1^{re} année Master STL

Rapport de projet

Dungeon crawler sur en haskell

Groupe :
Baaloudj Hakim
Bello Pablito

Juin 2020

Résumé

Ce document a été rédigé dans le cadre de l'unité d'enseignement Programmation avancée fonctionnelle. Le but est de mettre en pratique toutes les notions de modélisation et de logiciel sur vu en cours, dans la création d'un jeu vidéo de type "dungeon crawler" avec comme langage Haskell.

Table des matières

1 Manuel

1.1 Organisation des fichiers

Voici les différents dossiers du projet :

— app : contient le point d'entrée du code.
— lib : contient deux répertoires, un pour les assets du jeu et un pour les maps utilisés.
— src : code source
— test : contient l'ensemble des tests du projet

1.2 Dépendances

Il faut installer la librairie "Key" avec "cabal install keys", son utilisation est détaillée dans la partie qui parle du Main.

1.3 Utilisation

1.3.1 Création d'une carte

Voici un petit tutoriel pour créer une carte valide pour notre jeu. Chaque case correspond à une lettre de l'alphabet, et la composition de votre carte doit respecter plusieurs propriétés qui composent l'invariant de la carte.

— X -> Mur
— E -> Entrée
— S -> Sortie
— | -> Porte Nord Sud
— - -> Porte Ouest Est
— C -> Coffre
— T -> Trap (piège)
— P -> Portail de téléportation
— L -> Levier
— Toute autre case -> Case vide (un sol)

Et les propriétés que la carte doit respecter

— La carte doit être entourée de cases Mur
— Toute les portes ont deux murs de part et d'autre
    Exemple : X-X
— Il doit y avoir exactement une sortie et une entrée

— Il doit y avoir au moins un coffre
— les dimensions minimales que la carte doit respecter sont 3 en longueur et 10 en largeur (pour que l'affichage de l'inventaire et de la vie soit correct)
— Toutes les cases sont contenues dans la largeur et la longueur de la carte
— Le nombre de portails doit être divisible par 2, car ils viennent toujours par pair, et le nombre de leviers doit être égal au nombre de pièges additionné au nombre de portails divisé par 2, car pour une paire de portails, un seul levier seras connecté a eux.

Voici un exemple de carte valide :

```
XXXXXXXXXXXXXXXXXXXXXXX
X     |  S  T X        X
X  X  XXX-XX  X   P    X
X  X          X     L  X
X  XXXX       XX       X
X C   X  C     XXXXXXXXX
X    LX                X
XXXXXTX          XXXXXXXX
XE    P           |  L  X
XXX               XXXXXXXX
X                       X
X       C               X
XXXXXXXXXXXXXXXXXXXXXXX
```

1.3.2 Lancer le jeu

Pour exécuter le projet, on tape la commande run de stack, en passant en argument la carte sur laquelle on souhaite jouer. Les cartes jouables, c'est-à-dire les cartes qui respectent la spécification (décrite plus bas) se trouvent dans lib/maps/validMaps. Voici un exemple :

```
stack run lib/maps/validMaps/realmap.txt
```

Vous pouvez aussi d'abord lancer tous les tests pour être sur que votre carte est valide. (tous les autres tests passent mais ceux liés à la carte dépendent de la carte qu'on met dans le dossier. "validMap")

1.3.3 Comment jouer

Dès lors, le jeu se lance. Voici les différentes touches du clavier utilisées :
— Z : déplacer le personnage vers le haut.
— Q : déplacer le personnage vers la gauche.
— S : déplacer le personnage vers le bas.
— D : déplacer le personnage vers la droite.

— P : lancer une attaque sur les 4 cases adjacentes au personnage.
— Espace : interagir avec les portes/coffres/leviers sur les 4 cases adjacentes au personnage.
— Esc : quitter le jeu.

Les tests se lancent quant à eux simplement avec la commande suivante :

```
stack test
```

2 Présentation du projet

L'objectif du projet est la création d'un "dungeon crawler" sur, en 2D en utilisant la librairie externe SDL2 en Haskell.

Le joueur déplace son personnage dans un labyrinthe représenté par une grille discrète (les déplacements se font case par case.). Le labyrinthe contient des monstres, des pièges, des portes et des trésors.

L'objectif du joueur est de trouver la sortie, ou/et de rapporter à la sortie un trésor particulier.

Quand on parle de jeu vidéo "sûr", cela implique, en plus de l'implémentation du jeu, le respect d'un certain nombre de principes de modélisation de code et de type que nous nous sommes efforcé de suivre et comprendre, et d'une batterie de tests qui vérifient des propriétés qu'on aura au préalable décidé et codé.

