

Rapport de PAF: *Dungeon crawling*

Quentin PIOTROWSKI, Artemiy ROZOVYK

Juin 2020



1 Introduction

The great labyrinth of destiny est un jeu établi sur le modèle d'un *Dungeon crawling* développé en Haskell. Nous y déplacerons notre héros *Stormbreaker* dans différents niveaux afin de trouver la sortie au travers du labyrinthe, habité par différents monstres que l'on pourra combattre, de trésors à ramasser ainsi que de pièges à éviter.

Notre jeu s'appuie sur un modèle de programmation sûre, utilisant des invariants, pré-conditions et post-conditions qui sont toutes utilisés dans des test unitaires (HSpec)

Table des matières

1	Introduction	1
2	Manuel d'utilisation	3
3	Liste des propositions	3
3.1	Invariants de types	3
3.1.1	Invariants du type Carte	3

3.1.2	Invariants du type Environnement	3
3.1.3	Invariants du type Modèle	3
3.1.4	Invariant du type State	4
3.2	Opérations d'ajout	4
3.2.1	Ajout dans l'environnement	4
3.2.2	Ajout d'une entité au State	4
3.3	Opération d'enlèvement	4
3.3.1	Suppression d'une entité dans l'environnement	4
3.4	Opération de modification	4
3.4.1	Retour d'un élément dans l'Environnement	4
3.4.2	Récupération d'un élément dans le Modèle	5
4	Tests implémentés	5
4.1	Tests sur la carte	5
4.2	Tests sur l'environnement	5
4.3	Tests sur le modèle	5
4.4	Tests sur le Game State	6
4.5	Test implémentés	6
4.5.1	QuickCheckCarte	6
4.5.2	QuickCheckState	6
4.5.3	QuickCheckModel	6
5	Jeu de base	7
5.1	Terrain de Jeu	7
5.2	Environnement	7
5.2.1	Envi	7
5.2.2	Model	7
5.2.3	State	8
5.3	Moteur de Jeu	8
5.3.1	Boucle non-réactive	8
5.4	Déplacement des mobs	8
5.5	Main	8
6	Extensions	9
6.1	Générateur de niveaux aléatoire	9
6.2	Différents niveaux	9
6.3	Combats	9
6.4	Trésors	9
6.5	Piège	10
7	Conclusion	10

2 Manuel d'utilisation

Notre projet s'utilise directement sur Stack :

- On peut choisir de le compiler avec la commande "stack build"
- Pour lancer le jeu il suffit d'utiliser la commande "stack run"
- Pour lancer les différents tests il faut utiliser la commande "stack tests"

Une fois le jeu lancé, les commandes sont :

- Les touches Z, Q, S et D pour déplacer le joueur
- La touche R pour ramasser un trésor et pour attaquer les ennemis
- La touche E pour ouvrir une porte adjacente
- La touche ESCAPE quitte le jeu et fermer la fenêtre

L'objectif est de ramasser le trésor et d'atteindre ensuite la sortie du labyrinthe. Le joueur perd la partie si il marche sur un piège.

Une partie se décompose en trois niveaux succins : Le premier niveau utilise une carte crée par le générateur aléatoire de notre extension. Le second et troisième niveaux sont des niveaux classiques, implémentés par nos soins.

3 Liste des propositions

3.1 Invariants de types

3.1.1 Invariants du type Carte

Notre invariant `prop_inv_carte_saine` vérifie que le type carte respecte bien toutes les propriétés qu'une carte doit posséder afin d'être correcte. Cet invariant regroupe les 6 fonctions suivantes :

- `prop_carte1` vérifie que toutes les cases sont bien comprises entre la hauteur et la largeur de la carte.
- `prop_carte2` vérifie que chaque coordonné qui se trouve dans les limites de la carte ait bien une valeur associée.
- `prop_carte3` vérifie que la carte possède bien une unique entrée et une unique sortie.
- `prop_carte4` vérifie que la carte est entièrement entourée de murs.
- `prop_carte5` vérifie que chaque porte soit entourée de deux murs.
- `prop_carte6` vérifie qu'il existe bien un chemin entre l'entrée et la sortie. On effectue une recherche en parcourant les cases libres adjacents avec un parcours en largeur. On s'arrête quand on a trouvé la sortie ou lorsqu'il n'y a plus de cases à explorer.

