Proceedings of the
2nd International Workshop on Quantification

QUANTIFY 2015

Co-located with the
25th International Conference on Automated Deduction (CADE)
August 3, 2015
Berlin, Germany

Hubie Chen, Florian Lonsing, Martina Seidl

Preface

Quantifiers play an important role in language extensions of many logics. The use of quantifiers often allows for a more succinct encoding as it would be possible without quantifiers. However, the introduction of quantifiers affects the complexity of the extended formalism in general. In consequence, theoretical results established for the quantifier-free formalism may not directly be transferred to the quantified case. Further, techniques successfully implemented in reasoning tools for quantifier-free formulas cannot directly be lifted to a quantified version.

The Workshop on Quantification (QUANTIFY 2015) brings together researchers who investigate the impact of quantification from a theoretical as well as from a practical point of view. Quantification is a topic in different research areas, e.g., in SAT in terms of QBF, in CSP in terms of QCSP, in SMT, etc. After the first successful edition of QUANTIFY at the Vienna Summer of Logic as FLoC Workshop in 2014, this second edition is hosted by the 25th International Conference on Automated Deduction (CADE), Berlin, Germany.

QUANTIFY 2015 features an invited talk on "Proof Complexity of Quantified Boolean Formulas" by Olaf Beyersdorff and six contributed submissions which were reviewed and discussed by QUANTIFY's program committee. We would like to thank all the members of the PC for helping us with setting up the program. These informal proceedings archive and document these six contributions[1].

July 2015, Hubie Chen, Florian Lonsing, Martina Seidl

[1]The copyright remains with the authors.

Organization

Program Chairs

- Hubie (Hubert) Chen, Universidad del Pais Vasco and Ikerbasque
- Florian Lonsing, Vienna University of Technology, Austria
- Martina Seidl, University of Linz, Austria

Program Committee

- Olaf Beyersdorff, University of Leeds
- Nikolaj Bjorner, Microsoft Research
- Jasmin Blanchette, TU Munich
- Mikolas Janota, INESC-ID Lisboa
- Laura Kovacs, Chalmers University of Technology
- Francesco Scarcello, DIMES, University of Calabria
- Christoph Wintersteiger, Microsoft Research

Contents

1 A Survey on DQBF: Formulas, Applications, Solving Approaches

Gergely Kovasznai A Survey on DQBF: Formulas, Applications, Solving Approaches

A Survey on DQBF: Formulas, Applications, Solving Approaches

Gergely Kovásznai

IoT Research Center,
Eszterhazy Karoly University of Applied Sciences,
Eger, Hungary
`kovasznai.gergely@ekf.hu`

1 Introduction

Dependency Quantified Boolean Formulas (DQBF) are obtained by adding Henkin quantifiers to Boolean formulas and have seen growing interest in the last years. In contrast to QBF, the dependencies of a variable in DQBF are explicitly specified instead of being implicitly defined by the order of the quantifier prefix. This enables us to also use partial variable orders as part of a formula instead of only allowing total ones. As a result, problem descriptions in DQBF can possibly be exponentially more succinct. While QBF is PSpace-complete, DQBF was shown to be NExpTime-complete [14].

2 Applications

Many practical problems are known to be NExpTime-complete. This includes, e.g., partial information non-cooperative games [13] or certain bit-vector logics [12, 15] used in the context of Satisfiability Modulo Theories (SMT). More recently, also applications in the area of partial equivalence checking (PEC) problems [7, 8] have been discussed and DQBF has been shown to offer a natural encoding for PEC problems. For running experiments, one can access publicly available PEC problem instances and their DQBF encodings [3, 6, 9].

3 Solving Approaches

The first known direct solving approach for DQBF is an adaptation of QDPLL [5], which did not end up being very efficient.

Expansion-based techniques for DQBF were also investigated [1, 2] and yielded in a (not publicly available) expansion-based solver in [8] that uses an underlying SAT solver.

In [4], a refutational approach is proposed that is based on QBF abstraction, thus an uderlying QBF solver is used. This approach is incomplete since it only allows refutation of unsatisfiable formulas.

Based on the fact that Effectively Propositional Logic (EPR) is another logic which is NExpTime-complete, we investigated how to adapt an instantiation-based EPR solving approach, the Inst-Gen calculus [10, 11] to DQBF. We published those results and proposed a new instantiation-based DQBF solver, iDQ in [6]. As the experiments showed, iDQ turned out to be an efficient solver.

In [9], an elimination-based solving strategy is proposed and is implemented in the DQBF solver called HQS. Besides the powerful elimination strategy, HQS utilizes several optimizations, such as pure and unit literal detection, yields to an even more efficient DQBF solver than iDQ.

Apart from the solving technique we use, preprocessing techniques can speed up the solving significantly. Therefore it is worth to investigate if well-known SAT and QBF preprocessing techniques can be adapted to DQBF.

References

1. V. Balabanov, H. K. Chiang, and J. R. Jiang. Henkin quantifiers and boolean formulae. In *Proc. SAT'12*, 2012.
2. V. Balabanov, H. K. Chiang, and J. R. Jiang. Henkin quantifiers and boolean formulae: A certification perspective of DQBF. *Theoretical Computer Science*, 2013.
3. B. Finkbeiner and L. Tentrup. Fast DQBF refutation. In *Proc. SAT 2014*, pages 243–251, 2014.
4. B. Finkbeiner and L. Tentrup. Fast DQBF refutation. In *Proc. SAT'14*, 2014.
5. A. Fröhlich, G. Kovásznai, and A. Biere. A DPLL algorithm for solving DQBF. In *Proc. POS'12*, 2012.
6. A. Fröhlich, G. Kovásznai, A. Biere, and H. Veith. iDQ: Instantiation-based DQBF solving. In *Proc. POS 2014, aff. to SAT 2014*, pages 103–116, 2014.
7. K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, and B. Becker. Equivalence checking for partial implementations revisited. In *Proc. MBMV'13*, pages 61–70, 2013.
8. K. Gitina, S. Reimer, M. Sauer, R. Wimmer, C. Scholl, and B. Becker. Equivalence checking of partial designs using dependency quantified boolean formulae. In *Proc. ICCD'13*, pages 396–403, 2013.
9. K. Gitina, R. Wimmer, S. Reimer, M. Sauer, C. Scholl, and B. Becker. Solving DQBF through quantifier elimination. In *Proc. DATE 2015*, pages 1617–1622. EDA Consortium, 2015.
10. K. Korovin. Instantiation-based automated reasoning: From theory to practice. In *Proc. CADE'09*, pages 163–166, 2009.
11. K. Korovin. Inst-Gen - a modular approach to instantiation-based automated reasoning. In *Programming Logics*, pages 239–270, 2013.
12. G. Kovásznai, A. Fröhlich, and A. Biere. On the complexity of fixed-size bit-vector logics with binary encoded bit-width. In *Proc. SMT'12*, 2012.
13. G. Peterson, J. Reif, and S. Azhar. Lower bounds for multiplayer noncooperative games of incomplete information, 2001.
14. G. L. Peterson and J. H. Reif. Multiple-person alternation. In *Proc. FOCS'79*, pages 348–363, 1979.
15. C. M. Wintersteiger, Y. Hamadi, and L. de Moura. Efficiently solving quantified bit-vector formulas. In *Proc. FMCAD'10*, 2010.

2 Old Challenges and New Solutions: a Comprehensive Assessment of State-of-the-Art QBF Solvers

Paolo Marin, Massimo Narizzano, Luca Pulina, Armando Tacchella and Enrico Giunchiglia
Old Challenges and New Solutions: a Comprehensive Assessment of State-of-the-Art QBF Solvers

Old Challenges and New Solutions: a Comprehensive Assessment of State-of-the-Art QBF Solvers

P. Marin[2], M. Narizzano[1], L. Pulina[3], A. Tacchella[1], and E. Giunchiglia[1]

[1] DIBRIS, Università di Genova, Via Opera Pia, 13 – 16145 Genova – Italy
{giunchiglia,narizzano,tacchella}@unige.it
[2] Lehrstuhl für Rechnerarchitektur, Georges-Köhler-Allee 051 – 79110 Freiburg i.B. – Germany marin@informatik.uni-freiburg.de
[3] POLCOMING, Università di Sassari, Viale Mancini n. 5 – 07100 Sassari – Italy
lpulina@uniss.it

Since the first QBF Evaluation (QBFEVAL'03) [1], whose aim was to assess the relatively young state of the art in the QBF reasoning field, almost every year an evaluation event was organized. The purpose of that was to measure the progress in QBF reasoning techniques and encourage the submission of new problems which could be encoded in QBF. A report of the last QBFEVAL event in the series was published in [2]. After more than a decade of new solvers being developed and new challenging problems being proposed, we believe that QBFEVAL and, more recently, QBF Gallery [3] events offer a series of snapshots about QBF solving and related aspects, but somehow fail to provide a long-term picture about what has been achieved, which of the techniques proposed are still worth considering, and which problems are still relevant for current QBF solvers. In this work we gather numerous results which enable us to assess the contributions of complete off-the-shelf QBF solving tools to the state-of-the-art considering the whole course of QBFEVAL and QBF Gallery evaluations and competitions, and exposing the result in a historical perspective. We are then able to suggest potential research directions for solving older and actual challenging problems. In the following, we list the solvers and describe the problems we used in our experiments, then we present the results, and conclude with some final remarks.