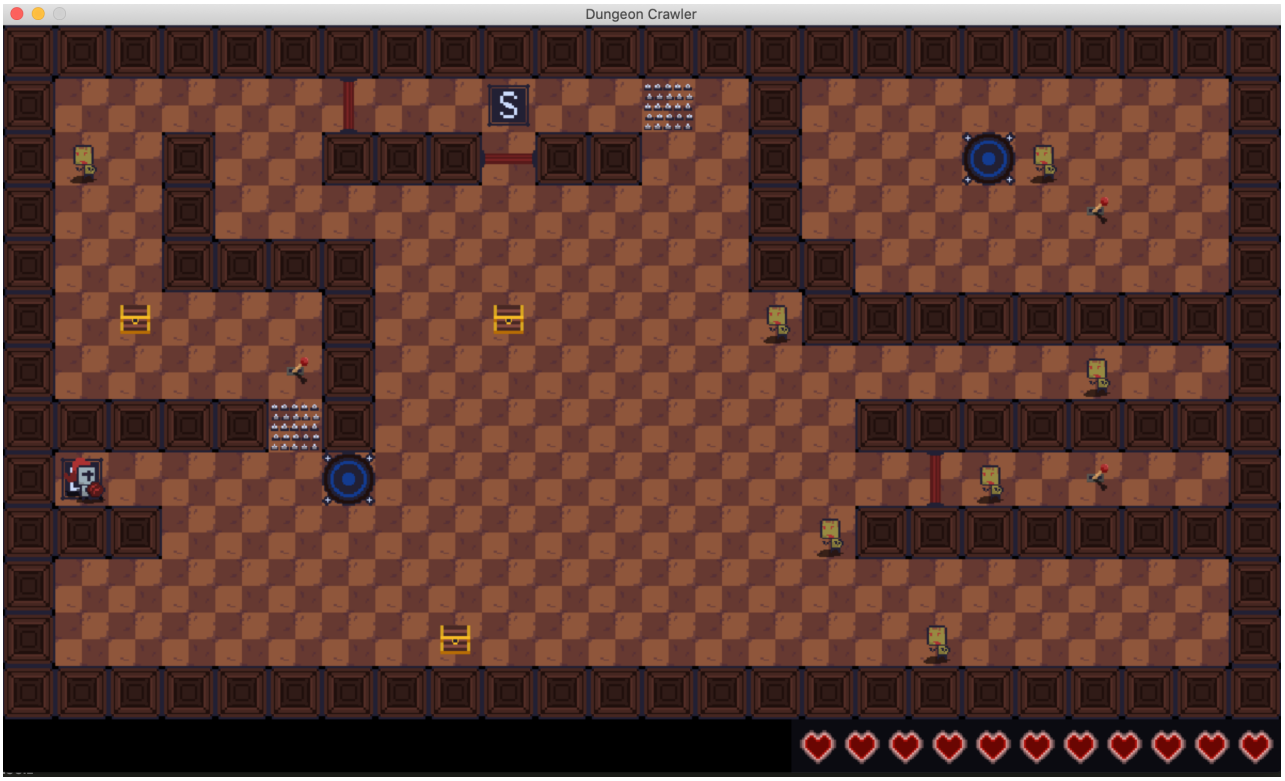
— Principe 1 : Les entités sont des sommes de produits/records

— Principe 2 : Les états remarquables ont des constructeurs dédiés

— Principe 3 : les états incohérents ne sont pas constructibles

— Principe 4 : Les entités ont (le plus souvent) des invariants (specifiés dans la prochaine partie, Propositions)

Dans la partie **Création d'une map** nous avons donné un exemple de carte valide, voici son affichage une fois en jeu.



3 Propositions

3.1 GameMap

Invariant du type GameMap :

— La carte doit etre entourée de case Mur

— Toute les portes ont deux murs de part et d'autre
Exemple : X-X

— Il doit y avoir exactement une sortie et une entrée

— Il doit y avoir au moins un coffre

— les dimensions minimales que la carte doit respecter sont 3 en longueur et 10 en largeur (pour que l'affichage de l'inventaire et de la vie soit correct

— Toute les cases sont contenu dans la largeur et la longueur de la carte

— Le nombre de portails doit être divisible par 2, car ils viennent toujours par pair, et le nombre de leviers doit être égal au nombre de pièges additionné au nombre de portails divisé par 2, car pour une paire de portails, un seul levier seras connecté a eux.

Pré-conditions des fonctions :

— checkCell : Les coordonnées sont incluses dans la hauteur et la largeur de la gamemap

— interactDoor : idem

— interactChest : idem

— interactSink : idem

3.2 Entité

Invariant du type Entité :

— Si l'Entité possède des points de vies, ils doivent être supérieurs ou égaux à zéro.

— Si l'Entité est un Joueur en état Unarmed, l'épée ne doit pas être contenue dans l'inventaire, s'il est en état Armed, l'épée doit être contenue dans l'inventaire.

— Si l'Entité possède un inventaire, il ne peut contenir que des entités immobiles.

3.3 Environnement

Invariant du type Environnement :

— Chaque Entité présente dans l'Environnement doit respecter l'invariant des Entité.

— Les Entité dans l'Environnement doivent toute avoir un id unique (donc on ne peut pas avoir d'Entité en double.).

Pré-conditions des fonctions :

— removeId : L'Entité donnée doit exister dans l'Environnement donné.

— moveId : L'Entité donnée doit exister dans l'Environnement et la coordonnée donnée doit exister dans l'Environnement donné.

— updateEntity : les id des deux Entité données doivent correspondre.

3.4 Model

Invariant du type Model :

— Les invariants de GameMap et Environnement doivent être respectés.

