We present CafeSat, a SAT solver written in the Scala programming language. CafeSat is a modern solver based on DPLL and featuring many state-of-the-art techniques and heuristics. It uses two-watched literals for Boolean constraint propagation, conflict-driven learning along with clause deletion, a restarting strategy, and the VSIDS heuristics for choosing the branching literal. CafeSat is both sound and complete.

In order to achieve reasonable performance, low level and hand-tuned data structures are extensively used. We report experiments that show that significant speedup can be obtained from translating a high level algorithm written in a relatively idiomatic Scala style to a more C-like programming style. These experiments also illustrate the importance of modern techniques used by SAT solver. Finally, we evaluate CafeSat against the reference SAT solver on the JVM: Sat4j.

Categories and Subject Descriptors
J6 [Computer-Aided Engineering]: Computer-Aided Design

General Terms
Algorithms, Verification

Keywords
Boolean satisfiability, constraint solving, verification.

1. INTRODUCTION

The Boolean satisfiability problem (SAT) is one of the most important problem in computer science. From a theoretical point of view, it is the first NP-complete problem. On the practical side, it is used as a target low level encoding for many applications. Since SAT solvers are well understood and have been engineered over many years, applications often choose to rely on them rather than developing a custom solver for the domain. Often those SAT solvers are also an important building block in the more general problem of constraint solving, and in particular as a basis for SMT solvers [4].

In the Boolean satisfiability problem, one is given a set of clauses, where each clause is a set of literals. A literal is either a propositional variable or the negation of a propositional variable. The goal is to find an assignment for the variables such that for each clause, at least one of the literal evaluates to true. This representation is called Conjunctive Normal Form (CNF).

In this paper, we present CafeSat, a complete SAT solver implemented in Scala. CafeSat is strongly inspired by MiniSat [3]. CafeSat implements many recent techniques present in modern SAT solvers. CafeSat is built around the DPLL scheme [2]. Boolean constraint propagation is implemented using the 2-watched literal scheme introduced by Chaff [8]. The branching heuristics is VSIDS, also introduced by Chaff.

A key component of modern SAT solver is the conflict-driven clause learning [10, 11], allowing for long backtracking and restarting. CafeSat supports an efficient conflict analysis, with the 1UIP learning scheme and a clause minimization inspired from MiniSat.

Additionally, CafeSat exports an API for Scala. This enables some form of constraint programming in Scala, as already promoted by ScalaZ3 [6]. We illustrate its ease of use in Figure 1. The code implements a sudoku solver. A sudoku input is represented by a matrix of Option[Int]. We then generate nine variables for each entry, and generate all constraints required by the rules of sudoku. The constraints state how variables from the same rows, columns and blocks of a sudoku grid must relate to each other. Variables and constraints can be naturally manipulated as would any regular boolean expression in Scala.

Our library provides a new boolean type and lifts the usual boolean operations of Scala to enable a natural declaration of constraints. Any SAT problem can be build by combining fresh boolean variables with the above operations. We implement a structure preserving translation to CNF [9]. This transformation avoids the exponential blow up of the naive CNF transformation by introducing a fresh variable for each sub-formula and asserting the equivalence of the new variable with its corresponding sub-formula.

We believe CafeSat could have applications in the Scala world. The current release of the Scala compiler integrates a small SAT solver for the pattern matching engine. It could benefit from a self-contained and efficient solver written entirely in Scala to avoid complex dependencies.
systems on the JVM such as Eclipse also start to include SAT solving technology for their dependency management engines [7].

Finally CafeSat, beside being a practical tool, is also an experiment in writing high performance software in Scala. Our goal it to prove — or disprove — that Scala is suitable to write programs that are usually built in C++. The initial results reported here show that it is necessary to sacrifice some of the advanced features of Scala in order to attain acceptable performance.

## 2. CAFESAT

In this section, we present the architecture and features of CafeSat. We discuss the different heuristics implemented and also describe some of the data structures used. The solving component of CafeSat is currently about 1,300 lines of code. This does not include the API layer. CafeSat is open source and available on GitHub\(^1\). As part of a bigger system, in development, intended to do constraint solving. In this system, CafeSat will play a central role.

