Examen Réparti 1 - PAF 2023: *Dune II*

PAF 2022 - Master STL S2

14 Mars 2023

Préliminaires

Aucun barême n'est donné, les copies seront évaluées sur les points suivants:

- qualité de la programmation fonctionnelle,
- maitrises des spécificités de Haskell vues en cours et en TD,
- pertinence de la modélisation : respect des principes vus au TD3, écriture systématique de propriétés (invariants, post-conditions), preuves et tests.

Le langage du partiel est le *Haskell* utilisé en cours et en TD. Les petites erreurs de syntaxe ne sont pas pénalisées.

Attention : La partie 1 est guidée, les parties suivantes ne le sont pas et évaluent les compétences de modélisation travaillées en cours, en TD et en TME. Les spécifications de ces parties sont **volontairement incomplètes**.

Description du jeu

Le but de cet examen est de spécifier un jeu vidéo de type *stratégie en temps réel* (RTS) sur une grille, similaire à *Dune II* (Westwood Studios, 1992).

Pendant la partie, chaque joueur est responsable d'*unités* (entités mobiles) et de *bâtiments* (entités immobiles) évoluant sur une carte (un terrain, vu du dessus). L'interface permet au joueur de donner des **ordres indépendants** à ses unités (déplacement, collecte, combat) et à ses bâtiments (production, construction). La création d'unités et de batiments supplémentaires nécessite de dépenser une ressource, qui se trouve présente sur le terrain et peut être collectée avec des unités spéciales. Les unités combattantes peuvent attaquer les unités et bâtiments des autres joueurs. Chaque unité ou bâtiment dispose d'un total de points de vie, réduit par les attaques adverses. Quand ce total arrive à 0, l'unité ou le bâtiment est détruit. Un joueur perd la partie quand son bâtiment principal est détruit. Un joueur gagne la partie quand tous les autres ont perdu.

```
-- Map vide
empty :: Map k a
-- cherche la valeur associee a une clef
lookup :: Ord k => k -> Map k a -> Maybe a
-- fold sur les valeurs du Map
foldr :: (a -> b -> b) -> b -> Map k a -> b
-- fold sur les clefs et les valeurs du Map
foldrWithKey :: (k -> a -> b -> b) -> b -> Map k a -> b
-- ajoute/remplace une association clef/valeur
insert :: Ord k => k -> a -> Map k a -> Map k a
```

Figure 1: Fonctions des Map

Partie 1 : Carte

Le jeu se passe sur une carte en deux dimensions, *vue du dessus*, décrivant le terrain. Chaque case de la carte possède des coordonnées et une *nature*. On utilisera le type suivant pour les coordonnées.

```
data Coord = C {cx :: Int,
                cy :: Int}
    deriving (Show, Eq)
```

Dans un premier temps la nature des cases de la carte pourra être : "herbe" (le terrain de base, constructible et passable par toutes les unités), "eau" (terrain "bloquant" inconstructible et impassable) et "ressource" représentant un gisement de ressources pouvant être collectées par les joueurs :

```
data Terrain =    Herbe
               | Ressource Int
               | Eau
```

Ainsi, une case dont la nature est `Ressource 42` représente un gisement de ressources contenant encore 42 unités de ressource.

Question 1.1 On décide qu'un gisement de ressource contient un nombre **strictement positif** de ressource. En déduire un invariant pour le type `Terrain`.

La carte sur laquelle le jeu se déroule est représentée par un `Map` dont les clefs sont des coordonnées, et les valeurs sont des éléments de `Terrain`.

```
import qualified Data.Map.Strict as M

newtype Carte = Carte {carte :: M.Map Coord Terrain}
```

Question 1.2 La Figure 1 rappelle les signatures de quelques fonctions usuelles sur les `Map`, et montre que l'utilisation d'un `Map k a` nécessite que le type `k` des clefs instantie la classe `Ord`. Donner une instantiation permettant au type `Carte` d'être correctement défini.

