Examen Réparti 1 - PAF 2024: *SimCity*

PAF 2024 - Master STL S2

05 Mars 2024

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.

Consigne : La partie 0 et 1 sont guidée, les parties 2a, 2b, 2c 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 *city builder* sur une grille, similaire à *SimCity* (Maxis, 1989).

Le jeu se déroule sur une carte vue du dessus, le haut de l'écran représente le Nord. Le joueur place sur la carte des routes, des batiments administratifs et des zones (résidentielles, commerciales, industrielles) afin de construire et faire évoluer une *ville*.

Une *simulation* gère en temps réel la vie des citoyens de la ville *individuellement*, qui dorment, travaillent et font leurs achats dans différents batiments au sein des zones de la ville.

Les citoyens se déplacent en temps réel dans la ville, ce qui génère du traffic routier, qui peut conduire à des embouteillages et force le joueur à planifier à l'avance l'urbanisation de sa ville.

Des systèmes de contraintes et d'évolution rendent le jeu ludique, par exemple, une économie (les citoyens recoivent un salaire, le dépense en loyer, nourriture et impôts, qui sont utilisés par le joueur pour construire de nouvelles infrastructures), des contraintes de services (des réseaux d'électricité et d'eau courante doivent être mis en place, des policiers patrouillent dans la ville), une notion de qualité des zones (la "valeur" d'une zone peut évoluer en fonction de différents facteurs locaux, ce qui a un impact sur les loyers et les salaires de cette zone)

Partie 0 : Formes

Le jeu se passe sur une carte en deux dimensions, *vue du dessus*, décrivant le terrain considéré comme une grille composée de *cases*. Chaque case de la carte possède deux coordonnées (abscisse et ordonnée), données par le type suivant.

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

Ainsi, la case de coordonnées `C -3 2` est située juste au Nord de la case de coordonnées `C -3 1` et juste à l'Est de la carte de coordonnées `C -4 2`.

Des **zones** (artificielles, comme une route ou un zone résidentielle, ou naturelles comme une rivière ou une montagne) couvrent certaines cases de la carte. Chaque zone possède une **forme**, qui représente un ensemble de cases. Dans un premier temps, on donne le type suivant pour `Forme` :

```
data Forme =      HSegment Coord Int
               | VSegment Coord Int
               | Rectangle Coord Int Int
```

Ainsi, une forme est soit :

- un *segment horizontal* (une suite de case contigües sur l'axe Ouest-Est), représenté par les coordonnées de son point le plus à l'Ouest et sa longueur,
- un *segment vertical* (une suite de case contigües sur l'axe Nord-Sud), représenté par les coordonnées de son point le plus au Nord et sa longueur,
- un *rectangle*, représenté par les coordonnées de son point le plus au Nord-Ouest, sa largeur (son étendue sur l'axe Ouest-Est) et sa hauteur (son étendue sur l'axe Nord-Sud)

(On aurait pu considérer les segments comme des rectangles de largeur ou hauteur 1, mais on garde cette distinction pour pouvoir traiter séparément les formes longilignes - routes, lignes haute-tension, ...).

Question 0.1 Ecrire une fonction `limites` qui prend en entrée une forme et renvoie quatre entiers représentant, dans l'ordre, l'ordonnée du point le plus au Nord de la forme, l'ordonnée du point le plus au Sud, l'abscisse du point le plus à l'Ouest et l'abscisse du point le plus à l'Est.

Question 0.2 Ecrire une fonction `appartient` qui prend en entrée des coordonnées, une forme, et décide si la case aux coordonnées donnés appartient à la forme.

Consigne : On pourra dans la suite utiliser une fonction `adjacent :: Coord -> Forme -> Bool` qui prend en entrée des coordonnées et une forme, et décide si la case aux coordonnées donnés est contigüe à (c'est-à-dire juste à côté de) la forme (on ne demande pas son code dans cette épreuve).

Consigne : Dans les questions suivantes, on ne fera plus de supposition sur les constructeurs de `Forme` (par exemple, on n'écrira plus `HSegment`), on utilisera uniquement les trois fonctions précédentes. (Ainsi, si on ajoute de nouveaux constructeurs à `Forme`, seules les trois fonctions précédentes devront être mises à jour).

