Utilisation du convertisseur

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

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

En comparant cela à 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*).

Utilisation du système de contraintes

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

On crée un convertisseur en définissant

```
val C = new Quantity;
val F = new Quantity;
CFconverter(C, F);
```


Un langage pour les contraintes

Développons maintenant un *langage de contraintes* qui permet 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 : contraintes primitives liées par des connecteurs.
- Les contraintes comme équations algébriques.

Exemple : 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.

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.
- P.ex.

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

Voici l'interface d'une quantité :

```
class Quantity with {
  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.

Implantation du système de contraintes

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

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.


```
> C forgetValue
Probe: Celsius temp  = ?
Probe: Fahrenheit temp = ?
> F setValue 212
Probe: Celsius temp  = 212
Probe: Fahrenheit temp = 100
```

Notez que le même réseau est utilisé pour 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, TK!solver.

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

Pour voir le réseau fonctionner, démarrons l'interpréteur :

```
siris constr.scala
> :l CFconversion.scala
```

et plaçons des sondes sur les quantités C et F :

```
> probe("Celsius temp", C);
> probe("Fahrenheit temp", F);
```

Ensuite, donnons une valeur à une des quantités :

```
> 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 "oublier" les anciennes valeurs d'abord :

Implantation des contraintes primitives

L'implantation des contraintes primitives est désormais facile.

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


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 → *c.newValue* → *q'.setValue*
q.forgetValue → *c.dropValue* → *q'.forgetValue*

Contraintes

L'interface d'une contrainte est simple.

```
abstract class Constraint with {
  abstract def newValue: Unit;
  abstract 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.

Le type *Option*

Le type *Option* est défini ainsi :

```
abstract class Option[a];
case class Some[a](value: a) extends Option[a];
case class None[a] extends Option[a];
```

L'idée est que *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 */
}
```

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)) do 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.

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 with {
  private var value: Option[Double];
  private var constraints: List[Constraint] = [];
  private var informant: Constraint = NoConstraint;
```


Constantes

Une constante est un cas particulier de contrainte.

On l'implante ainsi :

```
class Constant(q: Quantity, v: Double) extends Constraint with {
  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.

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.

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/Fahrenheits :

```
val C = new Quantity;
val F = new Quantity;‘
C * c(9) === (F + c(−32)) * c(5);
```


Sondes

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

Elle sont implantées de la manière suivante :

```
class Probe(name: String, q: Quantity) extends Constraint with {
  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 ⇒ "?"
    }
    System.out.println("Probe: " + name + " = " + vstr);
  }
  q connect this;
}
```


Voici l'implantation de *connect*.

```
def connect(c: Constraint) = {
  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.

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

```
def forgetValue(retractor: Constraint): Unit = {
  if (retractor == informant) {
    value = None;
    for (val c ← constraints; !(c == informant)) do 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.

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;
}
```


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.