Laetitia PHAM
Guillaume RIDEY

Projet de PAF
Rapport

Juin 2020

Science et Technologie du Logiciel
Sorbonne Université
Année 2019/2020

1 Description du projet

L'objectif principal de notre projet est de développer un jeu vidéo sûr, de type *Dungeon Crawler*, dans le langage *Haskell*. Durant ce processus, nous avons réalisé un scénario de jeu que nous allons vous décrire.

Le joueur, incarné par notre héros, doit atteindre le coffre au trésor situé dans le niveau final. De nombreux obstacles l'attente, en effet, ce donjon est peuplé de terribles monstres qui si le héros n'est pas muni d'une épée, pourront le tuer. Dans ce cas-là, le niveau courant *reset*. Les items à disposition du héros seront des épées, pour éliminer les monstres lorsque celui-ci les touchera et des clefs qui permettent d'ouvrir les portes. Par ailleurs, pour attendre les prochains niveaux, le héros devra trouver les escaliers.

1.1 Manuel d'utilisation

Pour exécuter le projet depuis le terminal, il suffit de lancer la commande:

```
stack run
```

Ainsi notre jeu, *Dungeon*, démarrera. Pour déplacer le personnage, il suffit d'utiliser les touches z (Nord), q (Ouest), s (Sud) et d (Est).

Par ailleurs, après avoir récupérer un clef, pour ouvrir une porte il suffit d'appuyer sur la touche "*k*" lorsque le héros est devant celle-ci. De plus, il y a deux manières de quitter le jeu, d'abord en fermant la fenêtre (avec la petite croix rouge) ou en appuyant sur la touche "*escape*".

Pour lancer les tests de *Spec* et les *QuickChecks*, il suffit de lancer la commande:

```
stack test
```

1.2 Les Extensions

Les extensions choisies dans notre projet sont :

Déplacement du héros : Celui-ci se déplace case par case et un *scrolling* de la carte est présente, c'est-à-dire que peu importe le déplacement que fera le héros notre carte défilera de telle sorte que le héros soit au milieu de celle-ci.

Scènes et Niveaux : Notre jeu possède différents états définissant des scènes telles que la scène d'introduction ou la scène de fin de jeu. Par ailleurs, il y a différents niveaux. Le héros est transporté à l'entrée du niveau suivant en empruntant les escaliers et le dernier niveau comporte un coffre au trésor qui permet de gagner le jeu.

Diversité des Monstres : Il y a cinq types de monstres, parmi eux, deux d'entre eux sont spéciaux. En effet, les fantômes sont les seuls monstres pouvant traverser les murs et les autres monstres. Tandis que les monstres "undead" (non-vivant) n'ont pas un cheminement prédéfini de direction contrairement aux autres.

```haskell
-- Situé dans Monster.hs
type Gen a = State R.StdGen a
class Arb a where
  arb :: Gen a
instance Arb Int where
  arb = state R.next
generate :: Gen Int
generate = liftM (`mod` 1000) arb
-- Retourne une direction arbitraire, ici n est la somme des coordonées du monstre plus celle se son compteur
getArbitraryDir :: Int -> String
getArbitraryDir n = let index = (foldr (+) 0 (evalState (sequence (replicate 10 (generate))) (R.mkStdGen n)))
    in ["Haut", "Bas", "Droite", "Gauche"]!!(index 'mod' 4)
```

Objets du jeu : Le héros peut récupérer une épée, dans ce cas-là il changera de couleur, mais également une clef. Si son inventaire est plein l'item restera au sol, en effet, il ne peut s'équiper que d'une épée et d'une clef. Par ailleurs, les items ne peuvent être utilisés qu'une seule fois.

Attaquer des monstres : Le héros ne possède qu'une vie, or si un monstre l'attaque c'est-à-dire rentre en collision avec celui-ci, le niveau redémarre. Cependant les monstres aussi ne possèdent qu'une vie, ainsi si le héros est équipé d'une épée il pourra les tuer.

