CafeSat: A Modern SAT Solver for Scala

Régis Blanc

Laboratory for Automated Reasoning and Analysis
École Polytechnique Fédérale de Lausanne

June 2, 2013

SAT Solvers

CafeSat is a SAT solver, still in development, written in Scala.

Boolean Satisfiability Problem (known as SAT):

- First known example of an NP-complete problem.
- Problem Statement:
  Given a propositional formula, is there an assignment of variables that makes it true ?

$$a \wedge (\neg a \vee (b \wedge a))$$

- Hard problem (NP-complete) but fast algorithms in practice.
- Annual conference (SAT) and bi-annual competition.
- Many existing solvers:
  MiniSAT, Grasp, ZChaff, Sat4j (JVM), ...

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Demo

Demo

`file.cnf`:

```
...
330 -355 0
330 -23 0
330 -22 0
330 -21 0
330 20 0
330 19 0
330 -18 0
330 -375 0
351 352 353 354 355 23 22 21 -20 -19 18 375 -330 0
...
```

Demo

`file.cnf`:

```
...
330 -355 0
330 -23 0
330 -22 0
330 -21 0
330 20 0
330 19 0
330 -18 0
330 -375 0
351 352 353 354 355 23 22 21 -20 -19 18 375 -330 0
...
```

```
reg@reg-laptop:~/vcs/scabolic (master) $ ./cafesat file.cnf
```

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Demo

`file.cnf`:

```
...
330 -355 0
330 -23 0
330 -22 0
330 -21 0
330 20 0
330 19 0
330 -18 0
330 -375 0
351 352 353 354 355 23 22 21 -20 -19 18 375 -330 0
...
```

```
reg@reg-laptop:~/vcs/scabolic (master) $ ./cafesat file.cnf
sat
reg@reg-laptop:~/vcs/scabolic (master) $
```

How to Use CafeSat ?

How to Use CafeSat ?

```
reg@reg-laptop:~/vcs/scabolic (master) $ ./cafesat file.cnf
```

How to Use CafeSat ?

```
reg@reg-laptop:~/vcs/scabolic (master) $ ./cafesat file.cnf
```

A constraint programming library:

- ASTs for formulas: `And`, `Not`, `Var`, ...
- Classic boolean operators: &&, ||, !
- Scala maps for representing models.
- Can mix with the expressiveness of Scala.

(Image: courtesy of Creative Commons)

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

An Example

Concise and fast sudoku solver:

```
def solve(sudoku: Array[Array[Option[Int]]]) = {
  val vars = sudoku.map(_.map(_ => Array.fill(9)(boolVar())))
  val onePerEntry = vars.flatMap(row => row.map(vs => Or(vs:_*)))
  val uniqueInColumns = for(c <- 0 to 8; k <- 0 to 8; r1 <- 0 to 7; r2 <- r1+1 to 8)
    yield !vars(r1)(c)(k) || !vars(r2)(c)(k)
  val uniqueInRows = for(r <- 0 to 8; k <- 0 to 8; c1 <- 0 to 7; c2 <- c1+1 to 8)
    yield !vars(r)(c1)(k) || !vars(r)(c2)(k)
  val uniqueInGrid1 =
    for(k <- 0 to 8; i <- 0 to 2; j <- 0 to 2; r <- 0 to 2; c1 <- 0 to 1; c2 <- c1+1 to 2)
      yield !vars(3*i + r)(3*j + c1)(k) || !vars(3*i + r)(3*j + c2)(k)
  val uniqueInGrid2 =
    for(k <- 0 to 8; i <- 0 to 2; j <- 0 to 2; r1 <- 0 to 2;
        c1 <- 0 to 2; c2 <- 0 to 2; r2 <- r1+1 to 2)
      yield !vars(3*i + r1)(3*j + c1)(k) || !vars(3*i + r2)(3*j + c2)(k)
  val forcedEntries =
    for(r <- 0 to 8; c <- 0 to 8 if sudoku(r)(c) != None)
      yield Or(vars(r)(c)(sudoku(r)(c).get - 1))
  val allConstraints =
    onePerEntry ++ uniqueInColumns ++ uniqueInRows ++ uniqueInGrid1 ++ uniqueInGrid2 ++ forcedEntries
  solve(And(allConstraints:_*))
}
```

A naive backtracking search is slow; a fully optimized search with deduction is large and complex to write.