Question 0.3 En utilisant `limites`, écrire une fonction (non-triviale) `collision_approx` qui prend en entrée deux formes et décide de manière approximative si les deux formes entrent en collision (c'est-à-dire, si une case appartient aux deux formes). Si les deux formes ont une case en commun, la fonction doit renvoyer `True`, mais on accepte qu'elle renvoie `True` si les deux formes n'ont pas de case en commun (on parle de *faux positif*).

Consigne : On pourra dans la suite utiliser une fonction `collision :: Forme -> Forme -> Bool` qui décide de manière exacte si deux formes entrent en collision, et une fonction `adjacentes :: Forme -> Forme -> Bool` qui prend en entrée deux formes et décide si les deux formes sont adjacentes (elle ne sont pas en collision, mais au moins une case de la première forme est adjacente à la deuxième).

Partie 1 : Zones

Une *zone* est un ensemble de cases de la carte qui sont dédiées à une construction particulières.

```
data Zone =     Eau Forme
              | Route Forme
              | ZR Forme [Batiment]
              | ZI Forme [Batiment]
              | ZC Forme [Batiment]
              | Admin Forme Batiment
```

Dans cette partie, on ne s'intéresse pas à la définition des types `Citoyen` et `Batiment`.
Ainsi une zone est soit :

- une étendue d'eau,
- une route,
- une zone résidentielle/commerciale/industrielle contenant zéro, un ou plusieurs bâtiments.
- une zone administrative contenant un unique bâtiment spécial.

Question 1.1 Ecrire une fonction `zoneForme` qui prend en entrée une zone et renvoie sa forme.

Pour pouvoir utiliser des structures de données, on définit des types représentant des identifiants (étiquettes) pour les zones, les batiments et les citoyens.

```
newtype ZonId = ZonId Int
newtype BatId = BatId Int
newtype CitId = CitId String
```

On définit un état du jeu, contenant l'intégralité des informations du jeu à un moment donné, par le type suivant :

```
data Ville = {
               viZones :: Map ZoneId Zone,
               viCit :: Map CitId Citoyen
             }
```

La figure 1 récapitule quelques fonctions usuelles sur les `Map`.

Question 1.2 Quelle(s) classe(s) de types doi(ven)t être instanciée(s) pour les types définis ci-dessus pour pouvoir définir le type `Ville` ? Instancier ces classes.

```
-- 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: Quelques fonctions utiles des `Map`

Question 1.3 Ecrire une fonction `prop_ville_sansCollision` qui prend en entrée une `Ville` et décide si toutes les zones de cette ville sont disjointes deux à deux ; c'est-à-dire si aucune zone n'entre en collision avec une autre zone.

Dans une partie de jeu, on veut qu'à tout moment :

- toutes les zones soit disjointes deux à deux,
- chaque zone résidentielle, commerciale, industrielle ou administrative est adjacente à au moins une zone de route.
- les routes sont connexes : pour chaque paire (z_d, z_f) de zones de route, il existe une séquence $(z_i)_{1 \leq i \leq n}$ telle que pour tout $1 \leq i \leq n-1$, z_i et z_{i+1} sont adjacentes, et $z_1 = z_d$, $z_n = z_d$.

Question 1.4 Ecrire un invariant pour `Ville`.

Conseil : Lire les 4 questions suivantes avant de comment à répondre à la 1.5

Question 1.5 Ecrire une fonction `construit` qui prend en entrée une `Ville e`, une `Zone z` et qui ajoute `z` dans `e`.

Question 1.6 Ecrire une précondition pour la fonction `construit` qui garantit que l'ajout d'une zone ne viole pas les conditions sur l'état décrites plus haut.

Question 1.7 Ecrire une post-condition pour la fonction `construit` qui vérifie que l'ajout d'une zone s'est fait correctement.

Question 1.8 Ecrire un jeu de tests *HSpec* pour les propriétés de la fonction `construit`.

Consignes : On peut traiter plus ou moins indépendamment les parties 2a, 2b, 2c. On ne demande pas de modélisation en détail, on pourra par exemple, pour certaines fonctions, se contenter d'en fournir une signature et une description.

Partie 2a : Bâtiments

Un bâtiment est un ensemble de cases à l'intérieur d'une zone.

Par exemple :

