Constraint Programming

Functional programming describes computation using **functions**

$$f : A \rightarrow B$$

Computation proceeds in **one direction**: from inputs (A) to outputs (B)

e.g. $f(x) = x / 2$ is a function that maps x to x/2

A more general view, **constraint programming**, works with **relations**

$$r \subseteq A \times B$$

which state only constraints between the quantities

e.g. $y + y = x$ is constraint between x and y

We can interpret a constraint as either mapping x to y, or mapping y to x

We discuss how to do constraint programming in a functional language

Why this is interesting?

First part: constraint propagation networks

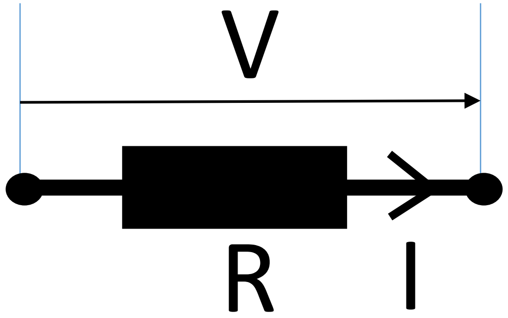
- Similar to circuit simulation
- It builds further on the idea of discrete event simulation: re-compute only what is needed
- Useful patterns, often used in user interface design

Second part: SAT solvers

- a method to check satisfiability of propositional formulas
- many useful problems can be reduced to SAT

Example: Ohm's Law

$$I \cdot R = V$$



where the meaning of symbols is:

- I - current flowing through the wire
- V - the voltage between endpoints of the wire
- R - resistance of the wire

The constraint applies regardless which ones of V, I, R are known

It encodes several functions, depending on what is known:

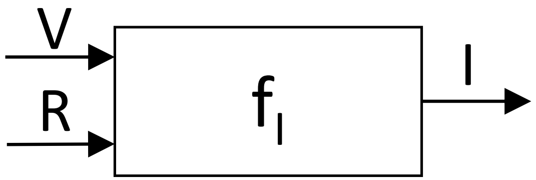
$f_I(V,R) = V/R$

$f_V(I,R) = I \cdot R$

$f_R(V,I) = V / I$

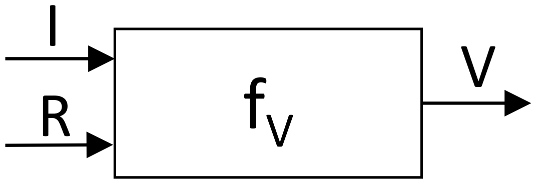
Schematic Display of Constraints vs Functions

$$f_I(V,R) = V/R$$



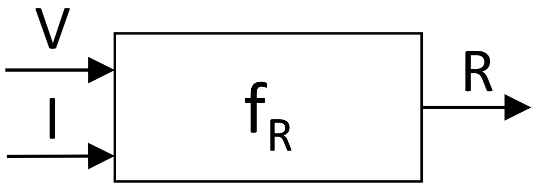

```
def fI(v:Double,r:Double) =
        v/r
```

$$f_V(I,R) = I \cdot R$$



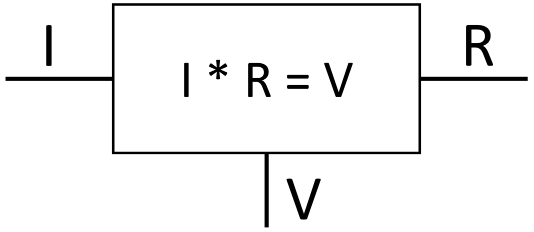

```
def fV(i:Double,r:Double) =
        i*r
```

$$f_R(V,I) = V \,/\, I$$




```
def fR(v:Double,i:Double) =
        v/i
```

$$I \cdot R = V$$



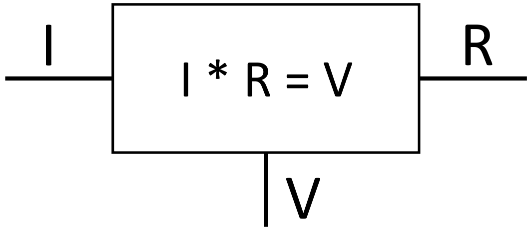
today:

Multiplier(I, R, V)

replaces all of above

How to Use Constraints

$I \cdot R = V$



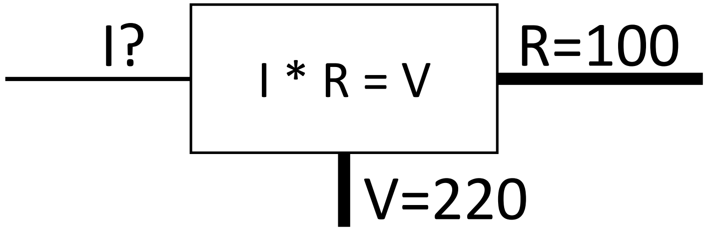
Multiplier(I, R, V)

We define Multiplier as a class whose construction establishes the constraint.

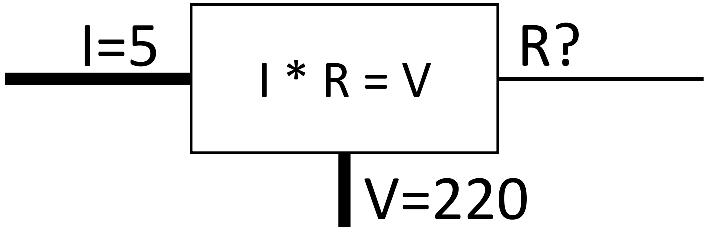
To allow dynamically setting which one of I,R,V is known, we define I,R,V as **Quantities**: objects that model the variables that participate in constraints

```
val I, R, V = new Quantity
Multiplier(I, R, V)
```

```
V setValue 220
R setValue 100
I getValue ➔ Some(2.2)
```




```
R forgetValue
I setValue 5
R getValue ➔ Some(44)
```