Clefs et Portes : Le jeu possède plusieurs obstacles pour attendre le trésor, dont les portes qui ne peuvent être ouvertes qu'en trouvant une clef. De plus, pour que la porte s'ouvre il faut appuyer sur la touche "k" du clavier.

1.3 Instructions pour créer des niveaux

Nous avons fait en sorte que la lecture d'un seul fichier texte soit utilisé pour initialiser un niveau, que ce soit pour la carte, les monstres ou les items. En effet, il existe une méthode *"readCarte"* dans les fichiers *Carte.hs, Monster.hs et Items.hs* qui permet d'initialiser chaque éléments du niveau. Celle fonction va analyser chaque ligne du texte pour récupérer les informations dont elle a besoin. Voici un exemple de la méthode *"readCarte"* situé dans *Monster.hs*.

```haskell
-- Recupere les données du fichier texte pour initialiser la liste des monstres
readCarte :: String -> [Monstre]
readCarte [] = []
readCarte txt = getMonsters  txt 0 0
    where getMonsters  (x:xs) cx cy =
        (if xs == []
            then []
            else case x of
                'o' ->(Monster Orc (C.C cx cy) 0 (getCptInit Orc) True):(getMonsters  xs (cx+50) cy)
                'f' ->(Monster Fantome (C.C cx cy) 0 (getCptInit Fantome) True):(getMonsters  xs (cx+50) cy)
                's' ->(Monster Skeleton (C.C cx cy) 0 (getCptInit Skeleton) True):(getMonsters  xs (cx+50) cy)
                'd' ->(Monster Demon (C.C cx cy) 0 (getCptInit Demon) True):(getMonsters  xs (cx+50) cy)
                'u' ->(Monster Undead (C.C cx cy) 0 (getCptInit Undead) True):(getMonsters  xs (cx+50) cy)
                '\n'-> getMonsters  xs 0 (cy+50)
                otherwise -> getMonsters  xs (cx+50) cy)
```

Ainsi pour créer un nouveau niveau, il faut d'abord ajouter un fichier texte dont le nom commence par "monde" puis le numéro du niveau. Ensuite, lors de l'élaboration de la nouvelle carte, il faut que celle-ci respecte les invariants de type **Carte** expliqués dans la section suivante.

```haskell
-- Changement de niveau dans Main.hs
gameLoop frameRate renderer tmap smap kbd gameState@(M.GameState _ _ _ _ _ _ _ _ (_,n,True) _) cpt = do
  putStrLn ("assets/monde"++(show (n+1)) ++".txt")
  file <- liftIO $ try $ readFile ("assets/monde"++(show (n+1)) ++".txt")
  case file :: Either IOException String of
    Left exception -> gameLoop frameRate renderer tmap smap kbd
                        (M.Bug "ERROR: Fichier du prochain niveau non trouvé.") cpt
    Right contents -> (do if (Carte.prop_post_readCarte contents)
       then gameLoop frameRate renderer tmap smap kbd (M.createGameState contents (n+1)) (cpt+1)
       else gameLoop frameRate renderer tmap smap kbd (M.Bug "ERROR: Carte du niveau suivant incorrect.") cpt)
```

Puis il suffit de rajouter les monstres et les items : "c" pour clef, "e" pour l'épée, "t" pour le trésor et "E" pour les escaliers. De plus, le dernier niveau doit posséder un trésor sinon le jeu finira sur la scène d'erreur.

2 Propositions Implémentées

2.1 Carte

Dans notre fichier *Carte.hs*, les types utilisés sont **Case**, **Coord** et **Carte**. En effet, notre carte est représentée par une table d'association entre des coordonnées (toutes multiples de cinquante) et des cases. Les propositions implémentées dans ce fichier sont :

Invariant du type Coord : La proposition *"prop_Coord"* vérifie que les coordonnées ne soient pas négatives et que celles-ci soient inférieures ou égales à la taille de la carte, hauteur et largeur, également passé en argument.