In general, we avoid recursion and try to use iterative constructs as much as possible. We use native JVM types whenever possible. We rely on mutable data structures to avoid expensive heap allocations. In particular, we make extensive use of \texttt{Array} with primitive types such as \texttt{Int} and \texttt{Double}. Those types are handled well by the Scala compiler, which is able to map them to the native \texttt{int[]} and \texttt{double[]} on the JVM.

The input (CNF) formula contains a fixed number \(N\) of variables, and no further variables are introduced in the course of the algorithm. Thus, we can represent variables by integers from 0 to \(N - 1\). Many properties of variables such as their current assignment and their containing clauses can then be represented using \texttt{Array} where the indices represent the variable. This provides a very efficient \(O(1)\) mapping relation. Literals are also represented as integers, with even numbers being positive variables and odd numbers being negative variables.

We now detail the important components of the SAT procedure.

### 2.1 Branching Decision

We rely on the VSIDS decision heuristic introduced initially by Chaff [8]. However, we implement the variation of the heuristic described in MiniSat [3]. We keep variables in a priority queue, sorted by their current VSIDS score. On a branching decision, we extract the maximum element of the queue that is not yet assigned. This is the branching literal.

We use a custom implementation of a priority queue that supports all operations in \(O(\log N)\), including a delete by value of the variables (without any use of pointers). The trick is to take advantage of the fact that the values stored in the heap are integers from 0 to \(N - 1\), and maintain an inverse index to their current position in the heap. The heap is a simple binary heap built with an array. In fact, we store two arrays, one for variables and one for their corresponding score. Having two separate arrays seem to be more efficient than one array of tuples.

### 2.2 Boolean Constraint Propagation

CafeSat implements the 2-watched literals described by the Chaff paper. We implement a custom \texttt{LinkedList} to store the clauses that are currently watching a literal. An important feature of our implementation is the possibility to maintain a pointer to elements we wish to remove, so that a remove operation can be done in \(O(1)\) while iterating over the clauses. This is a typical use case for the 2-watched literal, where we need to traverse all clauses that are currently watching the literal, find a new literal to watch, add the current clause to the watchers of the new literal while removing it from the previous one. All operations need to be very fast because they are done continuously on all unit propagation steps.

### 2.3 Clause Learning

In the original DPLL algorithm, the exhaustive search was explicit, setting each variable to true and false successively after exploring the subtree. A more recent technique consists in doing conflict analysis and then learning a clause before backtracking. The intuition is that this learnt clause is a reason why the search was not able to succeed in this branch. This learning scheme also enables the solver to do long backtracking, returning to the first literal choice that caused the clause to be unsatisfiable and not the most recent one.

In CafeSat, we implement a conflict analysis algorithm to learn new clauses. For this, we use the 1UIP learning scheme [11]. We also apply clause minimization as invented by MiniSat. We use a stack to store all assigned variable and maintain a history. We also store for each variable the clause (if any) responsible for its propagation. This implicitly stores the implication graph used in the conflict analysis.

### 2.4 Clause Deletion

```scala
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 uniquInColumns = 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 uniquInRows = 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 uniquInGrid1 = for(k <- 0 to 8; i <- 0 to 2; j <- 0 to 2; r <- 0 to 2; c1 <- 0 to 7; c2 <- c1+1 to 2)
    yield !vars(3*i + r)(3*j + c1)(k) || !vars(3*i + r)(3*j + c2)(k)
  val uniquInGrid2 = for(k <- 0 to 8; i <- 0 to 2; j <- 0 to 2; r1 <- 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 ++ uniquInColumns ++ uniquInRows ++ uniquInGrid1 ++ uniquInGrid2 ++ forcedEntries
  val backtrack = And(allConstraints)
  solve(backtrack)
}
```

Figure 1: Implementing a sudoku solver with CafeSat API.

\(^1\text{https://github.com/regb/scaboli}\)
We use an activity based heuristic similar to the one used for decision branching to select which clauses to keep and which ones to drop. We set a maximum size to our set of learnt clauses, and whenever we cross this threshold, we delete the clauses with the worst activity score. To ensure completeness and termination, we periodically increase this threshold.

Our current implementation simply stores a list of clauses and sorts them each time we need to remove the least active ones. We assume that clause deletion only happens after a certain number of conflicts, so it is not a very frequent operation. Besides, it could be cheaper to only sort the list each time it is needed, than to maintain the invariant in a priority queue for each operation.

