Semaine 9 : Contraintes

- Les programmes informatiques sont généralement organisés comme des calculs unidirectionnels qui consomment des entrées et produisent des sorties.
- La programmation fonctionnelle (pure) rend cela explicite dans le code source, puisqu'on a :

  entrée = argument de fonction sortie = résultat de fonction
- Les mathématiques, par contre, ne sont pas toujours unidirectionnelles.
- Par exemple, dans l'équation $d \cdot A \cdot E = F \cdot L$, on peut calculer la valeur de n'importe quelle variable au moyen de la valeur des quatre autres.
- Par ex.

$$d = F \cdot L/(A \cdot E)$$

$$A = F \cdot L/(d \cdot E), \text{ etc}$$

Un langage pour les contraintes

Développons maintenant un *langage de contraintes* qui permette à l'utilisateur de formuler des équations de ce type et de demander au système de les résoudre.

Il y a deux niveaux :

- Les contraintes comme réseau : des contraintes primitives liées par des connecteurs.
- Les contraintes comme équations algébriques.

Example: La relation entre les températures en degrés Celsius et Fahrenheit est :

$$C * 9 = (F - 32) * 5$$

Cela s'exprime ainsi sous forme d'un réseau de contraintes.

Utilisation du système de contraintes

Admettons que l'on veuille convertir entre Celsius et Fahrenheit.

On crée un convertisseur en définissant

$$\textbf{val}\ C,\ F = \textbf{new}\ Quantity$$
$$CFconverter\,(C,\ F\,)$$

Utilisation du convertisseur

Ici, *CFconverter* est une méthode qui construit un réseau de contraintes.

```
def CFconverter(c: Quantity, f: Quantity) = {
    val u, v, w, x, y = new Quantity
    Constant(w, 9);   Multiplier(c, w, u)
    Constant(y, 32); Adder(v, y, f)
    Constant(x, 5); Multiplier(v, x, u)
}
```

En comparant avec la représentation graphique du réseau, on constate que :

- les boîtes sont des contraintes, telles que *Multiplier*, *Adder*, *Constant*,
- les connecteurs sont des quantités (c.-à-d. des instances de la classe *Quantity*).

Pour voir le réseau fonctionner, démarrons l'interpréteur :

scalaint constr.scala
scala> *:compile Constraints.scala*
scala> ***import*** *CFconversion._*

definissons des quantités C et F et plaçons des sondes sur eux:

scala> ***val*** *C, F =* ***new*** *Quantity*
scala> *Probe("Celsius temp", C)*
scala> *Probe("Fahrenheit temp", F)*
scala> *CFconverter(C, F)*

Ensuite, donnons une valeur à une des quantités :

scala> *C setValue 25*
Probe: Celsius temp = 25
Probe: Fahrenheit temp = 77

Essayons maintenant de donner une valeur à F :

> F setValue 212
Error! contradiction: 77 and 212

Si nous désirons réutiliser le système avec de nouvelles valeurs, il

nous faut d'abord "oublier" les anciennes valeurs :

$> C.forgetValue$
$Probe: Celsius\ temp\ = ?$
$Probe: Fahrenheit\ temp = ?$
$> F\ setValue\ 212$
$Probe: Celsius\ temp\ = 100$
$Probe: Fahrenheit\ temp = 212$

Notez que le même réseau permet de calculer C à partir de F et F à partir de C.

Cette absence de direction de calcul est caractéristique des systèmes basés sur des *contraintes*.

De tels systèmes sont communs aujourd'hui ; toute une industrie s'y intéresse.

Exemples : ILOG Solver (et JSolver), TK!solver.

Souvent aussi, les systèmes de contraintes *optimisent* certaines quantités en fonction d'autres ; toutefois, nous ne traiterons pas cela ici.

Implantation du système de contraintes

L'implantation du système de contraintes est quelque peu similaire à l'implantation du simulateur de circuits logiques.

Un système de contraintes est composé de contraintes (boîtes) primitives et de quantités (connecteurs).

Les contraintes primitives simulent des équations simples entre des quantités x, y, z, telles que :

$$x = y + z,$$
$$x = y * z,$$
$$x = c$$

où c est une constante.

Une quantité est soit définie soit indéfinie.

Une quantité peut connecter un nombre quelconque de contraintes.

Voici l'interface d'une quantité :

```
class Quantity {
    def getValue: Option[double] = ...
    def setValue(v: double, setter: Constraint): unit = ...
    def setValue(v: double): unit = setValue(v, NoConstraint)
    def forgetValue(retractor: Constraint): unit = ...
    def forgetValue: unit = forgetValue(NoConstraint)
    def connect(c: Constraint) = ...
}
```

Explications :

getValue retourne la valeur actuelle de la quantité.

setValue donne la valeur, et *forgetValue* l'oublie.

Ces deux méthodes existent en deux variantes surchargées.

Une des variantes (utilisée en interne par le système de contraintes) passe la contrainte qui cause la modification ou l'oubli de valeur en paramètre.

connect déclare que la quantité participe à une contrainte.