```
data Batiment =    Cabane Forme Coord Int [CitId]
                 | Atelier Forme Coord Int [CitId]
                 | Epicerie Forme Coord Int [CitId]
                 | Commissariat Forme Coord
```

Chaque bâtiment se trouve dans une zone. Tous les bâtiments possèdent une forme et une *entrée* : une case adjacente au batiment (donc, qui n'est pas dans sa forme) et à la zone dans laquelle se trouve le batiment (donc, qui n'est pas dans cette zone) et qui doit appartenir à une zone de route.

Les bâtiments résidentiels/commerciaux/industriels possèdent une liste des citoyens qui y habitent/y sont clients/y sont employés ainsi qu'une capacité (le nombre maximum de citoyens que cette liste peut contenir).

Les cabanes/ateliers/epiceries ne se trouvent que dans les zones résidentielles/industrielles/commerciales.

Question 2a Intégrer les bâtiments dans le jeu en ajoutant des invariants, des opérations, des préconditions et des postconditions permettant de garantir les conditions ci-dessus, d'observer (par exemple, de récupérer la population et le taux de chomage d'une `Ville`) et de manipuler (créer, détruire, modifier) des batiments.

Partie 2b : Citoyens

Les citoyens sont les acteurs de la simulation :

```
data Citoyen  =   Immigrant Coord (Int, Int, Int) Occupation
                | Habitant Coord (Int, Int, Int) (BatId, Maybe BatId, Maybe BatId) Occupation
                | Emigrant Coord Occupation
```

Tous les citoyens possèdent des coordonnées indiquant leur position sur la carte et une *occupation* (ce qu'ils sont en train de faire : travailler, dormir, faire les courses, se déplacer vers une case précise).

Les habitants et les immigrants possèdent un état composé de trois entiers, modélisant l'argent sur leur compte en banque, leur niveau de fatigue et leur niveau de faim.

Les habitants possèdent une référence vers le bâtiment dans lequel ils habitent, celui dans lequel ils travaillent (s'ils travaillent) et celui dans lequel ils vont faire leurs courses (s'il en existe un).

L'état d'un citoyen et sa position détermine son occupation. Par exemple un citoyen sur son lieu de travail va travailler (et gagner de l'argent) jusqu'à être trop affamé ou trop fatigué. Auquel cas il va quitter son travail et se dirigers vers un magasin (pour y faire des courses) ou vers son domicile (pour y dormir).

Les immigrants sont des citoyens n'ayant pas encore intégré la ville (ils cherchent à rejoindre un bâtiment résidentiel avec une place disponible), les émigrants des citoyens qui cherchent à quitter la ville.

Question 2b Intégrer les citoyens dans le jeu en précisant le type `Occupation`, en ajoutant des invariants, des opérations, des préconditions et des postconditions permettant de garantir les conditions ci-dessus et de manipuler les citoyens (créer des immigrants, transformer un immigrant en habitant, ...)

Partie 2c : Simulation

Question 2c Ajouter des types, des invariants, des opérations, des préconditions et des postconditions permettant de gérer l'évolution de la simulation en temps réel (une fonction `etape` permet de passer d'un état à l'état suivant). Les citoyens changent d'occupation en fonction de leur état. Les temps de trajets sur les routes, et l'encombrement des routes doit être géré. L'arrivée d'immigrants et le départ d'émigrants se fait sous conditions (à inventer).

Sujet de Projet PAF 2024: *Sim City*

PAF 2022 - Master STL S2

Version du 04/03/2024

Ce document présente le sujet du projet de l'UE PAF 2024 et contient les attendus, la description du projet, une liste des extensions possibles et des consignes pour le rapport. Ce document complète le sujet d'Examen Réparti 1 du 04 mars.

1 Objectifs et Rendus

1.1 Objectifs

L'objectif principal de ce projet est le développement d'un **jeu** vidéo **sûr**, dans le langage **Haskell**. L'évaluation portera sur ces trois points:

1. programmation **sûre**: les types et les fonctions associées du projet doivent toutes être accompagnées de propositions d'**invariants**, de **pré-conditions**, et de **post-conditions**. Les opérations des entités doivent toutes être testées en utilisant, de façon raisonnablement complète, les **tests unitaires** (*HSpec*) et les **tests basés sur les propriétés** (*Quickcheck*).
