Jay LeCavalier
Computer Aided Verification
Final Project Report
Due 12/8/15

camlogic: a CDCL SAT solver for OCaml

1. Introduction

camlogic is a CDCL SAT solver written in OCaml. The program, once configured and compiled, can be run from the command line. It will take a file in the DIMACS file fomat [1] as an argument, parse the file into a SAT problem, and solve. If the formula specified is satisfiable, then the solver will print a satisfying assignment of variables to the terminal. Otherwise, the program will tell the user the formula is unsatisfiable. In either case, the program will also tell the user how long it took to solve the problem.

The structure of this report is as follows: In section 2, we discuss the system requirements for the program and how to compile and run the program. In section 3, we discuss some implementation details. In section 4, we discuss the algorithm used. Finally, in section 5, we present the runtime results of the program.

2. Requirements and Usage Instructions

In order to run camlogic, you will need to have OCaml installed (at least version 4.02) [2]. You will also need ocamlfind [3] and oasis [4], which are both OCaml packages. Although it is not required, it is strongly recommended that you install ocamlfind and oasis using opam [5], the OCaml package manager. It will make your life easier.

Once you have the prerequisite programs and packages installed, you are ready to compile and install the program. Once downloaded [5], navigate to the root directory of camlogic using your terminal and type “./configure”. This will give oasis important information about your system, including which compiler to use, etc. Then, type “make” to compile the program.

Once you have compiled the program, you can run it using the command “./main.native <files>”, where <file> is a list of one or more DIMACS format cnf files [1]. The program will tell you whether the formula is satisfiable or not and how long it took to come to that conclusion. The program will write results about how long it took to solve each of the listed files in the output file “time_results.dat”. If you need to use some example cnf files, they are located in the “specs” directory. You can also find some more examples from Rutger's University [7].

3. Implementation Details

The program is built using oasis [4], an incredibly useful tool for OCaml. oasis allows you to specify information about your OCaml project using a specification file called “_oasis” located in the root directory. oasis uses this file to generate the script which configures the program, generates the makefile, and many other useful things.

The program itself is divided into three modules. Here are there descriptions:

1. Main (src/main.ml) – This module invokes the lexer and parser to lex and parse the file specified by the user. The lexer and parser are generated using two specification files, “sys_parser.mly” and “sys_lexer.mll”. The lexer specification file is meant to be used with ocamllex [6], a program that generates a lexer for your program. It behaves similarly to flex, the C program. The parser specification file is meant to be used with ocamlyacc [6], a program that generates a parser for your program. It behaves similarly to yacc, the C program.

2. ParserUtils (src/parserUtils.ml) – This module contains functions that are invoked by the parser. These functions involve taking the raw data extracted from the specification file provided by the user and converting the problem into a SAT problem, verifying that the SAT problem specified is well-formed, and invoking the SAT solver.

3. Cdcl (src/cdcl.ml) – This is where the magic happens. This module contains the function sat (and all of its subroutines), which takes a SAT problem as an input and tells us whether the problem is satisfiable or not. We discuss the algorithm used to solve SAT in detail in the next section.

4. The CDCL Algorithm

To solve SAT, we employ the CDCL (conflict driven clause learning) algorithm. The algorithm runs as follows:

1. First, we preprocess the formula. If any clause is a unit clause, we are forced to make an assignment accordingly. If a literal is a unit clause and its negation is also a unit clause, we immediately halt and return unsatisfiable. If a literal occurs pure in the formula (its negation does not occur in the formula), then we assign that literal to true.

2. Next, we attempt to find a satisfying assignment for the formula. If we are able to do so, we halt, reveal the solution, and return satisfiable. However, sometimes while searching for a satisfying assignment, we are not forced to assign any variables to either true or false, so we will have to arbitrarily assign a variable true or false. In this case, we increment a global counter called the decision level and we go to step 3.