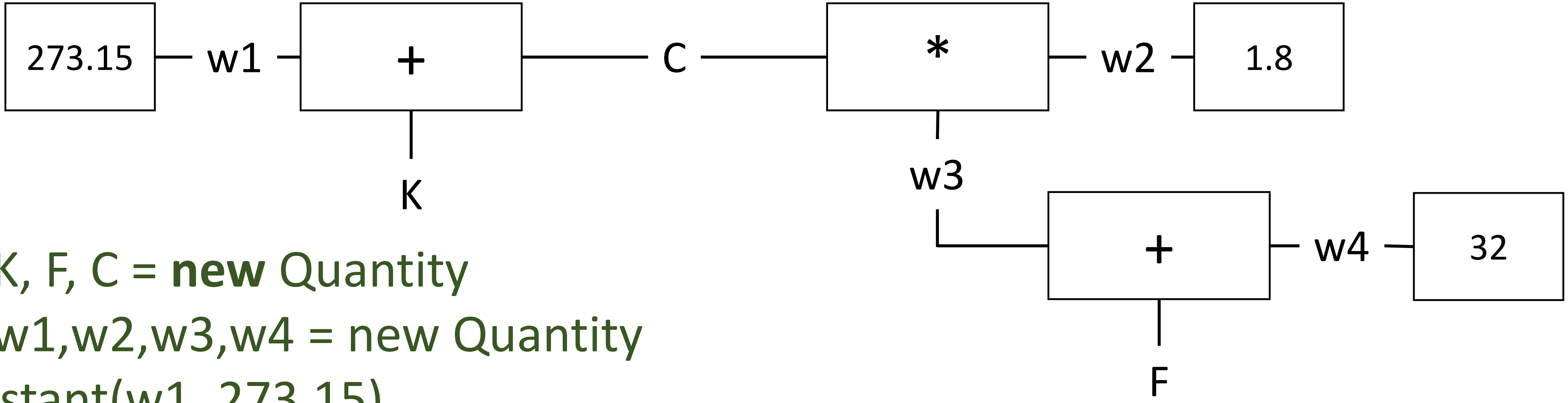
Connecting Constraints: Temperature Converter

K = C + 273.15

F = C * 1.8 + 32

each variable determines the other two

K – Kelvin, C – Celsius, F – Fahrenheit



val K, F, C = **new** Quantity

val w1,w2,w3,w4 = new Quantity

Constant(w1, 273.15)

Adder(w1, C, K)

Constant(w2, 1.8); Multiplier(C,w2,w3)

Adder(w3, w4, F); Constant(w4, 32)

F setValue 451

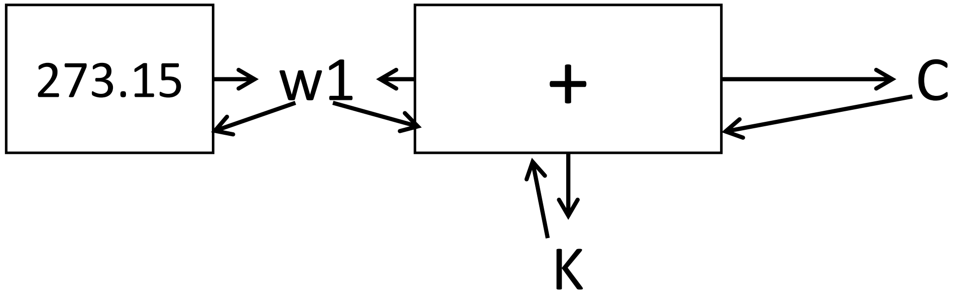
K getValue ➔ Some(505.93)

Quantities and Constraints are Doubly-Linked Objects

Quantities optionally store a value, if known:

```
class Quantity {
  private var value: Option[Double] = None
  def getValue: Option[Double] = value
  def setValue(v: Double) = setValue(NoConstr)
  def setValue(v : Double, setter : Constraint)
  def forgetValue = forgetValue(NoConstr)
  def forgetValue(retractor : Constraint)
  private var constraints: List[Constraint] = List()
  def connect(c : Constraint) }
```

Value can be set by a constraint, or by setValue



val K, C, w1 = **new** Quantity

Quantities start unconstrained

They can be **connect**ed to any number of **constraints**

Constraints create rules to set some quantities if others change.

Constant(w1, 273.15)

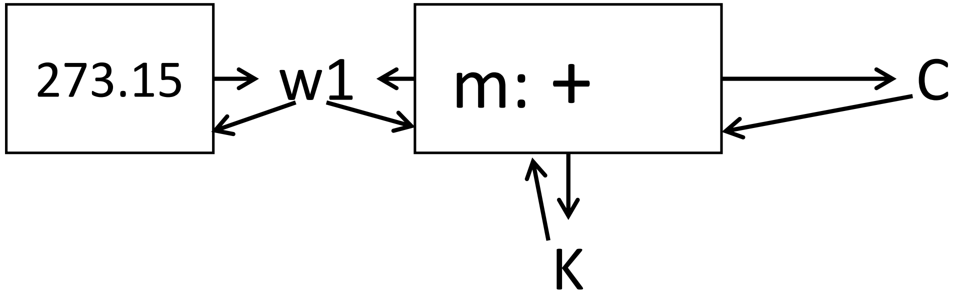
Keeps w1 set to 273.15

Adder(w1, C, K)

If two quantities are known, sets the third.