Question 1.3 Pour qu'une carte soit correcte, on demande d'une part, à ce que toutes ses cases correspondent à un terrain correct ; d'autre part, on laisse la forme de la carte plutôt libre (pas forcément rectangulaire), mais pour simplifier la gestion des itinéraires, on impose la contrainte de convexité suivante : si deux cases de coordonnées (x_1, y_1) et (x_2, y_2) font partie de la carte, alors :

- soit toutes les cases des segments $[(x_1, y_1), (x_1, y_2)]$ et $[(x_1, y_2), (x_2, y_2)]$ font partie de la carte,
- soit toutes les cases des segments $[(x_1, y_1), (x_2, y_1)]$ et $[(x_2, y_1), (x_2, y_2)]$ font partie de la carte,

(cela force l'existence d'un "chemin en coude" entre tout couple de points de la carte)

Donner un invariant pour le type `Carte`.

Question 1.4 Plus tard, des unités collectrices seront capables d'extraire des ressources de la carte. Ecrire une fonction `collecteCase :: Coord -> Int -> Carte -> (Int, Carte)` qui prend des coordonnées (x, y), une valeur r et une carte c et qui essaie d'extraire r ressources de la carte, c'est-à-dire qui renvoie un couple (v, nc) où v est la quantité de ressources effectivement extraite, et nc représente la carte c après extraction. Le comportement attendu est donné par :

- si les coordonnées (x, y) correspondent à une case de ressource contenant strictement plus de r ressources, l'extraction réussit et r ressources sont effectivement extraites et la nouvelle carte correspond à l'ancienne dans laquelle on a enlevé r ressources en (x, y) (une partie du gisement a été collectée).
- si les coordonnées (x, y) correspondent à une case de ressource contenant r ressources ou moins, l'extraction réussit et la totalité des ressources de la case sont effectivement extraites et la nouvelle carte correspond à l'ancienne dans laquelle on trouve un terrain d'`Herbe` en (x, y) (le gisement a disparu).
- si les coordonnées (x, y) ne correspondent pas à une case de ressource, l'extraction échoue, on récupère 0 ressource et la carte n'est pas modifiée.

Question 1.5 Donner une précondition pour `collecteCase` qui assure que l'extraction réussit.

Question 1.6 Donner une postcondition pour `collecteCase` qui assure sa correction.

Question 1.7 Donner un jeu de trois tests, dans le formalisme `HSpec` pour la fonction `collecteCase`.

Partie 2 : Environnement

L'environnement du jeu est constitué d'une carte, d'un ensemble d'unités, d'un ensemble de batiments et d'une liste de joueurs. On ne donne pas la définition des types correspondants à ces entités dans cette partie, on se servira uniquement des observateurs dont les types sont définis ci-dessous :

```
newtype JoueurId = JoueurId Int
data Joueur = ...
jid :: Joueur -> JoueurId

newtype BatId = BatId Int
data Batiment = ...
bcoord :: Batiment -> Coord
bproprio :: Batiment -> JoueurId

newtype UniteId = UniteId Int
data Unite = ...
ucoord :: Unite -> Coord
uproprio :: Unite -> JoueurId

data Environnement = {
                        joueurs :: [Joueur],
                        ecarte :: Carte,
                        unites :: Map UniteId Unite,
                        batiments :: Map BatId Batiment
                     }
```

Question 2 Un environnement est correct quand :

- les entités le composant sont toutes correctes,
- chaque bâtiment et chaque unité appartient à un joueur de l'environnement,
- chaque case `Eau` de la carte est vide (pas de bâtiment ni d'unité),
- chaque case `Herbe` de la carte contient au plus un bâtiment ou une unité (mais pas les deux),

- chaque case `Ressource` de la carte contient au plus une unité (mais pas de bâtiment).

Ecrire un invariant (ou plusieurs parties d'un invariant) pour l'environnement.

Partie 3 : Bâtiments

Les bâtiments disponibles sont :

- le *quartier général* : un bâtiment unique pour chaque joueur présent à la création de la partie, sa destruction entraîne l'échec de son propriétaire.
- la *raffinerie* : un bâtiment permettant aux collecteurs de rapporter des ressources.
- l'*usine* : un bâtiment permettant de produire des unités.
- la *centrale* : un bâtiment produisant de l'énergie.

Chaque joueur dispose d'un capital de *crédit*. Il en reçoit une somme fixe au début de la partie, et à chaque fois qu'un collecteur (cf. partie suivante) rapporte des ressources à une raffinerie, ces dernières sont converties en crédits. Les crédits sont dépensés dans la construction de bâtiments et la production d'unités.