3.1.2 Invariants du type Environnement

Notre invariant `prop_envi_inv` vérifie que l'environnement respecte bien toutes les propriétés pour être saint. Il fait appel aux 3 fonctions suivantes :

- `prop_envi_inv1` vérifie qu'il n'y a pas deux unités au même endroit.
- `prop_envi_inv2` vérifie qu'il n'y ait pas plus d'un seul joueur.
- `prop_envi_inv3` vérifie que toutes les coordonnées sont supérieures à zéro.

3.1.3 Invariants du type Modèle

L'invariant `prop_modele_inv` qui doit retourner True si le modèle est saint doit vérifier beaucoup de choses :

- Pour commencer, il est nécessaire de vérifier si la carte contenue dans le modèle est saine. Il doit donc faire appel à l'invariant `prop_inv_carte_saine` sur la carte contenue dans le modèle.
- Il est également nécessaire de vérifier si l'environnement contenu est lui aussi totalement saint : Pour cela il doit faire appel à la fonction `prop_envi_inv`.

- `prop_model_inv1` vérifie ensuite que toutes les entités de l'environnement se trouvent bien sur les coordonnées d'une case traversable.
- `prop_model_inv2` vérifie enfin que les entités sont sur des coordonnées qui sont contenues dans les limites de la carte.
- `prop_model_inv4` vérifie que le trésor soit accessible depuis l'entrée. La fonction effectue une recherche avec un parcours en largeur des cases traversables adjacentes jusqu'à ce que le trésor soit contenu dans la case ou que toutes les cases aient été explorés.

3.1.4 Invariant du type State

L'invariant `prop_state_inv` doit également vérifier beaucoup de propriétés pour être valide :

- Il faut déjà vérifier à l'aide de la fonction `prop_inv_carte_saine` si la carte du State est bien saine.
- Il est ensuite nécessaire de vérifier si l'environnement est saint à l'aide de la fonction `prop_envi_inv`.
- `prop_state_inv1` vérifie que `numTour` est un entier positif.
- `prop_state_inv2` vérifie que chaque coordonnée sur laquelle se trouve une entité correspond à une case traversable (Différente d'un mur ou d'une porte fermée).
- `prop_state_inv3` vérifie que toutes les entités se trouvent bien dans les limites de la carte.
- `prop_state_inv4` vérifie que les entités ne se chevauchent pas entre elles ni avec le joueur.

3.2 Opérations d'ajout

3.2.1 Ajout dans l'environnement

L'opération `pre_ajout_env` vérifie la pré-condition suivante : Il n'y a pas déjà une entité (Monstre ou Joueur) présent sur la case avant l'ajout.

3.2.2 Ajout d'une entité au State

`prop_pre_add_entity_state` regroupe quatre fonctions qui vérifient :

- Si les coordonnées de l'ajout sont bien comprises dans les limites de la carte.
- Si il n'y a pas déjà une entité sur la case.
- On vérifie également que le nombre de Monstres ne dépasse pas la moitié des cases vides de la carte (L'état serait trop peuplé sinon).
- On doit également vérifier que l'on ajoute pas un deuxième joueur à l'état (un seul joueur par état)

`prop_post_add_entity_state` vérifie après l'exécution de la fonction `add_entity_state` que l'entité a bien été ajoutée au State.

3.3 Opération d'enlèvement

3.3.1 Suppression d'une entité dans l'environnement

La fonction `pre_rmv_envi` est lancée avant la fonction `rmv_coor_envi` afin d'être sûr qu'il y ait bien une entité présente sur la case que l'on pourra alors supprimer.

La fonction `post_rmv_envi` vérifie qu'après le passage de la fonction `rmv_coor_envi` qu'il n'y ait plus d'entité présente sur la case dont on a retiré l'entité.

3.4 Opération de modification

3.4.1 Retour d'un élément dans l'Environnement

L'opération `post_getPlayer` s'assure que l'élément retourné par la fonction `getPlayer` est bien une coordonnée contenant un Joueur et non un monstre.