2. programmation **fonctionnelle**: le langage Haskell est imposé, l'application doit être écrite dans un style fonctionnel, une attention doit être donnée à la séparation de la logique (évolution de l'état du jeu) et les effets de bords (boucle de jeu, entrée/sortie, affichage). L'utilisation de spécificités du langage Haskell est encouragée: types définis comme sommes d'états remarquables, instanciation de classes de types, utilisation des structures algébriques (`Functor`, `Applicative`, `Monad`).
3. programmation **graphique**: le jeu doit utiliser la bibliothèque SDL2 afin de proposer une expérience visuelle, intuitive et fluide pour le joueur. Le jeu doit avoir un intérêt ludique non-négligeable.

Le projet sera évalué sur ces trois critères, ce qui veut dire, entre autres, que le projet peut s'éloigner des caractéristiques proposées dans ce document, tant qu'il satisfait les deux critères. Aucune définition (en particulier, les définitions de l'ER1) n'est imposée.

1.2 Rendu

Le projet est à faire en binôme (monôme possible avec autorisation de l'équipe enseignante, trinôme non-autorisé). Le rendu consiste en un projet GitLab contenant un rapport. Les enseignants doivent y être ajoutés comme *maintainer*.

La deadline est le **???**.

2 Description du Jeu

(repris du partiel 2024:)

L'objectif est de réaliser un jeu vidéo de type *city builder* sur une grille, similaire à *Sim City* (Maxis, 1989).

Le jeu se déroule sur une carte vue du dessus, le haut de l'écran représente le Nord. Le joueur place sur la carte des routes, des batiments administratifs et des zones (résidentielles, commerciales, industrielles) afin de construire et faire évoluer une *ville*.

Une *simulation* gère en temps réel la vie des citoyens de la ville *individuellement*, qui dorment, travaillent et font leurs achats dans différents batiments au sein des zones de la ville.

Les citoyens se déplacent en temps réel dans la ville, ce qui génère du traffic routier, qui peut conduire à des embouteillages et force le joueur à planifier à l'avance l'urbanisation de sa ville.

Des systèmes de contraintes et d'évolution rendent le jeu ludique, par exemple, une économie (les citoyens recoivent un salaire, le dépense en loyer, nourriture et impôts, qui sont utilisés par le joueur pour construire de nouvelles infrastructures), des contraintes de services (des réseaux d'électricité et d'eau courante doivent être mis en place, des policiers patrouillent dans la ville), une notion de qualité des zones (la "valeur" d'une zone peut évoluer en fonction de différents facteurs locaux, ce qui a un impact sur les loyers et les salaires de cette zone)

On pourra chercher sur le Web des documents plus ou moins théoriques décrivant ce style de jeux vidéo. Par exemple :

- *Wikipédia*: `https://en.wikipedia.org/wiki/City-building_game`
- *MicropolisJS* : `https://www.graememcc.co.uk/micropolisJS/`

2.1 Fonctionnalités

- l'état du jeu (carte, bâtiments, citoyens à l'extérieur des batiments, animations) doit être affiché à l'aide d'une interface graphique (en SDL2), qui représente la zone de jeu, "vue de dessus",
- les éléments de simulation décrits dans l'ER1 doivent être implémentés.
- les utilisateurs peuvent construire des bâtiments administratifs et des routes, et délimiter des zones R/C/I à la souris et au clavier..
- les extensions suivantes doivent être implémentée :
  - **Economie** : Des impots (réglables) sur les sociétés et les citoyens rapportent de l'argent au joueur. Le joueur utilise cet argent pour construire des infrastructures.
  - **Services** : (au moins) une centrale électrique doit être construite par le joueur, et l'électricité doit être acheminée (par contact entre zone, ou par des constructions spéciales) aux différentes zones pour qu'elles se développent. Des commissariats envoient régulièrement des véhicules de patrouille qui se déplacent aléatoirement (ou non) dans la ville.
