes vides ou de `None` s'il n'y a pas. Il faut faire attention à bien remplir tous les champs afin de correspondre à la définition d'un Permanent.

Une fois toutes les cartes créées, il est impératif de créer ses capacités. C'est ce qui est effectué dans `abilities_effect`. Une capacité prend en paramètre un `GameState` et une liste optionnelle de cartes et renvoie le `GameState` modifié par la capacité.

Enfin il faut définir un `GameState` initial qui ne contient aucune carte sur le champ de bataille, les cartes précédemment créées dans la main, la mana pool a zéro etc. Il est possible de l'instancier avec le constructeur `mkGameState` :

### Utiliser les fonctions

Maintenant que tous les éléments sont opérationnels, nous pouvons utiliser les fonctions de jeu tel que Cast, Resolve et jouer un terrain. Pour commencer, jouons un terrain vert :

```
Definition land_played := Play_land (Forest 1) Initial_GS.
```

La fonction `Play_land` prend donc en paramètres, la carte (ici le terrain que l'on souhaite jouer avec son identifiant) et le `GameState` sur lequel on joue le terrain et nous renvoie un nouveau GameState avec le terrain joué si les conditions l'y autorise.
Les autres fonctions de jeu fonctionnent de la même manière, Cast prend exactement les mêmes paramètres et joue la carte si les conditions sont réunies.

```
Definition cast_card := Cast (birgi 1) land_played.
```

La fonction Resolve, qui permet de résoudre le premier élément de la stack, prend en paramètres un GameState, un entier qui correspond au choix de la capacité d'un sort, une liste de cartes étant les cibles des capacités et il renvoie un nouveau GameState modifié.

```
Definition resolve_stack := Resolve cast_card 0 None.
```

Ici on indique juste que l'on souhaite que Birgi soit Cast, il ne s'agit pas d'un sort ni d'une capacité qui nécessite une cible.

Enfin, il est également possible d'activer une capacité avec la fonction `activate_ability` qui prend en paramètres un entier qui correspond à l'index de la capacité, une première liste de cartes qui correspond à un potentiel coût en cartes à payer, une liste de mana représentant le coût en mana de cette capacité, une deuxième liste de cartes sur lesquelles la capacité s'applique, la carte qui contient la capacité, le dictionnaire des capacités à activer et enfin l'état de jeu où la capacité est activé. Prenons un exemple avec Freed from the realm, où l'on souhaite tapper Birgi :

```
Definition freed_activated := activate_ability 4 None (Some [mkMana Blue 1]) (birgi 1) freed_from_the_realm Dict_AA resolve_stack.
```

L'index de la capacité est 4, comme indiqué dans la carte éponyme, le coût en mana est de 1 et la carte que l'on veut tapper est Birgi.

# Vérification des nombres premiers

Pour tester si un nombre est premier nous utilisons la méthode suivante :

Soit n le nombre que l'on veut tester

D'abord si n est égal à 0 ou 1 alors le nombre n'est pas premier
Puis si n est égal 2 ou 3 alors il est premier

Ensuite on vérifie que n est congru à 1 ou 5 modulo 6 (sinon il n’est pas premier)

Finalement on prend la racine carré de n et on regarde tous ses diviseurs pour vérifier s'il est premier ou non.