Verified LRAT and LPR Proof Checking with `cake_lpr`

Yong Kiam Tan   Marijn J. H. Heule   Magnus O. Myreen

I. Summary

We present the `cake_lpr` proof checker [1] which is capable of checking proofs in either Linear RAT (LRAT) or Linear PR (LPR) proof formats. The LPR format is a backwards-compatible extension of LRAT. The checker is formally verified using CakeML and the HOL4 theorem prover; its formal proof is discussed in [1] and briefly in Section III. The DRAT and DPR proof formats are supported using `DRAT-trim` and `DPR-trim` as preprocessing tools, respectively.
The verified proof checker is available at:
https://github.com/tanyongkiam/cake_lpr
The `DRAT-trim` and `DPR-trim` tools are available at:
https://github.com/marijnheule/drat-trim
https://github.com/marijnheule/dpr-trim

A. Example

An outline of an end-to-end LRAT proof checking run is as follows:

```
# Assume the problem is input.cnf in DIMACS

... run SAT solver on input.cnf ...
... generate proof file input.drat ...

# Run drat-trim on the DRAT proof and
# generate LRAT file

drat-trim input.cnf input.drat -L input.lrat

# Run cake_lpr on the resulting LRAT proof

cake_lpr input.cnf input.lrat
```

If the proof checks successfully, `cake_lpr` will print to standard output:

```
s VERIFIED UNSAT
```

All other error messages, such as proof checking error, parsing error, out-of-memory error, will be printed to `stderr`. Solvers capable of generating LRAT proofs directly can skip the use of `DRAT-trim`. End-to-end proof checking for LPR proofs can be done similarly, using `DPR-trim` as the preprocessor for DPR proofs. It is also possible to convert DPR proofs to DRAT, then use `DRAT-trim`, but this approach is **not recommended** as it is significantly slower than checking DPR (and LPR) proofs directly [1].

II. Supported Proof Formats

Formal descriptions of all proof formats are available in the cited publications [1], [2] and online. We give brief descriptions of the formats with concrete examples.

A. DRAT and LRAT

The DRAT format consists of a list of clause addition or deletion steps, one per line. All lines are terminated by `0`. Each added clause must have RAT redundancy with respect to the current formula.

```
<CLAUSE> 0
d <CLAUSE> 0
```

Concrete example:

```
1 -2 3 0     # Add clause x_1,!x_2,x_3
d 1 2 -3 0   # Del clause x_1,x_2,!x_3
```

The `DRAT-trim` tool can be used as a preprocessor to automatically convert an input DRAT proof to LRAT format. The latter format extends DRAT with a notion of clause IDs and proof hints for each line. The input CNF is assumed to be given IDs in ascending order from 1 to n where n is the number of clauses in the file. Addition lines in LRAT have the following format, where `<ID>` is a positive integer, `<IDs>` is a list of `<ID>`, and `[...]*` denotes 0 or more repetitions of the enclosed block:

```
<ID> <CLAUSE> 0 <IDs> [-<ID> <IDs>]* 0
```

The first `<ID>` is the clause ID to be assigned to `<CLAUSE>`. If `<CLAUSE>` has RAT redundancy, then the first literal in the clause is the pivot literal. The first block of `<IDs>` lists unit propagation steps starting from the blocking assignment for `<CLAUSE>`. If `<CLAUSE>` has RAT redundancy, then this first block is followed by 0 or more `-<ID> <IDs>` blocks, where `-<ID>` refers to the `<ID>`-th clause in the RAT proof and the corresponding `<IDs>` indicate unit propagation steps for that clause.

Deletion steps are written with a list of clause IDs rather than clauses. All the clauses with IDs in `<IDs>` are deleted.

```
<ID> d <IDs> 0
```

Concrete example:

```
# Add clause x_1,!x_2,x_3 at clause ID 15
# with RAT on pivot !x_2
15 -2 1 3 0 4 13 7 10 8 -5 2 4 -10 3 5 0
# Del clause IDs 13 14 15
# (ID 16 in front of the line is ignored)
16 d 13 14 15 0
```