To accomplish our task we considered some *legacy solvers* (S-LEGACY in the following), i.e., tools that were proposed in the literature, but are not considered in more recents comparative events and *new solvers* (S-NEW in the following), i.e., all the other tools that we consider and which are not legacy. In particular, out of 8 solvers considered, the legacy ones are AIGSOLVE [4], the only AIG-based QBF solver; AQME [5], a multi-engine tool whose back-end solvers were released in 2006; QUANTOR [6], QuBE [7], and sKizzo [8], which implement key QBF solving techniques like resolution and expansion, DLL-search, and skolemization, respectively; lastly, STRUQS [9], which dynamically combines very different solution techniques. These tools are chosen among winners of at least one category in the past QBFEVAL events, conditioned to their maintenance ending before 2010. The set of new solvers is assembled by including the winners of the last QBF Gallery 2014: DEPQBF [10], GHOSTQ [11, 12] and RAREQS [11]. We did not consider HIQQER [13] because we could not find a version available for download. We define the "state-of-the-art" (SOTA) solver abstraction, i.e., the ideal system that always fares the best time among the systems in our portfolio. Likewise SOTA-LEGACY and SOTA-NEW abstract from legacy and new solvers, to score the global performace of solvers in S-LEGACY vs. those in S-NEW. As for problems, we consider three pools assembled for previous evaluations. In particular, we consider from [13] (*i*) QBF Gallery 2014 Track 1 (276 instances) and (*ii*) Track 2 (735 instances), and (*iii*) challenging formulas which are those classified as "Hard" (unsolved) and "Medium-Hard" (solved by one tool only) in the QBFEVAL evaluations from 2004 to 2010. These are then split by year. Overall, the testset is purposefully biased towards recently submitted instances, in order to (try to) assess legacy solvers on problems that are probably "unseen" to them, i.e., for which their developers did not have a chance to optimize the solver. On the other hand, group (*iii*) lets us assess whether the progress advertised by more recent evaluations is due to novel solving techniques, or to the fact that some hard problems were no longer evaluated. The tools were fed with the QBF formulas in their original format, i.e., we made no external preprocessing. Yet, some tools apply preprocessing techniques before the complete solving phase.

Considering the pool QBF Gallery Track 1, only 6 solvers out of 9 were able to solve at least 25% of the test set. By ranking the solvers according to the number of problems solved, the first is AIGSOLVE, which can solve about 42% of the test set, followed by QUBE and AQME, closely followed by GHOSTQ. Taking the above cited abstractions, SOTA was able to cope with about 73% of QBFG-T1. Its main contributors are AIGSOLVE, DEPQBF, and RAREQS.We also report that SOTA-LEGACY solves about 31% more problems than SOTA-NEW. Considering the pool QBF Gallery 2014 Track 2, which was partitioned into 6 families, AIGSOLVE, RAREQS, and QUANTORshow 3 times into the top-three ranking, AQME, DEPQBF, GHOSTQ, and STRUQS twice, and QUBE once. RAREQS and STRUQS can uniquely solve 16 problems, AIGSOLVE 12. Each of the remaining solvers less than 4. Lastly, we consider the challenging formulas from the past (6) QBFEVAL events: AIGSOLVE is always in the top-three and can solve uniquely 114 formulas, GHOSTQ and QUBE get the best rankings 5 times and can uniquely solve resp. 114 and 47 instances. Notice that RAREQS appears only once, but can solve uniquely 71 formulas. Overall, 2 out of 3 systems are always S-LEGACY, only in 2006 and 2010 the best solver is GHOSTQ. SOTA is able to solve 83% of the 2006 dataset, in other cases no more than 75%. With the notable exception of 2010, SOTA-LEGACY outperforms SOTA-NEW for each year. More details of this analysis are listed in the Appendix.

In the paper we have shown the results of a massive evaluation of QBF solvers and benchmarks from an historical perspective, and what emerges is that new systems are the main contributors of a SOTA solvers, yet comparing the SOTA-LEGACY and SOTA-NEW abstractions we have also shown that legacy systems still outperform the new ones in many problem categories. Therefore, we believe that it would be interesting to fuse legacy techniques into new systems in order to really push forward the state of the art in the QBF arena.

References

1. Berre, D.L., Simon, L., Tacchella, A.: Challenges in the QBF arena: the SAT'03 evaluation of QBF solvers. In: Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT 2003). Volume 2919 of Lecture Notes in Computer Science., Springer Verlag (2003) 468–485
2. Peschiera, C., Pulina, L., Tacchella, A., Bubeck, U., Kullmann, O., Lynce, I.: The seventh qbf solvers evaluation (qbfeval10). In: Theory and Applications of Satisfiability Testing–SAT 2010, Springer Berlin Heidelberg (2010) 237–250
3. F. Lonsing, M. Seidl, A.V.G.: QBF gallery 2013 (2013) `http://www.kr.tuwien.ac.at/events/qbfgallery2013/`.
4. Pigorsch, F., Scholl, C.: An aig-based qbf-solver using sat for preprocessing. In: Design Automation Conference (DAC), 2010 47th ACM/IEEE, IEEE (2010) 170–175
5. Pulina, L., Tacchella, A.: A self-adaptive multi-engine solver for quantified Boolean formulas. Constraints **14**(1) (2009) 80–116
6. Biere, A.: Resolve and Expand. In: Seventh Intl. Conference on Theory and Applications of Satisfiability Testing (SAT'04). Volume 3542 of LNCS., Springer Verlag (2005) 59–70
7. Giunchiglia, E., Marin, P., Narizzano, M.: Qube7.0. JSAT **7**(2-3) (2010) 83–88
8. Benedetti, M.: Evaluating QBFs via Symbolic Skolemization. In: Eleventh International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2004). Volume 3452 of Lecture Notes in Computer Science., Springer Verlag (2004)
9. Pulina, L., Tacchella, A.: A structural approach to reasoning with quantified Boolean formulas. In: 21st International Joint Conference on Artificial Intelligence (IJCAI 2009). (2009) 596–602
10. Lonsing, F., Biere, A.: Depqbf: A dependency-aware QBF solver. JSAT **7**(2-3) (2010) 71–76
11. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving qbf with counterexample guided refinement. In: Theory and Applications of Satisfiability Testing–SAT 2012, Springer Berlin Heidelberg (2012) 114–128
12. Klieber, W., Sapra, S., Gao, S., Clarke, E.: A non-prenex, non-clausal qbf solver with game-state learning. In: Theory and Applications of Satisfiability Testing–SAT 2010. Springer (2010) 128–142
13. Jordan, C., Seidl, M.: The QBF Gallery 2014 (2014)

Appendix

This Appendix shows the detailed results of our empirical analysis. We ran all the experiments on a cluster of Intel Xeon E31245 PCs at 3.30 GHz equipped with 64 bit Ubuntu 12.04. Each solver was limited to 600s of CPU time and 4GB of memory.

Solver	Total		True		False		Unique	
	#	Time	#	Time	#	Time	#	Time
AIGSolve	116	5333.01	56	2177.45	60	3155.56	22	1458.26
QuBE	106	8764.73	53	3997.78	53	4766.95	8	1195.58
aqme	97	3287.20	39	1098.00	58	2189.20	–	–
ghostq	91	4814.73	48	2912.38	43	1902.17	4	158.97
depqbf	88	2388.32	39	1163.15	49	1225.17	5	454.77
rareqs	79	2588.64	32	1593.25	47	995.39	6	787.33
sKizzo	51	948.81	18	556.76	33	392.06	–	–
quantor	50	1498.37	28	911.72	22	586.65	2	161.67
StruQS	43	6092.64	31	4052.98	12	2039.66	1	16.53

Table 1. Performance of the involved solvers on qbfg-t1. The table consists of nine columns that for each solver reports its name (column "Solver"), the total number of instances solved and the cumulative time to solve them (columns "#" and "Time", group "Total"), the number of instances found satisfiable and the time to solve them (columns "#" and "Time", group "True"), the number of instances found unsatisfiable and the time to solve them (columns "#" and "Time", group "False"), and, finally, the number of instances uniquely solved and the time to solve them (columns "#" and "Time", group "Unique"); a "–" (dash) means that the solver did not solve any instance. Finally, the table is sorted in descending order, according to the number of instances solved, and, in case of a tie, in ascending order according to the cumulative time taken to solve them.