3.4.2 Récupération d'un élément dans le Modèle

post_prevoit permet de vérifier qu'il y a bien au moins une action proposé.

post_openDoor : Vérifie que chaque porte autour du joueur ait bien été ouverte.

La fonction pre_moveGenerique s'applique avant moveGenerique, on vérifie avant le mouvement d'un joueur si la coordonnée de destination est toujours bien contenue dans la carte. Attention : La fonction ne vérifie pas si la case est traversable et cela est voulu : Le cas est traité par la fonction moveGenerique et ne doit pas être interdit.

La fonction post_moveGenerique vérifie si le joueur a bien été déplacé sur les coordonnées souhaitées.

post_interact_object : Vérifie qu'il n'y ait plus de trésor non ramassé ou d'ennemis non-blessé.

4 Tests implémentés

4.1 Tests sur la carte

Dans le premier test injustement nommé carteTest2, on effectue une vérification des tous les invariants sur l'exemple de la carte "carte1".

Le second test nommé carteTest1 vise à vérifier si une case précise de la carte est traversable.

4.2 Tests sur l'environnement

Le premier test sur l'environnement consiste à vérifier si l'environnement vide "envi1" est saint, pour cela il fait appel à la fonction prop_envi_inv qui fait appel à tous les invariants qui concernent l'environnement.

Nous allons ensuite tester l'ajout d'un élément (Un monstre en l'occurrence) avec nos pre/post conditions :

- La fonction pre_add_env va tester la case sur laquelle on veut y ajouter notre mob et va retourner "True" car l'environnement à ces coordonnées est vide.
- La fonction post_add_env va ensuite vérifier après l'application de la fonction d'ajout sur l'environnement résultant "res" si le Monstre a bien été ajouté.

Nous allons enfin tester le retrait d'une entité de notre environnement.

- On applique la fonction pre_rmv_env qui va tester la présence de notre monstre aux coordonnées de son ajout.
- Une fois la fonction d'enlèvement appliquée, on teste avec la fonction post_rmv_env pour vérifier que l'environnement ne contient plus d'entité aux coordonnées du retrait.

4.3 Tests sur le modèle

Afin de créer un modèle, il faut commencer par créer une carte saine ainsi qu'un environnement auquel on aura ajouté un trésor ainsi que le joueur. Ainsi, l'invariant de modèle qui vérifie que la carte, l'environnement et le modèle sont saint, retournera "True".

Nous allons ensuite tester les pre/post conditions de la fonction moveGenerique (Appelée pour déplacer le Joueur)

- Nous allons appliquer la fonction pre_moveGenerique sur les coordonnées situés au dessus de la position du joueur. La fonction retournera "True" car les coordonnées indiqués sont toujours situés sur la carte.
- On applique ensuite la fonction post_moveGenerique sur l'environnement sur lequel nous avons appliqué au préalable le mouvement du Joueur. La fonction trouve bien le joueur sur la position de son déplacement, elle doit donc retourner "True".

4.4 Tests sur le Game State

Pour réaliser le test sur un State, on commence par créer une carte et par générer un State avec cette carte. On teste ensuite avec la fonction `prop_state_inv` les différents invariants qui retourneront "True" si le State est saint.

Nous allons ensuite vérifier la cohérence de l'ajout d'une entité au State grâce aux pre/post conditions :

- La fonction `prop_pre_add_entity_state` vérifie que notre état contient bien moitié plus de case libres que de cases occupés, que les coordonnées sont cohérentes et sans autre entité dessus.
- La fonction `prop_post_add_entity_state` vérifie que l'entité est bien présente après l'application de la fonction d'ajout.

4.5 Test implémentés

Finalement, afin d'utiliser la puissance de **property-based testing** nous mettons en oeuvre une jeu de tests `QuickCheck`.

4.5.1 QuickCheckCarte

Suite à la définition d'un générateur de carte, nous pouvons désormais définir une instance de type `Arbitrary`, pour indiquer à `QuickCheck` la manière dont les cartes seront générées :

```
1 instance Arbitrary Carte where
2   arbitrary = genCarte
```

Ensuite, nous procédons au test de la fonction `prop_inv_carte_saine` qui regroupe les 6 invariants cités précédemment, (qui sera testé pour les 100 cartes générées aléatoirement).

