Week 9: Constraints

- Programs are generally organized as unidirectional computations that consume inputs and produce outputs.
- (Pure) functional programming makes this explicit in the source code, since we have:

  input = function argument   output = function result

- Mathematics, on the other hand, is not always unidirectional.
- For example, in the equation $d \cdot A \cdot E = F \cdot L$, we can calculate the value of any variable by using the values of the other four.
- For example,

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

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

A Language for Constraints

We now develop a *constraint language* that allows the user to formulate equations like this, and have the system solve them.

There are two levels:

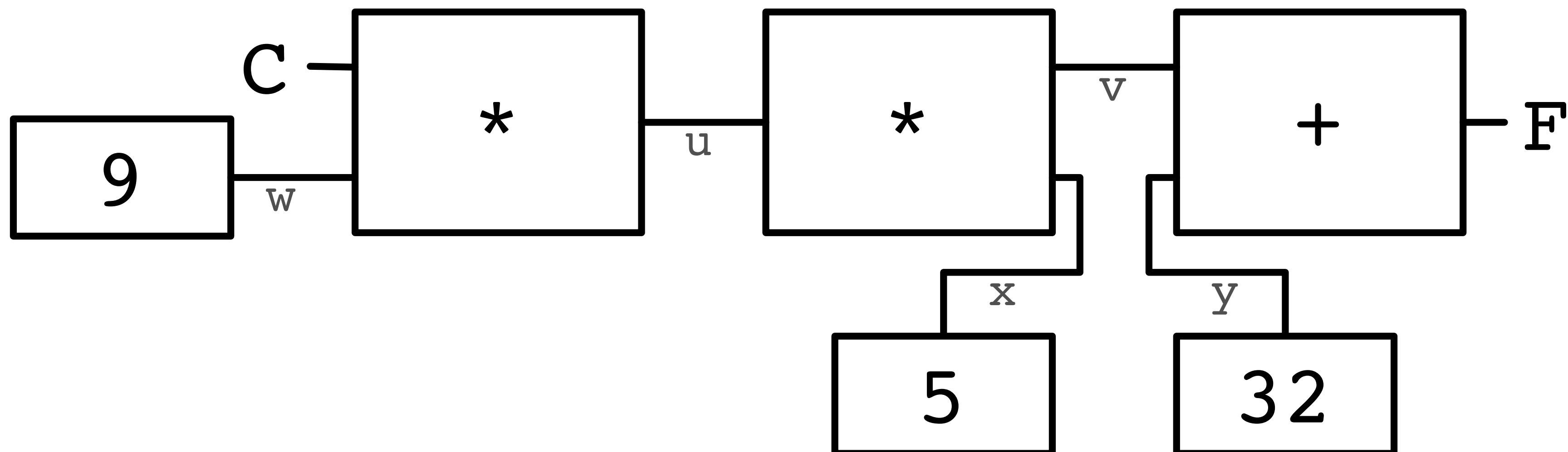
- Constraints like networks: primitive constraints linked by connectors.
- Constraints as algebraic equations.

Temperature Conversions

Exemple : The relationship between temperatures in Celsius and Fahrenheit is:

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

This can be expressed as a constraint network as follows:



Using the Constraint System

Suppose we want to convert between Celsius and Fahrenheit.

We create a converter by defining

val *C*, *F* = ***new*** *Quantity*
CFconverter(*C*, *F*)

Using the Converter

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

```
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)
}
```

By comparing with the graphical representation of the network, we find that:

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

To see the network running, start up the interpreter:

```
scala> :load week09.scala
defined module constr
scala> import constr._
import constr._
```

and place probes on the quantities C and F:

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

Then, give a value to one of the quantities:

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

Now try to give a value to F :

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

If we would like to reuse the system with new values, we must first “forget" the old values.

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

Note that the same network can be used to compute C from F and F from C.

This lack of direction is characteristic of systems based on *constraints*.

Such systems are common today; an entire industry is interested in them.

Examples: ILOG Solver (and JSolver), TK!solver.

Often, constraint systems *optimize* some quantities based on other quantities; but we'll not cover that here.

Implementing Constraint Systems

The implementation of a constraint system is somewhat similar to the implementation of a logical circuit simulator.

A constraint system is composed of primitive constraints (boxes) and of quantities (connectors).

Primitive constraints simulate simple equations between the quantities x, y, z, such that:

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

where c is a constant.

A quantity is either defined or undefined.

A quantity can connect any number of constraints.