Family	Solver	Total		True		False		Unique	
		#	Time	#	Time	#	Time	#	Time
bomb (132)	AIGSolve	83	1003.23	40	165.20	43	838.03	–	–
	rareqs	83	1420.61	34	165.41	49	1255.19	6	1094.71
	quantor	82	923.25	53	217.92	29	705.33	–	–
	aqme	80	674.38	53	345.02	27	329.36	–	–
	depqbf	67	2410.16	40	1693.36	27	716.79	–	–
	sKizzo	57	609.41	31	2.39	26	607.03	–	–
	ghostq	56	532.47	29	42.66	27	489.81	–	–
	QuBE	47	1168.86	23	470.47	24	698.39	–	–
	StruQS	36	1051.46	19	813.58	17	237.88	–	–
complexity (104)	rareqs	75	1559.65	29	466.77	46	1092.88	15	1148.51
	depqbf	49	1553.73	22	1086.35	27	467.38	–	–
	ghostq	42	1791.86	11	499.21	27	467.38	–	–
	QuBE	39	1273.95	19	277.87	20	996.09	–	–
	aqme	33	528.28	15	188.76	18	339.52	–	–
	quantor	26	170.44	11	11.29	15	159.14	–	–
	StruQS	21	1855.53	13	1677.81	8	177.72	–	–
	AIGSolve	15	70.26	7	12.24	8	58.02	–	–
	sKizzo	9	316.60	4	315.82	5	0.78	–	–
dungeon (107)	quantor	104	525.30	18	54.81	86	470.48	–	–
	aqme	104	1121.43	18	86.11	86	1035.32	–	–
	AIGSolve	87	1220.22	17	417.12	70	803.10	–	–
	rareqs	57	1870.73	18	54.89	39	1815.85	–	–
	depqbf	44	535.22	18	300.44	26	234.77	–	–
	QuBE	34	1429.60	7	212.89	27	1216.71	–	–
	ghostq	7	385.11	4	4.62	3	380.49	–	–
	sKizzo	2	0.99	–	–	2	0.99	–	–
	StruQS	1	21.96	1	21.96	–	–	–	–
hardness (114)	StruQS	88	7826.42	1	372.74	87	7453.68	12	3033.19
	QuBE	76	1346.11	–	–	76	1346.11	2	328.75
	ghostq	51	2649.30	2	239.56	49	2409.74	1	224.22
	aqme	50	265.14	–	–	50	265.14	–	–
	rareqs	14	1431.05	–	–	14	1431.05	–	–
	AIGSolve	12	2038.84	–	–	12	2038.84	–	–
	depqbf	8	617.99	–	–	8	617.99	–	–
	quantor	–	–	–	–	–	–	–	–
	sKizzo	–	–	–	–	–	–	–	–
planning (147)	AIGSolve	147	2371.36	38	114.02	109	2257.34	10	861.70
	rareqs	137	1093.01	38	125.66	99	967.35	–	–
	quantor	131	6750.13	37	122.68	94	6627.44	–	–
	aqme	123	9263.25	37	464.97	86	8798.28	–	–
	sKizzo	74	71.57	34	24.02	40	47.55	–	–
	depqbf	57	5134.24	29	1876.90	28	3257.34	–	–
	QuBE	14	1270.35	12	743.61	2	526.74	–	–
	ghostq	11	2155.26	8	1420.71	3	734.55	–	–
	StruQS	4	1229.67	4	1229.67	–	–	–	–
testing (131)	aqme	71	2675.64	64	2339.68	7	335.95	1	3.00
	StruQS	65	1770.09	63	1488.09	2	282.00	4	236.18
	depqbf	57	692.38	46	672.96	11	19.42	2	359.15
	AIGSolve	51	4194.44	46	4163.65	5	30.79	2	11.88
	QuBE	41	765.08	31	734.85	10	30.23	1	1.24
	rareqs	34	428.00	22	317.04	12	110.95	1	0.53
	ghostq	32	269.13	29	66.10	3	203.03	–	–
	quantor	26	121.15	25	110.52	1	10.63	–	–
	sKizzo	1	0.02	1	0.02	–	–	–	–

Table 2. Performances of QBF solvers on qbfg-t2: The table is split in six horizontal parts, one for each family. The first column contains families names, as well as its total amount of instances. The rest of the table is organized as Table 1.

Year	Solver	Total		True		False		Unique	
		#	Time	#	Time	#	Time	#	Time
2004 (167)	AIGSolve	62	2658.96	46	1781.89	16	877.07	15	1401.38
	ghostq	51	541.62	36	370.74	15	170.88	18	194.05
	QuBE	28	4783.92	23	3566.83	5	1217.09	6	626.75
	sKizzo	23	749.18	16	691.76	7	57.42	–	–
	StruQS	14	1081.04	9	885.70	5	195.34	–	–
	quantor	13	916.32	11	905.49	2	10.83	–	–
	depqbf	2	192.64	–	–	2	192.64	–	–
	rareqs	1	0.24	–	–	1	0.24	1	0.24
2005 (168)	AIGSolve	43	3386.36	32	2119.95	11	1266.41	18	1861.10
	ghostq	27	217.17	17	46.44	10	170.43	13	190.25
	QuBE	19	2653.56	16	2365.20	3	288.37	4	359.28
	rareqs	8	5.57	–	–	8	5.57	3	0.54
	StruQS	8	921.27	6	687.03	2	234.24	–	–
	sKizzo	7	595.39	7	595.39	–	–	–	–
	quantor	5	144.64	4	137.29	1	7.34	–	–
	depqbf	1	243.79	–	–	1	243.79	–	–
2006 (103)	ghostq	80	1577.10	80	1577.10	–	–	5	15.33
	AIGSolve	71	608.62	63	432.35	8	176.27	6	40.29
	QuBE	61	1108.04	57	506.54	4	601.49	1	190.78
	StruQS	37	373.31	36	298.82	1	74.48	–	–
	rareqs	4	277.89	1	102.27	3	175.62	–	–
	depqbf	1	22.85	–	–	1	22.85	–	–
	quantor	–	–	–	–	–	–	–	–
	sKizzo	–	–	–	–	–	–	–	–
2007 (281)	QuBE	88	7239.68	13	2729.53	75	4510.16	13	643.41
	rareqs	83	2706.28	10	1905.49	73	800.79	31	1741.36
	AIGSolve	61	765.34	36	547.74	25	217.61	34	445.93
	depqbf	50	2902.71	6	378.32	44	2524.39	5	264.06
	ghostq	49	1699.99	41	360.23	8	1339.76	19	275.34
	quantor	14	1302.44	11	1225.27	3	77.17	6	517.17
	StruQS	11	2051.88	11	2051.88	–	–	–	–
	sKizzo	5	953.55	1	50.44	4	903.11	–	–
2008 (961)	AIGSolve	335	13128.70	237	8439.00	98	4689.70	135	8250.56
	ghostq	304	8299.71	257	5325.55	47	2974.16	49	1493.92
	QuBE	198	19753.10	71	13501.39	127	6251.71	21	2320.72
	rareqs	126	4220.58	16	2744.06	110	1476.52	36	2255.13
	depqbf	96	4626.18	7	495.77	89	4130.41	4	261.53
	sKizzo	57	3599.21	26	1664.18	31	1935.02	1	275.69
	StruQS	50	5458.74	47	4844.87	3	613.87	–	–
	quantor	19	1646.95	18	1621.94	1	25.02	15	1166.44
2010 (96)	ghostq	29	591.60	25	281.30	4	309.76	10	342.94
	AIGSolve	22	282.18	20	262.94	2	19.24	5	208.02
	sKizzo	8	564.39	8	564.39	–	–	–	–
	QuBE	4	210.40	1	19.08	3	191.32	2	50.80
	rareqs	2	17.41	–	–	2	17.41	1	0.50
	depqbf	1	22.85	–	–	1	22.85	–	–
	quantor	–	–	–	–	–	–	–	–
	StruQS	–	–	–	–	–	–	–	–

Table 3. Performances of QBF solvers on challenging instances: The table is split in six horizontal parts, one for each family. The first column contains the QBFEVAL-related year families names, as well as its total amount of instances. The rest of the table is organized as Table 1.

	2004	**2005**	**2006**	**2007**	**2008**	**2010**
SOTA	88	65	85	207	601	37
SOTA-NEW	53	34	75	148	393	**30**
SOTA-LEGACY	**69**	**49**	**77**	**150**	**492**	26

Table 4. Performance of state-of-the-art solvers on CHALLENGING formulas. The table is organized as follows. The first column reports considered SOTA(s), while the remaining columns denote the pool of challenging formulas. In cells is shown the total amount of solved instances by the related SOTA. In bold we denote the best performance between SOTA-NEW and SOTA-LEGACY.

3 Encodings of Reactive Synthesis

Peter Faymonville, Bernd Finkbeiner, Markus N. Rabe and Leander Tentrup Encodings of Reactive Synthesis

Encodings of Reactive Synthesis

Peter Faymonville[1], Bernd Finkbeiner[1], Markus N. Rabe[2], and Leander Tentrup[1]

[1] Saarland University
[2] University of California, Berkeley

Abstract. In this talk, we present and compare several encodings of the bounded synthesis problem for linear-time temporal logic (LTL). The bounded synthesis problem for natural bound n is to decide whether there exists a strategy (generated by a transition system with n states) that satisfies an LTL specification—and in the positive case to construct such a strategy. We give an overview of previously studied encodings that use SMT, antichains, and BDDs, as well as new encodings using (quantified) propositional logics. Furthermore, we evaluate the constraint based approaches (SMT, SAT, QBF, etc.) with respect to solving time and implementation quality using certifying theory solvers.

1 Extended Abstract

Synthesis is the task of creating correct-by-construction implementations from formal specifications, thus avoiding the need for manual implementations. In recent years, synthesis has gained a lot of attention and modern synthesis tools emerged [1–4]. Last year, this development culminated in the first competition of synthesis tools [7].

In this talk, we consider the bounded synthesis [6] problem, that is the problem of synthesizing a strategy of size n, such that the strategy satisfies the LTL specification φ. For an LTL formula φ, we assume a partitioning into variables O that are controllable by the strategy and variables I that are given by the environment. A strategy $f : (2^I)^* \to 2^O$ maps sequences of valuations from the environment to a valuation of the controllable variables. We represent strategies as finite-state transition systems and identify the size of a strategy with the size of the transition system.