Le type *Option*

Le type *Option* est défini ainsi :

```
trait Option[+a]
case class Some[+a](value: a) extends Option[a]
case object None extends Option[Nothing]
```

L'idée est que la fonction *getValue* retourne

- *None* si aucune valeur n'est définie, ou
- *Some*(*x*) si la valeur de la quantité est *x*.

Les clients de *getValue* utilisent alors le filtrage de motifs pour décomposer la valeur :

```
q.getValue match {
    case Some(x) ⇒ /* fait quelque chose avec la valeur 'x' */
    case None    ⇒ /* traite la valeur indéfinie */
}
```

Covariance

La définition de *Option* illustre plusieurs aspects du système de types de Scala.

- Le + devant le paramètre de type *a* indique que *Option* est un constructeur de type co-variant:

  Si T est un sous-type de S (noté $T <: S$), alors $Option[T]$ est un sous-type de $Option[S]$.

  Par exemple, $Option[String]$ est un sous-type de $Option[Object]$.
- Sans le + dans la définition de la classe *Option*, $Option[String]$ et $Option[Object]$ seraient deux types incomparables.
- Question : Pourquoi les constructeurs de classe ne peuvent-ils pas toujours être covariants ?

- *None* est défini comme un objet “cas”. Autrement dit, c’est l’unique valeur qui hérite de $Option[Nothing]$.
- Le type *Nothing* est un sous-type de n’importe quel autre type. Par exemple, $Nothing <: String <: Object$.
- Étant donné que *Option* est covariant, cela signifie que *None* est une valeur de n’importe quel type de la forme $Option[T]$. Par exemple, $Option[Nothing] <: Option[String] <: Option[Object]$.

Contraintes

L'interface d'une contrainte est simple.

```
abstract class Constraint {
    def newValue: unit
    def dropValue: unit
}
```

Il n'y a que deux méthodes, *newValue* et *dropValue*.

newValue est appelée lorsqu'une des quantités connectée à une contrainte reçoit une nouvelle valeur.

dropValue est appelée lorsqu'une des quantités connectée à une contrainte perd sa valeur.

Lorsqu'elle est "réveillée" par un appel à *newValue*, une contrainte essaie de calculer la/les valeur(s) des quantités auxquelles elle est connectée.

Si elle y arrive, elle *propage* ces valeurs en appelant *setValue* pour tous les participants connectés.

Lorsqu'elle est "réveillée" par un appel à *dropValue*, une contrainte dit simplement à tous ses participants d'oublier également leur valeur.

On a donc deux séquences d'appels mutuellement récursifs.

$q.setValue \rightarrow c.newValue \rightarrow q'.setValue$
$q.forgetValue \rightarrow c.dropValue \rightarrow q'.forgetValue$

Implantation des contraintes primitives

L'implantation des contraintes primitives est désormais facile.

```
case class Adder(a1: Quantity, a2: Quantity, sum: Quantity)
                    extends Constraint {
  def newValue = Triple(a1.getValue, a2.getValue, sum.getValue) match {
    case Triple(Some(x1), Some(x2), _) ⇒ sum.setValue(x1 + x2, this)
    case Triple(Some(x1), _, Some(r))  ⇒ a2.setValue(r − x1, this)
    case Triple(_, Some(x2), Some(r))  ⇒ a1.setValue(r − x2, this)
    case _ ⇒
  }
  def dropValue: unit = {
    a1.forgetValue(this); a2.forgetValue(this); sum.forgetValue(this)
  }
  a1 connect this
  a2 connect this
  sum connect this
}
```

Explications :

- *newValue* fait un filtrage de motifs sur les trois quantités connectées par l'additionneur.
- Si deux des valeurs sont définies, la troisième est calculée et définie.
- *dropValue* se propage aux quantités connectées.
- Le code d'initialisation connecte l'additionneur avec les trois quantités passées.

Exercice : Écrivez une contrainte de multiplication. La contrainte devrait "savoir" que $0 * x = 0$, même si x est indéfini.

Constantes

Une constante est un cas particulier de contrainte.

On l'implante ainsi :

```
case class Constant(q: Quantity, v: double) extends Constraint {
    def newValue: unit = error("Constant.newValue")
    def dropValue: unit = error("Constant.dropValue")
    q connect this
    q.setValue(v, this)
}
```

Remarques :

- Les constantes ne peuvent être redéfinies ou "oubliées". C'est pourquoi *newValue* et *dropValue* produisent une erreur.
- Les constantes donnent immédiatement une valeur à la quantité attachée.

Quantités

Il nous reste à implanter les quantités.

L'état d'une quantité est donné par trois valeurs :

- sa valeur courante (*value*),
- les contraintes qui y sont attachées (*constraints*),
- "l'informateur", c.-à-d. la contrainte qui a causé la dernière définition de la valeur (*informant*).