— L'ensemble des coordonnées de GameMap doit être égal à l'ensemble des coordonnées de Environnement.

— Un coffre doit contenir un seul trésor.

Pré-conditions des fonctions :

— move : L'Entité donnée doit exister dans l'Environnement du Model donné et la coordonnée donnée doit exister dans l'Environnement du Model donné.

— tour : L'Entité donnée doit exister dans l'Environnement du Model donné.

— interactWithChest : La coordonnée donnée doit exister dans la GameMap du Model donné.

— interactWithDoor : La coordonnée donnée doit exister dans la GameMap du Model donné.

— canMoveTo : L'Entité donnée doit exister dans l'Environnement du Model donné.

— canOpen : La coordonnée donnée doit exister dans la GameMap du Model donné.

— canAttack : L'Entité donnée doit exister dans l'Environnement du Model donné.

— attack : L'Entité donnée doit exister dans l'Environnement du Model donné et la liste des Entité attaquables donnée doit correspondre au résultat de canAttack.

— cleanAttacked : L'Entité donnée doit exister dans l'Environnement du Model donné.

— possibleSolutions : L'Entité donnée doit exister dans l'Environnement du Model donné.

— playerIsNear : L'Entité donnée doit exister dans l'Environnement du Model donné.

— provide : L'Entité donnée doit exister dans l'Environnement du Model donné.

— decide : L'Entité donnée doit exister dans l'Environnement du Model donné et la liste d'Ordre donnée doit contenir au moins un Ordre.

3.5 Engine

Invariant du type GameMap :

— Respecter l'invariant de gamemap

— Respecter l'invariant de environnement

— "ent round" et "env round" doivent être cohérent, car "ent round" contient les entités de l'environnement

Pré-conditions des fonctions :

— setEntRound : L'environnement doit etre correct

— newState : le round ne doit pas être le premier et le model doit être valide

4 Tests

Les fichiers de test se trouvent dans le dossier test, chaque fichier test les fonctions du fichier correspondant dans src, sauf Spec.hs, qui lance tous les tests. Tous les fichiers contiennent des tests Hspec.

4.1 Hspec généralités

Pour chaque fonction d'un fichier, on créer une fonction qui va vérifier que la fonction originelle respecte les invariants nécessaires et les pré/post conditions.

Prenons ici l'exemple de la fonction canMoveTo dans Model.hs, celle-ci prend un Model, une Entité, et renvoie l'ensemble de direction vers lesquels l'Entité peut se déplacer. Voici sa fonction prop_canMoveTo :

```
prop_canMoveTo :: Model -> Entite -> Property
prop_canMoveTo model entity =
property $ (modelInv model) && (canMoveToPre model entity)
```

Cette fonction renvoie une propriété qui vérifie que canMoveTo respecte l'invariant et sa pré-condition, si canMoveTo renvoyais un nouveau Model, il aurait fallu vérifier que le model renvoyer respecte également l'invariant.

Une fonction canMoveToSpec est également créer :

```
canMoveToSpec model = do
    describe "canMoveTo" $ do
        it "Respecte l'invariant et sa pre condition" $ do
            property $ prop_canMoveTo model (exJoueur1)
        it "Retourne la liste des directions dans vers lesquels
            l'entite peux bouger" $ do
            canMoveTo model (exJoueur1) `shouldBe` [N]
```

C'est cette fonction qui sera appelée pour le test de canMoveTo, on vérifie dans un premier temps que canMoveTo respecte l'invariant et sa précondition, puis qu'elle retourne le résultat attendu, dans cet exemple le Joueur exJoueur1 ne peut bouger qu'au Nord.

Cette façon, de procéder, est commune à toutes les fonctions du projet, a part pour les tests de GameMap où on va mapper dynamiquement des fonctions qui testent des propriétés de l'invariant GameMap sur des dossiers remplis de fichiers qui illustrent a chaque fois la propriété qui doit être respectée (et les fichiers ne la respectent pas, volontairement pour montrer que nos fonctions détectent bien les cas invalides).

4.2 QuickCheck

Par manque de temps, nous avons utilisé quickcheck de manière très sommaire. Nous avons écrit des générateurs dans EntiteSpec.hs et EnvironnementSpec.hs pour créer un générateur d'environnements qui respectent l'invariant de Environnement.hs

5 Implémentation

5.1 Jeu de base

Dans cette section, nous allons parler du projet de base, qui correspond au rendu 1. Premièrement, nous allons parler de la modélisation du projet par les types. Nous avons suivi le guide proposé dans le sujet, voilà donc la structure du jeu.