Given such a specification φ, we build a universal co-Büchi automaton $\mathcal{U}_\varphi$. A transition system is accepted by $\mathcal{U}_\varphi$ if each run in the unique run graph on $\mathcal{U}_\varphi$ has only finitely many visits to the rejecting states of $\mathcal{U}_\varphi$. The acceptance of a finite-state transition systems on $\mathcal{U}_\varphi$ can be characterized by the existence of an annotation on the product of transition system and automaton [6]. This annotation maps a pair (s, q), where s is a state in the transition system and q is a state in the automaton, to the number of maximal visits to rejecting states on all runs that lead to (s, q). In the original formulation [5], the labeling and transition functions of the transition system as well as the correct annotation were encoded as SMT constraints. The SMT encoding uses uninterpreted functions and a theory that supports ordering constraints (like the theory of integers).

We show how to modify the encoding to use only uninterpreted functions and propositional constraints. Based on this modification, we give a reduction to the satisfiability problem for quantified Boolean formulas (QBF) and propositional satisfiability (SAT). Let $S = \{s_1, \ldots, s_n\}$ be the number of states in the transition system. The quantifiers in the QBF encoding make the transition function symbolic in the inputs, i.e., a part of the quantification header has the form $\forall \boldsymbol{i}.\, \exists \boldsymbol{t}_{s,s'}$ for all $s, s' \in S$, meaning that there is a transition from state s to s' in the transition system, if the Skolem function $f_{t_{s,s'}}$ evaluates to true for the given environment input $\boldsymbol{i}$.

We investigated experimentally which encoding is best given the current state of solver technology. With regard to the SMT encoding, we compare different theories to encode the ordering constraints and different levels of quantifications. For the propositional encoding, we compare the QBF encoding, which uses quantification for input-symbolic transition functions, with the variant that unrolls the quantification to a pure SAT encoding.

Modern solvers have the ability to construct models from satisfiable queries, e.g., Skolem functions in the case of QBF. As the existence of a transition system is encoded in the bounded synthesis query, we can easily construct an implementation using certifying solvers. We compare the quality of these implementations with respect to the different encodings, ranging from models generated by an SMT solver, Skolem functions extracted from QBF proofs, and assignments given by a SAT solver.

References

1. Bloem, R., Egly, U., Klampfl, P., Könighofer, R., Lonsing, F.: SAT-based methods for circuit synthesis. In: Proceedings of FMCAD. pp. 31–34 (2014)
2. Bloem, R., Gamauf, H., Hofferek, G., Könighofer, B., Könighofer, R.: Synthesizing robust systems with RATSY. In: Proceedings of SYNT. pp. 47–53 (2012)
3. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.: Acacia+, a tool for LTL synthesis. In: Proceedings of CAV. pp. 652–657 (2012)
4. Ehlers, R.: Symbolic bounded synthesis. Formal Methods in System Design 40(2), 232–262 (2012)
5. Finkbeiner, B., Schewe, S.: SMT-based synthesis of distributed systems. In: Proceedings of AFM (2007)
6. Finkbeiner, B., Schewe, S.: Bounded synthesis. STTT 15(5-6), 519–539 (2013)
7. Jacobs, S., Bloem, R., Brenguier, R., Ehlers, R., Hell, T., Könighofer, R., Pérez, G.A., Raskin, J., Ryzhyk, L., Sankur, O., Seidl, M., Tentrup, L., Walker, A.: The first reactive synthesis competition (SYNTCOMP 2014). CoRR abs/1506.08726 (2015), http://arxiv.org/abs/1506.08726

4 Model Finding for Recursive Functions in SMT

Andrew Reynolds, Jasmin Christian Blanchette and Cesare Tinelli Model Finding for Recursive Functions in SMT

Model Finding for Recursive Functions in SMT*

Andrew Reynolds[1], Jasmin Christian Blanchette[2,3], and Cesare Tinelli[4]

[1] École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
[2] Inria Nancy & LORIA, Villers-lès-Nancy, France
[3] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[4] Department of Computer Science, University of Iowa, USA

Many solvers based on SMT (satisfiability modulo theories) can reason about quantified formulas using incomplete instantiation-based methods [3, 7]. These methods work well in the context of proving (i..e, showing unsatisfiability), but they are of little help for finding models (i.e., showing satisfiability). Often, a single universal quantifier in one of the axioms of a problem is enough to prevent the discovery of models.

In the past few years, techniques have been developed to find models for quantified formulas in SMT. Ge and de Moura [4] introduced a complete instantiation-based procedure for formulas in the essentially uninterpreted fragment. This fragment is limited to universally quantified formulas where all variables occur as direct subterms of uninterpreted functions—e.g., $\forall x.\ \mathsf{f}(x) \approx \mathsf{g}(x) + 5$. Other syntactic criteria extend this fragment slightly, including cases when variables occur as arguments of arithmetic predicates. Subsequently, Reynolds et al. [8,9] introduced techniques for finding finite models for quantified formulas over uninterpreted types and types having a fixed finite interpretation. These techniques can find a model for a formula such as $\forall x, y : \tau.\ x \approx y \vee \neg\, \mathsf{f}(x) \approx \mathsf{f}(y)$, where τ is an uninterpreted type.

Unfortunately, none of these fragments can accommodate the vast majority of quantified formulas that correspond to recursive function definitions: The essentially uninterpreted fragment does not allow the argument of a recursive function to be used inside a complex term on the right-hand side, whereas the finite model finding techniques are not applicable for functions over infinite domains such as the integers or algebraic datatypes. A simple example where both approaches fail is $\forall x : \mathsf{Int}.\ \mathsf{ite}\big(x \leq 0,\ \mathsf{p}(x) \approx 1,\ \mathsf{p}(x) \approx 2 * \mathsf{p}(x-1)\big)$. This state of affairs is unsatisfactory, given the frequency of recursive definitions in practice and the impending addition of a dedicated command for introducing them, `define-fun-rec`, to the SMT-LIB standard [1].

We present a method for translating formulas involving recursive function definitions into formulas where finite model finding techniques can be applied. The recursive functions must meet a semantic criterion to be admissible. This criterion is met by well-founded (terminating) recursive function definitions.

We define a translation for a class of formulas involving admissible recursive function definitions. The main insight is that a recursive definition $\forall x : \tau.\ \mathsf{f}(x) \approx t$ can be translated to $\forall a : \alpha_\tau.\ \mathsf{f}(\gamma_{\mathsf{f}}(a)) \approx t[\gamma(a)/x]$, where α_τ is an uninterpreted abstract type and γ_{f} converts the abstract type to the concrete type. The translation preserves satisfiability and unsatisfiability, and makes finite model finding possible for problems in this class.

*This research is partially supported by the Inria technological development action "Contre-exemples Utilisables par Isabelle et Coq" (CUIC).

Our empirical evaluation on benchmarks from the IsaPlanner proof planner [5] and the Leon verifier [2] provides evidence that this translation improves the effectiveness of the SMT solvers CVC4 and Z3 for finding counterexamples to verification conditions. The approach is implemented as a preprocessor in CVC4 .

In future work, it would be interesting to evaluate the approach against other counterexample generators, notably Leon and Nitpick, and enrich the benchmark suite with problems exercising CVC4's support for coalgebraic datatypes [6]. We also plan to integrate CVC4 as a counterexample generator in proof assistants. Finally, future work could also include identifying further sufficient conditions for admissibility, thereby enlarging the applicability of the translation scheme presented here.

This abstract is based on a regular submission to the SMT 2015 workshop, which itself is based on a longer technical report. Both are available online.[1]

References

[1] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB standard—Version 2.5. `http://smt-lib.org/language.shtml`, to appear.

[2] R. Blanc, V. Kuncak, E. Kneuss, and P. Suter. An overview of the Leon verification system—Verification by translation to recursive functions. In *Scala '13*. ACM, 2013.

[3] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In F. Pfenning, editor, *CADE-21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.

[4] Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiability modulo theories. In *CAV '09*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[5] M. Johansson, L. Dixon, and A. Bundy. Case-analysis for rippling and inductive proof. In *ITP 2010*, pages 291–306, 2010.

[6] A. Reynolds and J. C. Blanchette. A decision procedure for (co)datatypes in SMT solvers. In A. Felty and A. Middeldorp, editors, *CADE-25*, LNCS. Springer, 2015.

[7] A. Reynolds, C. Tinelli, and L. de Moura. Finding conflicting instances of quantified formulas in SMT. In *FMCAD 2014*, pages 195–202. IEEE, 2014.

[8] A. Reynolds, C. Tinelli, A. Goel, and S. Krstić. Finite model finding in SMT. In N. Sharygina and H. Veith, editors, *CAV 2013*, volume 8044 of *LNCS*, pages 640–655. Springer, 2013.

[9] A. Reynolds, C. Tinelli, A. Goel, S. Krstić, M. Deters, and C. Barrett. Quantifier instantiation techniques for finite model finding in SMT. In M. P. Bonacina, editor, *CADE-24*, volume 7898 of *LNCS*, pages 377–391. Springer, 2013.

