Série d'exercices #11

IFT-2035

June 12, 2025

11.1 Objets géométriques

Soit le langage hypothétique Claskell qui combine Haskell avec un système orienté objet à base de classes traditionnelles.

```
class Drawable { method draw :: Image → Image; };
class Square extends Drawable
{ size :: Float; method draw(d :: Image) {...}; };
class Circle extends Drawable
{ radius :: Float; method draw(d :: Image) {...}; };
```

Sachant donc que *Circle* $\subseteq$ *Drawable* et *Square* $\subseteq$ *Drawable*, indiquer quelles relations de sous-typage seraient valides:

1. *Square* $\subseteq$ *Square*
2. *Circle* $\to$ *Circle* $\subseteq$ *Circle* $\to$ *Drawable*
3. *Circle* $\to$ *Circle* $\subseteq$ *Drawable* $\to$ *Drawable*
4. List *Square* $\subseteq$ List *Drawable*
5. (*Square* $\to$ *Drawable*) $\to$ *Circle* $\subseteq$ (*Square* $\to$ *Circle*) $\to$ *Drawable*

Soit un objet c :: *Circle* et soit une fonction *zoom* qui prend un facteur d'agrandissement et un objet de la classe *Drawable*, et renvoie un objet équivalent mais agrandi. Donner le type de:

6. *zoom*
7. *zoom* 2.0 c

Finalement, un programmeur tout aussi hypothétique décide de traduire le code vers Haskell (et ses classes de type):

```
class Drawable a with draw :: Image → Image
data Square = Square { size :: Float }
data Circle = Circle { radius :: Float }
instance Drawable Square where
```

```
  draw d = ...
instance Drawable Circle where
  draw d = ...
c :: Circle = ...
```

Donner le type Haskell qui en résulte pour:

6. *zoom*
7. *zoom* 2.0 c

11.2 Typage d'une méthode nue

Soit un langage orienté objet à base de classes, similaire à Java. Et soit un code source tel que:

```
myMethod (Drawable x1, Drawable x2) {
  x1.display (self);
  x2.display (self);
}
```

Le code de bas niveau généré par le compilateur pourrait alors ressembler à:

```
myMethod (Display self, Drawable x1, Drawable x2) {
  c1 = x1->class;
  m1 = c1->display;
  m1 (x1, self);
  c2 = x2->class;
  m2 = c2->display;
  m2 (x2, self);
}
```

Si on imagine que le langage de bas niveau est typé statiquement, quel pourrait être le type de *m1*? et de *c1*?
Soit le code alternatif de bas niveau ci-dessous:

```
myMethod (Display self, Drawable x1, Drawable x2) {
  c1 = x1->class;
  m1 = c1->display;
  m1 (x1, self);
  m1 (x2, self);
}
```

Ce code pourrait être dangereux, cependant.

1. Donner un scénario où cette version mène à un "core dump".
2. Discuter de comment il faudrait changer les types de *c1* et *m1* pour que le vérificateur de type du langage de bas-niveau le refuse?

11.3 Structure de données infinie

Les listes infinies sont souvent appelées *streams*. En Haskell, l'ordre d'évaluation utilisé permet d'utiliser n'importe quelle structure de donnée infinie sans effort particulier. Prenons par exemple les définitions ci-dessous:

```
zeros = 0 : zeros
uns = 1 : uns
numbers = 0 : zipWith (+) uns numbers
f = 0 : 1 : zipWith (+) f (tail f)
```

De plus, Haskell prédéfini les opérations suivantes:

```
(x:_)  !! 0 = x
(_:xs) !! n = xs !! (n - 1)

take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n - 1) xs
```

Avec les définitions ci-dessus, nous avons:

$$\begin{array}{lll} \textit{take}\ 5\ \textit{uns} & \rightsquigarrow^* & [1,1,1,1,1] \\ \textit{take}\ 5\ \textit{numbers} & \rightsquigarrow^* & [0,1,2,3,4] \\ \textit{numbers}\ !!\ 3 & \rightsquigarrow^* & 3 \end{array}$$

1. Montrer les étapes de l'évaluation de:

   ```
   f !! 3
   ```

   Expliquer en détail la notation que vous avez choisi d'utiliser. Attention à l'utiliser de manière cohérente et rigoureuse.

2. Soit la liste infinie de nombres suivante:

   ```
   (1 1/2 1/6 1/24 1/120 ... 1/n! ...
   ```

   Définir cette liste récursivement en utilisant *zipWith*, la liste infinie des entiers *numbers* et la liste circulaire `uns`.