A Language for Constraints

We now develop a *constraint language* that allows the user to formulate equations like these and to request from the system to solve them.

There are two levels:

- Constraints as networks: Primitive constraints joined by connectors.
- Constraints as algebraic equations.

Example: The relationship between Celsius and Fahrenheit is:

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

This is expressed as a constraint network as follows.

Week 10: Constraints

- Computer programs are usually organized as one-directional computations which consume certain inputs and produce certain outputs.
- (Pure) functional programming makes this explicit in the source program, with

  input = function argument   output = function result
- Mathematics, on the other hand, is not always uni-directional.
- For instance, in an equation $d \cdot A \cdot E = F \cdot L$, any four of the quantities d, A, E, F, L can produce the fifth.
- E.g.

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

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

Using the Converter

Here *CFconverter* is a method that constructs a constraint network.

```
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);
}
```

Compared to the graphical representation of the network, we have:

- Boxes are constraints such as *Multiplier*, *Adder*, *Constant*.
- Connectors are quantities (i.e. instances of class *Quantity*).

Using the Constraint System

Let's say we want to convert between Celsius and Fahrenheit.

A converter is created by defining

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

To watch the network in action, start the interpreter

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

and place probes on the C and F quantities.

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

Next, set one of the two quantities:

```
> C setValue 25
Probe: Celsius temp    = 25
Probe: Fahrenheit temp = 77
```

Now we can try to set F to a different value.

```
> F setValue 212
Error! contradiction: 77 and 212
```

If we want to re-use the system with new values, we have to forget old values first:


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

Note that the same network is used to compute C given F and to compute F given C.

This non-directionality is characteristic of *constraint-based* systems.

Such systems are by now quite common; there is an industry building them.

Examples: ILOG solver, TK!solver.

Constraint systems usually also *optimize* some quantities given others; however, this is not treated here.

Here is the interface of a quantity.

```
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) = ...;
}
```

Explanations:

getValue returns the current value of the quantity.

setValue sets the value, and *forgetValue* forgets it.

Both methods exist in two overloaded variants.

One of the variants (which is called internally by the constraint system) passes the constraint that causes the set or forget as additional parameter.

connect declares that the quantity participates in a constraint.

Implementing the Constraint System

The implementation of the constraint system is a bit similar to the implementation of the digital circuit simulator.

A constraint system is built from primitive constraints (boxes) and quantities (connectors).

Primitive constraints simulate simple equations between quantities x, y, z, such as

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

where c is a constant.

A quantity is either defined or undefined.

A quantity can connect any number of constraints.

Constraints

The interface of a constraint is quite simple.

```
abstract class Constraint with {
  abstract def newValue: Unit;
  abstract def dropValue: Unit
}
```

There are two methods, *newValue* and *dropValue*.

newValue is called when one of the quantities connected to a constraint has been set to a new value.

dropValue is called when one of the quantities connected to a constraint has lost its value.

When woken up by a *newValue*, a constraint tries to compute the value(s) of all quantities connected to it.

If successful, it *propagates* these values by calling *setValue* on all connected participants.

Type *Option*

The *Option* type is defined by:

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

The idea is that

- *getValue* returns *None* if no value is defined, and
- it returns *Some(x)* if the quantities value is *x*.

Clients of *getValue* then use pattern matching to decompose values:

```
q.getValue match {
  case Some(x) ⇒ /* do something with value 'x' */
  case None    ⇒ /* handle undefined value */
}
```

Implementing Primitive Constraints

The implementation of primitive constraints is now straightforward.

```
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;
}
```


When woken up by a *dropValue*, a constraint simply tells all its participants to drop their values as well.

So we have two mutually recursive calling sequences.

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

Constants

A constant is a special case of constraint.

It is implemented as follows.

```
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);
}
```

Remarks:

- Constants cannot be redefined or undefined. That's why *newValue* and *dropValue* give errors.
- Constants immediately set the value of the attached quantity.

Explanations:

- *newValue* does a pattern match over the three quantities connected by an adder.
- If two of the values are defined, it computes and sets the third.
- *dropValue* propagates to all connected quantities.
- The initialization code connects the adder with the three passed quantities.

Exercise: Write a multiplier constraint. The constraint should "know" that $0 * x = 0$, even if x is undefined.

Here is the implementation of *getValue* and *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);
```

Method *setValue* signals an error if an attempt is made to change an already defined value.

Otherwise, it propagates the change along by calling *newValue* of all attached constraints except the informant.

Quantities

It remains to implement quantities.

The state of a quantity is given by three values:

- its current *value*,
- the *constraints* attached to it,
- the *informant*, i.e. the constraint that caused the last value to be set.

The informant acts like a trail that avoids infinite cyclic propagation.

```
class Quantity with {
  private var value: Option[Double] = None;
  private var constraints: List[Constraint] = [];
  private var informant: Constraint = NoConstraint;
```

Here is the implementation of *connect*.

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

The method adds the given constraint to the *constraints* list.

If the quantity has a value, it also calls *newValue* on the newly attached constraint.

Here is the implementation of *forgetValue* and *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);
```

Method *forgetValue* resets the value to undefined (*None*), but only if the call came from the same constraint that set the value.

It then propagates the change along by calling *dropValue* on all attached constraints except for *informant*.

A call of *forgetValue* by somebody other than *informant* is ignored.

A Refinement

The presented constraint system works OK, but constraints are tedious to define.

Compare the equation:

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

with the code that defines *CFconverter*.

Would it not be nice if we could build a constraint system from an equation like the one above?

We can get almost as far in Scala. Here's an alternative way to express the Celsius/Fahrenheits conversion.

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


Probes

Probes are special constraints that simply print all changes to the value of the attached quantity.

They are implemented as follows.

```
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;
}
```

Summary

We have learned a new paradigm of computation: computing by solving *relations* or *constraints*.

The distinguishing characteristic of this paradigm is that computation can flow in more than one direction, depending on what's defined and what is not.

The presented implementation is based on a network of constraints (nodes) and quantities (edges).

Solving constraints implies propagating changes in values along the edges and across the nodes.

The network is expressed as a set of objects some of which encapsulate state.

Here,

- * and + are new methods in class *Quantity* that take a quantity and return a new quantity attached to the corresponding constraint.
- *c* is a function that returns quantity attached to a constant constraint.
- === is a method in *Quantity* that takes a quantity and builds an equality constraint.

For instance, here is an implementation of the + method in class *Quantity*:

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