TME8 - Générateurs aléatoires monadiques de QuickCheck

Sorbonne Université - Master Informatique M1 - STL

MU4IN510 Programmation Avancée en Fonctionnel - 2023

Dans ce TME nous étudions la définition de générateurs aléatoires pour le test basé sur les propriétés et QuickCheck. Ce thème est en rapport avec le cours puisque l'architecture des générateurs de QuickCheck est basée sur le concept de monade.

Attention : nous ne faisons quasiment pas de *property-based testing* dans ce TME, nous nous intéressons principalement à la définition de générateurs *custom* (le TME4 explique la partie test proprement dite).

Une fois n'est pas coutûme ce TME est plutôt du type tutoriel. Le principe est de répondre aux différents **questions** et également, d'évaluer dans GHCI (une fois le module chargé) tous les exemples d'interactions (vous pouvez remplacer les `???????` par les exemples de valeurs obtenues). Comme nous travaillons sur des générateurs aléatoires, on obtient des valeurs potentiellement différentes à chaque exécution, donc il est utile d'exécuter plusieurs fois les mêmes interactions.

Pour ce TME, nous allons utiliser directement les définitions de la bibliothèque QuickCheck.

```
import Test.QuickCheck
import System.Random (Random)
```

Première Partie : Architecture des générateurs QuickCheck

L'architecture des générateurs de QuickCheck est basée sur le type `Gen a` défini de la façon suivante :

```
>>> :info Gen
newtype Gen a
  = Test.QuickCheck.Gen.MkGen {Test.QuickCheck.Gen.unGen :: Test.QuickCheck.Random.QCGen
                                                            -> Int -> a}
    -- Defined in ‘Test.QuickCheck.Gen’
instance [safe] Applicative Gen -- Defined in ‘Test.QuickCheck.Gen’
instance [safe] Functor Gen -- Defined in ‘Test.QuickCheck.Gen’
instance [safe] Monad Gen -- Defined in ‘Test.QuickCheck.Gen’
instance [safe] Testable prop => Testable (Gen prop)
  -- Defined in ‘Test.QuickCheck.Property’
```

L'information la plus importante est que `Gen` est un contexte monadique (et donc applicatif et fonctoriel). L'intuition principale est que les générateurs se combinent essentiellement avec *bind*, et donc que la notation `do` est disponible. Nous allons y revenir mais nous allons d'abord exploiter la disponibilité d'un certain nombre de générateurs par défaut.

Générateurs par défaut avec Arbitrary

Une autre composante importante de la génération aléatoire avec QuickCheck est la *typeclass* `Arbitrary`.

```
>>> :info Arbitrary
class Arbitrary a where
  arbitrary :: Gen a
  shrink :: a -> [a]
  {-# MINIMAL arbitrary #-}
    -- Defined in ‘Test.QuickCheck.Arbitrary’
```

La fonction générique `arbitrary` peut être vue comme le générateur par défaut pour le type `a`. La fonction `shrink` implémente (lorsque cela a du sens) un algorithme de réduction de donnée qui propose une liste de valeurs «plus petites» à partir d'une valeur initiale. Nous n'aborderons pas cette aspect dans le TME, mais c'est ce qui permet à QuickCheck de trouver des contre-exemples les plus petits possibles en cas de propriété invalidée.

Il existe de nombreuses instances de `arbitrary` prédéfinies, comme par exemple des générateurs (et *shrinkers*) pour les types de base :

```
instance Arbitrary Bool
instance Arbitrary Char
instance Arbitrary Integer
instance Arbitrary Int
instance Arbitrary Float
instance Arbitrary Double
```

Pour tester ces générateurs, nous pouvons utiliser les fonctions `sample` ou `sample'` dont les signatures sont les suivantes :

```
sample :: Show a => Gen a -> IO ()
sample' :: Gen a -> IO [a]
```

La première affiche la série de valeurs choisies arbitrairement, et la seconde retourne une liste. Toutes les deux ne peuvent être utilisées que dans le contexte de `IO` car le générateur aléatoire (de nombre) par défaut n'est pas pur (il utilise une graine qui change à chaque initialisation du générateur).