Invariants du type Carte : La proposition "prop_inv_Carte" vérifie l'ensemble des propositions liées à la carte. Dans un premier temps, cela vérifie la taille de la carte (prop_TailleCarte), c'est-à-dire la hauteur et la largeur par rapport aux nombres de cases, mais également l'existence de chaque case. Puis, la proposition "prop_CaseCarte" est appelée, elle vérifie que chaque case possède des coordonnées correctes grâces à l'invariant de *Coord* et que la carte soit bien entourée de murs. Ensuite, "prop_CasePorte" est la proposition qui permet de vérifier que chaque porte soit bien maintenue par des murs. Finalement, "prop_EntreeCarte" vérifie s'il y a bien une entrée dans la carte, qui sera le positionnement par défaut du héros lors de l'initialisation du niveau.

Opération ReadCarte : Cette opération permet d'initialiser une carte à partir d'un *String*, obtenue à la lecture d'un fichier. La pré-condition, "prop_pre_readCarte", vérifie si le *String* n'est pas vide. La post-condition, "prop_post_readCarte", teste l'opération et vérifie que la carte retournée par celle-ci valide l'invariant de Carte.

Opération openDoor : Cette opération permet d'ouvrir une porte, c'est-à-dire transformé la case représentant la porte en une case vide, s'il y en a une autour des coordonnées données en paramètre. La pré-condition, "prop_pre_openDoor", vérifie que la carte soit valide par l'invariant de Carte et que les coordonnées de position soient comprise dans la carte.

La post-condition, "prop_post_openDoor", vérifie que la carte retournée soit valide par l'invariant de Carte et que la porte ait bien été ouverte, donc qu'il y avait bien une porte à ouvrir.

2.2 Monstres

Dans notre fichier *Monster.hs*, les types utilisés sont **Monstre** et **Espece**. En effet, nos monstres représentent des entités et sont définis par une espèce, une coordonnée, une direction, un compteur et un booléen d'affichage. Les propositions implémentées dans ce fichier sont :

Invariant du type Monstre : La proposition "prop_inv_Monstre" vérifie l'ensemble des invariants d'un Monstre. La proposition "prop_inv_coord_monstre" vérifie que les coordonnées d'un monstre ne soient pas négatives et qu'elles soient bien multiples de 50.

La proposition "prop_inv_direction_monstre" vérifie que la direction du monstre, représentée par un entier, soit positive et strictement inférieur à la longueur du pattern du monstre.

La proposition "prop_inv_cpt_monstre" vérifie que le compteur, représentant le nombre de pas que le monstre fait dans une direction, ne soit pas négatif et inférieur ou égale à sa valeur initiale.

Opération getMonsterPattern : Cette opération nous permet de récupérer un vecteur de direction correspondant au pattern d'une Especе. Il n'y a pas de pré-condition car la fonction prend en entrée que l'Especе qui est un paramètre déjà vérifié par le système de type de *Haskell*. La post-condition, "prop_post_getMonsterPattern", vérifie que le pattern rendu ne contient que des directions valides.

Opération moveToDir : Cette opération permet de modifier les coordonnées selon la direction choisie. La pré-condition, "prop_pre_MoveToDir ", vérifie que le *String* correspond bien à une direction valide et que les coordonnées initiales soient multiples de 50 et positives. De manière analogue, la post-condition, "prop_post_MoveToDir", vérifie cela avec les coordonnées retournées par l'opération.

Opération initMonster : Cette méthode nous permet d'initialiser des monstres selon une liste de couples, composé de coordonnées et de *String* décrivant l'espèce du monstre. La pré-condition, "prop_pre_initMonstres ", vérifie que les *String* correspondent bien à des types Especе. Elle vérifie aussi que les coordonnées soient positives et multiples de 50. La post-condition, "prop_post_initMonstres", vérifie que tous les monstres de la liste obtenue respectent tous les invariants des Monstres.

Opération moveMonster : Cette opération nous permet de déplacer un monstre dans une direction, elle prend en paramètre un monstre et rend un monstre. La pré-condition "prop_pre_moveMonster" vérifie que le monstre en entré respecte l'ensemble des invariants des Monstres. De plus, la post-condition "prop_post_moveMonster" vérifie la même chose avec le monstre en sorti de fonction.