L'informateur permet d'éviter la propagation infinie des valeurs en présence de cycles.

```
class Quantity {
    private var value: Option[double] = None
    private var constraints: List[Constraint] = List()
    private var informant: Constraint = NoConstraint; ... }
object NoConstraint extends Constraint { ... }
```

Voilà l'implantation de *getValue* et *setValue* :

```
def getValue: Option[double] = value

def setValue(v: double, setter: Constraint) = value match {
    case Some(v1) ⇒
        if (v != v1) error("Error! contradiction: " + v + " and " + v1)
    case None ⇒
        informant = setter; value = Some(v)
        for (val c ← constraints; c != informant) c.newValue
}
def setValue(v: double): unit = setValue(v, NoConstraint)
```

La méthode *setValue* signale une erreur lorsqu'on essaie de modifier une valeur déjà définie.

Sinon, elle propage le changement en appelant *newValue* sur toutes les contraintes attachées, informateur excepté.

Voici l'implantation de *forgetValue* et *connect* :

```
def forgetValue(retractor: Constraint): unit = {
    if (retractor == informant) {
        value = None
        for (val c ← constraints; c != informant) c.dropValue
    }
}
def forgetValue: unit = forgetValue(NoConstraint)
```

La méthode *forgetValue* "oublie" la valeur (en la remettant à *None*) seulement si l'appel provient de la contrainte à l'origine de la valeur.

Elle propage ensuite la modification en appelant *dropValue* sur toutes les contraintes attachées, informateur excepté.

Un appel à *forgetValue* provenant de quelqu'un d'autre que l'informateur est ignoré.

Voici l'implantation de *connect*.

```
def connect (c: Constraint): unit = {
    constraints = c :: constraints
    value match {
        case Some(_) ⇒ c.newValue
        case None ⇒
    }
}
```

Cette méthode ajoute la contrainte à la liste *constraints*.

Si la quantité a une valeur, elle appelle aussi *newValue* sur la nouvelle contrainte.

Sondes

Les sondes sont des contraintes particulières, qui affichent simplement tous les changements de la quantité attachée.

Elle sont implantées de la manière suivante :

```
case class Probe(name: String, q: Quantity) extends Constraint {
    def newValue: unit = printProbe(q.getValue)
    def dropValue: unit = printProbe(None)
    private def printProbe(v: Option[double]): unit = {
        val vstr = v match {
            case Some(x) ⇒ x.toString()
            case None ⇒ "?"
        }
        Console.println("Probe: " + name + " = " + vstr)
    }
    q connect this
}
```

Amélioration

Le système présenté fonctionne, mais les contraintes restent fastidieuses à définir.

Comparez l'équation :

$$C * 9 = (F - 32) * 5$$

avec le code qui définit *CFconverter*.

Ne serait-il pas agréable de pouvoir construire un système de contraintes directement à partir d'une équation telle que celle ci-dessus ?

On peut presque y arriver en Scala. Voici une nouvelle manière d'exprimer la conversion Celsius/Fahrenheit :

```
val C, F = new Quantity
C * constant(9) === (F + constant(-32)) * constant(5)
```

Ici,

- $*$ et $+$ sont de nouvelles méthodes de la classe *Quantity* qui prennent une quantité et retournent une nouvelle quantité attachée à la contrainte correspondante.
- c est une fonction qui retourne une quantité attachée à une contrainte constante.
- $===$ est une méthode de *Quantity* qui prend une quantité et construit une contrainte d'égalité.

Par exemple, voici une implantation de la méthode $+$ dans la classe *Quantity* :

```
def + (that: Quantity): Quantity = {
    val sum = new Quantity
    Adder (this, that, sum)
    sum
}
```

Les vues

La relation Celsius/Fahrenheit présentée plus haut peut être exprimée telle quelle en Scala. Scala introduit en effet la notion de "vue" entre types.

Soit deux objets de types incompatibles tels que *double* et *Quantity* l'utilisateur peut définir une conversion *implicite* indiquant au compilateur comment se comporter lorsqu'il rencontre un objet de type *int* lorsque le type attendu est *Quantity*.

Example:

```
implicit def constant (x: double): Quantity = {
    val q = new Quantity
    Constant (q, v)
    q
}
implicit def intConstant (x: int): Quantity = constant (x)
```

Si la conversion implicit est visible sans prefix (i.e importé ou defini dans la portée de l'utilisation), on obtient:

$$\textbf{val}\ C, F = \textbf{new}\ Quantity$$
$$C * 9.0 === (F - 32.0) * 5.0$$

(ou bien)

$$C * 9 === (F - 32) * 5$$

Résumé

Nous avons appris un nouveau paradigme de calcul : le calcul par résolution de *relations* ou *contraintes*.

La caractéristique principale de ce paradigme est que le calcul peut s'effectuer dans plus d'une direction, en fonction de ce qui est défini et de ce qui ne l'est pas.

L'implantation présentée ici est basée sur un réseau de contraintes (nœuds) et de quantités (arêtes).

La résolution des contraintes implique la propagation des changements de valeurs le long des arêtes et à travers les nœuds.

Le réseau est modélisé par un ensemble d'objets dont certains contiennent un état.