Dans GHCI on peut par exemple écrire les expressions suivantes (après avec chargé le module `PAF_TME8_GenMonad`, ou directement avec `stack ghci`) :

```
>>> sample' (arbitrary :: Gen Integer)
[0,0,3,3,6,7,7,-8,3,-7,11]

>>> sample' (arbitrary :: Gen Integer)
[0,2,-2,-3,2,-10,-2,14,-2,-5,20]

>>> sample' (arbitrary :: Gen Char)
"\393105\DC1h\839920n\GSY?\758572\765624\516695"

-- >>> sample' (arbitrary :: Gen Char)
-- "Z\1082697+\304938C\609601\557853\&9\246852\US\SUB"
```

Bien sûr, les résultats changent à chaque appel, ce sont des actions et non des fonctions pures.

Il y a aussi des générateurs pour les types polymorphes de base :

```
instance Arbitrary a => Arbitrary (Maybe a)
instance Arbitrary a => Arbitrary [a]
instance (Arbitrary a, Arbitrary b) => Arbitrary (Either a b)
instance (Arbitrary a, Arbitrary b) => Arbitrary (a, b)
instance (Arbitrary a, Arbitrary b, Arbitrary c) => Arbitrary (a, b, c)
-- etc ... jusqu'aux 10-uplets !
```

La fonction `sample'` retourne des listes de 11 données, ce qui est parfois une peu trop gros, donc nous allons plutôt utiliser la fonction suivante :

```
samples :: Int -> Gen a -> IO [a]
samples n gen = do
  l <- sample' gen
  return $ take n l
```

Par exemple :

```
>>> samples 4 (arbitrary :: Gen [Integer])
???????
```

```
>>> samples 10 (arbitrary :: Gen (Maybe Int))
???????

>>> samples 3 (arbitrary :: Gen (Double, Bool))
???????
```

Question : : donner une expression avec `samples` permettant de générer des données de la forme suivante :

```
[[],[Left (-1),Left 2],[Right Nothing,Left 1,Right (Just False),Left 0]]
```

```
-- REPONDRE ICI
```

Générateurs custom

Nous allons maintenant nous intéresser à la définition proprement dite de générateurs.

Pour cela, nous allons étudier un certain nombre de *combinateurs monadiques* de génération. On a tout d'abord quelques générateurs élémentaires.

Générateurs atomiques.

Le principe de génération élémentaire est un générateur uniforme d'entiers, que l'on peut simplement utiliser pour générer des valeurs des types numériques de bases, ainsi que des types énumérés (comme `Bool` ou `Char`). Ces générateurs atomiques sont fournis par le module `System.Random` de la bibliothèque *random* (cf. https://www.stackage.org/lts-20.10/package/random-1.2.1.1).

```
>>> :info Random
class Random a where
  -- ... def pas importante ici --
  System.Random.randomR :: System.Random.RandomGen g =>
    -- Defined in ‘System.Random’
instance Random Word -- Defined in ‘System.Random’
instance Random Integer -- Defined in ‘System.Random’
instance Random Int -- Defined in ‘System.Random’
instance Random Float -- Defined in ‘System.Random’
instance Random Double -- Defined in ‘System.Random’
instance Random Char -- Defined in ‘System.Random’
instance Random Bool -- Defined in ‘System.Random’
```

La fonction de quickcheck permettant de transformer une instance de `Random` en un générateur est la suivante :

```
choose :: Random a => (a, a) -> Gen a
```

Le premier argument est un couple `(m, n)` correspondant à un intervalle avec `m` et `n` inclus:

```
>>> sample' $ choose (4, 10)
???????

>>> sample' $ choose ('a', 'z')
???????
```

Question : Définir un générateur avec la signature suivante :

```
chooseNat :: (Random a, Num a) => a -> Gen a
```

tel que `genNat n` ne génère que des nombres entre 1 et `n` (inclus)

```
chooseNat :: (Random a, Num a) => a -> Gen a
chooseNat _ = undefined
```

```
>>> sample' $ chooseNat 10
???????
```

Combinateurs élémentaires

En complément des générateurs atomiques on trouve des opérateurs assez simples permettant de composer des générateurs atomiques.