- Un module GameMap qui encapsule les types et les fonctions implémentant la carte de jeu. En résumé : une Map strict de coordonnée vers des cases, une hauteur et une largeur, et on stocke ces 3 informations dans un enregistrement.
- Un module Entite qui encapsule les types et les fonctions implémentant les entités du jeu, le joueur, les ennemies...
- Un module Environnement qui encapsule les types et les fonctions implémentant l'environnement du jeu, c'est-à-dire l'état des entités sur une carte donné à un instant donné. En résumé : une Map strict de coordonnées vers une liste d'entités, et des fonctions utilitaires pour gérer et accéder aux entités dans l'environnement
- Un module Model qui encapsule les types et les fonctions implémentant le modèle du jeu, ou, autrement dit, toute la logique du gameplay. En résumé : un enregistrement stockant une carte, un environnement, un générateur standard, un log, un gestionnaire d'événement clavier, et des fonctions qui implémentent le gameplay de base, mouvements du joueur et des ennemies, intéractions, etc...
- Un module Engine qui encapsule les types et les fonctions implémentant le moteur du jeu, autrement dit l'état du jeu a un certain instant (un tour/round), c'est le moteur qui, a chaque tour, relance le modèle du jeu et récupère son nouvel état pour le transmettre au Main et avoir ainsi le nouvel état du jeu à tour donné.
- Un module Keyboard récupéré sur le TME6, gère les événements claviers
- Un module Sprite récupéré sur le TME6, qui gère les images en tant que tel.
- Un module SpriteMap récupéré sur le TME6, encapsule une map qui nous permet de lier des chaînes de caractères a des images, et gérer cette map.
- Un module TextureMap récupéré sur le TME6, qui gère le chargement de texture a partir de fichiers.
- Un module Main qui concentre toute la partie effet de bord de notre projet, on charge les assets, on initialise la carte, l'environnement, le modèle, le moteur... Et ont appel la boucle de jeu qui s'occupe d'afficher l'état du jeu a un instant T.

5.1.1 Points forts/important

Un des premier point important dans l'implantation du jeu de base est principalement en rapport avec le model du jeu.

On trouve au sein du model les fonctions `tour`, `provide` et `decide`. Ces fonctions permettent respectivement de jouer le tour d'une entité, récupérer l'ensemble des actions possibles pour

une entité autonome lors de son tour et enfin choisir parmi ces actions possibles laquelle l'entité doit effectuer.

La fonction tour parcours donc l'ensemble des entités, si l'entité est un joueur, on prend la dernière entrée clavier pour effectuer l'action qui y est associée, s'il n'y a pas d'entrée clavier, on ne fait rien. Si l'entité est une entité autonome, `provide` est appelée :

```
provide :: Model -> Entite -> [(Int, Ordre)]
provide model zomb@(Zombie _ _ _) =
  let l = possibleSolutions model zomb in
  -- si le Zombie peut attaquer
  if (canAttack model zomb) /= [] then [(100, A)]
  -- si un Joueur est dans le champ de detection du Zombie
  else if isJust $ playerIsNear model zomb then
    [(100, fromJust $ playerIsNear model zomb)]
  -- sinon, comportement par defaut
  else aux 0 l
  where
    aux :: Int -> [Ordre] -> [(Int, Ordre)]
    aux acc [] = []
    aux acc (t:q) =
      case t of
        R -> (100 - acc, R):(aux 100 q)
        otherwise -> (20, t):(aux (acc + 20) q)
```

Elle renvoie une liste d'action possibles pour l'entité, chaque action étant associée à un pourcentage de chances de l'effectuer.

On récupère d'abord toutes les actions liées aux déplacements que le zombie peut effectuer avec un appel à `possibleSolutions`. Ensuite, on regarde si le zombie peut attaquer ou s'il détecte un joueur, mais concerne des extensions. Ici c'est la partie "comportement par défaut" qui nous intéresse. A partir de la liste des déplacements possibles, on fixe un pourcentage de chance de faire chaque déplacement à 20% de chances, puis le reste va à "ne rien faire". Ce choix est bien sûr totalement arbitraire.

Si une entité peut donc se déplacer au Nord et au Sud, elle aura 20% de chances d'aller au Nord, 20% d'aller au Sud, puis 100 - 20 - 20 = 60% de chances de ne rien faire. Le tableau final serait [(N,20),(S,40), (R, 100)], cela indique simplement que si on tire un nombre entre 1 et 20 le zombie ira au Nord (20% de chances), entre 21 et 40 il ira au Sud (20% de chances) et entre 41 et 100 il ne fera rien (60% de chances).

Si elle peut aller dans les quatre directions, alors elle aura uniquement 20% de chances de ne rien faire. Un autre choix possible aurait été de fixer les chances de ne rien faire et d'adapter le pourcentage des autres actions en fonction.

Finalement, la fonction `decide` tirera un nombre entre 1 et 100, et applique l'action associée.

Un autre point fort dont nous sommes fier et qu'on aimerais mettre en avant dans ce rapport car notre trouvons cette solution élégante. C'est la gestion des tests automatique. Effectivement nous avons vite eu la problématique de devoir tester beaucoup de configuration de cartes différentes, et de ne pas le faire a la main.

Nous avons donc codé un système qui récupère automatiquement les fichiers dans des dossiers précis, et qui applique les tests sur les fichiers texte de carte. C'est classique jusque la, mais nous avons eu a nous poser certaines questions qui ont trouvé solutions dans l'utilisation pertinente des monades.

Effectivement, nous avons commencé par écrire une fonction qui récupère le contenu d'un dossier sous forme de liste, et qui map un filtre sur cette liste pour avoir une liste des fichier texte uniquement.