A Constraint Can Update and Reset Quantities

```
abstract class Constraint {
  // subclasses have fields pointing to Quantities of the constraint
  def newValue: Unit  // rules to compute quantities from known ones
  def dropValue: Unit  // forgetValue-s all quantities of the constraint
}
```



C setValue 100

→ m.newValue

→ K.setValue(373.15, m)

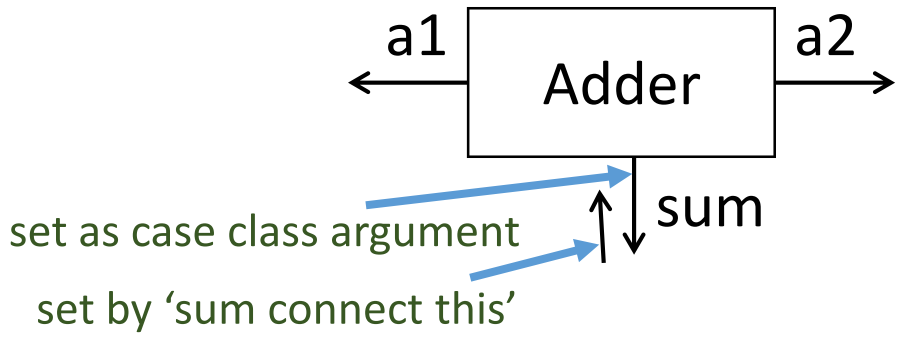
C forgetValue

→ m.dropValue

→ K.forgetValue(m)

Implementation of the Adder Constraint

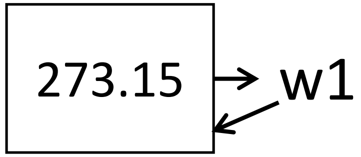
```
case class Adder(a1: Quantity, a2: Quantity, sum: Quantity) extends Constraint {
  def newValue = (a1.getValue, a2.getValue, sum.getValue) match {
    case (Some(x1), Some(x2), _) => sum.setValue(x1 + x2, this)
    case (Some(x1), _, Some(r))  => a2.setValue(r - x1, this)
    case (_, Some(x2), Some(r))  => a1.setValue(r - x2, this)
    case _ =>
  }
  def dropValue {
    // quantitities ignore irrelevant forgetValue calls, so we can just call it on all of them
    a1.forgetValue(this); a2.forgetValue(this); sum.forgetValue(this)
  }
  a1 connect this            // tell each quantity to add a back link to us
  a2 connect this
  sum connect this
}
```



Implementation of the Multiplier Constraint

```
case class Multiplier(a1: Quantity, a2: Quantity, prod: Quantity) extends
Constraint {
  def newValue = (a1.getValue, a2.getValue, prod.getValue) match {
    case (Some(0), _, _) => prod.setValue(0, this)
    case (_, Some(0), _) => prod.setValue(0, this)
    case (Some(x1), Some(x2), _) => prod.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 _ =>
  }
```

Constant Constraint



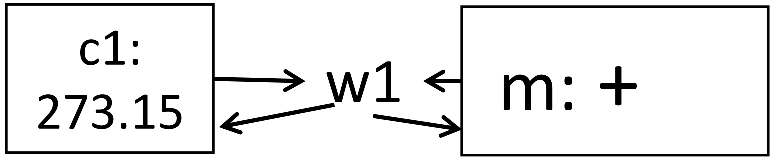
Constant(w1, 273.15)

```
case class Constant(q: Quantity, v: Double) extends Constraint {
  def newValue: Unit = ???
  def dropValue: Unit = ???
  q connect this
  q.setValue(v, this)
}
```

- Constants cannot be redefined or forgotten
- That's why \`newValue\` and \`dropValue\` produce an error – w1 will never call them
- Constants immediately give a value to the attached quantity.

More on Quantities: Summary of Their Fields

```
class Quantity {
  private var value: Option[Double] = None
  private var constraints: List[Constraint] = List()
  private var informant: Constraint = NoConstr;   ... }
object NoConstr extends Constraint {  } // not an actual constraint
```




```
w1:    value = Some(273.15)
       constraints = List(c1, m)
       informant = c1
```

More on Quantities: setValue

```
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
}
```

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 that called it

It remembers the informant, so it knows who is responsible for the value

More on Quantities: forgetValue

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

Forgets the value (by resetting it to `None`) only if the call comes from the constraint that the value originated from

It 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

More on Quantities: connect

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

Adds the constraint to the list `constraints`

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

Callbacks to Monitor Changes

What if we want to take an action when some quantity gets a new value?

We could keep traversing all quantities, but that is inefficient and unnecessary

```
case class Notification(q: Quantity,  action : Option[Double] => Unit)
        extends Constraint {
  def newValue: Unit = action(q.getValue)
  def dropValue: Unit = action(None)
  q connect this
}
```

Example: print quantity C when it changes: Notification(C, println(_))

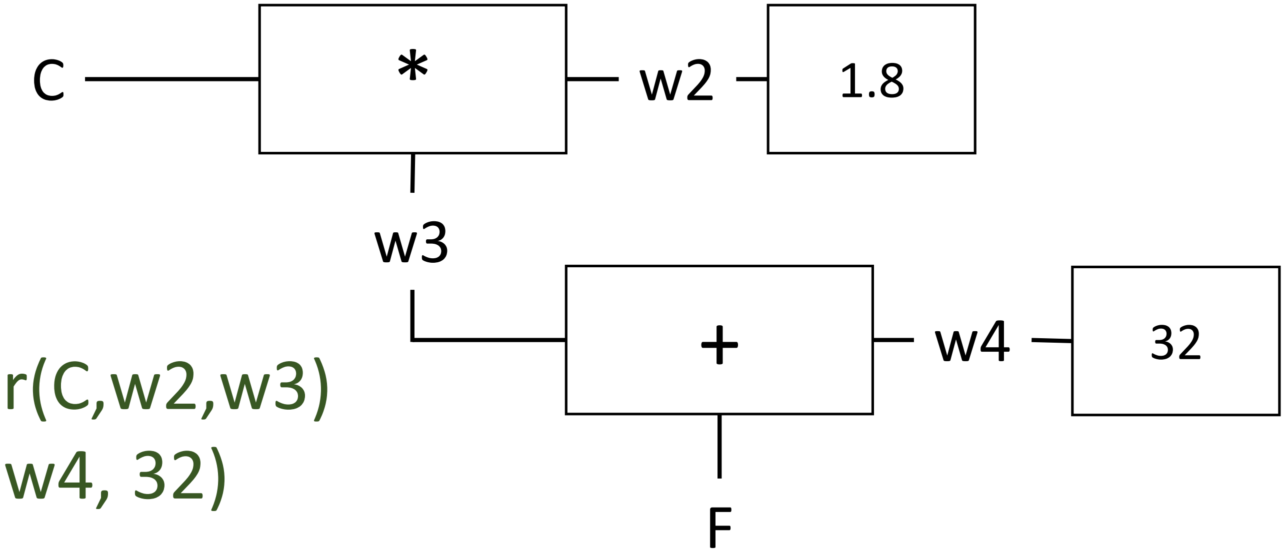
Notation: Constraints vs Math

math:

F = C * 1.8 + 32

Scala:

```
Constant(w2, 1.8); Multiplier(C,w2,w3)
Adder(w3, w4, F); Constant(w4, 32)
```



Can we make our Scala code more like math?