Par exemple, la fonction `elements :: [a] -> Gen a` retourne un générateur «constant» pour une valeur prise au hasard dans la liste spécifiée.

```
>>> sample' $ elements [42, 2, 17]
???????

>>> sample' $ elements ['a' .. 'e']
???????
```

Le générateur `listOf :: Gen a -> Gen [a]` génère une liste de `a`.

```
>>> samples 5 $ listOf (chooseNat 10)
???????
```

La taille de la liste générée est bornée par un paramètre `size` qui est passé (via *bind* nous allons y revenir) entre générateurs. Il est possible d'influencer sur la taille avec :

```
resize :: Int -> Gen a -> Gen a
```

Par exemple :

```
>>> samples 3 $ resize 10 $ listOf $ chooseNat 10
???????
```

Les deux premières listes sont de petite taille, mais la 3ème est bien de longueur 10. En fait, l'indicateur de taille est plus une sorte de demande informelle qu'une information à prendre précisément en compte. Pour de nombreux générateurs prédéfinis, cette taille est prise comme une borne supérieure non-stricte, et les premières valeurs générées sont souvent de petite taille. On verra un peu plus loin comment contrôler la taille un peu plus finement.

Un autre générateur utile est `oneof:: [Gen a] -> Gen a`. Il s'agit de choisir, de façon arbitraire, un générateur parmi la liste proposée. La contrainte est que tous les générateurs sont pour le même type `a`.

Par exemple :

```
>>> sample' $ oneof [choose (1, 10), choose (100, 110), choose (1000, 1010)]
???????
```

Un opérateur plus général permet de donner plus ou moins de poids à un générateur. Le poids est indiqué par un entier, et bien sûr une valeur plus grande augment la probabilité d'utiliser le générateur correspondant.

La signature de cette fonction est la suivante :

```
frequency :: [(Int, Gen a)] -> Gen a
```

Attention: il ne faut pas voir ici un contrôle fin de la distribution aléatoire, car cette dernière n'est pas maîtrisable du fait de l'enchaînement de générateurs arbitraires. On fait ici plus de la génération *arbitraire* que de la *véritable* génération aléatoire (avec contrôle de la distribution des valeurs générées).

Voici une variante de l'exemple précédent :

```
>>> sample' $ frequency [(60, choose (1, 10)), (30, choose (100, 110)), (10, choose (1000, 1010))]
???????
```

On aurait ici, *grosso modo*, 60% de chance de tirer un nombre entre 1 et 19, 30% entre 100 et 110 et 10% entre 1000 et 1010.

Voici une petite vérification «maison» de cette distribution attendue :

```
genFreq :: Gen Integer
genFreq = frequency [(60, choose (1, 10)), (30, choose (100, 110)), (10, choose (1000, 1010))]

freqStats :: (Num a, Ord a) => [a] -> (Double, Double, Double)
```

```haskell
freqStats xs =
  aux 0 0 0 xs
  where aux nb1 nb2 nb3 [] =
          let tot = nb1 + nb2 + nb3
              coef = 100.0 / (fromIntegral tot)
          in (fromIntegral nb1 * coef, fromIntegral nb2 * coef, fromIntegral nb3 * coef)
        aux nb1 nb2 nb3 (x:xs) | x <= 10 = aux (nb1+1) nb2 nb3 xs
                               | x <= 110 = aux nb1 (nb2+1) nb3 xs
                               | otherwise = aux nb1 nb2 (nb3+1) xs

checkFrequency :: (Random a, Num a, Ord a) => Gen [a] -> IO (Double, Double, Double)
checkFrequency gen = do
  xs <- generate gen
  return $ freqStats xs
```

Essayez d'invoquer plusieurs fois cette fonction de vérification de la fréquence.

```
>>> checkFrequency $ resize 100000 $ listOf genFreq
???????
```

On obtient des résultats plutôt cohérents avec la spécification, mais tout cela dépend des générateurs utilisés.

Le dernier générateur que l'on présente permet de filtrer des valeurs en entrée, par un principe de *génération avec rejet*. L'idée est très simple : on prend des valeurs en entrée et on ne conserve que celles qui vérifient une propriété donnée. Le combinateur concerné possède la signature suivante :

```haskell
suchThat :: Gen a -> (a -> Bool) -> Gen a
```

Si l'on veut par exemple uniquement générer des entiers pairs, on pourra utiliser l'expression suivante :

```
>>> sample' $ chooseNat 100 `suchThat` even
???????
```