[1] `http://lara.epfl.ch/~reynolds/SMT2015-recfun/`

5 Congruence Closure with Free Variables

Haniel Barbosa and Pascal Fontaine Congruence Closure with Free Variables

Congruence Closure with Free Variables (Work in Progress)

Haniel Barbosa and Pascal Fontaine

LORIA–INRIA, Nancy, France
{Haniel.Barbosa, Pascal.Fontaine}@inria.fr

Abstract. This paper presents preliminary work on the definition of a general framework for handling quantified formulas in SMT solving. Its focus is on the derivation of instances conflicting with a ground context, redefining the approach introduced in [11]. An enhanced version of the classical congruence closure algorithm, able to handle free variables, is presented.

1 Introduction

SMT solvers (see [3] for a general presentation of SMT) are extremely efficient at handling large ground formulas with interpreted symbols, but they still struggle to manage quantified formulas. Quantified first-order logic is best handled with resolution-based theorem proving [10]. Although there are first attempts to unify SMT and resolution [8], the main approach used in SMT is still *instantiation*: quantified formulas are freed from quantifiers and are refuted with the help of decision procedures for ground formulas. Even though such techniques as E-matching [6] and model based quantifier instantiation (MBQI) [7] have been used successfully in state-of-the-art solvers, there are still far more instances produced than would actually be needed. Reynolds et al. [11] present an alternative approach: instances are generated such that they are conflicting, by construction, with the ground context produced by the solver. This provides a strong advantage due to its finer instantiation guideline: less instances are needed to prove a formula unsatisfiable, as their experimentation data indicates.

Since their method is restricted to problems in pure first-order logic with equality, it has strong reminiscence of the (non-simultaneous) rigid E-unification problem. Unifying two expressions with free variables modulo a set of equations is equivalent, as shown in [12], to finding instances conflicting with a ground context. In this preliminary work we try to exploit this relation while revisiting the technique: defining an enhanced version of the classic congruence closure procedure capable of handling *free variables* unification accordingly. We aim for a better integration of ground conflicting instance generation and E-matching techniques within core SMT algorithms (namely the congruence closure decision procedure), with MBQI being used as last resort.

2 Notations and basic definitions

We recall here some usual notions of first-order logic. For simplicity, we work in mono-sorted (in contrast to many-sorted) languages. A *first-order language* is a tuple $\mathscr{L} = \langle \mathcal{X}, \mathcal{P}, \mathcal{F} \rangle$ where $\mathcal{X}$, $\mathcal{P}$ and $\mathcal{F}$ are enumerable sets of *variable*, *predicate* and *function symbols*, respectively. Every function and predicate symbol has an arity. Nullary functions and predicates are called *constants* and *propositions*, respectively. *Formulas* and *terms* are generated by

$$t ::= x \mid f(t, \ldots, t) \qquad \varphi ::= t \approx t \mid p(t, \ldots, t) \mid \neg\varphi \mid \varphi \vee \varphi \mid \forall x_1 \ldots x_n.\varphi$$

in which $x, x_1, \ldots, x_n \in \mathcal{X}$, $p \in \mathcal{P}$ and $f \in \mathcal{F}$. The symbol $\approx$ stands for *equality.* We will mostly work in predicate-free languages. The usual conventions for *disequality*, *existential quantification* and connectives are assumed. In particular we use $\wedge$ for conjunction. The terms in a formula φ are denoted by $\mathbf{T}^{\varphi}$. *Atoms* are formulas of the form $t \approx t$ and $p(t, \ldots, t)$. A *literal* is an atom or its negation. A subformula *appears positively (resp. negatively) in* φ iff it is under an even (resp. odd) number of negations. A subformula $\forall x_1 \ldots x_n.\psi$ of φ is *weakly (resp. strongly) quantified* iff it appears positively (resp. negatively) in φ. A formula is in *Skolem form* iff it has no strong quantifiers.

Whenever convenient, an enumeration of symbols $s_1, \ldots, s_n$ will be represented as $\mathbf{s}$. Analogously, an enumeration of binary operations s_1 *op* $t_1, \ldots, s_n$ *op* t_n is represented as $\mathbf{s}$ ***op*** $\mathbf{t}$.

Terms and formulas without variables are denoted *ground*. *Free* and *bound* variables are defined in the usual way. A *substitution* is a function from variables to terms such that $\sigma = \{\mathbf{x} \mapsto \mathbf{t}\}$ maps each variable $x_i \in \mathbf{x}$ into the term $t_i \in \mathbf{t}$ and every other variable not in $\mathbf{x}$ to itself. $\varphi\sigma$ (resp. $t\sigma$) denotes the recursive application of σ in the structure of the formula (resp. term) in a capture-avoiding way, while not substituting bound variables. The *domain of* σ is the set $\mathit{dom}(\sigma) = \{x \mid x \in \mathcal{X} \text{ and } x\sigma \neq x\}$, while the *range of* σ is $\mathit{ran}(\sigma) = \{x\sigma \mid x \in \mathit{dom}(\sigma)\}$. σ is a *ground substitution* iff every term in $\mathit{ran}(\sigma)$ is ground. The *composition* of two substitutions σ_1 and σ_2 is defined such that $\sigma_1 \circ \sigma_2 = \{x \mapsto (x\sigma_2)\sigma_1 \mid x \in \mathcal{X}\}$. It is commutative for ground substitution, that is, everywhere in this text. A formula ψ_1 is an *instance* of a formula ψ_2 iff there is a substitution σ such that $\psi_1 = \psi_2\sigma$.

An *interpretation* is represented as a tuple $\mathcal{M} = \langle \mathcal{D}, \mathcal{I}, \mathcal{V} \rangle$, in which $\mathcal{D}$ is a non-empty *domain*; $\mathcal{I}$ is a function mapping each function symbol f to a function $f^{\mathcal{I}} : \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \to \mathcal{D}$ and each predicate symbol p to a predicate $p^{\mathcal{I}} : \mathcal{D}_1 \times \cdots \times \mathcal{D}_n \to \{\top, \bot\}$; $\mathcal{V}$ is a *valuation* assigning an element of $\mathcal{D}$ to every variable. $\mathcal{M}$ assigns a value in $\mathcal{D}$ to every term t, denoted $[\![t]\!]^{\mathcal{M}}$, and a *truth value* $(\top, \bot)$ to every formula φ, denoted $[\![\varphi]\!]^{\mathcal{M}}$, through the usual recursive definition. $\mathcal{M}$ *satisfies* φ, written $\mathcal{M} \models \varphi$, iff $[\![\varphi]\!]^{\mathcal{M}} = \top$, in which case $\mathcal{M}$ is a *model* of φ. φ is *satisfiable* iff it has a model. It is *unsatisfiable* otherwise. A set of formulas Γ *entails* a set of formulas Δ, written $\Gamma \models \Delta$, iff all interpretations satisfying every $\varphi \in \Gamma$ also satisfy every $\psi \in \Delta$.

An interpretation $\mathcal{M}$ *propositionally (resp. groundly) satisfies* φ, written $\mathcal{M} \models^{\mathrm{p}} \varphi$ (resp. $\mathcal{M} \models^{\mathrm{g}} \varphi$), iff it is a model of its *propositional (resp. ground)* *abstraction*, which is a propositional (resp. ground) formula where every non-propositional atom (resp. quantified subformula) is mapped into a fresh propositional symbol. These notions carry out accordingly to other definitions.

3 Congruence Closure with Free Variables

Modern SMT solvers handle quantified formulas using instantiation. That is, while checking the satisfiability of a formula φ in a theory $\mathcal{T}$, the ground abstraction of the formula is given to the ground SMT solver, which provides a groundly $\mathcal{T}$-satisfiable set of literals. These abstracted literals correspond to (concrete) ground literals $\mathcal{L}$ and quantified[1] formulas $\mathcal{Q}$, with $\mathcal{L} \cup \mathcal{Q} \models \varphi$. If $\mathcal{L} \cup Q$ is $\mathcal{T}$-satisfiable, then so is φ. This satisfiability check could be done by a model finder. As in [11] we focus here on the problem of finding instances from $\mathcal{Q}$ that groundly refute $\mathcal{L}$. Repeatedly adding such instances conjunctively to the original formula, combined with MBQI [7], provides a practical and powerful procedure to deal with unsatisfiable formulas in SMT. We believe that better integrating ground conflicting instance generation [11] and MBQI within the core SMT algorithm will generate new heuristics and ideas for a layered approach to quantifier handling.

It is assumed, for simplicity, that $\mathcal{L}$ contains only equality literals and that each formula in $\mathcal{Q}$ is of the form $\forall \mathbf{x}.\psi$, where ψ is quantifier-free and consists of a single clause of non-ground equality literals. We further assume that $\mathcal{T}$ is the empty theory, that is, we work in pure first-order logic with equality. Since any unsatisfiable formula in pure first-order logic is also unsatisfiable in any theory, the techniques here can also be seen as an incomplete algorithm for SMT, whatever the background theories.