Opération elimineMonster : Cette opération change le booléen d'affichage des monstres se trouvant aux coordonnées passées en paramètre. La pré-condition "prop_pre_elimineMonstres" vérifie que l'abscisse et l'ordonnée passées en paramètre soit positives et multiples de 50. Elle vérifie aussi que tous les monstres de la liste passée en paramètre respecte l'ensemble des invariants des Monstres. De la même manière, la post-condition "prop_post_elimineMonstres" vérifie cela avec la liste des monstres en sortis et que l'affichage des monstres éliminés soit bien passée à faux.

Opération collisionMonstres : Cette méthode vérifie si un des monstres de la liste se situe à la même position que les coordonnées fournis en paramètre. La pré-condition, "prop_pre_collisionMonstres", vérifie donc que ces coordonnées soient correctes ainsi que chaque monstre de la liste valide l'invariant des Monstres.

Opération readCarte : Cette fonction permet d'initialiser une liste de monstres à partir d'un *String*. La post-condition, "prop_post_readCarte", teste si chaque monstre créé valide bien l'invariant des Monstres.

2.3 Items

Dans notre fichier *Item.hs*, les types utilisés sont **Type** et **Item**. En effet, nos items sont définis par un type et un booléen d'affichage. Les propositions implémentées dans ce fichier sont :

Invariant de type Item : L'invariant, "prop_inv_ItemType", vérifie que le type de l'item ne soit pas de type *ErrorItem*.

Opération initItems : Cette fonction permet d'initialiser les items grâce à une liste de couple, coordonnées et *String*. La pré-condition "prop_pre_initItems" vérifie que les coordonnées soient correctes et que les *String* correspondent à des Type. Quant à la post-condition "prop_post_initItems", celle-ci teste si les items dans la liste initialisée sont valides.

Opération changeItems : Cette opération désactive l'affichage d'un item.
La pré-condition "prop_pre_changeItems" vérifie que l'item de ce type existe et que les coordonnées passées en paramètre le soient aussi. La post-condition "prop_post_changeItems" vérifie que l'item en question a bien l'affichage désactivé et que l'item soit toujours valide selon l'invariant des Items.

Opération readCarte : Cette fonction permet d'initialiser les items grâce au *String* provenant de la lecture d'un fichier. La post-condition "prop_post_readCarte" vérifie les items initialisés soit valide et que leur affichage soit à vrai.

2.4 Modèle

Dans notre fichier *Model.hs*, le type utilisé est **GameState**. En effet, celui-ci représentent un état du jeu à un instant donné.
Il y a l'état "Title" qui est composé de la lecture du fichier du premier niveau, l'état "Victory", l'état "Bug" composé d'un message d'erreur, l'état "End" composé de deux entiers et d'un *String*, et finalement l'état "GameState". Cette état est composé des positions du personnage, de sa vitesse de déplacement, des items dont il est équipé ou non, de la carte courante, d'une liste de monstres, d'une map avec pour clef des coordonnées et pour valeur un item et enfin d'un triplet composé de la carte initiale, du numéro de niveau et un booléen précisant s'il réussit le niveau ou pas.
Les propositions implémentées dans ce fichier sont :

Invariant du type Model : La proposition "prop_inv_GameState" vérifie l'ensemble des invariants du GameState. La proposition "prop_inv_Perso" vérifie que les coordonnées du personnage soient positives, inférieur à la taille de la carte et multiples de 50. La proposition "prop_inv_Monsters" vérifie que tous les monstres, présent dans l'état *GameState*, respectent l'ensemble des invariants des Monstres. La proposition "prop_inv_Items" vérifie que tous les items, présent dans l'état *GameState*, respectent l'ensemble des invariants des Items.