Attention : il ne faut pas abuser du `suchThat` (que l'on utilise le plus souvent en position infixe, pour des questions de lisibilité) car les valeurs rejetées peuvent être nombreuses. En particulier, il faut être à peu près certain que le prédicat concerné est «souvent» vrai pour le générateur qu'il filtre. Voici un bon contre-exemple :

```
>>> sample' $ chooseNat 100000 `suchThat` (==1)
???????
```

Ici, *QuickCheck* a bien retourné une liste de 1 mais après un temps assez long, et l'exemple suivant ne s'arrête bien sûr jamais (ne pas le lancer !) :

```
>>> sample' $ choose (1, 10) `suchThat` (==11)
-- ... infinite loop
```

Il existe d'autres combinateurs intéressants dans la bibliothèque fournie par *QuickCheck*, on n'hésitera pas à consulter la documentation associée (rubrique *Generator combinators*).

Combinateurs monadiques

On peut créer des générateurs intéressants avec les générateurs atomiques et les combinateurs élémentaires. Cependant, pour avoir un contrôle plus fin de la génération, il faut exploiter le fait que les fonctions de signature `a -> Gen a` sont monadiques, c'est-à-dire peuvent être combinées pour produire des générateurs plus complexes.

Nous allons exploiter le fait que `Gen` est un contexte monadique. Notamment, si on a un générateur `gen` de type `Gen a` et une fonction monadique `f` de type `a -> Gen b` alors l'expression `gen >>= f` produit un générateur `Gen b` qui dépend potentiellement de `gen`.

Le générateur le plus simple que l'on puisse imaginer est celui qui génère toujours la même valeur. On cherche donc la signature `a -> Gen a` et c'est le fameux `pure` des applicatifs (qui est aussi le `return` des monades).

Par exemple :

```
>>> sample' $ (pure 42 :: Gen Int)
???????
```

Illustrons maintenant l'utilisation du *bind*. Considérons comme exemple un générateur qui inverse les valeurs produites par un générateur de nombres.

```
chooseInv :: Num a => Gen a -> Gen a
chooseInv gen = gen >>= (\x -> return (- x))
```

Par exemple :

```
>>> sample' $ chooseInv $ chooseNat 10
???????
```

Question : utiliser la notation `do` pour définir une fonction `chooseInv2` qui définit exactement le même générateur.

```
chooseInv2 :: Num a => Gen a -> Gen a
chooseInv2 _ = undefined
```

Par exemple :

```
>>> sample' $ chooseInv2 $ chooseNat 10
???????
```

A présent, étudions la génération de couples de valeurs. De façon un peu étonnante, il n'existe pas de générateurs pour les n-uplets dans la bibliothèque de base de Quickcheck. On aimerait, par exemple, pouvoir disposer d'un générateur avec la signature suivante :

```
genPair :: Gen a -> Gen b -> Gen (a, b)
```

La bibliothèque tierce *checkers* (cf. https://www.stackage.org/lts-18.28/package/checkers-0.5.7) propose un tel combinateur, sous le nom `(>*<)`, que l'on peut redéfinir aisément en exploitant la notation `do ...` :

```
genPair :: Gen a -> Gen b -> Gen (a, b)
genPair genFst genSnd = do
  x <- genFst
  y <- genSnd
  return (x,y)
```

Par exemple :

```
>>> samples 5 $ genPair (chooseNat 10) (arbitrary :: Gen Bool)
???????
```

Question : une variante applicative

On peut voir que `x` et `y` sont traités de façon indépendante, on peut même dire «parallèle», puisque on aurait pu les tirer dans un autre sens. Dans ce genre de situation, ou l'aspect séquentiel des monades n'est pas exploité, le contexte applicatif est souvent suffisant. A partir de cette remarque, définir un générateur `genPair2` qui génère des couples dans un contexte applicatif.

```
-- genPair2 :: Applicative f => f a -> f b -> f (a, b)
genPair2 :: Gen a -> Gen b -> Gen (a, b)
genPair2 _ _ = undefined
```

Par exemple :

```
>>> samples 5 $ genPair2 (chooseNat 10) (arbitrary :: Gen Bool)
???????
```

On peut bien sûr généraliser à tout type produit. Prenons l'exemple suivant :

```
data Personne = Personne { nom :: String, prenom :: String, age :: Int }
  deriving (Show, Eq)
```

Question : Définir(en style applicatif) un générateur `genPersonne` pour ce type. Pour le nom est le prénom on indiquera une taille désirée de 10, et pour l'age se sera entre 7 et 99 ans.

```haskell
genPersonne :: Gen Personne
genPersonne = undefined
```

Par exemple :

```
>>> samples 2 $ genPersonne
???????
```

Bien sûr, on peut définir des générateurs pour les types sommes (et sommes de produits).