La fonction "getAllMapsOf" en question

```
getAllMapsOf :: FilePath -> IO[String]
getAllMapsOf [] = error "directory path missing"
getAllMapsOf path = do
    allFiles <- D.getDirectoryContents path
    let f = (\filename -> T.isSuffixOf (T.pack ".txt") filename)
    let allMapsFiles = filter  $ T.pack <$> allFiles
    return $ (T.unpack <$> allMapsFiles)
```

Une fois qu'on récupère cette liste, dans le contexte monadique IO du main grâce a "<-", nous avons certes une liste de string classique, mais maintenant nous voulons ouvrir chacun de ces fichiers en utilisant leurs noms.

Car nous voulons créer dynamiquement des cartes à partir de cette liste de noms de fichiers. Donc naïvement nous mappons la fonction "readFile" sur cette liste. Et la, le premier problème arrive, nous nous retrouvons avec une liste de type

```
[IO String]
```

Mais on a besoin d'une liste de String, car nous voulons utiliser la fonction read de GameMap pour lire une string et sortir une map. Nous avons donc découvert la fonction "séquence" du module standard de Haskell. Voici son type

```
Sequence :: Monad a => [a b] -> a [b]
```

Ici c'est gagné, car nous transformons notre liste de IO string, en une liste de type

```
IO [String]
```

Nous pouvons donc récupérer notre liste de String avec "<- ", car nous sommes dans le contexte monadique IO de la fonction main. Et transformer notre liste de string en liste de gamemap.

```
let surrgamemaps = GM.gameMapRead <$> surrmaps
```

Deuxième problème qui s'est posé a nous, maintenant nous voulons mapper la fonction "hspec" (sur une fonction de tests elle même mappé sur une liste de gamemap), le problème, c'est que le type de hspec est le suivant :

```
hspec :: Spec -> IO ()
```

Et impossible de mapper une fonction a effet de bord avec la fonction map classique.

Nous avons donc découvert une fois encore une nouvelle fonction : "mapM_" qui est un map version monadique, et donc très utile dans notre cas, car nous allons pouvoir mapper une fonction à effet de bord sur une liste.

Nous avons également utilisé les avantages des foncteurs applicatifs, car dans certaines fonctions de tests liés à la carte, le résultat attendu est envoyé en paramètre. Voici un exemple :

```
mapM_ hspec (GMS.mapSurrounedByWallSpec <$> surrgamemaps <*> [False])
```

Ici, nous mappons monadiquement hspec, sur le map de la fonction de tests qui verifie si la carte donné en parametre est bien entourée de murs (voir specifications). Et grace aux applicatives, nous pouvons donner en deuxieme argument le résultat attendu, qui est False car ici les cartes "surrgamemaps" utilisées sont volontairement non entourées de mur.

Une dernière chose que nous trouvons particulièrement intéressant dans notre code, se trouve ici du coté du module Main.

Nous voulions mapper une fonction d'affichage de sprite sur des Map Strict (afficher la carte, et l'environnement) de manière élégante, mais nous avions besoin de la clé de chaque élément de ces Map strict car c'est des coordonnées et cela nous est primordial pour savoir à quel endroit afficher les images.

Nous avons donc trouvé une librairie externe du nom du "Key", et plus particulièrement une fonction qui répondait directement a notre problème

```
mapWithKeyM_ :: (FoldableWithKey t, Monad m) => (Key t -> a -> m b) -> t a -> m ()
```

Ici, nous pouvons mapper une fonction a effet de bord sur une structure pliable **a clé**. C'est donc exactement ce qu'il nous fallait, nous pouvons donc afficher tout les éléments de notre carte en 2 lignes uniquement.

```
Key.mapWithKeyM_ cellToSprite map
Key.mapWithKeyM_ entiteToSprite env
```

Avec "cellToSprite" et "entiteToSprite" deux lambdas expressions d'une ligne chacune, qui attendent deux arguments (coordonnée et case/entité) et qui produit un effet de bord pour afficher l'image en question.

5.1.2 Difficultés rencontrés

Les principales difficultés rencontrées lors de cette première version du jeu ont été dans un premier temps liées à la modélisation par les types. Comprendre la logique du guide fournis, comment s'imbriquent les différentes parties, etc.

Et dans un second temps nous avons eu une longue réflexion sur la partie sûreté du code, les invariants, les pré et post condition, comment doit s'imbriquer tout ça dans notre code, doit ont les utiliser dans le code lui-même du jeu, ou uniquement dans les tests...

Les tests sont bien sur un bon moyen pour vérifier que les fonctions d'un programme font bien ce qu'on attend, mais ils ne sont jamais exhaustifs.

Vérifié que les différentes propriétés sont respectées tout au long de l'exécution d'un programme améliorerait peut-être la sûreté de celui-ci, nous y avons donc réfléchis, mais cela paraissait long à mettre en place, dans le sens où le code aurait sûrement doublé de volume, aurais été moins lisible..

Étant dans le doute, nous vous avons donc consulté l'équipe pédagogique qui nous a indiqué que pour ce projet, la vérification des propriétés pouvais être faite uniquement dans la partie test du projet, mais que le sujet restais à débattre, et était avant tout un choix de développement. Nous avons donc choisit de séparer correctement le jeu des tests.

