Efficiently grounding FOL using bit vectors

Lucas Van Laer, Simon Vandevelde, Joost Vennekens

[1] KU Leuven, De Nayer Campus, Dept. Of Computer Science,
[2] Leuven.AI – KU Leuven institute for AI, B-3000 Leuven, Belgium
[3] Flanders Make – DTAI-FET
{lucas.vanlaer, s.vandevelde, joost.vennekens}@kuleuven.be

Introduction

Systems for declarative problem solving often use a "ground and solve" approach, in which a user-made "high-level" specification is transformed into an equivalent "low-level" specification that can be used by a solver to search for solutions. This transformation involves grounding out first-order variables, by replacing them with all possible values in their domain. Typically, we can then also reduce the formula using information about the problem instance. For example:

$$\forall x, y : x \neq y \wedge \mathit{border}(x, y) \Rightarrow \mathit{colour}(x) \neq \mathit{colour}(y).$$

can be ground out to the following, using a known *borders/2* to simplify:

$$\mathit{colour}(\mathit{USA}) \neq \mathit{colour}(\mathit{Canada}) \wedge \mathit{colour}(\mathit{USA}) \neq \mathit{colour}(\mathit{Mexico}) \wedge \cdots$$

In our LPNMR 2024 paper [5], we present a new method to efficiently perform such grounding for first order logic by using bit vectors to leverage the *Single Instruction, Multiple Data (SIMD)* parallelism of modern processors.

Implementation

In essence, our implementation relies on calculating the *satisfying sets* of formulas. A satisfying set for a formula ϕ with free variables $x_1, \ldots, x_n$ consists of those tuples of domain elements $d_1, \ldots, d_n$ that satisfy the formula. For example, consider the formula $p(x) \vee q(y)$, with the domains of p and q the natural numbers from 1 to 3. If both predicates are defined (e.g., as $p = \{1, 3\}$, $q = \{2\}$), we can calculate the satisfying set of the formula by representing both as a bit vector, and applying a logical or: $[101] \vee [010] = [111]$. Such bit vector operations can be straightforwardly extended to other logical operations and quantification.

In the case where only some symbols are interpreted, the approach described earlier can be used to simplify formulas using a bottom-up approach. Indeed, when grounding away a quantifier, we do not need to consider all values of a domain but can instead limit ourselves to the values that still matter. For instance, if $q(x)$ is interpreted in $\forall x : p(x) \vee q(x)$, we can effectively rewrite the formula as $\forall x : \neg q(x) \Rightarrow p(x)$, and only ground out those domain values that do not satisfy q. Any arbitrary formula containing interpreted symbols can be re-written using this approach, thereby allowing us to (in some cases) greatly simplify the resulting grounding.

Evaluation

We implemented our bit vector-based grounding in a new grounder named SLI [1]. It accepts problems formalized in the FO(·) language, an extension of FO, which it grounds to SMT-LIB that is passed on to the Z3 solver. SLI is written in Rust, and is available under the LGPLv3 license.

To evaluate the grounder, we implemented three grounding methods: the bit vector approach presented in this paper (*SLI vec*), and a naive approach of grounding by checking each tuple of domain elements the formula, with and without simplification (*SLI naive* and *SLI no reduc*). We also include gringo v5.7.1 [4] and IDP-Z3 v0.11.1 [3] solvers in our benchmark. The benchmark is run on a Xeon Silver 4210R CPU with 16 GB of RAM with a timeout of 600 seconds.

To benchmark grounding, we used a number of benchmarks that require only trivial solving (CI, CS and TG), and split the first two into SAT and UNSAT instances. In addition, we also included four benchmark from the ASP2013 competition (GG, GC, PPM, WSP). For more information, we refer to the full text. All benchmarks and scripts are available online [2].

Table 1 shows the average grounding time for each instance. The averages have been calculated by running all the instances of each benchmark, with timeouts omitted from the averages but listed between parentheses instead. The grounding averages show that *SLI vec* and *SLI naive* both outperform the other approaches on most benchmarks. Of these two, *SLI vec* seems faster overall, but the difference is never very large. However, the TG benchmark shows a weakness of the *SLI vec* approach. In this benchmark, the goal is to identify all edges that form triangles in a graph, for which the bit vectors need to reserve a bit position for every one of the n^3 *potential* triangles. Gringo vastly outperforms all SLI approaches here, as its bottom-up approach only considers the $O(n^2)$ actual edges in the graph. On the ASP competition benchmarks, SLI vec and gringo both alternate as being the fastest.

To conclude, our paper presents a novel grounding method for FOL using bit vectors. Through our evaluation, we have shown that it can be considered competitive w.r.t other state-of-the-art approaches.

Table 1: Average grounding time in seconds for given benchmark

Benchmark	SLI vec	SLI naive	SLI no reduc	gringo	IDP-Z3
CI-SAT(50)	0.920	**0.860**	2.179(30)	3.744	timeout
CI-UNSAT(50)	**0.681**	0.806	1.661(28)	2.844	timeout
CS-SAT(50)	**0.774**	0.893	1.771(24)	3.429	timeout
CS-UNSAT(50)	0.765	**0.710**	1.764(31)	3.604	timeout
TG(11)	0.075(4)	2.06(5)	3.57(6)	**0.0238**	6.81(6)
GG(10)	**0.035(2)**	0.045(2)	0.036(2)	0.036(3)	0.232(2)
GC(60)	0.052(42)	0.068(42)	0.082(43)	**0.024(32)**	0.309(42)
PPM(30)	**0.069**	0.097(5)	0.084	7.810(4)	2.55(1)
WSP(30)	0.031	0.033	0.0304	**0.0165**	0.205

References

1. https://gitlab.com/EAVISE/sli/SLI
2. https://gitlab.com/EAVISE/sli/vectorized_interpretation_benchmark
3. Carbonnelle, P., Vandevelde, S., Vennekens, J., Denecker, M.: IDP-Z3: A reasoning engine for FO (.). arXiv preprint arXiv:2202.00343 (2022)
4. Gebser, M., Kaminski, R., Kaufmann, B., Schaub, T.: Multi-shot ASP solving with clingo (2017)
5. Van Laer, L., Vandevelde, S., Vennekens, J.: Efficiently grounding fol using bit vectors. Proceedings of LPNMR 2024 (2024)