Problem description Given some formula $\forall \mathbf{x}.\psi \in \mathcal{Q}$, if there exists a substitution σ such that $\mathcal{L} \models \neg\psi\sigma$, then, by Lemma 1, there is a ground substitution σ' such that $ran(\sigma') \subseteq \mathbf{T}^{\mathcal{L}}$. Such a substitution refutes $\mathcal{L}$ and is called *ground conflicting.*

As shown in [12], computing a ground conflicting substitution is equivalent to solving a non-simultaneous rigid E-unification problem. Therefore it becomes the problem of finding a conjunctive set Σ of equalities $x \approx t$ ($x \in \mathrm{FV}(\psi)$ and $t \in \mathbf{T}^{\mathcal{L}}$) such that $\mathcal{L} \models \neg\psi \wedge \Sigma$, each variable occurring at most once in Σ. Since this is a ground problem, classical SMT solving tools, i.e., *Congruence Closure* (see e.g. [9]) can be adapted to solve it: unification of free variables must be handled, associating variables in $\mathrm{FV}(\psi)$ to ground terms already occurring in $\mathcal{L}$.

Lemma 1. *Consider a ground formula φ and a formula ψ with free variables. If there exists a substitution σ such that $\varphi \models \psi\sigma$, then there is a ground substitution σ' such that $\varphi \models \psi\sigma'$ and* $\mathrm{ran}(\sigma') \subseteq \mathbf{T}^{\varphi}$.

[1] Assuming φ is Skolemized, we can safely assume $\mathcal{Q}$ only contains weakly quantified formulas.

Proof. Let $\mathcal{M}$ be model of φ such that its domain elements are the interpretation of terms in $\mathbf{T}^{\varphi}$. Since $\varphi \models \psi\sigma$, $\mathcal{M}$ is a also a model of $\psi\sigma$. Therefore, for each variable $x \in dom(\sigma)$ there is a term $t \in \mathbf{T}^{\varphi}$ such that $[\![x\sigma]\!]^{\mathcal{M}} = [\![t]\!]^{\mathcal{M}}$. Thus, a substitution $\sigma' = \{x \mapsto t \mid x \in dom(\sigma), t \in \mathbf{T}^{\varphi}, [\![x\sigma]\!]^{\mathcal{M}} = [\![t]\!]^{\mathcal{M}}\}$ fulfills the desired condition.

Algorithm CCFV

The CCFV procedure shown in Figure 1 derives, if any, a ground substitution σ such that $\mathcal{L} \wedge \psi\sigma$ is unsatisfiable. It computes a sequence of substitutions $\sigma_0, \ldots, \sigma_k$ such that, for $\neg\psi = l_1 \wedge \cdots \wedge l_k$,

$$\sigma_0 = \varnothing;\ \sigma_{i-1} \subseteq \sigma_i \text{ and } \mathcal{L} \models l_i\sigma_i$$

which guarantees that $\mathcal{L} \models \neg\psi\sigma_k$.

Example 1. Consider a set of literals $\mathcal{L} = \{f(c) \approx a,\ f(a) \approx b,\ f(a) \not\approx f(b)\}$ and a quantified formula $\forall x_1, x_2.\ (f(x_1) \not\approx a \vee f(x_2) \approx b)$. Successively evaluating $\neg\psi = (f(x_1) \approx a \wedge f(x_2) \not\approx b)$ yields $\sigma_1 = \{x_1 \mapsto c\}$ such that $\mathcal{L} \models (f(x_1) \approx a)\sigma_1$ and $\sigma_2 = \{x_1 \mapsto c, x_2 \mapsto b\}$ such that $\mathcal{L} \models (f(x_2) \not\approx b)\sigma_2$. Therefore $\sigma = \sigma_2$ is a ground conflicting substitution, since $\mathcal{L} \wedge \psi\sigma$ is groundly unsatisfiable.

Preliminaries $\mathfrak{C}$ and $\mathfrak{D}$ are initially, respectively, the set of equalities and of disequalities in $\mathcal{L}$.

Given a term $s \in \mathbf{T}^{\mathcal{L}\cup\{\psi\}}$, $[s]$ denotes the *congruence class* of s in the partition of $\mathbf{T}^{\mathcal{L}\cup\{\psi\}}$ induced by $\mathfrak{C}$, i.e., $[s] = \{t \mid t \in \mathbf{T}^{\mathcal{L}\cup\{\psi\}},\ \mathfrak{C} \models s \approx t\}$. Operations on substitutions are performed modulo the current partition, so that, e.g., $\theta \subseteq_{\mathfrak{C}} \sigma$ iff, for every $x \in dom(\theta)$, $\mathfrak{C} \models x\sigma \approx x\theta$, rather than requiring $x\sigma = x\theta$.

$\varDelta_{\mathbf{x}}$ denotes the set of *unfeasible substitutions*, i.e., $\delta \in_{\mathfrak{C}} \varDelta_{\mathbf{x}}$ iff there is no σ such that $\delta \subseteq_{\mathfrak{C}} \sigma$ and $\mathcal{L} \models \neg\psi\sigma$. It is initially empty.

$\textsc{sel}$ denotes a function mapping variables to themselves or ground terms, such that $\textsc{sel}(x) = x$ iff $[x]$ contains no ground terms, otherwise $\textsc{sel}(x)$ is some ground term $t \in [x]$.

Algorithm The *current instantiation* in $\mathfrak{C}$ is obtained depending on the congruence classes of $\mathbf{x}$ containing ground terms or not. Thus $\{x \mapsto \textsc{sel}(x) \mid x \in \mathbf{x}\}$ denotes the current instantiation in $\mathfrak{C}$.

For each $l \in \neg\psi$, the procedure $\textsc{Handle}$ checks its consistency w.r.t. $\mathfrak{C} \cup \mathfrak{D}$. If they are incompatible then the current instantiation is unfeasible, which triggers its addition to $\varDelta_{\mathbf{x}}$ and backtracking. Otherwise it tries to extend the current instantiation to some σ, not in $\varDelta_{\mathbf{x}}$, for which $\mathcal{L} \models l\sigma$. If it fails to do so, then again the current instantiation is unfeasible. If every literal in $\neg\psi$ is asserted successfully, the current instantiation represents a conflicting substitution and is outputed by CCFV. On the other hand, if $\varnothing$ is added to $\varDelta_{\mathbf{x}}$, there is no σ such that $\mathcal{L} \models \neg\psi\sigma$. So the procedure terminates without producing a ground conflicting substitution.

```
proc CCFV($\mathcal{L}, \psi$)
    $\mathfrak{C} \leftarrow \{s \approx t \mid s \approx t \in \mathcal{L}\}$;   $\mathfrak{D} \leftarrow \{s \not\approx t \mid s \not\approx t \in \mathcal{L}\}$;   $\Delta_{\mathbf{x}} \leftarrow \varnothing$        // Init
    foreach $l \in \neg\psi$ do
        if not(HANDLE($\mathfrak{C}, \mathfrak{D}, \Delta_{\mathbf{x}}, l$)) then
            $\Delta_{\mathbf{x}} \leftarrow \Delta_{\mathbf{x}} \cup \{\{x \mapsto \mathrm{SEL}(x) \mid x \in \mathbf{x}\}\}$
            if $\varnothing \in \Delta_{\mathbf{x}}$ then return $\varnothing$                  // No $\sigma$ s.t. $\mathcal{L} \models \neg\psi\sigma$
            RESET($\mathfrak{C}, \mathfrak{D}, \neg\psi$)                               // Backtracking
    return $\{x \mapsto \mathrm{SEL}(x) \mid x \in \mathbf{x}\}$                             // $\mathcal{L} \models \neg\psi\sigma$

proc HANDLE($\mathfrak{C}, \mathfrak{D}, \Delta_{\mathbf{x}},\ l$)
    match $l$ :
        $u \approx v$ :
            if $\mathfrak{C} \cup \mathfrak{D} \models u \not\approx v$ then return $\bot$            // Checks consistency
            $\mathfrak{C} \leftarrow \mathfrak{C} \cup \{u \approx v\}$                                // Updates $\mathfrak{C} \cup \mathfrak{D}$
        $u \not\approx v$ :
            if $\mathfrak{C} \models u \approx v$ then return $\bot$
            $\mathfrak{D} \leftarrow \mathfrak{D} \cup \{u \not\approx v\}$
    $\delta \leftarrow \{x \mapsto \mathrm{SEL}(x) \mid x \in \mathcal{X}\}$                          // Current instantiation
    $\Lambda \leftarrow (\mathrm{UNIFY}\ \delta\ l) \setminus_{\mathfrak{C}} \Delta_{\mathbf{x}}$                      // $\mathcal{L} \models l\sigma$, for every $\sigma \in \Lambda$
    if $\Lambda \neq \varnothing$ then
        let $\sigma \in \Lambda$ in
            $\mathfrak{C} \leftarrow \mathfrak{C} \cup \bigcup_{x \in dom(\sigma)}\{x \approx x\sigma\}$
        return $\top$
    return $\bot$
```

Fig. 1: Congruence Closure with Free Variables.

Backtracking is handled by the procedure RESET, which simply resets $\mathfrak{C}$ and $\mathfrak{D}$ to their initial states, while the literals in $\neg\psi$ are marked to be reevaluated by the loop. This is a naïve approach, for simplicity. Smarter backtracks are achievable through careful analysis of the dependencies for the inconsistency found.

The function UNIFY in Figure 2 takes a set of literals $\mathcal{L}$, a substitution θ and a literal $l\theta$ as input, computing the set of substitutions σ such that $\theta \subseteq_{\mathfrak{C}} \sigma$ and $\mathcal{L} \models l\sigma$. It is invoked with the current instantiation and the literal being asserted. If the resulting set is empty, a failure is reported, showing the unfeasibility of the current instantiation. Otherwise one of its elements is chosen and the current instantiation is updated accordingly.