Pour rendre le jeu plus intéressant, une notion d'énergie est introduite : le QG et les centrales produisent une valeur fixe d'énergie, les usines et les raffineries consomment une valeur fixe d'énergie. A chaque début de tour, si les bâtiments d'un joueur consomment plus d'énergie qu'ils n'en produisent (globalement), alors toutes les raffineries et les usines de ce joueur sont fermées (elles sont inutilisables tant que la production d'énergie n'est pas suffisante).

Chaque bâtiment de la carte possède un nombre de "points de vie". Quand ils atteignent zéro (suite à des attaques d'unités adverses), le bâtiment est détruit (et le terrain qu'il occupait est à nouveau disponible).

Une usine en production doit "se souvenir" de l'unité qu'elle produit, et du temps restant (en tours de jeu) à sa production.

Un bâtiment peut être construit sur une case donnée si le joueur qui veut le construire dispose de suffisamment de crédit et si la case est constructible (une case d'herbe, vide).

Question 3.1 Ecrire un constructeur intelligent pour `Environnement` qui prend en entrée une carte, une liste de coordonnées, et qui crée une partie (un environnement) avec des QGs pour des joueurs différents pour tous les coordonnées de la liste.

Question 3.2 Modéliser les bâtiments et les opérations sur les bâtiments en respectant les principes du TD3 (notamment en fournissant invariants, préconditions et postconditions).

Partie 4 : Unités

Les unités disponibles sont :

- le *collecteur* : il extrait des ressources des gisements présents sur la carte et les stocke dans une cuve interne (de capacité fixée) et les rapporte à une raffinerie, où ils sont convertis en crédits.
- le *combattant* : lorsqu'il est à proximité d'une entité (unité, batiment) ennemie, il l'attaque (lui enlève des points de vie à chaque tour)

Chaque unité possède un nombre de "points de vie". Quand ils atteignent zéro (suite à des attaques d'unités adverses), l'unité est détruite.

Les unités peuvent se déplacer selon les quatres directions cardinales. Elles ne peuvent pas se déplacer dans une case non-vide, non-existante, ou non-passable.

L'"intelligence artifielle" des unités est gérée ainsi :

- chaque unité possède une liste *d'ordres* donnés par le joueur (comme collecter des ressources dans une zone, se déplacer vers un point, patrouiller entre deux points, ...)
- chaque unité possède une *but* qui représente l'action qu'elle essaye d'entreprendre (se déplacer vers un point, extraire des ressources, attaquer une unité ennemie à portée, ...)

- une fonction `etape` gère le comportement d'une unité pendant un tour de jeu, en fonction de son but, de ses ordres et son état, par exemple (non exhaustif et non contractuel) :
  - une unité dont le but est de se déplacer vers un point calcule si elle est plus loin sur l'axe Est-Ouest ou sur l'axe Nord-Sud de ce point et se déplace d'une case dans la direction dans laquelle elle est la plus loin,
  - un collecteur non-plein qui a un ordre de collecte et qui commence son tour sur une case de ressource change son but pour extraire la ressource et se met à collecter de la ressource à chaque tour, jusqu'à ce qu'il soit plein ou que la case sur laquelle il se trouve se change en herbe,
  - un collecteur plein qui a un ordre de collecte, change son but pour se déplacer vers la raffinerie indiquée dans son ordre de collecte,
  - un combattant qui a un ordre de patrouille en tête de liste, et qui commence son tour à portée d'une entité ennemie se met à attaquer cette unité (changeant son but), en réduisant ses points de vie,
  - un combattant qui a un ordre de patrouille en tête de liste, un but d'attaque, mais qui n'a plus d'unité ennemie à portée, change son but pour de déplacer vers le premier point de son ordre de patrouille,
  - un combattant qui a un ordre de patrouille en tête de liste, un but de déplacement, et qui atteint le premier point de son ordre de patrouille, inverse les deux points de son ordre de patrouille et change son but pour se déplacer vers l'autre point,
  - ...

Question 4 Modéliser les unités et écrire des équations pour la fonction `etape` spécifiant le comportement à chaque tour des unités.

Partie 5 : Jeu

Il reste à définir des règles du jeu, en interprétant les commandes des joueurs (gérées dans le modèle à un niveau abstrait) pour donner des ordres aux unités, construire des bâtiments, produire des unités.

Il faut en outre gérer un tour de jeu en appelant la fonction `etape` sur toutes les unités du jeu.