Yes, it is possible to write e.g.

F === (C * k(1.8)) + k(32)

How?

F === (C * k(1.8)) + k(32)

Notation: Constraints vs Math

Introduce additional binary methods on quantities:

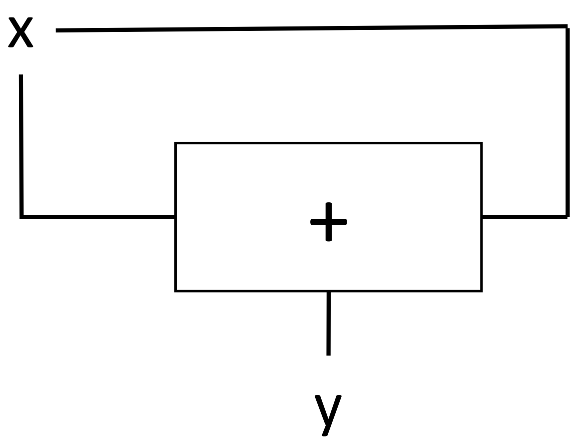
```
class Quantity { …
  def +(that: Quantity): Quantity = {
    val sum = new Quantity
    Adder(this, that, sum)
    sum
  }
  def *(that: Quantity): Quantity = {
    val product = new Quantity
    Multiplier(this, that, product)
    product
  }
  def ===(that: Quantity): Unit =
     Equalizer(this, that)
}
```

```
def k(x : Double) : Quantity = {
  val qx = new Quantity
  Constant(qx, x); qx
}
case class Equalizer(left: Quantity, right: Quantity)
              extends Constraint {
  def newValue = (left.getValue, right.getValue) match {
    case (Some(l), _) => right.setValue(l)
    case (_, Some(r))  => left.setValue(r)
     case _ =>
  }
  def dropValue {
    left.forgetValue(this); right.forgetValue(this)
 }
  left connect this
  right connect this
}
```

Remarks on Constraint Propagation Networks

- They work well and are fast when constraints have structure of **trees**
- Current implementation does not make it easy to remove constraints or add them only temporarily and revert to previous state
- Propagation is limited as a solving technique: it does not produce results for directed acyclic graphs with sharing, even if they are known:

    Adder(x, x, y); y.setValue(10); x.getValue → None



How to Solve More General Constraints?

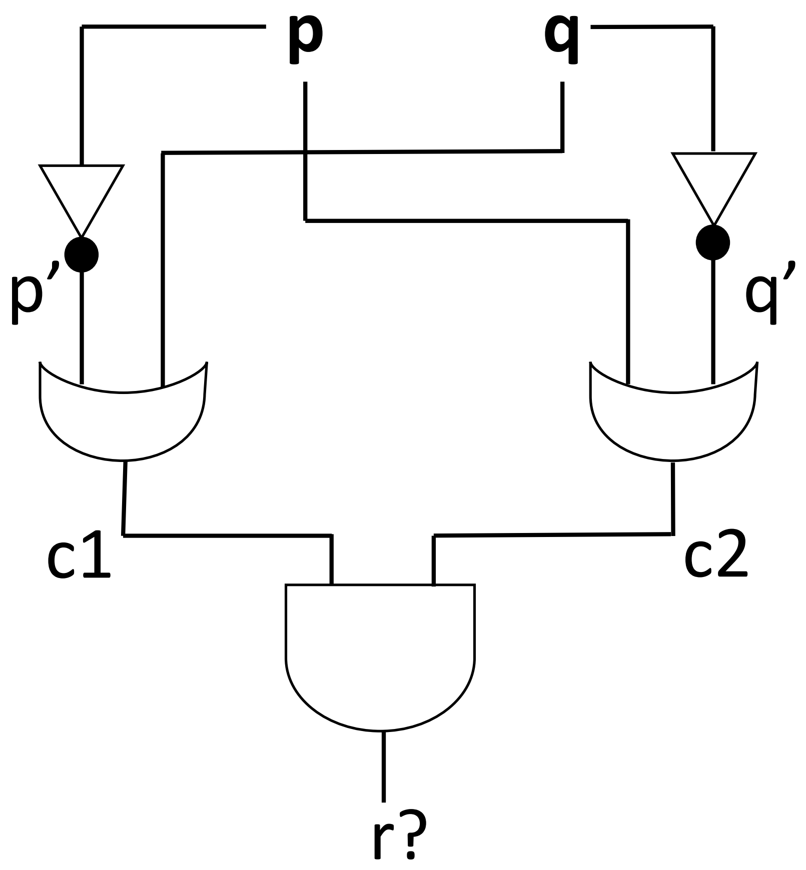
Depends on the type of Quantities and types of Constraints we have

- Double-s with approximate precision: numerical analysis techniques (e.g. iterative solvers for non-linear equations, Newton's method, …)
- rational numbers with only Adders and constants: Gaussian elimination
- BigInts where with only Adders and constants: solving Diophantine equations
- BigInts with Adders, Multipliers, constants: there exists no general algorithm (Hilbert's 10^{th} problem, final step in 1970ies shown by Matiyasevich)

- For finite domains: we could try all possibilities, but in practice we use combinatorial search teachnique, often **Satisfiability (SAT) Solvers**
- Solutions are not always unique, we are interested in any solution