Computing feasible instantiations Adapting the *recursive descent* E-*unification* algorithm in [1], the function UNIFY computes the set of $(\mathfrak{C}, \mathfrak{D})$-unifiers of given equality literals $u \approx v$ and $u \not\approx v$, respectively, extending an initial substitution σ. The resulting unifiers solve the E-unification problem for $\mathcal{L}$ and the given literal. In the presentation, u and $f(\mathbf{u})$ represent non-ground terms, v and $f(\mathbf{v})$ terms that may or may not be ground and t and $f(\mathbf{t})$ ground terms, with subscripts or not.

The *merging* of two ground substitutions σ_1 and σ is defined by

$$\sigma_1 \oplus \sigma_2 = \begin{cases} \sigma_1 \circ \sigma_2 & \textbf{if } x \in (\mathit{dom}(\sigma_1) \cap \mathit{dom}(\sigma_2)) \text{ only if } \mathfrak{C} \models x\sigma_1 \approx x\sigma_2 \\ \varnothing & \textbf{otherwise} \end{cases}$$

The merging of two sets of ground substitutions is the result of pairwisely merging their members. If either set is empty, so is their merging.

Example 2. Given substitutions $\sigma_1 = \{x \mapsto a\}$ and $\sigma_2 = \{x \mapsto b\}$, their merging is the empty set unless $\mathfrak{C} \models a \approx b$.

$$\begin{aligned}
&(1)\ \textsc{Unify}\ \mathcal{L}\ \sigma\ t_1 \approx t_2 &&= \begin{cases} \{\sigma\} & \textbf{if } \mathfrak{C} \models t_1 \approx t_2 \\ \varnothing & \textbf{otherwise} \end{cases} \\
&(2)\ \textsc{Unify}\ \mathcal{L}\ \sigma\ x \approx t &&= \{\{x \mapsto t\} \circ \sigma\} \\
&(3)\ \textsc{Unify}\ \mathcal{L}\ \sigma\ f(\mathbf{u}) \approx t &&= \bigcup\nolimits_{f(\mathbf{t}) \in [t]} (\textsc{Unify}\ \mathcal{L}\ \sigma\ u_1 \approx t_1) \oplus \cdots \oplus (\textsc{Unify}\ \mathcal{L}\ \sigma\ u_n \approx t_n) \\
&(4)\ \textsc{Unify}\ \mathcal{L}\ \sigma\ u \approx v &&= \bigcup\nolimits_{t \in \mathbf{T}^{\mathcal{L}}} \bigcup\nolimits_{\theta \in \textsc{Unify}\ \mathcal{L}\ \varnothing\ v \approx t} \textsc{Unify}\ \mathcal{L}\ \sigma\theta\ u\theta \approx v\theta \\
&(5)\ \textsc{Unify}\ \mathcal{L}\ \sigma\ u \not\approx t &&= \bigcup\nolimits_{t_1 \not\approx t_2 \in \mathfrak{D},\, t_1 \in [t],\, t' \in [t_2]} \textsc{Unify}\ \mathcal{L}\ \sigma\ u \approx t' \\
&(6)\ \textsc{Unify}\ \mathcal{L}\ \sigma\ u \not\approx v &&= \bigcup\nolimits_{t \in \mathbf{T}^{\mathcal{L}}} \bigcup\nolimits_{\theta \in \textsc{Unify}\ \mathcal{L}\ \varnothing\ u \approx t} \textsc{Unify}\ \mathcal{L}\ \sigma\theta\ u\theta \not\approx v\theta
\end{aligned}$$

Fig. 2. Unify function

Rules are applied through pattern matching on the given literal, following the presented order. Cases (5) and (6) handle the unification with the disequalities in $\mathfrak{D}$, which ultimately also rely on the partition induced by $\mathfrak{C}$. Cases (1-4) make the computation of the feasible substitution by recursively going through the terms in the equality, eventually matching modulo $\mathfrak{C}$. The crucial test occurs in (1), when the substitution computed stands only if the equality over ground terms holds in $\mathfrak{C}$. There is no need to check if $x \in \mathit{dom}(\sigma)$ in (2), since every free variable in the given literals is not in the domain of the given substitution.

Many optimizations may be devised to improve Unify, (if u is a function term $f(\mathbf{u})$, one does not need to go through every term in $\mathbf{T}^{\mathcal{L}}$, it is sufficient to check any terms of the form $f(\mathbf{t})$; and so on) but the above definition is sufficient for generating every unifier solving the given E-unification problem, as established in Lemma 2.

Lemma 2. *Consider a set of equality literals $\mathcal{L}$ and a substitution θ. Let $\mathfrak{C}$ and $\mathfrak{D}$ be the sets of equalities and disequalities in $\mathcal{L}$, respectively, the former inducing* *a partition of* $\mathbf{T}^{\mathcal{L}}$ *in congruence classes. Then, given literals* $u \approx v$*,* $u \not\approx v$*,*

$$\begin{aligned}\textsc{Unify}\ \mathcal{L}\ \theta\ u\theta \approx v\theta &= \{\sigma \mid \theta \subseteq \sigma, \mathfrak{C} \models u\sigma \approx v\sigma\}\\ \textsc{Unify}\ \mathcal{L}\ \theta\ u\theta \not\approx v\theta &= \{\sigma \mid \theta \subseteq \sigma, \mathfrak{C} \cup \mathfrak{D} \models u\sigma \not\approx v\sigma\}\end{aligned}$$

Proof (sketch). Induction on the structure of the literals (rules 1-4 for $\mathfrak{C}$-unifiers; 5-6 for $\mathfrak{D}$-unifiers).

Example 3. Let CCFV be applied on the following input:

$$\begin{aligned}\mathfrak{C} &= \{a \approx f(c), b \approx f(a)\} & \Delta_{\mathbf{x}} &= \varnothing\\ \mathfrak{D} &= \{f(a) \not\approx f(b)\} & \neg\psi &= f(x_1) \approx a \wedge f(x_2) \not\approx b.\end{aligned}$$

For $l = f(x_1) \approx a$, $\textsc{Handle}(\mathfrak{C}, \mathfrak{D}, \Delta_{\mathbf{x}}, l)$ adds l to $\mathfrak{C}$, since it is consistent with $\mathfrak{C} \cup \mathfrak{D}$. The current instantiation is $\varnothing$, so $\textsc{Unify}\ \mathcal{L}\ \varnothing\ f(x_1) \approx a$ is invoked. According to its definition,

$$\begin{aligned}\textsc{Unify}\ \mathcal{L}\ \varnothing\ f(x_1) \approx a &= \textsc{Unify}\ \mathcal{L}\ \varnothing\ f(x_1) \approx f(c) &&\\ &= \textsc{Unify}\ \mathcal{L}\ \varnothing\ x_1 \approx c && \text{[Rule 3]}\\ &= \{\{x_1 \mapsto c\}\} && \text{[Rule 2]}\end{aligned}$$

since $f(c)$ is the only term in $[a]$ unifiable with $f(x_1)$: for every other $t \in [a]$, $\textsc{Unify}\ \mathcal{L}\ \varnothing\ f(x_1) \approx t = \varnothing$. Since $\{x_1 \mapsto c\}$ is not in $\Delta_{\mathbf{x}}$, it is a feasible substitution, triggering the addition of $\{x_1 \approx c\}$ to $\mathfrak{C}$ and the successful termination of this invocation of Handle.

For $l = f(x_2) \not\approx b$, Handle adds l to $\mathfrak{D}$ (no inconsistency) and invokes $\textsc{Unify}\ \mathcal{L}\ \delta\ f(x_2) \not\approx b$, with $\delta = \{x_1 \mapsto c\}$ as the current instantiation. Thus,

$$\begin{aligned}\textsc{Unify}\ \mathcal{L}\ \delta\ f(x_2) \not\approx b &= \textsc{Unify}\ \mathcal{L}\ \delta\ f(x_2) \approx f(b) && \text{[Rule 5]}\\ &= \textsc{Unify}\ \mathcal{L}\ \delta\ x_2 \approx b && \text{[Rule 3]}\\ &= \{\{x_2 \mapsto b\} \circ \delta\} && \text{[Rule 2]}\\ &= \{\{x_1 \mapsto c, x_2 \mapsto b\}\} && \text{[Composition]}\end{aligned}$$

since $f(a) \not\approx f(b)$, the sole disequality in $\mathfrak{D}$, is such that $f(a) \in [b]$ and $f(b)$ is unifiable with $f(x_2)$, deriving $\{x_2 \mapsto b\} \circ \delta$. Since $\{x_1 \mapsto c, x_2 \mapsto b\}$ is not in $\Delta_{\mathbf{x}}$, $\{x_2 \approx b\}$ is added to $\mathfrak{C}$.

With no more literals to process in $\neg\psi$, the current instantiation $\sigma = \{x_1 \mapsto c, x_2 \mapsto b\}$ is outputed. It is a ground conflicting substitution for $\mathcal{L} \wedge \psi$.

The correctness of CCFV is established in Theorem 1.