3. If we make a decision about which literals to assign to true, there may be implications about this. For example, if I have the clause (A v ~B) and I assign A to false, then I must also assign B to false to satisfy the whole formula. When an implication like this occurs, we add it to a structure called the implication graph. The nodes of the implication graph contain literals and the decision level at which that literal was assigned true, and the edges represent an implication. If a node occurs in the graph, the literal of that node is implied to be assigned true. Sometimes, while constructing this graph, we add a node containing a literal and another node containing the negation of that literal. This is called a conflict, as we are not allowed to assign both a literal and its negation to true. If this occurs, we resolve the conflict in step 4. Otherwise, we go back to step 2.

4. When a conflict occurs, we must figure out why the conflict occurred, give the solver some additional information about how to avoid that conflict, and backtrack to an appropriate decision level, where we had not yet generated a conflict. This involves a complicated subroutine (depicted graphically in [8]), which we describe as follows:
i) First, we divide the graph into two sub-graphs. The first sub-graph is called the conflict side. This graph contains the conflicting node and every node at the same decision level as the conflicting node (except for the decision variable for that decision level, which is the variable we arbitrarily assigned at this decision level). The other side is called the reason side. This contains all the nodes that aren't on the conflict side.
ii) Now, we restrict the reason side to the nodes that caused the conflict. These are precisely the nodes on the reason side that have an edge which enters the conflict side. Note that the literal that was arbitrarily chosen at this decision level is guaranteed to have an edge which enters the conflict side.
iii) We look through the nodes we selected in step ii and find the maximum decision level that is NOT

the decision level at which the conflict occurred. This decision level is guaranteed to be a safe place to back up to (safe in the sense that no conflict was implied at that decision level). If there is no safe decision level to return to, we immediately halt and return unsatisfiable, as this means that the conflict cannot be resolved.
iv) We “back up” to the decision level found during step iii by removing all assignments that occurred after the backup level and clearing the implication graph (Aside from the assignments made at the appropriate level). This effectively resolves the conflict and all of the decisions that led to the conflict.
v) Finally, to make sure that we do not make the same decision that implied the conflict we just resolved, we take all of the literals from the nodes we collected in step ii and conjoin them together. This conjunction is essentially the reason the conflict occurred, as each conjunct represents a fact which directly implies something on the conflict side. If we negate this conjunction, we get a clause. We add that clause to our list of clauses, which will not change the satisfiability of the formula and will also guarantee that we do not make the same decisions that led to the conflict we just resolved.

After this subroutine completes, we go back to step 2.

5. Summary of Results

Here is a table which summarizes the results of camlogic running on different files. In each of the problem categories, 100 different problems were tested. All of the problems were downloaded from Rutger's University [7].

	Average Runtime	Fastest Runtime	Slowest Runtime
Satisfiable problems with 20 variables and 91 clauses	0.003876 seconds	Negligible (close to zero seconds)	0.024 seconds
Satisfiable problems with 50 variables and 218 clauses	0.48944 seconds	0.012 seconds	3.344 seconds
Unsatisfiable problems with 50 variables and 218 clauses	1.19576 seconds	0.24 seconds	10.132 seconds
Satisfiable problems with 75 variables and 325 clauses	14.11968 seconds	0.028 seconds	2.19 minutes
Unsatisfiable problems with 75 variables and 325 clauses	38.46464 seconds	2.668 seconds	4.5962 minutes

6. References

[1] Information about the DIMACS file format: http://people.sc.fsu.edu/~jburkardt/data/cnf/cnf.html

[2] How to install Ocaml: https://github.com/realworldocaml/book/wiki/Installation-Instructions

[3] ocamlfind: https://opam.ocaml.org/packages/ocamlfind/ocamlfind.1.5.5/

[4] Setting up and using oasis: http://oasis.forge.ocamlcore.org/

[5] Setting up and using opam: https://github.com/realworldocaml/book/wiki/Installation-Instructions

[6] ocamllex and ocamlyacc: http://caml.inria.fr/pub/docs/manual-ocaml-4.00/manual026.html

[7] gigabytes of free sat problems from Rutger's University:
http://www.cs.ubc.ca/~hoos/SATLIB/benchm.html

[8] Visual representation of conflict clause generation: http://www.mimuw.edu.pl/~tsznuk/tmp/dpll.pdf