4.5.2 QuickCheckState

Pour vérifier l'invariant et les propriétés du module `State` nous allons instancier `Arbitrary` de manière suivante :

```
1 instance Arbitrary Etat where
2   arbitrary = do
3     carte <- (arbitrary :: Gen Carte)
4     moment <- choose (0::CDouble, 20::CDouble)
5     numberOfMobs <- choose(2,6)
6     seed <- choose (0, 1000)
7     let gen = mkStdGen seed
8     return $ init_state carte numberOfMobs moment gen
```

Ainsi nous vérifions les états contenant entre 2 et 6 entités de type `Mob`. L'invariant ainsi que la postcondition de la fonction `add_entity` sont testés avec des états arbitraires.

4.5.3 QuickCheckModel

De même manière, nous définissons une façon de générer les `Modele`. Le point intéressant étant la vérification de `post_interact_object` qui nécessite un générateur de coordonnées particulier (et non celui défini par défaut `QuickCheckCarte`). Nous introduisons donc un générateur de coordonnées à part et indiquons à `QuickCheck` de l'utiliser.

```
1 genMove :: Gen Coord
2 genMove = do
3   (x,y) <- elements [(1,0), (-1,0), (0,1), (0,-1)]
4   return (C x y)
5 ...
6 describe "Object interaction" $ do
7   property (forAll genMove (prop_post_interact_object ))
```

5 Jeu de base

5.1 Terrain de Jeu

Notre carte suit le même modèle que le guide à savoir :

- Une largeur de la carte : "cartel"
- Une hauteur de la carte "carteh"
- Un attribut "carte_contenu" qui est une Map associant pour chaque type coordonné une case.

L'implémentation de la fonction `prop_carte6` est assez intéressante : Il s'agit de l'invariant qui vérifie que la sortie est accessible depuis l'entrée.

La fonction auxiliaire prends en paramètre la carte que l'on souhaite tester ainsi que deux listes :

- La première liste, vide à l'initialisation, est celle des cases dites "mortes", elle contient les cases dont on a inspecté toutes les voisines et que l'on ne souhaite pas inspecter à nouveau.
- La seconde liste ne contient que l'entrée à l'initialisation. Elle servira à référencer les cases "à visiter". On y ajoutera progressivement chaque case accessible et on retirera en retirera les cases dont on a inspecté toutes les voisines pour les ajouter placer dans la première liste des "cases mortes".

```
1 aux1 (Carte larg haut cases) [] [getEntreeCase (Carte larg haut cases)]
```

L'algorithme se déroule comme suit :

La première case de la liste de cases "à visiter" est extraite. On renvoie True si cette case se trouve être la sortie. Sinon, on commence par regarder la case voisine de droite : Si cette case n'est pas un mur et qu'elle n'appartient ni à la liste des cases "mortes", ni à celle des cases "à visiter", on ajoute alors la case voisine à la liste des cases "à visiter" ainsi que la case courante que l'on remplace. Si cette case est déjà présente dans l'une des deux listes ou qu'il s'agit d'un mur : On passe à la case voisine suivante (celle du bas). Si aucune des 4 cases voisines n'a été ajoutée, on place alors la case courante dans la liste des case "mortes". Si la liste des cases "à visiter" se trouve vide, c'est que la sortie ne fait pas partie des cases accessibles depuis l'entrée.

```
1 -> case M.lookup (C x (y+1)) cases of
2   Nothing -> error "Should not occur"
3   Just e -> let (C x1 y1, e1) = (C x (y+1), e) in
4             if notElem (C x1 y1, e1) v && e1 /= Mur && notElem (C x1 y1, e1
5             ) xs
6             then let xss = (xs <> [(C x1 y1, e1)]) in
                  aux1 (Carte larg haut cases) (v <> [(C x y, caase)]) ((C x
y, caase) <> xss)
```

5.2 Environnement

5.2.1 Envi

Notre environnement se compose d'un map contenant des Coordonnées Coord associés à une liste d'entités. Il y a quatre entités : L'avatar du héros (Player) et les monstres (mob), chacun d'eux ayant un identifiant, des points de vie, les mobs ayant un âge et le joueur un booléen indiquant si il a ramassé le trésor. Les autres entités sont le trésor et le piège.