A complexity analysis for the LRAT proof format is given in [2, Theorem 2], where asymptotically (keeping all parameters constant except number n of steps in proofs), the complexity is reported as $O(n^2 \log n)$; `cake_lpr` slightly improves

```
⊢ cake_lpr_run cl fs mc ms ⇒                                          (1)
  machine_sem mc (basis_ffi cl fs) ms ⊆                               (2)
    extend_with_resource_limit
      { Terminate Success (cake_lpr_io_events cl fs) } ∧
  ∃ out err.                                                          (3)
    extract_fs fs (cake_lpr_io_events cl fs) =
      Some (add_stdout (add_stderr fs err) out) ∧
    if ...                                                            (4)
    else if length cl = 3 then
      if out = «s VERIFIED UNSAT\n» then
        inFS_fname fs (el 1 cl) ∧
        ∃ fml.
          parse_dimacs (all_lines fs (el 1 cl)) = Some fml ∧
          unsatisfiable (interp fml)
      else out = «»
    else ...
```

Fig. 1. The end-to-end correctness theorem for the CakeML LPR proof checker. (Some irrelevant cases are elided with . . . for brevity).

the asymptotic bound to $O(n^2)$ because it uses constant-time rather than logarithmic-time lookup data structures [1]. Empirically, we have observed that most proofs generated by solvers in past SAT competitions are dominated by simple (non-RAT) steps. In that case, one may expect near-linear scaling from `cake_lpr`.

B. DPR and LPR

The DPR format extends DRAT so that added clauses are *propagation redundant* with respect to the current formula. Here, `<WIT>` is a list of literals which must start with the first literal in `<CLAUSE>`. Note that this is syntactically backwards compatible with DRAT (when `<WIT>` is empty).

```
<CLAUSE> <WIT> 0
```

The `DPR-trim` tool can be used as a preprocessor to automatically convert an input DPR proof to LPR format. The latter format extends DPR with clause IDs and proof hints in the same way LRAT extends DRAT. The only syntactic addition is the optional `<WIT>` after `<CLAUSE>`.

```
<ID> <CLAUSE> <WIT> 0 <IDs> [-<ID> <IDs>]* 0
```

The proof checking procedure for LPR is backwards compatible with LRAT, using `<IDs>` and `[-<ID> <IDs>]*` as unit propagation hints for propagation redundancy [1]. Deletion lines are identical for LPR and LRAT.

Concrete example:

```
# Add clause x_1,!x_2,x_3 at clause ID 15
# with PR witness !x_2,x_5
15 -2 1 3 -2 5 0 4 13 7 10 8 -5 2 4 -10 3 5 0
# Del clause IDs 13 14 15
# (ID 16 in front of the line is ignored)
16 d 13 14 15 0
```

The proof checking procedure for LPR is essentially the same as LRAT, i.e., with $O(n^2)$ complexity (assuming all other parameters are held constant).

III. Proof Checker Verification

Our proof checker, `cake_lpr`, is formally verified down to the level of its x64 machine code implementation, which eliminates the possibility of bugs arising from, e.g., compiler errors, code extraction, or other, unverified additions to (verified) source code. This is achieved by compiling its formally verified CakeML source code implementation, with a formally verified compiler for CakeML [1].

The key correctness theorem is shown in Fig. 1. To informally summarize:

- Line (1) assumes that the `cake_lpr` binary is executed in an x64 machine environment set up according to the standard CakeML assumptions.
- Lines (2) guarantees that `cake_lpr` will terminate successfully (i.e., no out of bounds array accesses, etc.); it may run out of either heap or stack memory (resource limits).
- Lines (3) says that, according to the CakeML file system model, there will be some strings printed to standard output and standard error.
- Lines (4) says (among other things) that, IF the string "s VERIFIED UNSAT" is printed onto standard output, then the first command line argument corresponds to a file, which parsed in DIMACS format, to a formula that is unsatisfiable. The DIMACS parser is verified to be left inverse to the DIMACS printer.

References

[1] Y. K. Tan, M. J. H. Heule, and M. O. Myreen, "cake_lpr: Verified propagation redundancy checking in CakeML," in *TACAS*, ser. LNCS, J. F. Groote and K. G. Larsen, Eds., vol. 12652. Springer, 2021, pp. 223–241.

[2] L. Cruz-Filipe, M. J. H. Heule, W. A. Hunt Jr., M. Kaufmann, and P. Schneider-Kamp, "Efficient certified RAT verification," in *CADE*, ser. LNCS, L. de Moura, Ed., vol. 10395. Springer, 2017, pp. 220–236.