### 2.5 Restarting Strategy

We use a restart strategy based on a starting interval that slowly grows over time. The starting interval is $N$ which is the number of conflicts until a restart is triggered. A restart factor $R$ will increase the interval after each restart. This increases in the restart interval guarantees completeness of the solver. In the current implementation, $N = 32$ and $R = 1.1$.

### 3. EXPERIMENTS

We ran a set of experiments to evaluate the impact of various optimizations that have been implemented over the development of CafeSat. The goal is to give some insight on how incremental refinement of a basic SAT solver can lead to a relatively efficient complete solver. We selected a few important milestones in the development of CafeSat, and compared their performance on a set of standard benchmarks.

Our results are summarized in Table 1. The experiments have been run on an Intel core i5-2500K with 3.30GHz and 8 GiB of RAM. A timeout was set to 30 seconds. The running time is showed in seconds. The versions are organized from the most ancient to the most recent one, their description is as follows:

- **naive.** Based on the straightforward implementation techniques using AST to represent formulas, and recursive functions along with pattern matching for DPLL and BCP.
- **counters.** Uses specialized clauses. Each variable is associated with adjacency lists of clauses containing the variable. It uses counters to quickly determine whether a clause becomes SAT or leads to a conflict.
- **conflict.** Introduces conflict-driven search with clause learning. This is a standard architecture for modern SAT solver. However the implementation at this stage suffers from a lot of overhead.
- **2-watched.** Implements the BCP based on 2-watched literals.
- **minimization.** Focuses on a more efficient learning scheme. The conflict analysis is optimized and the clause learnt is minimized. It also introduces clause deletion.
- **optimization.** Applies many low level optimizations. A consistent effort is invested in avoiding object allocation as much as possible, and overhead is reduced thanks to the use of native Array with Int as much as possible. We implemented dedicated heap and stack data structures, as well as a linked list optimized for our 2-watched literal implementation.

Our solver is competitive with Sat4j on the instances of standard problem from SATLIB. They contain up to about 300,000 clauses. Our results are summarized in Table 1. The experiments are run on an Intel core i5-2500K with 3.30GHz and 8 GiB of RAM. A timeout was set to 30 seconds. The running time is showed in seconds. The versions are organized from the most ancient to the most recent one, their description is as follows:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>CafeSat</th>
<th>Sat4j</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Suc.</td>
<td>Time (s)</td>
<td>% Suc.</td>
</tr>
<tr>
<td>uf50</td>
<td>100</td>
<td>92</td>
</tr>
<tr>
<td>uf100</td>
<td>100</td>
<td>92</td>
</tr>
<tr>
<td>uuf100</td>
<td>100</td>
<td>92</td>
</tr>
<tr>
<td>uf125</td>
<td>100</td>
<td>92</td>
</tr>
<tr>
<td>uf200</td>
<td>100</td>
<td>92</td>
</tr>
<tr>
<td>uf250</td>
<td>100</td>
<td>92</td>
</tr>
<tr>
<td>bmc</td>
<td>100</td>
<td>92</td>
</tr>
</tbody>
</table>

**Table 2: CafeSat vs Sat4j: Showdown.**
instances. That CafeSat is slower than Sat4j should not come as a shock. Sat4j has been under development for more than 5 years and is considered to be the best SAT solver available on the JVM.

4. CONCLUSION

We presented CafeSat, a modern SAT solver written in Scala. CafeSat offers solid performance and provides Scala programmers with a library for constraint programming. This library makes access to SAT solving capabilities very easy in the Scala ecosystem offering a native solution with the usual feeling of a Scala DSL.

CafeSat is a DPLL based SAT solver. It is both sound and complete. It integrates many state-of-the-art techniques and heuristics that are currently in use in some of the most popular SAT solvers.

We used an extensive set of standard benchmarks to evaluate the improvement of CafeSat over time. These results give some insight on the importance of good heuristics and careful hacking. We also compared CafeSat to Sat4j, and despite Sat4j being superior, our new solver shows some promising initial results.

We plan to build a complete constraint solver on top of CafeSat. To that end, we will extend CafeSat with incremental SAT solving. We also aim to provide a constraint programming API to use our extended system. We hope to make CafeSat a solid infrastructure on which Scala programmers can build.

5. ACKNOWLEDGMENTS

The author would like to thank Viktor Kuncak and Alexandre Duc for comments on this report, as well as Philippe Suter for precious advices on implementation details.

6. REFERENCES