  - **Qualité de vie** : des caractéristiques doivent être ajoutés aux batiments des zones en fonction de paramètres locaux (par exemple pour un batiment résidentiel, le nombre d'habitant, leur richesse, la sécurité de la zone, la proximité avec des usines polluantes, ...). Ces caractéristiques permettent

aux zones d'évoluer positivement ("cabane" devient "maisonette" et le changement s'accompagne d'une hausse des loyers et de la qualité de vie) ou négativement.
- ***Pathfinding*** : Les citoyens savent comment se déplacer efficacement d'un point à un autre, en évitant les bouchons, si possible.
- **Immigration et Emigration** : Des places (logements, emplois) disponibles dans une ville attractive attirent les immigrants. A l'inverse, les citoyens trop fatigués, affamés ou sans argent quittent la ville. Une ou plusieurs zone *d'interface* peuvent être créé pour représenter les points de départ/arrivée des immigrants/émigrants.

3 Contenu et Extensions

3.1 Consignes

Pour obtenir une **note satisfaisante** à l'évaluation, il faut que le projet respecte les contraintes suivantes:
- implémentation des **fonctionnalités** (listées ci-dessus) en Haskell + SDL2 (ou une modification du jeu de contenu équivalent),
- rédaction de **propriétés** pre/post/inv pour les types et les opérations du jeu,
- test systématique des propriétés pre/post/inv incluant **une utilisation pertinente de tests basés sur les propriétés** (*Quickcheck*).
- quelques extensions basiques, afin de proposer un contenu ludique minimal.

Pour obtenir **la note maximale**, le projet doit inclure de multiples extensions parmi celles proposées ou non (c'est encore mieux si elles sont originales), une utilisation pertinente **du test basés sur les propriétés** et **des classes algébriques**.

La motivation des extensions est importante: il faudra montrer ce qu'elles apportent au jeu, mais surtout ce qu'elles apportent au projet: complexité de la rédaction de propositions, élégance du code fonctionnel utilisé, défi pour obtenir des tests complets, ... Une extension qui ne contiendrait pas de contenu "logiciel sûr" (propositions et tests) ou de contenu "programmation fonctionnelle" (utilisation de construction propres à la programmation fonctionnelle ou à Haskell pour la mettre en oeuvre) n'ajouterait rien à l'évaluation.

3.2 Tests Basés sur la propriété

Le projet doit contenir une utilisation pertinente de *QuickCheck*, conforme à celles proposées en TME. Un soin particulier sera apporté à l'efficacité des générateurs (cf. le sujet de TME sur l'utilisation de `suchThat`).

La rédaction et l'utilisation de générateurs complexes pour des types non-triviaux est fortement encouragée. Dans tous les cas, les tests doivent être décrits dans le rapport et la méthode de rédaction des générateurs doit y être explicitée.

3.3 Classes de types

Les projets doivent rendre explicite la maîtrise des classes de types :
- manipulation correcte, élégante et efficace de `Maybe`, `Either`, `IO`, ...,
- instanciation de classes algébriques par des types définis dans le projet (par exemple `instance Monad Machin where ...`) et utilisation des bénéfices apportés par l'instanciation (primitives, *do-notations*, compositions, ...)

La programmation "avancée" avec les algébriques (rédaction de *transformers*, ...) est attendue pour une évaluation maximale.

3.4 Extensions

Voici quelques extensions possibles renforçant l'intéret du projet en tant que "jeu vidéo" (on rappelle que les extensions originales sont encouragées) :

Transports. Une gestion poussée des transports est proposée, avec plusieurs modes pour les citoyens (à pied, en voiture, en bus, en train, en avion, ...) et une gestion fine des encombrements des voies, des intersections, des potentiels accidents, ...

Services supplémentaires. Des établissement scolaires (de différents types) permettent d'éduquer les citoyens et de les faire accéder à des emplois plus rémunérateurs. Les citoyens peuvent tomber malade ou mourir et le joueur peut construire des hopitaux et des cimetières. Des déchets sont produits par les citoyens, qui doivent ensuite être traités par des services spécialisés. Tous ces services génèrent du trafic sur les routes.

