The SAT Compiler in Picat

Neng-Fa Zhou (CUNY Brooklyn College & Graduate Center)

November, 2018

SAT solvers’ performance has drastically improved during the past 20 years, thanks to the inventions of techniques from conflict-driven clause learning, backjumping, variable and value selection heuristics, to random restarts [2, 4, 21]. SAT has become the backbone of many software systems, including specification languages for model generation and checking [8, 16, 17, 20], planning [19, 28], program analysis and test pattern generation [29], answer set programming [5, 11], and solvers for general constraint satisfaction problems (CSPs) [4, 14, 18, 27, 30, 31, 32].

In order to fully exploit the power of SAT solvers, a compiler is needed to Booleanize constraints as formulas in the conjunctive normal form (CNF) or some other acceptable form. The encodings of constraints have big impact on the runtime of SAT solvers [3]. Several encodings of domain variables have been proposed, including sparse encoding [12, 32], order encoding [6, 22, 31], and log encoding [15, 10]. The sparse and order encodings can easily blow up the code size, and the log encoding is perceived to be a poor choice, despite its compactness, due to its failure to maintain arc consistency, even for binary constraints. This dilemma of the eager approach has led to the emergence of the lazy approach, as represented by SMT solvers that use integer arithmetic as a theory [1, 7, 24] and lazy clause generation (LCG) solvers that combine SAT and constraint propagation [9, 26]. Both the eager and lazy approaches have strengths and weaknesses [25]. For problems that require frequent checking of arithmetic constraints the lazy approach may not be competitive due to the overhead, even when checking is done incrementally and in a priori manner. From an engineering perspective, the eager approach also has its merit, just like the separation of computer hardware and language compilers is beneficial.

We have developed an optimizing compiler in Picat [34], called PicatSAT, which adopts the sign-and-magnitude log-encoding for domain variables. For a domain with the maximum absolute value \( n \), it uses \( \log_2(n) \) Boolean variables to encode the domain. If the domain contains both negative and positive values, then another Boolean variable is employed to encode the sign. Each combination of values of the Boolean variables represents a valuation for the domain variable. The addition constraint is encoded as logic adders, and the multiplication constraint is encoded as logic adders using the shift-and-add algorithm.

Log-encoding for constraints resembles the binary representation of numbers used in computer hardware, and many algorithms and optimization opportunities have been exploited by hardware design systems. PicatSAT adopts some optimizations from CP systems, language compilers, and hardware design systems for encoding arithmetic constraints into compact and efficient SAT code: it preprocesses constraints before compilation in order to remove no-good values from the domains of variables whenever possible; it eliminates common subexpressions so that no primitive constraint is duplicated; it uses a logic optimizer to generate optimized code for adders. PicatSAT also incorporates an optimization, named equivalence reasoning, [33] which identifies values or equivalence relationships of Boolean variables in primitive arithmetic constraints at compile time. These optimizations significantly improve the quality of the generated code.

PicatSAT is provided in Picat as a solver module, named sat, which follows the common constraint interface. Other solver modules that implement the same constraint interface in Picat include cp, smt, and mip. The following gives a Picat program for the Sudoku problem:
import sat.

main =>
  Board = [
    [2,_,_,6,7,_,_,_,_],
    [_,6,_,_,2,1,],
    [4,_,_,_,_,8,_,_],
    [5,_,_,_,_,9,3,_,_],
    [_,3,_,_,_,_,5,_,_],
    [_,2,8,_,_,_,7],
    [_,_,2,_,_,_,_,7],
    [7,_,8,_,_,_,6,_,_],
    [_,_,_,_,5,3,_,_,8]],
  sudoku(Board),
  foreach(Row in Board) writeln(Row) end.

sudoku(Board) =>
  N = Board.len,
  Vars = Board.vars(),
  Vars :: 1..N,
  foreach (Row in Board) all_different(Row) end,
  foreach (J in 1..N)
    all_different([Board[I,J] : I in 1..N])
  end,
  M = round(sqrt(N)),
  foreach (I in 1..N-M, J in 1..N-M)
    all_different([Board[I+K,J+L] : K in 0..M-1, L in 0..M-1])
  end,
  solve(Vars).

The first line imports the sat module, which defines the used built-ins by this program, including the operator ::, the global constraint all_different, and the solve predicate for labeling variables. For a given board, the sudoku predicate retrieves the length of the board (Board.len), extracts the variables from the board (Board.vars()), generates the constraints, and calls solve(Vars) to label the variables. The first foreach loop ensures that each row of Board has different values. The second foreach loop ensures that each column of Board has different values. The list comprehension [Board[I,J] : I in 1..N] returns the Jth column of Board as a list. Let M be the size of the sub-squares (M = round(sqrt(N))). The third foreach loop ensures that each of the N M×M squares contains different values. As demonstrated by this example, Picat’s language constructs such as functions, arrays, loops, and list comprehensions make Picat as powerful as other modeling languages, such as OPL [13] and MiniZinc [23], for CSPs. The common constraint interface that Picat provides for the solver modules allows seamless switching from one solver to another.

PicatSAT, which is implemented in Picat, has over 10,000 lines of code, excluding comments. PicatSAT competed in the MiniZinc Challenge (http://www.minizinc.org/challenge.html) and the XCSP competition (http://www.xcsp.org/) in 2018. In both competitions, PicatSAT used Lingeling (version 587f, http://fmv.jku.at/lingeling/) as the underlying SAT solver. PicatSAT won the first place in the COP track and the third place in the CSP track in the 2018 XCSP competition, and won the silver medal in the free-search class and the bronze medal in the
parallel-search class in the 2018 MiniZinc Challenge. The competition results demonstrate the competitiveness of the implementation.

In addition to the advancement of SAT solvers and the optimizations implemented in the compiler, the competition results of PicatSAT are also attributed to Picat, the implementation language. The log encoding is arguably more difficult to implement than the sparse and order encodings. Picat’s features, such as attributed variables, unification, pattern-matching rules, and loops, are all put into good use in the implementation. There are hundreds of optimization rules, and they can be described easily as pattern-matching rules in Picat. Logic programming has been proven to be suitable for language processing in general, and for compiler writing in particular; PicatSAT has provided another testament.

Acknowledgements

The author would like to thank Håkan Kjellerstrand for helping tune and test the compiler, Olivier Roussel and Christophe Lecoutre for the feedback from the XCSP competition, and Andreas Schutt for the feedback from the MiniZinc Challenge.

References