Voici par exemple un générateur alternatif pour `Maybe` :

```haskell
genMaybe :: Gen a -> Gen (Maybe a)
genMaybe gen = do
  x <- arbitrary -- ici on tire un booléen
  case x of
    False -> return Nothing
    True -> gen >>= (pure . Just)  -- ou  do { x <- gen ; pure (Just x) }
```

Par exemple :

```
>>> samples 6 $ genMaybe $ choose (1, 10)
???????
```

Question : Soit le type suivant

```haskell
data Geom = Point
  | Rect { longueur :: Int, largeur :: Int }
  | Circle { rayon :: Int }
  deriving (Show, Eq)
```

Définir un générateur `genGeom` qui tire dans 20% des cas un point, dans 50% des cas un rectangle de longueur dans l'intervalle (5, 60) et de largeur minimale 1 et maximale 40 (en faisant attention que la longueur soit au moins aussi grande que la largeur). Dans les autres cas un cercle de rayon entre 1 et 10.

```haskell
genGeom :: Gen Geom
genGeom = undefined
```

Par exemple :

```
>>> samples 3 genGeom
???????
```

Générateurs récursifs

On peut bien sûr implémenter un algorithme récursif de génération.

En guise d'illustration, nous pouvons définir un générateur pour nombres naturels, en partant du type suivant :

```haskell
data Nat = Z | S Nat
  deriving (Show, Eq, Ord)
```

Comme tout type récursif, le principe est de générer un arbre dont les feuilles sont des constructeurs atomiques (comme `Z` pour zéro) et les noeuds internes sont les constructeurs récursifs (ici `S` pour successeur).

```haskell
genNat :: Gen Nat
genNat = do
  m <- getSize  -- on utilise le paramètre de taille comme borne max
  n <- choose (1, m)
  genAux n Z
    where genAux :: Int -> Nat -> Gen Nat
          genAux n k | n > 0 = genAux (n-1) (S k)
                     | otherwise = return k
```

Par exemple :

```
>>> samples 1 $ resize 10 $ genNat
???????
```

On se rappelle que le générateur `listOf` interprète le paramètre de taille des générateurs comme un ordre de grandeur. On peut donc définir une variante qui interprête la taille de manière plus stricte.

Question : définir un générateur `listOfSize :: Int -> Gen a -> Gen [a]` qui retourne systématiquement des listes de la taille spécifiée en argument.

```
listOfSize :: Gen a -> Int -> Gen [a]
listOfSize _ _ = undefined
```

Par exemple :

```
>>> samples 4 $ listOfSize (chooseNat 10) 5
???????
```

Nous pouvons en déduire un combinateur qui utilise le paramètre de taille implicite de la monade `Gen`.

```
sizedList :: Gen a -> Gen [a]
sizedList gen = do
  size <- getSize
  listOfSize gen size
```

Par exemple :

```
>>> samples 4 $ resize 5 $ sizedList $ chooseNat 10
???????
```

Il existe une autre façon d'écrire des variantes pour les générateurs de la forme `Int -> Gen a` avec la fonction QuickCheck suivante :

```
sized :: (Int -> Gen a) -> Gen a
```

Ici on peut donc écrire :

```
sizedList' :: Gen a -> Gen [a]
sizedList' gen = sized (listOfSize gen)
```

Par exemple :

```
>>> samples 4 $ resize 5 $ sizedList' $ chooseNat 10
???????
```

Nous allons prendre comme dernier exemple récursif le cas des arbres binaires. Nous partons du type suivant :

```
data BinTree a = Tip | Node a (BinTree a) (BinTree a)
  deriving (Show, Eq)
```

Nous allons supposer que la taille de l'arbre correspond au nombres de noeuds internes `Node`.