5.2.2 Model

Le modèle de notre jeu se compose d'une Carte, d'un envi, d'une graine aléatoire ainsi que d'un type Keyboard. La fonction `prop_model_inv4` est une invariant qui permet de s'assurer que le trésors est accessible depuis l'entrée (et donc de la sortie par effet de cause de l'invariant 6 sur la carte). Il s'agit du même type de recherche que l'on effectue pour rechercher la sortie sauf que cette

fois il est nécessaire de regarder dans l'environnement pour vérifier les coordonnées du trésor.

5.2.3 State

L'état du jeu nommé `State` peut prendre trois formes différentes : `Perdu`, `Gagne` et `Tour`. Le type `Tour` contient Un numéro de tour, une carte, un environnement, un générateur aléatoire ainsi qu'une map d'objectifs.

Nous avons a disposition une fonction d'initialisation `init_state` qui permet d'initialiser un état à partir d'une carte. On y place sur des cases libres une liste d'entités composée d'un trésor, d'un piège ainsi que de plusieurs monstres. On fini par placer le joueur sur l'entrée, de stocker les objectif et de retourner l'état prêt pour le jeu.

5.3 Moteur de Jeu

5.3.1 Boucle non-réactive

L'avancement de l'état du jeu est basé sur le temps et les déplacements du personnage (induites par l'utilisateur). Nous utilisons la fonction `time` de `SDL.Time` qui rend un `CDouble` correspondant à l'instant¹ donné. En passant cette valeur dans `note_gameLoop`, nous pouvons initier un déplacement des `Mobs` toutes les N secondes. Pour cela, à la création d'un `Mob`, nous choisissons une valeur aléatoire qui servira pour la comparaison avec l'instant courant. L'idée est d'avoir un point de départ différent pour tout les `Mobs` ce qui rend leur manière de se déplacer plus réaliste.

5.4 Déplacement des mobs

Nous nous sommes inspirés de la solution proposé par le guide du sujet. Chaque `Mob` prévoit l'ensemble des actions qui sont possibles en regardant autour de lui. Si la liste obtenue n'est pas vide (il n'est pas condamné), une action est choisie aléatoirement et est effectué.

5.5 Main

Notre fichier `main` possède une fonction `loadGeneric` permettant de charger les différents fichier images (au format PNG ou BMP sur windows). La valeur de retour de cette fonction est une paire (`TextureMap`, `SpriteMap`) encapsulés dans une monade `IO`. Donc, afin de charger efficacement l'ensemble des images dans les deux map en question, nous utilisons la fonction `foldM` :

```
1 (tmp0', smp0') <- foldM (\(t,s) path->loadGeneric rdr path t s 0 0) (t0,s0) toLoad
```

De la même manière, afin d'afficher les *sprites* à l'écran, nous *mappons* la fonction `displaySprite :: Renderer -> TextureMap -> Sprite -> IO()` aux *sprites* qui sont récupérées grâce aux informations se trouvant dans la carte et dans l'état.

```
1 mapM_ (S.displaySprite renderer tmap) (fetchSpriteFromEnv gameState smap mpTiles)
```

Chaque élément de la liste (`[Sprite]`) retourné par la fonction `fetchSpritesFrom*` sera donné à la fonction `displaySprite` qui prouira une action de `IO` (l'affichage à l'écran). Nous utilisons la fonction `mapM_` (à noter l'*underscore*) car nous n'utilisons pas le résultat.

Lors de l'initialisation du game `State`, les montres, trésors et pièges sont ajoutés à des emplacements aléatoire.

La fonction `main` commence par initialiser la première carte générée aléatoirement ainsi que de charger les deux autres cartes "classiques". On initialise ensuite les 3 States avec les cartes que l'on a crée et on appelle ensuite la boucle de jeu qui va gérer les différents événements (lorsqu'une touche du clavier est pressée par exemple).

