Rapport de projet PAF

-

Belkacem Gueliane
Jae-Soo LEE

Manuel d'utilisation

Afin de lancer notre version du jeu, lancer la commande *stack run* à partir du dossier projetPaf. La fenêtre de jeu apparaît et il est par la suite possible de cliquer sur les lemmings avec la souris et leur attribuer un rôle via les touches du clavier:
Q -> creuseur
S -> Stoppeur
D -> Poseur

Il est possible de lancer les tests en lançant la commande *stack test* à partir du même dossier.

Les propositions

Les propositions implémentées sont comme suit:

-Pour le type Coord:
prop_bougeCoordGaucheDroite vérifie que bouger une coordonnée vers la gauche puis la rebouger vers la droite revient à ne pas bouger.

-Pour la classe Placable:
prop_bougePlacableGauche vérifie qu'appliquer bougeP vers la gauche sur une instance de Placable est équivalent à lui appliquer deplaceP vers la case à sa gauche.

-Pour le type Lemming
lemming_inv vérifie qu'un Lemming est conforme, ie il se situe bien à l'intérieur du niveau qui lui est associé et les deux cases qu'il occupe sont soit vides, soit des cases L (dans le cas du Lemming stoppeur)

-Pour le type Niveau

prop_NiveauCorrect vérifie que le Niveau est conforme. (Nous aurions dû l'appeler niveau_inv vu qu'il s'agit d'un invariant de type). Cet invariant vérifie plusieurs propriétés:

prop_NiveauUniqueES vérifie qu'il existe exactement une entrée et une sortie dans le niveau.

prop_FermetureMetal vérifie que toutes les cases en bordure du niveau sont métalliques

prop_entreeSurVide vérifie que l'entrée du niveau se situe au dessus d'une case vide.

prop_sortieSurMetal vérifie que la sortie du niveau se situe au dessus d'une case métallique.

prop_associated vérifie que tout l'espace du Niveau est associé à une case et, inversement que toutes les cases du Niveau sont associé à une coordonnée à l'intérieur du Niveau.

-Pour le type GameState

gameState_inv vérifie que le GameState est conforme, ie que le Niveau du GameState est conforme (respecte son invariant de type) et tous les Lemmings sont eux aussi conformes.

post_ajout_lemming vérifie qu'après l'ajout d'un lemming à un GameState, nous avons bien un lemming d'identifiant (i - 1) situé sur la case en dessous de l'entrée.

pre_manip_lemming est une précondition sur toutes les opérations manipulant les lemmings d'un GameState. Elle vérifie que le lemming d'identifiant i existe bien dans le GameState.

post_suppr_lemming vérifie qu'après avoir enlevé le lemming d'identifiant i du GameState, il n'existe pas de lemming d'identifiant i dans le GameState.

post_déplacer_lemming vérifie qu'une fois le lemming d'identifiant i déplacé à une coordonnée c, ce lemming se trouve bien à l'emplacement voulu.

post_modifier_lemming vérifie qu'après avoir modifié le rôle d'un lemming i, ce lemming a bien pris son nouveau rôle et les deux cases qu'il occupe sont soit vides, soit L si son nouveau rôle est le stoppeur.

post_moteurJeu vérifie que l'invariant de type de GameState est bien respecté par le GameState suivant un tour de jeu.

Les Tests

Dans le Hspec de notre projet, nous avons implémenté les tests vérifiant les fonctions de Coordinates:

coordSpec0 et coordSpec1 vérifient *prop_bougeCoordGaucheDroite* et *prop_bougePlacableGauche* sur des Coordonnées/ Lemmings placés à ces coordonnées générés aléatoirement.

niveauSpec vérifie que les niveaux du jeu respectent l'invariant de type de Niveau.

deroulementTest simule les 500 premiers tours de jeu pour le niveau 1 (sans interventions du joueur) et vérifie que le GameState vérifie son invariant de type à chaque fin de tour.

supprTest simule le 500e tour de jeu d'un mini niveau et vérifie *post_suppr_lemming* lorsqu'on enlève un lemming du GameState.

Rapport

Pour la réalisation de ce projet nous nous sommes basés sur la structure proposée en partiels. Nous avons donc les types Coord, Niveau, Case, Lemming ainsi que la classe Placable.

Niveau

Pour le type Niveau, nous avions donc dans un premier temps suivi la structure

```
data Niveau = Niveau { hNiveau :: Int,
                       lNiveau :: Int,
                       casesNiveau :: Map Coord Case}
                       deriving Eq
```

Dans un effort de repenser le type afin qu'il implémente Monad, nous avons tenté l'implémentation suivante:

```
data Niveau = Niveau Int Int a
```

puis par la suite

```
instance  Monad Niveau where
(>>=) (Niveau x y c) f = f c
```

mais nous avons été confrontés à l'implémentation du return qui ne permettait pas dans la structure choisie d'imposer la hauteur et largeur d'un niveau en sortie.
Le type Grille qui par la suite a remplacé le type du champ casesNiveau est un monument aux divers essais infructueux afin de forcer cette implémentation et nous n'avons pas osé restaurer l'implémentation initiale de peur de casser le code (et de ne pas avoir le temps de le corriger proprement au vu de la contrainte de temps; ce qui a été un choix difficile car notre première implémentation était plus propre…)
L'implémentation de ce type suit par ailleurs d'assez près celle proposée par le sujet du partiel.