Commerce et Industrie. Les biens produits par les industries locales, ou importés sont vendus dans les commerces locaux, ou exportés. Les citoyens ont des besoins spécifiques en biens de consommation, liés à leur niveau de vie.

Désastres. Des incendies (et/ou des tornades, des tsunami, des ouragants) se produisent de manière aléatoire (ou non), forçant le joueur à agir de manière proactive (construction de casernes de pompiers) et réactive (reconstruction).

Environnement local comonadiques. La "qualité" de vie d'un batiment dépend de plusieurs facteurs dans son entourage. Cette approche suggère une utilisation des `Comonad` pour traiter ces transformations à haut niveau.

Inspiration On pourra s'inspirer de jeux existants pour ajouter des fonctionnalités au projet :

- *SimCity* (*Maxi*, 1989): une série de jeux de référence pour le *city building*. Le premier est disponible dans un navigateur dans une version libre (appelée *MicropolisJS*). Attention, à l'exception du dernier jeu de la série (*SimCity 2013*), ces jeux ne simulent pas individuellement le comportement de chaque citoyen (et le dernier jeu de la série le fait d'une manière "originale").
- *Caesar II* (*Impressions Games*, 1995) (et tous les jeux de la série : *Zeus Pharaoh*, *Emperor*): des *city-builders* historiques, qui ne sont pas des simulations totales, mais qui on introduit la distribution de service par des entités qui "patrouillent" sur les routes, et affectent les batiments adjacents à leur trajet.
- *Cities: Skylines* (*Colossal Order*, 2015): deux *city-builders* récents qui proposent une simulation totale.
- ...

4 Rapport

Le rapport de projet se compose de quatre parties:

- un **manuel** d'utilisation succint expliquant comment tester le jeu depuis le projet *stack*.
- une liste exhaustive des **propositions** (invariants/pré-conditions/post-conditions) implémentées dans le projet, classées par types / opérations et contenant une brève description de ce que vérifie la propriété.
- une description des **tests** implémentés par le `Spec.hs` du projet.
- un **rapport** proprement dit, qui décrit le projet, les extensions choisies et leur implémentation et qui, en plus, explicite les points importants de l'implémentation, en mettant en évidence du code, des propositions, des tests pertinents (ou des bugs mis découvertes grâce à la méthode de développement sûre).

De manière générale, il est fortement conseillé de consigner dans le rapport (sous la forme d'un paragraphe) toutes les difficultés rencontrées dans le développement du projet et tous les *points forts* (tests basés sur la propriété, utilisation d'algébriques, invariants de type complexe, ...) du projet afin de les mettre en valeur pour l'évaluation.

5 Barême Prévisionnel

Cette suggestion de barême n'est pas opposable, il s'agit d'un exemple possible pour la notation du projet, destiné à faire comprendre les priorités d'évaluation:

- 4 points pour **le rapport**: le rapport doit être clair, lisible, et contenir les informations nécessaires (manuel, propositions, tests). Deplus, le rapport doit présenter suffisamment de points intéressants du développement du projet.
- 4 points pour l'implémentation correcte d'un **jeu minimum** en SDL2.
- 2 points pour **les propositions**: la rédaction des invariants/pre-conditions/post-conditions est systématique, pertinente, et commentée.
- 2 points pour **les tests**: les tests sont complets, correctement structurés, utilisent la puissance du *property based testing* et commentés.
- 1 point pour l'utilisation pertinente **du test basé sur la propriété**.
- 1 points pour l'utilisation pertinente **des classes algébriques**.
- 6 points pour **les extensions**: un nombre suffisant d'extensions ont été choisies, implémentées dans un style fonctionnel et (autant que possible) "haskellien", augmentées de propositions et de tests adéquats (on rappelle qu'une extension qui n'est pas accompagnée de inv/pre/post et de tests n'est pas évaluée). La pertinence, la difficulté et l'originalité des extensions choisies sont prises en compte.