A SAT Solver in Scala (Almost) Fits on One Slide

```scala
sealed trait Formula {
  def subst(x: Var, v: Boolean)
  def vars: Set[Var]
  def eval: Boolean //if no free variable
}
case class And(f1: Formula, f2: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class Var(id: Int) extends Formula
case class Lit(b: Boolean) extends Formula

def isSat(f: Formula): Boolean = {
  val vs = f.vars
  if(vs.isEmpty)
    f.eval
  else {
    val v = vs.head
    isSat(f.subst(v, true)) || isSat(f.subst(v, false)) }
}
```

A SAT Solver in Scala (Almost) Fits on One Slide

```scala
sealed trait Formula {
  def subst(x: Var, v: Boolean)
  def vars: Set[Var]
  def eval: Boolean //if no free variable
}
case class And(f1: Formula, f2: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class Var(id: Int) extends Formula
case class Lit(b: Boolean) extends Formula

def isSat(f: Formula): Boolean = {
  val vs = f.vars
  if(vs.isEmpty)
    f.eval
  else {
    val v = vs.head
    isSat(f.subst(v, true)) || isSat(f.subst(v, false)) }
}
```

Is it correct?

A SAT Solver in Scala (Almost) Fits on One Slide

```scala
sealed trait Formula {
  def subst(x: Var, v: Boolean)
  def vars: Set[Var]
  def eval: Boolean //if no free variable
}
case class And(f1: Formula, f2: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class Var(id: Int) extends Formula
case class Lit(b: Boolean) extends Formula

def isSat(f: Formula): Boolean = {
  val vs = f.vars
  if(vs.isEmpty)
    f.eval
  else {
    val v = vs.head
    isSat(f.subst(v, true)) || isSat(f.subst(v, false)) }
}
```

Is it correct? Yes!

Is it fast?

A SAT Solver in Scala (Almost) Fits on One Slide

```
sealed trait Formula {
  def subst(x: Var, v: Boolean)
  def vars: Set[Var]
  def eval: Boolean //if no free variable
}
case class And(f1: Formula, f2: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class Var(id: Int) extends Formula
case class Lit(b: Boolean) extends Formula

def isSat(f: Formula): Boolean = {
  val vs = f.vars
  if(vs.isEmpty)
    f.eval
  else {
    val v = vs.head
    isSat(f.subst(v, true)) || isSat(f.subst(v, false)) }
}
```

Is it correct? Yes!

Is it fast? Not so much

How slow?

A SAT Solver in Scala (Almost) Fits on One Slide

```scala
sealed trait Formula {
  def subst(x: Var, v: Boolean)
  def vars: Set[Var]
  def eval: Boolean //if no free variable
}
case class And(f1: Formula, f2: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class Var(id: Int) extends Formula
case class Lit(b: Boolean) extends Formula

def isSat(f: Formula): Boolean = {
  val vs = f.vars
  if(vs.isEmpty)
    f.eval
  else {
    val v = vs.head
    isSat(f.subst(v, true)) || isSat(f.subst(v, false)) }
}
```

Is it correct? Yes!

Is it fast? Not so much

How slow? $> 30s$ on 20 variables, timeout $5m$ on 50 variables

A Better Approach

Previous approach was brute-force. Need to be smarter.
Use Conjunctive Normal Form (CNF) instead of arbitrary formula:

$$(a \vee b) \wedge (a \vee \neg b \vee c) \wedge (b \vee c)$$

A Better Approach

Previous approach was brute-force. Need to be smarter.
Use Conjunctive Normal Form (CNF) instead of arbitrary formula:

$$(a \vee b) \wedge (a \vee \neg b \vee c) \wedge (b \vee c)$$

DPLL algorithm: a backtracking search.

1. Pick an unassigned variable
2. Force unit clause (BCP)
3. Backtrack if conflict, go to 1.

Straighforward implementation in Scala:
How fast?

A Better Approach

Previous approach was brute-force. Need to be smarter.
Use Conjunctive Normal Form (CNF) instead of arbitrary formula:

$$(a \vee b) \wedge (a \vee \neg b \vee c) \wedge (b \vee c)$$

DPLL algorithm: a backtracking search.

1. Pick an unassigned variable
2. Force unit clause (BCP)
3. Backtrack if conflict, go to 1.

Straighforward implementation in Scala:
How fast? 0.17s on 50 variables, $\sim$30s on 100 variables

Key Technology in CafeSat

- VSIDS decision heuristic,
- Two-watched literals for unit propagation (BCP),
- 1UIP clause learning,
- Clause deletion based on activity.

Implementation not so straightforward:

```
var decisionLevel = 0
var trail: FixedIntStack = null
var qHead = 0
var reasons: Array[Clause] = null
var levels: Array[Int] = null
var conflict: Clause = null
var model: Array[Int] = null
var watched: Array[ClauseList] = null
var seen: Array[Boolean] = null
var cnfFormula: CNFFormula = null
var status: Status = Unknown
var restartInterval = Settings.restartInterval
var nextRestart = restartInterval
val restartFactor = Settings.restartFactor
```

How fast?