5.2 Extensions

5.2.1 Combats

Cette extension permet au joueur d'attaquer les monstres autour de lui.

Le joueur peut lancer une attaque qui touchera les quatre cases adjacentes, nous avons fait ce choix, car il nous semblait plus naturel d'avoir une touche dédiée à l'attaque. Une autre touche aurait pu être ajouté pour donner une direction à l'attaque afin de ne toucher qu'une case à la fois, mais nous avons estimé que cela ne représentait pas un grand intérêt au niveau de la programmation de l'extension.

Cette extension ajoute beaucoup au gameplay, surtout couplée avec l'extension Ennemis Dangereux. Le joueur peut ainsi prendre part à de vrais combats où chaque coup portée ou reçu (extension Ennemis Dangereux) sera déterminant pour la suite de son aventure.

Deux fonctions permettent la réalisation de l'extension toute les deux se trouvant dans le Model :

— canAttack : Permet à partir d'un Model et d'une Entite donné, de récupérer la liste de toutes les Entite attaquables. Cette fonction regarde simplement en fonction de l'Entité qu'on lui passe, quelles Entite qui se trouvent autour d'elles peuvent être attaqués. Elle servira aussi d'extension pour Ennemis Dangereux. Ainsi, les Zombie ne peuvent attaquer que les Joueur et les Joueur ne peuvent attaquer que les Zombie.
Cette fonction possède une pré-condition : L'Entite donnée doit exister dans l'Environnement du Model donné.

— attack : l'Entite donnée réalise une attaque sur la liste des entités données, la fonction requiert également le Model associé. Cette fonction servira également pour l'extension Ennemis Dangereux. Les dégâts de l'attaque se font en fonction de la valeur d'attaque de l'Entite, disponible dans ses stats. Si l'Entite est un Joueur, on additionne l'attaque de base du Joueur à l'attaque bonus dont le Joueur dispose (grâce à ses équipements), s'il en a.

Deux pré-conditions sont nécessaire : L'Entite donnée doit existé dans l'Environnement du Model donné et la liste d'Entite attaquables doit correspondre au retour de canAttack pour l'Entite donnée.

Cette fonction n'appelle pas elle même canAttack car on réalise une attaque seulement si on peut attaquer des Entite, un autre choix aurait été de simplement attaquer toutes les Entite sur les cases adjacentes, peut importe qu'il y en ait ou non.

5.2.2 Ennemis Dangereux

Cette extension permet aux ennemis d'avoir un comportement plus ou moins agressif, de façon pré-déterminée. Ainsi, les Zombie se comportent de la manière suivante :

— Si un Joueur se trouve sur une case adjacente au Zombie, le Zombie l'attaque.

— Si un Joueur est présent à trois cases ou moins du Zombie alors celui-ci avance vers le Joueur.

— Sinon, le Zombie se déplace de manière aléatoire (20% de chances pour chaque direction où il peut aller, le reste des % étant dédiés à l'action "ne rien faire". Ainsi, si un Zombie se trouve sur une case où il peut se déplacer seulement au Nord et au Sud, et qu'aucun Joueur ne se trouve à moins de trois cases, il aura 20% de chances d'aller au Nord, 20% de chances d'aller au Sud, et 60% de chances de ne rien faire.

L'extension apporte bien sur beaucoup au gameplay, surtout couplée avec l'extension Combats, mais elle est aussi intéressante à mettre au point : il faut réfléchir à un algorithme pour que les Zombie puissent détecter les Joueur aux alentours.

Voici la liste des fonctions qui permettent la mise en œuvre de l'extension :

— canAttack et attack : déjà décrites dans l'extension Combats, ses fonctions sont réutilisées ici, en effet, il n'aurait pas été efficace de créer une fonction pour les Joueur et une pour les Ennemis, le pattern matching nous permettant de facilement tout rassembler.

— playerIsNear : cette fonction prend une Entite et un Model, si un Joueur est dans le rayon de détection de l'Entite, elle renverra un Just Joueur, sinon elle renverra Nothing.

L'algorithme appliqué est un simple algorithme récursif, contenu dans la fonction auxiliaire detect. Elle parcourt toutes les cases adjacentes qu'un Zombie peut parcourir : quand un Joueur est détecté, on renvoie la parcourue, qui est stockée dans un accumulateur. Ensuite, on trouve parmi tous les chemins possibles vers un Joueur lequel est le plus court et on renvoie la direction associé que le Zombie doit suivre.

Le problème majeur de cet algorithme est qu'il n'est pas très efficace, tout simplement, car nous n'avons pas eu le temps d'implanter une version dans laquelle on se souvient des cases déjà parcourut (pour éviter d'appeler l'algorithme sur une case qu'on aurait déjà parcouru), on évite néanmoins les appels infinis en arrêtant de rappeler l'algorithme si le chemin est plus long que la capacité de détection de l'Entite.

La fonction playerIsNear à une pré-condition associée : l'Entite donnée doit exister dans l'Environnement du Model donné.

5.2.3 Coffres au trésor

Cette extension apporte des coffres dans le jeu, avec une entité trésor contenu dans un coffre, que le joueur doit avoir dans son inventaire en sortant du labyrinthe s'il veut gagner, cette extension a donc été développé en parallèle de l'extension liée a l'inventaire et aux équipements.