Si le jeu est dans un état de Victoire ou de Défaite, il affichera l'image associée et attendra que le joueur presse la touche "Escape". Si le jeu est dans un état "Tour", il continuera de faire tourner

1. Converti à partir du temps système.

le jeu et attendra les différentes actions du joueur.

Le jeu ne sera en état de "Victoire" qui si les 3 niveaux ont été passé avec succès.

6 Extensions

6.1 Générateur de niveaux aléatoire

Notre générateur aléatoire est défini dans le fichier Carte.hs. La consigne stipulant que le générateur devait retourner un niveau intéressant, il est évident que l'on ne pouvait pas générer chaque case aléatoirement. Le niveau, en plus de ne pas être forcément être saint (même si on a des invariants pour le vérifier) aurait forcément été incohérent d'un point de vue pratique : Des cases espaces creux sans accès auraient pu accueillir des monstres ou des pièges, des portes auraient pu être contournable (Si une porte est bien entourée de deux blocs de murs pour respecter l'invariant mais que les 3 blocs se trouvent au milieu d'un grand espace vide cela ne fait pas beaucoup sens...). Nous avons donc choisis une approche avec un pattern dans lequel nous plaçons les murs, les portes, l'entrée et la sortie de façon aléatoire mais qui reste cohérent et intéressant à jouer.

La fonction `carteVerifiee` prend en paramètre une graine aléatoire et doit retourner une carte. Elle

tire un nombre aléatoire conséquent et fait appelle à la fonction `carteGenerator` pour générer une carte. La fonction va vérifier avec un invariant si la carte est saine. Si c'est le cas : La carte sera alors immédiatement retournée, sinon, une nouvelle carte sera générée aléatoirement et testée. Si la carte générée n'a pas été saine dix fois de suite, alors la fonction retourne une erreur en signalant que la génération a échouée. (Cela afin d'éviter un boucle infinie).

La fonction `carteGenerator` prend un entier aléatoire en paramètre et retourne une carte. Elle dé-

compose ce nombre aléatoire en plusieurs autres nombres aléatoire afin de définir les emplacements des murs, portes, de l'entrée et de la sortie. Toutes ces variables sont liés et cela est voulut . En effet : Il est nécessaire que certaines variables soient définies en fonction des autres afin d'aider a construire une carte cohérente. (éviter que des murs ne bloquent les portes ou la sortie par exemple) La carte générée est stockée dans un fichier que l'on utilisera dans le main pour jouer le niveau.

6.2 Différents niveaux

Notre jeu est composé de 3 niveaux différents :

- Le premier niveau, généré aléatoirement par notre fonction.
- Le second niveau plus "classique".
- Le troisième niveau lui aussi déjà présent dans un fichier.

Il est nécessaire de gagner le niveau actuel pour accéder au niveau suivant. Il est possible de quitter n'importe quel niveau en appuyant sur "Échappe" (Cette possibilité nous a fait revoir la moitié de notre implémentation du main).

6.3 Combats

Lorsque le héros se trouve à proximité d'un monstre, (sur une case adjacente) il peut, en appuyant sur la touche "R" l'attaquer et réduire ses points de vie de 25. Une fois son nombre de points de vie à 0, le monstre est retiré du jeu.

6.4 Trésors

Lorsque le joueur se trouve sur une case adjacente au trésor, ce dernier peut appuyer sur la touche "R" pour le ramasser. Cette action retire le trésor du l'environnement du `gameState` et permet au joueur de gagner la partie si atteint la sortie (ou de passer au niveau suivant).

6.5 Piège

Le piège est clairement représenté sur la carte pour un intérêt pédagogique. Si le joueur marche dessus, il meurt et cela déclenche une défaite. (Une erreur survient sur windows en cas d'interaction "R" à coté du piège. Il signale un cas non exhaustif qui est pourtant traité de manière exhaustif)

7 Conclusion

Notre jeu se trouve être assez complet, il propose en effet des combats, des trésors à ramasser ainsi que des pièges à éviter qui sont des extensions que l'on est en droit d'attendre d'un jeu de ce genre. En plus de cela, notre jeu propose plusieurs niveaux ainsi que l'utilisation d'un générateur de niveau aléatoire qui permet une rejouabilité quasi infinie.