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Key Technology in CafeSat

- VSIDS decision heuristic,
- Two-watched literals for unit propagation (BCP),
- 1UIP clause learning,
- Clause deletion based on activity.

Implementation not so straightforward:

```
var decisionLevel = 0
var trail: FixedIntStack = null
var qHead = 0
var reasons: Array[Clause] = null
var levels: Array[Int] = null
var conflict: Clause = null
var model: Array[Int] = null
var watched: Array[ClauseList] = null
var seen: Array[Boolean] = null
var cnfFormula: CNFFormula = null
var status: Status = Unknown
var restartInterval = Settings.restartInterval
var nextRestart = restartInterval
val restartFactor = Settings.restartFactor
```

How fast? 0.226s on 100 variables, 40% timeouts 30s on 200 variables

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Pushing the Limits

My personal experience to improve performance in this context:

- A core engine of about 700 lines of code:
  - Global variables that share states accross functions.
  - Almost exclusively imperative.
  - Look very much like C code.
- Hand crafted data structures for the problem.
- Use as much primitive types as possible.

How fast?

Pushing the Limits

My personal experience to improve performance in this context:

- A core engine of about 700 lines of code:
  - Global variables that share states accross functions.
  - Almost exclusively imperative.
  - Look very much like C code.
- Hand crafted data structures for the problem.
- Use as much primitive types as possible.

How fast? 0.183 on 100 variables, 2.131 on 200 variables

How Many Times Faster ?

Version	*dpll*		*conflict*		*optimization*	
Benchmark	Succ.	Time	Succ.	Time	Succ.	Time
uf20	100	0.171	100	0.052	100	0.052
uf50	100	0.171	100	0.084	100	0.081
uuf50	100	0.507	100	0.111	100	0.095
uf75	100	3.948	100	0.138	100	0.122
uf100	30	27.05	100	0.225	100	0.183
uuf100	44	25.42	100	0.369	100	0.275
uf125	0	NA	100	0.393	100	0.317
uf200	0	NA	60	6.688	100	2.131
uf250	0	NA	22	25.46	64	16.01

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

How Many Times Faster ?

Version	*dpll*		*conflict*		*optimization*	
Benchmark	Succ.	Time	Succ.	Time	Succ.	Time
uf20	100	0.171	100	0.052	100	0.052
uf50	100	0.171	100	0.084	100	0.081
uuf50	100	0.507	100	0.111	100	0.095
uf75	100	3.948	100	0.138	100	0.122
uf100	30	27.05	100	0.225	100	0.183
uuf100	44	25.42	100	0.369	100	0.275
uf125	0	NA	100	0.393	100	0.317
uf200	0	NA	60	6.688	100	2.131
uf250	0	NA	22	25.46	64	16.01

A straightforward implementation of DPLL can be more than 100 times slower !

The Big Picture

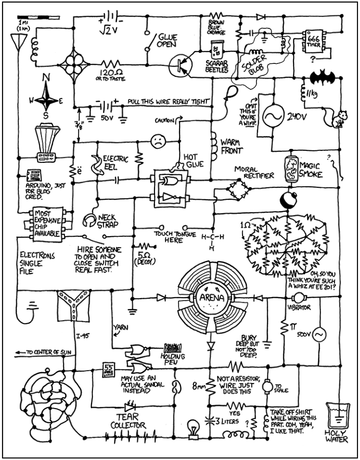
We started from:

```scala
sealed trait Formula {
  def subst(x: Var, v: Boolean)
  def vars: Set[Var]
  def eval: Boolean //if no free variable
}
case class And(f1: Formula, f2: Formula) extends Formula
case class Not(f: Formula) extends Formula
case class Var(id: Int) extends Formula
case class Lit(b: Boolean) extends Formula

def isSat(f: Formula): Boolean = {
  val vs = f.vars
  if(vs.isEmpty)
    f.eval
  else {
    val v = vs.head
    isSat(f.subst(v, true)) || isSat(f.subst(v, false)) }
}
```

The Big Picture (Literally)

And ended up here:



(Courtesy of xkcd.com)

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

The Big Picture

Question:
Do we always have to go back to low level, imperative code to get the best performance ?
Some advantage with Scala:

- Can abstract the messy parts.
- Higher order functions to provide modularity for heuristics.
- Flexible syntax to create powerful API.

Conclusion

- CafeSat is an open source SAT solver.
- It achieves reasonable performance using modern techniques and some implementation tricks.
- It provides a high level API to program with boolean constraints in Scala.

Work in progress:

- Improving performance.
- More general constraints (SMT).
- Available on GitHub: `http://github.com/regb/scabolic`