Pour attraper un trésor, le joueur peut interagir avec les coffres de la même manière qu'il interagit avec des portes, c'est-à-dire ouvrir/fermer. Et si, quand il ouvre un coffre, un trésor était contenu dedans, le joueur gagne donc le trésor, et son prochain objectif seras d'atteindre la sortie pour gagner.

Voici la liste des fonctions importantes, accompagnées de leurs pré/post conditions, qui accompagnent cette extension

- **interactChest** dans GameMap.hs : prend une coordonnée, une gamemap, et ouvre le coffre a ces coordonnées

  interactChestPre : les coordonnées sont incluse dans la gamemap
- **interactWithChest** dans Model.hs : regarde aux cases adjacente du joueur, et interagis avec tout les coffres qui s'y trouvent, en enrichissant l'inventaire du joueur si les coffres contiennent des objets

  interactWithChestPre : le joueur existe et les coordonnées sont incluse dans la gamemap
- La condition de victoire dans Engine.hs a aussi été modifiée, maintenant, le joueur doit avoir le trésor dans son inventaire, et etre sur la case de sortie

Cette extension a aussi impliquée un enrichissement de certains types.

- Le type "Cell" dans GameMap.hs, a été enrichis du type Chest, avec "CloseableState" un type qui représente tout ce qui peut être ouvert ou fermé, et Maybe Entite pour modéliser le fait qu'un coffre peut être vide, ou remplis par une entité (Un trésor uniquement pour l'instant)

  ```
  Chest CloseableState (Maybe Entite)
  ```

  Evidemment, les instances de Show et Read pour Cell ont été mis a jour
- Le type "Entite" dans Entite.hs a été enrichis par un nouveau type "Treasure"
- Le type Ordre dans Model.hs se voit rajouter un constructeur "IC" qui représente une action d'intéraction avec un coffre

  ```
  data Ordre = N | S | E | O | R | A | ID IDoor | IC
  ```

Nous avons donc décider de faire des coffres, des elements statiquement placé sur la carte avec le fichier texte, cependant, initialement, tout les coffres sont vide, et dynamiquement, a l'initialisation du jeu, nous remplissons les coffres aleatoirement, pour donner un certain interet pour le joueur meme si c'est lui qui a crée sa propre carte.

Avec de nouveaux types, des invariants ont aussi été enrichis

- L'invariant de GameMap.hs est enrichis d'une nouvelle propriété, il doit y avoir au minimum 1 coffre sur la carte, car c'est nécessaire pour avoir la possibilité de gagner.
- L'invariant de Model est enrichis d'une nouvelle propriété, on vérifie qu'il y'a, exactement **un seul** trésor contenu dans un coffre

5.2.4 Équipements et inventaire

Comme dit plus haut, cette extension a été implémentée en parallèle des coffres et du trésor, car le trésor, c'est un objet qu'on doit "ramasser" pour sortir avec et gagner, donc cela impliquais un inventaire. En plus de ça, cette extension rajoute le concept d'équipement.

Effectivement, maintenant, on remplit les coffres avec un trésor, mais aussi des équipements, ici une épée, qui embarque un bonus d'attaque. Si le joueur trouve cette épée dans un coffre, il devient armé, et a donc ce bonus d'attaque de manière permanente. Ici on remarque donc que les équipements ont impliqué le rajout de statistique au joueur.

La quasi-totalité des changements se sont fait sur les types dans **Entite.hs**, avec une seule fonction principale

- **expandInventory** dans Entite.hs : étend l'inventaire du joueur si on lui donne un objet en parametre.

Le type Entite a été enrichis avec l'ajout de

```haskell
type BonusAttack = Int

data PlayerState =
Unarmed
| Armed BonusAttack deriving Eq

data Stats = Stats {vie :: Int, atk :: Int} deriving Eq

--Dans le type Entite
Sword BonusAttack
```

Intuitivement, une arme a un bonus d'attaque, qui est une valeur entière, on a un état du joueur car initialement, il est non armé, et il peut trouver une arme dans un coffre, une fois armée, il gagne le bonus, qui est directement utilisé pendant les combats (si armée : atk + BonusAttack, sinon atk).

Cette extension a aussi impliquée un enrichissement d'invariants

— L'invariant de Entite.hs :

si le joueur est en état unarmed, l'épée ne doit pas être contenue dans l'inventaire, si il est en état armed, l'épée doit être contenu dans l'inventaire

l'inventaire ne peut contenir que des entités immobiles (pas de Zombie ou de Joueur..)

5.2.5 Pièges/Téléportations avec leviers

Et enfin, l'extension qui rajoute des cases particulières connecté a un levier sur la carte, levier qui peut être activé ou non, et qui désactiveras ou non, la ou les cases auxquelles il est connecté.

C'est un ajout intéressant en terme de gameplay car cela nous permet de faire du game design dans la construction de nos cartes et rendre le jeu amusant et intéressant.

Nous allons d'abord préciser tous les ajouts avant d'expliquer le système a proprement parlé. Nous voulons modéliser des cases qui représentent des pièges, et des cases représentant des téléporteurs.