Un générateur naïf est par exemple le suivant :

```
genBinTreeNaive :: Gen a -> Int -> Gen (BinTree a)
genBinTreeNaive _ 0 = return Tip
genBinTreeNaive gen 1 = gen >>= \x -> return $ Node x Tip Tip
genBinTreeNaive gen size = do
  x <- gen
  lsize <- choose (0, size-1)
  l <- genBinTreeNaive gen lsize
  r <- genBinTreeNaive gen (size - lsize)
  return $ Node x l r
```

Par exemple :

```
>>> samples 1 $ genBinTreeNaive (choose (1, 5)) 5
??????
```

Question : Définir un générateur `GenBinTree` qui utilise le paramètre implicite de taillle de la monade `Gen`.

```
genBinTree :: Gen a -> Gen (BinTree a)
genBinTree gen = sized $ genBinTreeNaive gen
```

Par exemple :

```
>>> samples 1 $ resize 5 $ genBinTree (choose (1, 5))
???????
```

Structures avec invariants (complément au TME4)

Pour cette dernière partie de notre TME/tutoriel, nous discutons du cas des structures de données avec invariants, que l'on pourrait aussi appeler des structures sémantiquement *riches*. C'est important par exemple dans le cadre du projet puisque l'état du jeu sera évidemment de cette nature.

Remarque : cette partie du TME correspond plus à un approfondissement et à une suggestion d'usage pour Quickcheck dans le cadre du projet PAF que d'un exercice proprement dit.

Pour illustrer notre propos, nous allons reprendre l'exemple `Bridge` du TME4. Normalement, vous avez implémenté cet exemple et validé les tests associés. Dans ce TME nous livrons la version «squelette» du TME, qu'il faudra complèter pour que les exemples ci-dessous fonctionnent.

Nous rappelons la structure de donnée principale :

```
data IslandBridge =
  BridgeOpened Int Int Int Int
  | BridgeClosed Int Int Int
  deriving Show
```

Pour le pont ouvert `BridgeOpened` on a quatre accesseurs (tous de type `Int`) :

- `bridgeLimit` : le nombre maximum de véhicules autorisées sur l'île
- `nbCarsOnIsland` : le nombre de véhicules déjà sur l'île
- `nbCarsToIsland` : le nombre de véhicules en route vers l'île (et donc sur le pont)
- `nbCarsFromIsland` : les véhicules sur le point et qui partent de l'île

Pour le pont fermé `BridgeClosed` le nombre de véhicule sur l'île est calculable donc il n'est pas spécifié dans la structure.

On a aussi défini `nbCars` pour représenter le nombre total de véhicule présentes dans le système.

Venons-en à l'invariant du système :

```
islandBridge_inv :: IslandBridge -> Bool
islandBridge_inv b@(BridgeOpened lim _ _ _) =
  0 <= nbCars b && nbCars b <= lim
islandBridge_inv b@(BridgeClosed lim _ _) =
  nbCars b == lim
```

Cet invariant vérifie principalement que le nombre maximum de véhicules autorisés n'est pas dépassé.

Dans le TME4 nous avions proposé un premier générateur pour cette structure, sans aucune contrainte :

```
genBridgeFree :: Gen IslandBridge
genBridgeFree = do
    lim <- choose (1, 100)  -- la limite initiale
    nbI <- choose (0, 50)
    nbTo <- choose (0, 50)
    nbFrom <- choose (0, 50)
    return $ if nbI + nbTo + nbFrom == lim
             then BridgeClosed lim nbTo nbFrom
             else BridgeOpened lim nbTo nbI nbFrom
```

Ce générateur ne devrait plus poser de problème de compréhension. Cependant, on peut dire qu'il y a relativement peu de chance qu'il satisfasse l'invariant.

Par exemple :

```
>>> sample' $ fmap islandBridge_inv genBridgeFree
???????
```

Bien sûr, un tel générateur est intéressant pour effectuer des tests *hors-limites*, notamment pour tester que les vérifications d'invariants, de pré et post-conditions sont bien effectuées. Mais pour effectuer des tests fonctionnels ce n'est pas une approche très intéressante.