Here is the interface of a quantity:

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

Explanation:

getValue returns the current value of the quantity.

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

These two methods exist in two overloaded variants.

One of the variants (used internally by the constraint system) passes the constraint that causes the modification or the reset of the parameter value.

connect declares that the quantity is involved in a constraint.

The *Option* Type

The *Option* type is defined as:

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

The idea is that the function *getValue* returns

- *None* if no value is specified, or,
- *Some*(*x*) if the value of the quantity is *x*.

The clients of *getValue* then use pattern matching to decompose the value:

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

Covariance

The definition of $Option$ illustrates several aspects of Scala's type system.

- The + before the type parameter a indicates that $Option$ is a covariant type constructor:

  If T is a subtype of S (note $T <: S$), then $Option[T]$ is a subtype of $Option[S]$.

  For example, $Option[String]$ is a subtype of $Option[Object]$.
- Without the + in the class definition of $Option$, $Option[String]$ and $Option[Object]$ will be two incomparable types.
- Question: Why can't class constructors always be covariant?

- *None* is defined as a *case object*. In other words, it's the only value that inherits from $Option[Nothing]$.
- The type *Nothing* is a subtype of every other type. For example, $Nothing <: String <: Object$.
- Since *Option* is covariant, this means that *None* is a value of any type of the form $Option[T]$. For example, $Option[Nothing] <: Option[String] <: Option[Object]$.

Constraints

The interface of a constraint is simple.

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

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

newValue is called when one of the quantities connected to a constraint receives a new value.

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

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

If this happens, it *propagates* these values by calling *setValue* for all the connected participants.

When it is woken up by a call to *dropValue*, a constraint simply tells all participants to forget their value.

We have therefore two sequences of mutually recursive calls.

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

Implementation of primitive constraints

Now it's easy to implement primitive constraints.

```
case class Adder(a1: Quantity, a2: Quantity, sum: Quantity)
                extends Constraint {
  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 {
    a1.forgetValue(this); a2.forgetValue(this); sum.forgetValue(this)
  }
  a1 connect this
  a2 connect this
  sum connect this
}
```

Explanations:

- *newValue* does a pattern match on the three quantities connected by the adder.
- If two of the values are defined, the third is computed and defined.
- *dropValue* is propagated to the connected quantities.
- The initialization code connects the adder to the three passed quantities.

Exercise: Write a multiplication constraint. The constraint should "know" that $0 * x = 0$, even if x is not defined.

Constants

A constant is a special case of a constraint.

We implement it as follows:

```
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)
}
```

Remarks:

- Constants cannot be redefined or forgotten. That's why *newValue* and *dropValue* produce an error.
- Constants immediately give a value to the attached quantity.

Quantities

We still have to implement the quantities.

The state of a quantity is given by three values:

- its current value (*value*),
- the constraints that are attached to it (*constraints*),
- the informant, i.e., the constraint that has caused the latest definition of the value (*informant*).

The informant can prevent the infinite propagation of values in the presence of cycles.

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

This is how *getValue* and *setValue* are implemented:

```
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 (c <- constraints if c != informant) c.newValue
}
def setValue(v: Double): Unit = setValue(v, NoConstraint)
```

The method *setValue* signals an error when one tries to modify a value that is already defined.

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

This is how *forgetValue* and *connect* are implemented:

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

The method *forgetValue* forgets the value (by resetting it to *None*) only if the call comes from the constraint that the value originated from.

It then propagates the modification by calling *dropValue* on all the attached constraints, except the informant.

A call to *forgetValue* coming from somewhere else than the informant is ignored.

Here is the implementation of *connect*.

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

This method adds the constraint to the list *constraints*.

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

Probes

Probes are special constraints that simply output all the changes of the attached quantity.

They are implemented as follows:

```
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]) {
    val vstr = v match {
      case Some(x) => x.toString()
      case None => "?"
    }
    println("Probe: " + name + " = " + vstr)
  }
  q connect this
}
```

Improvement

The presented system works, but the constraints remain tedious to define.

Compare the equation:

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

to the code that defines $CFconverter$.

Wouldn't it be nice to be able to build a constraint system directly based on an equation such as the one above?

We can almost do this in Scala. Here is a new way of expressing the Celsius/Fahrenheit conversion:

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

Here,

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

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

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

Summary

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

The main feature of this paradigm is that the computation can take place in more than one direction, depending on what is defined and what is not.

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

Constraint resolution involves the propagation of changes of values along the edges and across the nodes.

The network is modeled by a set of objects some of which contain a state.