Premièrement, voyons les types qui ont été rajouté et/ou modifiés :

— dans GameMap.hs

```
data ActionableState = Activated | Desactivated deriving Eq
type LinkedTo = Either (Coord,Coord) Coord
type OtherSide = Maybe Coord

--Dans le type Cell
..
| Portal ActionableState OtherSide
| Trap ActionableState
| Sink ActionableState (Maybe LinkedTo)
..
```

Ici, plusieurs choses a expliquer, "ActionableState" est le type qui modélise tout objet qui peut être activé ou désactivé.

Donc ici, les leviers, les cases pièges, les cases de téléportation. Ensuite, le type "LinkedTo" modélise le fait qu'un levier peut être soit : connecté à un piège, soit a un couple de portails de téléportation, représentés par leurs coordonnées respective.

Le type "OtherSide" représente l'autre extrémité d'un portail de téléportation. Donc au final, un portail, c'est un constructeur qui prend un état actionnable, et une extrémité, un piège est un constructeur avec un état uniquement, activé (et occasionne des dommages) ou désactivé.

Finalement, un levier, a un état activé ou désactivé, et est soit connecté a rien, soit connecté a un type "LinkedTo", modélisé par un type Maybe.

Le système a proprement parlé fonctionne comme ça : les leviers et les cases spéciales sont des cases statique qu'on met sur le fichier texte de la carte, donc initialement, les leviers sont

connecté a rien (d'ou le Maybe dans son type), et les cases de téléportations ne sont liées à aucune extrémité.

Ensuite, à l'initialisation du jeu, ont appel les fonctions "linkSinkToPortals" et "linkSinkToTraps" qui connecte aléatoirement des couples de portails et des pièges a des leviers.

Voici un extrait de la fonction "linkSinkToPortals" :

```
...
linkSinkToPortals_aux pts sks pick m = do
  portal1 <- pick pts
  let restPortals = delete portal1 pts
  portal2 <- pick restPortals
  let restPortals' = delete portal2 r
  sink <- pick sks
  let restSinks = delete sink sks
  -- on active les portails en les liant l'un avec l'autre
  -- et a la fin on lie un levier a ce couple de portails
  let linkp1Top2 = M.adjust (\_ -> Portal Activated (Just portal2)) portal1 m
  let linkp2Top1 = M.adjust (\_ -> Portal Activated (Just portal1)) portal2 linkp1Top2
  let f = (\_ -> Sink Desactivated (Just (Left (portal1,portal2))))
  let sinkLink = M.adjust f sink linkp2Top1
  linkSinkToPortals_aux restPortals' restSinks pick sinkLink
```

ici on voit l'extrait de la fonction, qui illustre le calcul principal, qui prend au hasard deux portails, un levier, connecte les deux portails entre eux, et le levier aux deux portails, puis rappel la fonction avec ce qui reste de leviers, de portails, la fonction aleatoire, et une liste avec ce qui a déja été lié.

5.2.6 Points forts

On peut voir dans le code, qu'après avoir implémenté un système gérant les portes, et les actions du joueur de manière relativement modulaire, il a été plutôt naturel de développer les coffres et les leviers, car nous avons simplement réécrit le même pattern de fonction, "interact", "interactWith", enrichir la fonction "canInteact" et connecter la touche du clavier avec la détection d'action dans la fonction tour.

5.2.7 Difficultés rencontrés

Les principales difficultés rencontrées lors de l'implémentation des extensions ont été de bien réfléchir a comment enrichir et modifier la structure des types, quand c'était nécessaire, sans pour autant briser ce qui existait déjà, pour ne pas avoir a recorder entièrement des parties.

Cela dit, cette difficulté nous a aussi permis de réarranger des petits bouts de code qui n'étaient pas satisfaisant, pas assez modulaire. Au final, nous sommes satisfaits de nos extensions, et de leurs insertions dans le code existant.

6 Conclusion

À travers le développement de notre Dungeon Crawler en Haskell, nous avons pu mettre en pratique les principes de modélisation et de programmation dirigée par les types, comme vu en cours, ainsi que des concepts propre a Haskell et de manière générale a la programmation fonctionnelle.

Ce projet et cette UE nous ont sensibilisé sur la programmation sûre, effectivement, en plus des bonnes pratiques de programmation, de modélisation, et de la puissance d'un système de type riche et d'un compilateur exigeant, nous nous sommes efforcé d'écrire une batterie de tests conséquente. Tout en ayant posé une réflexion, tout au long du projet, sur l'importance des invariants et pré/post conditions, et de comment les utiliser.

L'un principes fondamentale de la programmation qui a été abordé tout au long du semestre : la composition, a eu également une place très importante dans notre projet, nous avons essayé au mieux de construire notre architecture en pensant a la composition.

Ce qui a rendu l'implémentation de certaines extensions, naturelle. Nous avons aussi conscience des faiblesses de notre code, (nous savons que certaines fonctions auraient pu être simplifiées et/ou factorisées (si on écrit ça, faut donner des exemple)).

Pour finir, le contexte du projet a rendu sa réalisation agréable et ludique, nous avons pu en apprendre d'avantage sur les problématiques et les stratégies d'implantation d'un jeu vidéo simple.