Combinational Circuits as Constraint Networks

Propagation values from inputs to outputs in a DAG is evaluation and works



p setValue 1

q setValue 0

Propagation can compute:

→p’ setValue 0

→q’ setValue 1

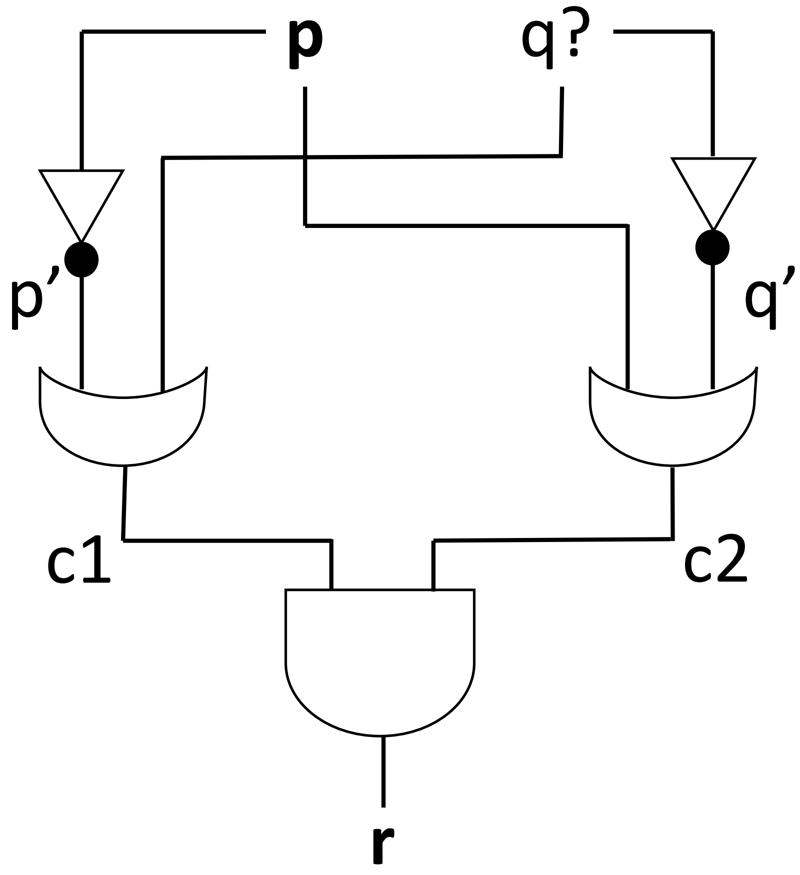
→c1 setValue 0

→c2 setValue 1

→r setValue 0

Computing Backwards Can Also Work Sometimes

What if instead we set p and r and ask for q?



p setValue 1

r setValue 0

Propagation can compute:

→p’ setValue 0 (inverter)

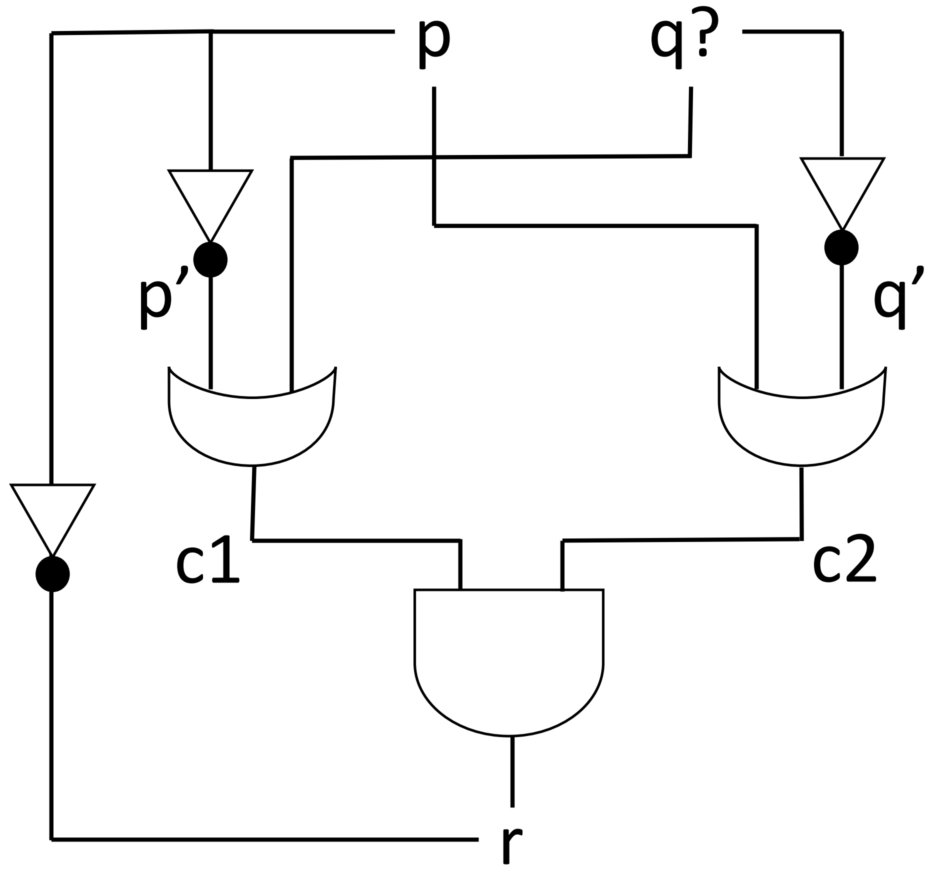
→c2 setValue 1 (or)

→c1 setValue 0 (and)

→q setValue 0 (or)

Propagation Alone is Not Sufficient

What if instead we just ask for p to be inverse of r? What is the value of q?