Lorsque l'invariant est simple, comme c'est le cas ici, on peut parfois trouver une méthode de génération directe. Pour le TME4 voici ce que nous proposions.

```
genBridgeOk :: Gen IslandBridge
genBridgeOk = do
    lim <- choose (1, 100)  -- la limite initiale
    nbCs <- choose (0, lim)  --
    nbI <- choose (0, nbCs)
    nbTo <- choose (0, nbI)
    let nbFrom = nbCs - (nbI + nbTo)
    return $ mkBridge lim nbTo nbI nbFrom
```

Par exemple :

```
>>> sample' $ fmap islandBridge_inv genBridgeOk
???????
```

Cependant, cette méthode directe est souvent fastidieuse, et parfois beaucoup trop complexe. D'une certaine façon, il s'agit d'imaginer à l'avance et une fois pour toute tous les domaines de valeurs acceptables.

Hors, c'est justement le but de la programmation sûre avec les opérations, préconditions et/ou postconditions : décrire les domaines acceptables par le biais des opérations que l'on souhaite effectuer.

Pour le `Bridge`, les opérations principales sont les suivantes :

- `initBridge :: Int -> IslandBridge` qui construit un état initial avec la limite du nombre de véhicules autorisés
- `enterToIsland :: IslandBridge -> IslandBridge` qui fait entrer un véhicule sur le pont en direction de l'île
- `leaveToIsland` qui fait sortir un véhicule du pont pour entrer dans l'île
- `enterFromIsland :: IslandBridge -> IslandBridge` qui fait entrer un véhicule sur le pont depuis l'île
- `leaveFromIsland` qui fait quitter un véhicule du point et du système

Pour construire un générateur qui respecte l'invariant, et qui par la même occasion permet de tester les opérations proprement dite, on peut procéder de la façon suivante.

1. on commence par définir un type somme décrivant les opérations qui nous intéressent
2. on écrit un interprète pour cette structure, qui applique les actions spécifiées à partir du type précédent. On peut imaginer un interprète *libre* qui ne fait qu'exécuter les actions, et un interprête *sûr* qui vérifie au passage les préconditions et les invariants.
3. un générateur pour le système procède en deux étapes : a. génération d'une liste d'actions à effectuer b. exécution par l'interprête de la liste d'actions, qui produit finalement un état *arbitraire* du système.

Cette méthode de définition d'un générateur est très efficace en terme de test fonctionnel, mais il faut faire attention que l'on n'a aucune idée dans la plupart des cas de la distribution aléatoire obtenue par ce procédé de génération. Il faut se rappeler que le but est de générer des structures correctes mais totalement arbitraires. Il est très difficile par exemple de parler précisément de couverture. Mais c'est de toute façon une question très complexe dans le cas des structures à invariants (hors cas triviaux comme le `Bridge`).

Illustrons cette méthode étape par étape, en commençant par notre structure de description des actions.

```haskell
data BridgeAction =
  Act_BridgeInit Int
  | Act_EnterToIsland
  | Act_LeaveToIsland
  | Act_EnterFromIsland
  | Act_LeaveFromIsland
  deriving (Show, Eq)
```

Une liste correcte d'actions commence toujours par `Act_BridgeInit` et des propriétés de bon parenthèsage, comme par exemple que `Act_LeaveToIsland` doit être précédé d'un `Act_EnterToIsland` dans la liste (mais pas forcément le prédécesseur), etc. On a donc transformé des contraintes sémantiques sur l'invariant en des contraintes structurelles sur les listes d'opérations. Des structures plus complexes que les listes peuvent parfois être mises en oeuvre, mais cela dépasse le cadre de ce TME.

La deuxième étape consiste à définir un interprète pour les listes d'actions. Nous allons prendre un interprète sûr qui retourne un `Either String IslandBridge`. Le cas `Left` correspond à l'invalidation d'un invariant ou d'une précondition, et le cas `Right` construit un état correct.

```haskell
interpBridge :: [BridgeAction] -> Either String IslandBridge
interpBridge ((Act_BridgeInit lim):acts)
  | initBridge_pre lim = interp acts (initBridge lim)
  | otherwise = Left $ "Initialisation error: precondition failed"
  where interp :: [BridgeAction] -> IslandBridge -> Either String IslandBridge
        interp [] bridge = Right bridge
        interp (a:as) bridge = do
          bridge' <- interp1 a bridge
          interp as bridge'

        interp1 :: BridgeAction -> IslandBridge -> Either String IslandBridge
        interp1 (Act_BridgeInit _) _ = Left $ "Initialization in the middle of the list"
        interp1 act@Act_EnterToIsland bridge =
           interpOp act enterToIsland_pre enterToIsland bridge
        interp1 act@Act_LeaveToIsland bridge =
           interpOp act leaveToIsland_pre leaveToIsland bridge
        interp1 act@Act_EnterFromIsland bridge =
           interpOp act enterFromIsland_pre enterFromIsland bridge
        interp1 act@Act_LeaveFromIsland bridge =
           interpOp act leaveFromIsland_pre leaveFromIsland bridge

interpBridge _ = Left $ "List of actions must be non-empty and start with an initialization"

interpOp :: BridgeAction -> (IslandBridge -> Bool) -> (IslandBridge -> IslandBridge)
         -> IslandBridge -> Either String IslandBridge
interpOp act precond operation bridge
  | precond bridge = Right $ operation bridge
  | otherwise = Left $ "Action " <> (show act) <> " error: précondition failed"
```