Lemming

Nous avons implémenté ce type sous forme d'enregistrement.

```
data Lemming = Lemming { position :: Coord, role :: Role, niveau :: Niveau, direction
:: Deplacement, chute :: Int}
  deriving Eq
```

Un lemming est doté d'un positionnement, d'un rôle, d'un niveau d'une direction et d'un compteur.
Les rôles:
-Marcheur
-Tombeur
-Creuseur
-Stoppeur
-Fini
-Mort

Chaque lemming agit suivant une fonction tour propre à son rôle, à l'exception du Stoppeur qui n'agit plus à partir du moment qu'il prend ce rôle.

GameState

```
data GameState = GameState{niveauS :: Niveau, ensLemmings :: Map Int Lemming,
lemmingId :: Int}
data Fin = Victoire | Defaite
```

Notre implémentation de l'état du jeu est un enregistrement contenant le niveau du jeu, l'ensemble des lemmings présent dans le niveau ainsi que l'identifiant du prochain lemming à venir (qui fait aussi office de compteur).

Nous avons implémenté les fonctions pour modifier/enlever/ajouter des lemmings au jeu, et défini le type Fin:

data Fin = Victoire | Defaite

afin de pouvoir détecter les conditions de fin (en fonction du nombre de lemmings à sauver que nous visons).

Afin d'optimiser le temps de détection des events (clicks de souris, touche clavier), nous avons décidé d'incrémenter l'état du jeu tous les 10 tours.
Effectivement, si nous incrémentons l'état du jeu à chaque tour, le jeu devient trop rapide et injouable. Si nou augmentons les delays de chaque frame, le temps de détection des événements devient trop court et il est alors quasiment impossible d'interagir avec le jeu.

```
moteurJeu :: GameState -> Int -> GameState
moteurJeu gs i
  | rem i 50 == 0 && (lemmingId gs) < 10  = let (niv, hs) = Map.foldlWithKey
stateChanger ((niveauS gs), Map.empty) (ensLemmings gs) in
    ajouterLemming (GameState niv hs (lemmingId gs))
  | rem i 10 == 0 = let (niv, hs) = Map.foldlWithKey stateChanger ((niveauS gs),
Map.empty) (ensLemmings gs) in
    GameState niv hs (lemmingId gs)
  | otherwise = gs
```

Dans la boucle de jeu telle que nous l'avons défini, un nouveau lemming est injecté dans la partie tous les 50 tours (jusqu'à un total de 10 lemmings injectés) et comme dit précédemment l'état du jeu est incrémenté tous les 10 tours.

Pour chaque incrémentation nous effectuons un fold sur l'ensemble des lemmings et lui appliquons tourLemming sur le Niveau du jeu (qui est alors modifié selon l'action du lemming).

Le Main

A chaque boucle la gameLoop du main affiche tous les éléments du GameState ainsi que l'image de fond, modifie l'état du jeu en cas d'intéraction de la part du joueur, vérifie que les conditions de fin ne sont pas satisfaites puis lance le prochain tour de jeu.

La difficulté que nous avons rencontré est, comme dit auparavant, l'implémentation efficace de la détection de souris/clavier.

En ce qui concerne le choix des images, n'ayant pas trouvé d'ensemble d'image d'un personnage spécifique, nous avons décidé d'associer une image d'un personnage différent pour chaque rôle de lemming.

Conclusion

Ce projet a été très intéressant car il nous a permis d'appliquer le langage Haskell dans un contexte concret.
Nous sommes particulièrement satisfait d'avoir atteint une meilleure compréhension de ce langage mais déplorons malheureusement du manque de temps afin d'implémenter une version plus élégante de ce jeu (Le type Lemming aurait été de notre avis mieux implémenté en type somme de produits, et le temps nous a manqué en ce qui concerne la restructuration de ce projet afin d'y implémenter les monads).
Bien que nous ayons suivi les UE d'OCAML les années/semestres précédents, nous pensons que l'apprentissage d'haskell nous a montré une approche bien plus poussée de la programmation fonctionnelle.

Nous pensons continuer l'implémentation de deux types de monstres ainsi qu'un lemming guerrier dans les jours à venir. (hors évaluation bien sûr)
Je vous invite, si vous le souhaitez, à venir voir l'implémentation.