Nothing is set, nothing propagates

So we must speculate (“decide”)

q setValue 1
→ q’ setValue 0
→ c1 setValue 1

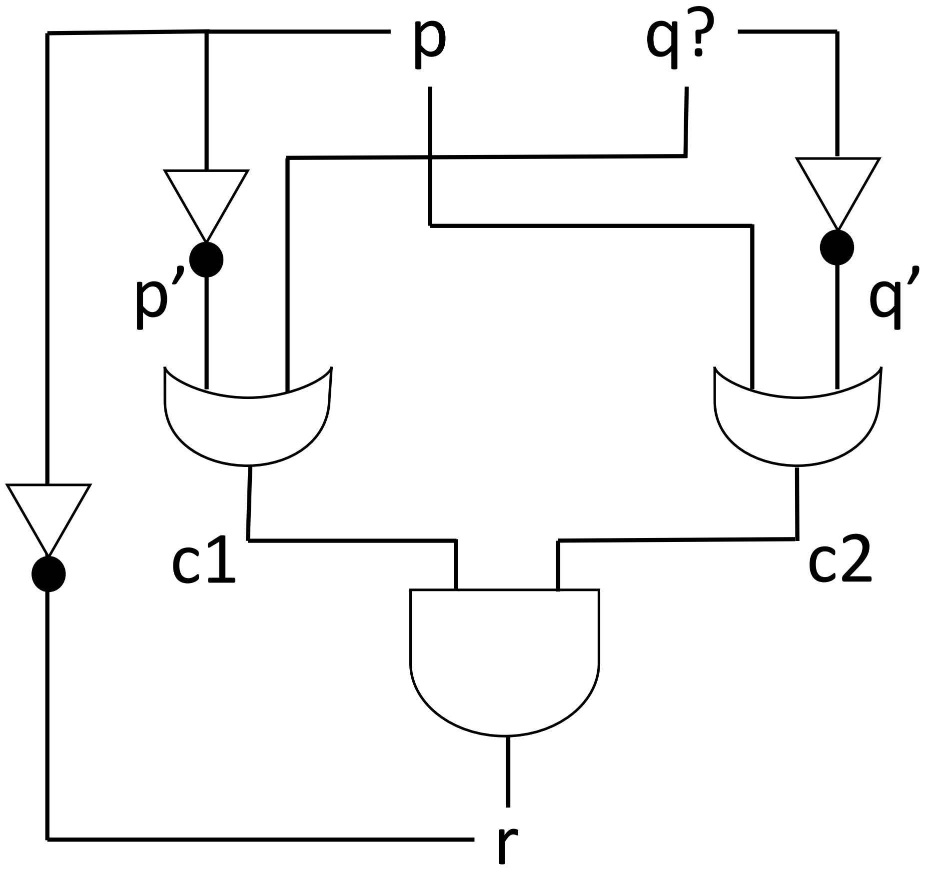
Nothing more propagates. But it does not mean that q is the right value. We need to find examples of p,r

p setValue 1
→ r setValue 0
→ p’ setValue 1
→ c2 setValue 1
→ r setValue 1
CONFLICT

p setValue 0
→ r setValue 1
→ p’ setValue 1
→ c2 setValue 0
→ r setValue 0
CONFLICT

Chosen Value of q Was Wrong. Had to try p to see that

What if instead we just ask for p to be inverse of r? What is the value of q?



Nothing is set, nothing propagates

So we must speculate (“decide”)

q setValue 1 ← wrong decision

→ q’ setValue 0

→ c1 setValue 1

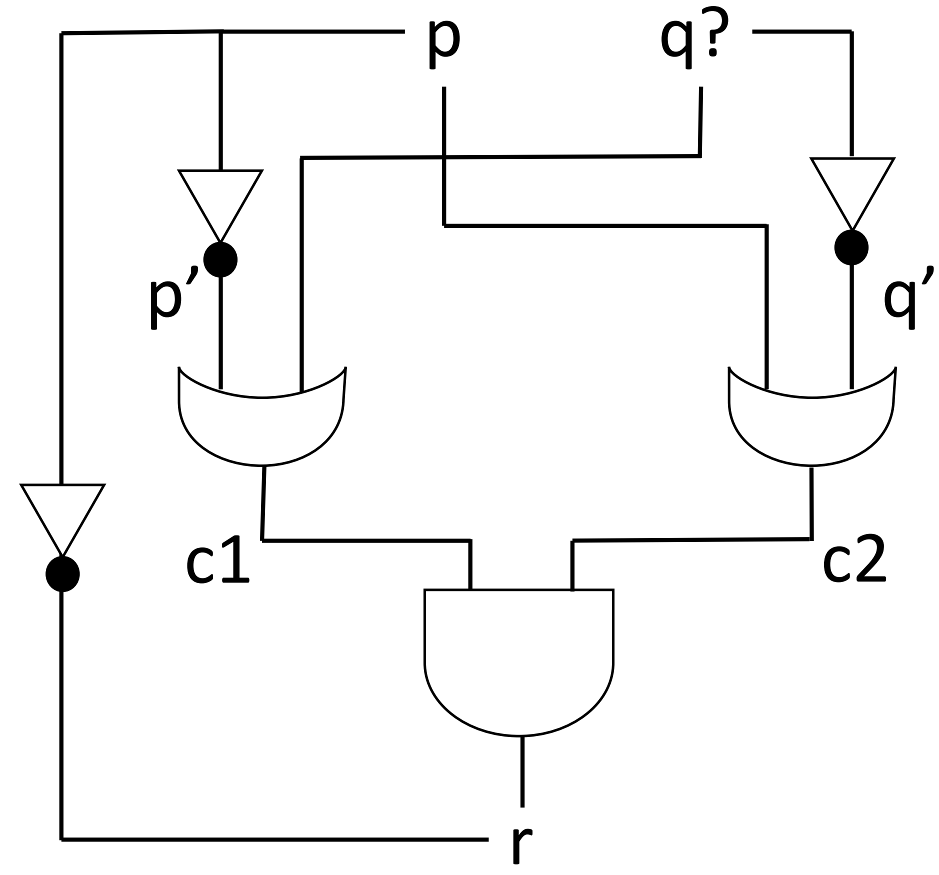
Nothing more propagates. But it does not mean that q is the right value. We need to find examples of p,r

p setValue 1
→ r setValue 0
→ p’ setValue 1
→ c2 setValue 1
→ r setValue 1
CONFLICT

p setValue 0
→ r setValue 1
→ p’ setValue 1
→ c2 setValue 0
→ r setValue 0
CONFLICT