On peut imaginer plusieurs stratégies pour notre générateur. Nous allons adopter une stratégie simple qui consiste à choisir le nombre de véhicules à faire entrer dans le système à l'avance. Pour chaque véhicule on va choisir entre 4 listes d'actions possibles. Et finalement, nous allons entrelacer les séquences obtenues de façon aléatoire, en utilisant la fonction suivante :

Ce générateur consiste simplement à mélanger les éléments de la liste fournie en entrée.

```haskell
vehicleActs :: [BridgeAction]
vehicleActs = [Act_EnterToIsland, Act_LeaveToIsland, Act_EnterFromIsland, Act_LeaveFromIsland]

genVehicle :: Gen [BridgeAction]
genVehicle = do
```

```
  nbAct <- choose (1, 4)
  return $ take nbAct vehicleActs
```

Par exemple :

```
>>> samples 1 genVehicle
???????
```

On va ensuite générer une liste de ces véhicules.

```
genVehicles :: Int -> Gen [[BridgeAction]]
genVehicles nb = listOfSize genVehicle nb
```

Arrivé à ce stade nous avons tous nos véhicules mais il faut encore mélanger les actions. Le problème, bien sûr, est qu'il ne faut pas mélanger les actions qui concerne un véhicule donné.

Pour cela, nous allons prendre une fonction qui extrait une action d'un des véhicules, et retire le véhicule si ce dernier n'a plus d'action disponible.

```
ntake :: Int -> [[a]] -> (a, [[a]])
ntake _ [] = error "Empty list"
ntake 0 ([x]:xss) = (x,xss)
ntake 0 ((x:xs):xss) = (x, xs:xss)
ntake k (xs:xss) =
  let (y,xss') = ntake (k-1) xss
  in (y, xs:xss')
```

```
>>> ntake 2 [[1, 2, 3], [4, 5], [6, 7, 8], [9, 10]]
???????

>>> ntake 2 [[1, 2, 3], [4, 5], [7, 8], [9, 10]]
???????

>>> ntake 2 [[1, 2, 3], [4, 5], [8], [9, 10]]
???????
```

```
shuffleLists :: [[a]] -> Gen [a]
shuffleLists [] = pure []
shuffleLists xss = do
          k <- choose (0, (length xss) -1)
          let (y,yss) = ntake k xss
          ys <- shuffleLists yss
          return (y:ys)
```

```
>>> fmap head $ samples 1 $ shuffleLists [["a1", "a2", "a3"], ["b1", "b2"], ["c1","c2","c3"]]
???????
```

On remarque que les actions produites respectent l'ordre de chaque sous-liste, et au final tous les éléments ont été choisis.

Voici finalement notre générateur pour notre exemple `Bridge`.

```
genBridgeLim :: Int ->  Gen IslandBridge
genBridgeLim lim = do
  nbVehicles <- choose (1, lim)
  baseVehicles <- genVehicles nbVehicles
  vehicles <- shuffleLists baseVehicles
  let actions = (Act_BridgeInit lim):vehicles
  case interpBridge actions of
    Left msg -> error $ "Generation failed: " <> msg
    Right bridge -> return bridge
```

```haskell
genBridge :: Gen IslandBridge
genBridge = sized genBridgeLim
```

Par exemple :

```
>>> samples 1 $ genBridgeLim 10
-- ... erreur tant que les opérations sont undefined

>>> samples 1 $ resize 10 genBridge
-- ... erreur tant que les opérations sont undefined
```