Opération createGameState : Cette fonction permet d'initialiser le *GameState*. Elle prend en paramètre la lecture du fichier, représentant le niveau, et le numéro du niveau. La pré-condition "prop_pre_createGameState" vérifie que le *String* ne soit pas vide et que le niveau soit positif. La post-condition "prop_post_createGameState" vérifie l'ensemble des invariants sur le *GameState* créé.

Opérations moveLefyt, moveRight, moveUp et moveDown : Ces quatre opérations permettent de faire bouger notre personnage d'un pas dans la direction voulue. Elles prennent en entré le *GameState* et le retourne modifié. La pré-condition "prop_pre_move" vérifie que le *GameState* en entré respecte l'ensemble des invariants du **GameState**. La post-condition "prop_post_move " vérifie aussi cela avec le *GameState* retourné.

Opération moveMonsters : Cette fonction permet de faire avancer d'un pas tous les monstres lorsque le compteur est à zéro. De plus, si un monstre avance vers une case inaccessible comme un mur ou la prochaine position d'un autre monstre, ce monstre passera son tour (sauf si c'est un fantôme). La pré-condition "prop_pre_move-Monsters" verifie que le compteur soit positif et que le *GameState* en entré respecte tout les invariants du **GameState**. La post-condition "prop_post_moveMonsters" vérifie que le gameState en retours respecte bien l'ensemble des invariant.

Opération changeItems : Cette fonction permet d'équiper l'item sur la même case que le héros s'il y en a un. La pré-condition "prop_pre_changeItems" vérifie que le *GameState* en entrée respecte l'ensemble des invariants du **GameState**. La post-condition "prop_post_changeItems " vérifie l'ensemble des invariants sur le *GameState* en sorti.

Opération changeMonstres : Cette fonction gère les collisions entre le héros et les monstres. La pré-condition "prop_pre_change-Monstres" vérifie que le *GameState* en entré respecte l'ensemble des invariants du **GameState**. La post-condition "prop_post_changeMonstres " vérifie l'ensemble des invariants sur le *GameState* en sorti.

Opération activePorte : Cette méthode permet d'ouvrir une porte si le héros possède une clef. La pré-condition "prop_pre_active-Porte" vérifie que le *GameState* en entré respecte l'ensemble des invariants du **GameState**. La post-condition "prop_post_active-Porte " vérifie l'ensemble des invariants sur le *GameState* en sorti.

3 Tests implémentés

Dans le fichier **Spec.hs**, nous testons toutes les propositions que nous avons implémentées en appelant la méthode "cFunSpec" de *CarteSpec.hs, ItemsSpec.hs, MonsterSpec.hs, ModelSpec.hs* et les **QuickChecks**: *MonstreQuickCheck.hs et ItemsQuickCheck.hs.*

Pour les tests des *Spec.hs*, nous générons manuellement un état initial qui respecte les pré-conditions d'une opération. Ensuite nous testons la post-condition qui applique l'opération et vérifie si le résultat est valide. De plus, nous générons également des états incorrects qui ne doivent pas être validé par la pré-condition de l'opération, ou par la post-condition. Ainsi nous obtenons des tests variés pour chaque opération.

En ce qui concerne les *QuickChecks*, nous en avons généré pour les monstres et les items, pour cela nous avons créé des générateurs de types correspondant aux paramètres d'entrées des fonctions que nous voulions tester. Cette génération est aléatoire, mais tout de même restreinte par les pré-conditions. Nous générons ensuite des listes de 100 éléments qui seront par la suite utilisées comme paramètres des fonctions à tester et nous vérifions que le résultat retourné valide les post-conditions.

Les propositions testés dans *QuickCheck* sont :
- Pour Items :
  - prop_inv_ItemType
  - prop_pre_initItems
  - prop_post_initItems
  - aux_prop_coord_changeItems
- Pour Monster :
  - prop_inv_Monstre
  - prop_post_getMonsterPattern
  - prop_pre_initMonstres
  - prop_post_initMonstres
  - prop_pre_moveMonster
  - prop_post_moveMonster
  - prop_pre_elimineMonstres
  - prop_post_elimineMonstres
  - prop_pre_collisionMonstres