Search Tree

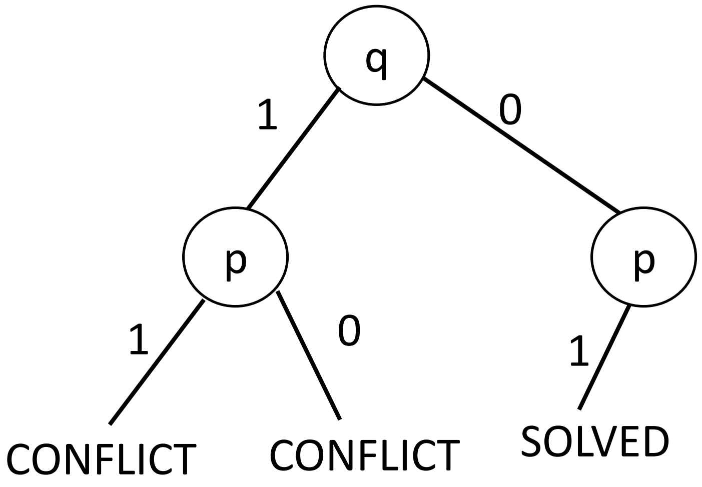
What if instead we just ask for p to be inverse of r? What is the value of q?



Nothing is set, nothing propagates

The only alternative (if there is any):

q setValue 0

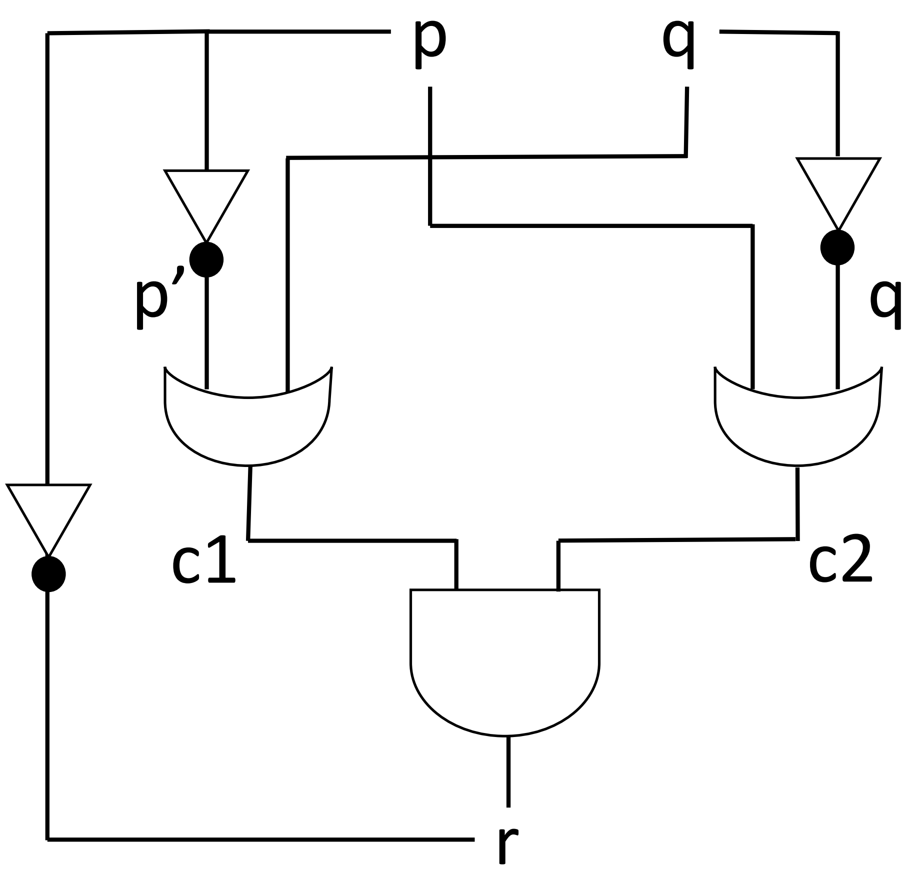


→ q’ : 1
→ c2: 1
Decide p: 1
→ p’:0
→ c1:0
→ r:0
All values assigned!
All constraints true!

We found a solution q:0, p:1, r:0

SAT Solver

Given an arbitrary circuit, a **SAT solver** needs to answer one of these two:

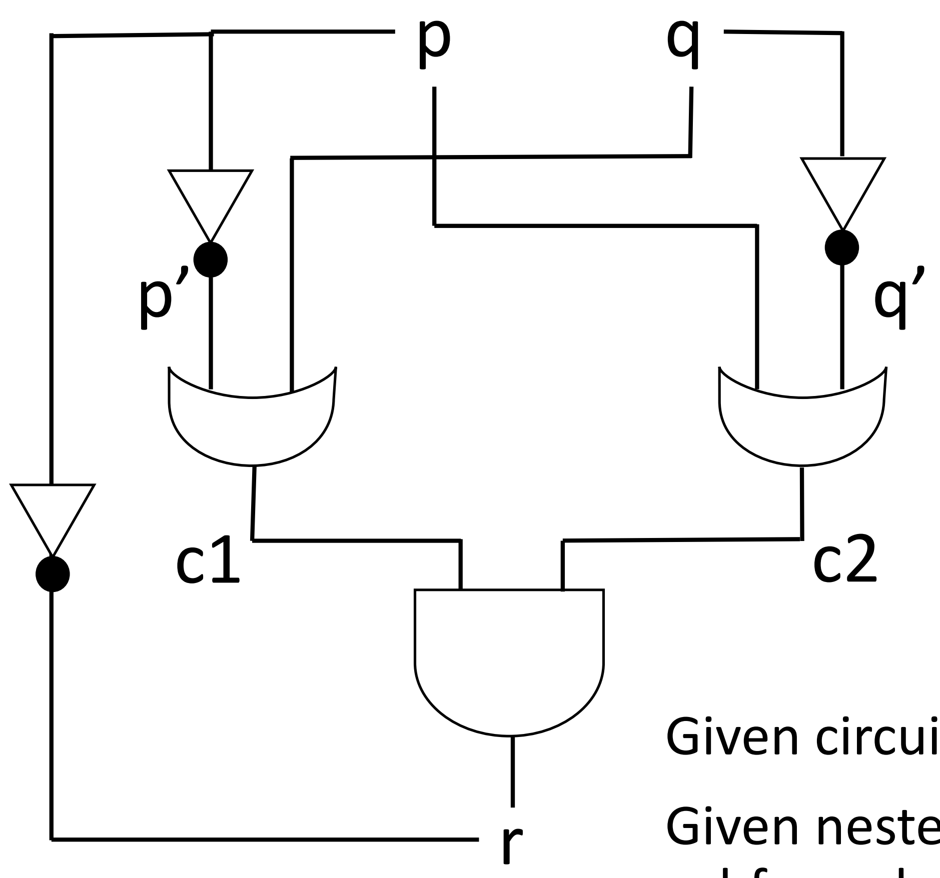


1. SAT: Gives back a satisfying assignment of 0/1 to all Boolean quantities such that all constraints hold
2. UNSAT: Says “there are no satisfying assignments”

Techniques in SAT solvers:

- constraint propagation: always deduce consequences of current decisions; use efficient data structures such as “2-watched literals scheme”
- backtracking search: always maintain candidate partial solution and update it
- clause learning (CDCL): deduce new clause representing minimal reason for a conflict
- heuristics on which variable to decide
- restarts if no progress after some time

SAT for Circuits = SAT for Propositional Formulas



r == !p &&
p' == !p &&
c1 == p' || q &&
q' == !q &&
c2 == p || q' &&
r == c1 && c2

Given circuit: write each constraint, conjoin them all

Given nested formula: introduce fresh variables to denote subformulas, express operations using &&,||,! - obtain a circuit.

SAT for Propositional Formulas = SAT for CNF

CNF = Conjunctive Normal Form, formula is conjunction of **clauses**

A **clause** is a disjunction of **literals**

A **literal** is a propositional variable or its negation

Converting to CNF: L==R becomes (!L || R) && (L || !R)

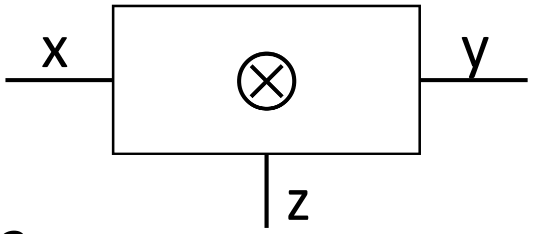
```
r == !p          &&
p’ == !p         &&
c1 == p’ || q    &&
q’ == !q         &&
c2 == p || q’    &&
r == c1 && c2
```

```
(!r || !p) && (r || p) &&
(!p’ || !p) && (p’ || p) &&
(!c1 || p’ || q) && (c1 || !p’) && (c1 || !q) &&
(!q’ || !q) && (q’ || q) &&
(!c2 || p || q’) && (c2 || !p) && (c2 || !q’) &&
(!r || c1) && (!r || c2) && (r || !c1 || !c2)
```

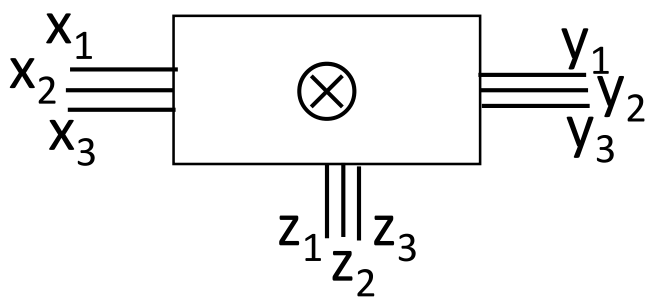
“DIMACS format for CNF formulas”

Encoding Constraints on Finite Domains

$D = \{a_0, a_2, \ldots, a_{N-1}\}$ arbitrary finite set of N elements.
Let $\otimes \subseteq D \times D \times D$ be constraint on D
Let M be such that $2^M \geq N$
Using M bits we can represent all numbers 1,…,N
Represent a_k using binary representation of number k, e.g., for N=6, M=3



a_0	000
a_1	001
a_2	010
a_3	011
a_4	100
a_5	101



Represent each finite-domain variable using M Boolean variables
Represent the constraint $\otimes$ using a circuit

Applications of SAT

- Scheduling problems
- Checking plugin dependencies
- Checking pattern matching in Scala compiler
- Checking correctness of microprocessors before they are fabricated (verification tools of companies such as Cadence, Synopsys, Mentor Graphics)
- Solving puzzles: e.g. Sudoku solver
- Solving planning problems in AI: find a sequence of actions to meet the goal

…

In your homework you will be using a SAT solver written in Scala

What if domains are not finite?

We can use extension of SAT called SMT

SMT = Satisfiability Modulo Theories

Constraints involving not only Booleans but also a mixture of

- linear integer/real constraints (Simplex algorithm, branch and cut, …)
- fixed-width integers
- arrays, sequences/strings
- finite trees (unification constraints) – we will see them in later lectures

A convenient way to use Z3 SMT solver in Scala:

http://lara.epfl.ch/w/ScalaZ3

An open-source SMT solver CVC4: http://cvc4.cs.nyu.edu/web/

Conclusions

- Constraint propagation networks can solve certain classes of constraints
- They are graphs consisting of quantity objects and constraint objects
- Propagation of quantities is done by setting and invalidating quantities according to the meaning of local constraints

- More complex constraints require search or algebraic reasoning
- It is worth considering to use an outside solver and express your problem in its constraint language
- An important class of constraint solvers are SAT solvers, which we can use to solve constraints over finite domains
- SAT solvers perform propagation but also search and have many heuristics