Theorem 1. *Consider a ground formula* $\mathcal{L}$ *and a formula* ψ *with free variables. If there are ground conflicting substitutions for* $\mathcal{L}$ *and* ψ*, then there exists a ground substitution* σ *such that* $\sigma = \mathrm{CCFV}(\mathcal{L}, \psi)$*,* $\mathrm{ran}(\sigma) \subseteq \mathbf{T}^{\mathcal{L}}$ *and* $\mathcal{L} \models \neg\psi\sigma$*.*

Proof (sketch). Relying on Lemma 2, the computation of ground conflicting substitutions by CCFV can be proven through induction on the structure of ψ.

4 Future work

Since completeness is frequently lost for theories more expressive than the empty one, handling quantifiers is essentially an effort of having efficient but incomplete techniques that, for some fragments, allow decision procedures, but often not.

We want to define a framework encompassing not only the derivation of ground conflicting but of *model conflicting instances*, that is, instances refuting a ground model extended into a candidate model for the original formula. This would amount to extend CCFV to handle MBQI, an effective approach for quantifiers in SMT. Furthermore, it is essential to include theory reasoning in these processes.

We are starting to integrate the CCFV algorithm within the SMT solver veriT [5], in order to evaluate and improve our approach of ground conflicting instance generation [11]. We believe CCFV is also strongly related to approaches like bounded simultaneous rigid E-unification [2]. It is however not clear how to reconcile both approaches since CCFV considers only one quantified formula at a time.

While extending CCFV for deriving substitutions conflicting modulo theories, the foremost theory to consider will be *LIA*. We will investigate techniques such as *Hierarchic Theorem Proving* [4], which obtains an effective procedure for handling quantified formulas with interpreted terms. To do so they combine ground and resolution-based reasoning, besides a model refinement technique with similarities to MBQI.

References

1. F. Baader and J. H. Siekmann. Unification theory. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume2, Deduction Methodologies*, pages 41–126. 1994.
2. P. Backeman and P. Rümmer. Theorem proving with bounded rigid E-unification. (To appear).
3. C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, Feb. 2009.
4. P. Baumgartner, J. Bax, and U. Waldmann. Finite Quantification in Hierarchic Theorem Proving. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Automated Reasoning*, volume 8562 of *Lecture Notes in Computer Science*, pages 152–167. Springer International Publishing, 2014.
5. T. Bouton, D. C. B. de Oliveira, D. Déharbe, and P. Fontaine. veriT: an open, trustable and efficient SMT-solver. In R. A. Schmidt, editor, *Proc. Conference on Automated Deduction (CADE-22)*, 2009.
6. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.
7. Y. Ge and L. de Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In A. Bouajjani and O. Maler, editors, *CAV 2009*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

8. L. Moura and N. Bjørner. Engineering DPLL(T) + saturation. In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 475–490, Berlin, Heidelberg, 2008. Springer-Verlag.
9. R. Nieuwenhuis and A. Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557 – 580, 2007. Special Issue: 16th International Conference on Rewriting Techniques and Applications.
10. R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science B.V., 2001.
11. A. Reynolds, C. Tinelli, and L. M. de Moura. Finding conflicting instances of quantified formulas in SMT. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 195–202, 2014.
12. A. Tiwari, L. Bachmair, and H. Rueß. Rigid e-unification revisited. In D. McAllester, editor, *Automated Deduction CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 220–234, Pittsburgh, PA, jun 2000. Springer-Verlag.

6 Quantifiers, Computation, and Cognition

Jakub Szymanik Quantifiers, Computation, and Cognition

Quantifiers, Computation, and Cognition

Jakub Szymanik[*]

Institute of Logic Language and Computation, University of Amsterdam
J.K.Szymanik@uva.nl

Generalized quantifier theory studies the semantics of quantifier expressions, like, ‘every’, ‘some’, ‘most’, ‘infinitely many’, ‘uncountably many’, etc. The classical version was developed in the 1980s, at the interface of linguistics, mathematics, and philosophy. In logic, generalized quantifiers are often defined as classes of models closed on isomorphism (topic neutral). For instance, the quantifier ‘infinitely many’ may be defined as a class of all infinite models. Equivalently, in linguistics generalized quantifiers are formally treated as relations between subsets of the universe. For example, in the sentence ‘Most of the students are smart’, quantifier ‘most’ is a binary relation between the set of students and the set of smart people. The sentence is true if and only if the cardinality of the set of smart students is greater than the cardinality of the set of students who are not smart. Generalized quantifiers turned out to be one of the crucial notions in the development of formal semantics but also logic, theoretical computer science and philosophy [11]. In this talks we survey recent results combining classical generalized quantifier themes and a computational complexity perspective, with an outlook toward applications in cognitive science and linguistics [16].

We focus on the complexity of meaning of natural language quantifiers. The general question we aim to answer is why the meanings of some sentences are more difficult than the meanings of others. For instance, why we will probably all agree that it is easier to evaluate sentence (1) than to evaluate sentence (2) and why sentence (3) seems hard while sentence (4) sounds odd.

(1) Every book on the shelf is yellow.
(2) Most of the books on the shelf are yellow.
(3) Less than half of the members of parliament refer to each other.
(4) Some book by every author is referred to in some essay by every critic.

The tools of logic and computability theory are useful in making such differences precise. The complexity analysis of the quantifier sentences in natural language allows drawing and testing empirical predictions about cognitive difficulty of language processing, and about specific cognitive resources (working memory, executive functions, etc.) involved in it.

We will start by introducing the notion of monadic quantifiers—the most important class of generalized quantifiers that captures the meanings of natural language simple determiners. In particular, we will introduce so-called semantic automata theory that associates each quantifier with a simple computational device. We will also discuss some classical definability results connecting expressibility with semantic automata, e.g., all quantifiers definable in the first-order

[*]The research was supported by Veni Grant NWO-639-021-232.

logic are recognizable by acyclic finite-automata [1, 9, 4]. In doing that we will use fundamental notions of automata theory and draw some connections with psychology, for instance, we will show that the distinction between finite-automata and push-down automata quantifiers matters for psycholinguistics [8, 13, 14, 17, 20, 18].

Next we will survey current literature concerned with polyadic quantification. We will explain how polyadic quantifiers result from semantically natural operations applied to monadic quantifiers, like iteration, cumulation, or Ramseyification. In addition to discussing definability issues, we will also demonstrate how the semantic automata framework can be extended to iterations, showing that if Q_1 and Q_2 are recognizable by finite-automata (push-down automata) then also their iteration must be recognizable by finite-automata (push-down automata) [12]. Furthermore, we will discuss computational complexity results on more kinds of polyadic quantifiers—among others—proving a dichotomy result for Ramsey quantifiers [15, 2], namely, we show that the Ramseyification of polynomial-time and constant-log-bounded monadic quantifiers result in polynomial time computable Ramsey quantifiers while assuming the Exponential Time Hypothesis. Moreover, we will discuss how such complexity results correlate with linguistic distributions [19, 3].

In the final, most technical part, we will show how the standard generalized quantifier theory, originally designed to deal with distributive quantification, can be extended to cover collective quantifiers. We will discuss type-lifting strategies constructing collective readings from distributive readings. We will also introduce the notion of second-order generalized quantifier that is a natural mathematical extension of Lindström quantifiers to the collective setting. We will introduce the definability theory for second-order generalized quantifiers [5] and discuss related computational complexity results [6]. In particular, we will show that the question whether a second-order generalized quantifier $\mathcal{Q}_1$ is definable in terms of another quantifier $\mathcal{Q}_2$, the base logic being monadic second-order logic, reduces to the question if a quantifier $\mathcal{Q}^{\star}{}_1$ is definable in $FO(\mathcal{Q}^{\star}{}_2, <, +, \times)$ for certain first-order quantifiers $Q_1^{\star}$ and $Q_2^{\star}$. We use our characterization to show new definability and non-definability results for second-order generalized quantifiers [7]. We will conclude with a more general methodological discussions, using definability and complexity results we will ask about the expressivity bounds of everyday language [10].

References

1. Johan van Benthem. *Essays in logical semantics*. Reidel, 1986.
2. Ronald de Haan and Jakub Szymanik. A dichotomy result for Ramsey quantifiers. In *Proceedings of the 22nd Workshop on Logic, Language, Information and Computation*, 2015.
3. Nina Gierasimczuk and Jakub Szymanik. Branching quantification vs. two-way quantification. *The Journal of Semantics*, 26(4):329–366, 2009.
4. Makoto Kanazawa. Monadic quantifiers recognized by deterministic pushdown automata. In F. Roelofsen M. Aloni, M. Franke, editor, *Proceedings of the 19th Amsterdam Colloquium*, pages 139–146, 2013.

5. Juha Kontinen. *Definability of second order generalized quantifiers.* PhD thesis, Helsinki University, 2004.
6. Juha Kontinen and Jakub Szymanik. A remark on collective quantification. *Journal of Logic, Language and Information*, 17(2):131–140, 2008.
7. Juha Kontinen and Jakub Szymanik. A characterization of definability of second-order generalized quantifiers with applications to non-definability. *Journal of Computer and System Sciences*, 80(6):1152 – 1162, 2014.
8. Corey T. McMillan, Robin Clark, Peachie Moore, Christian Devita, and Murray Grossman. Neural basis for generalized quantifier comprehension. *Neuropsychologia*, 43:1729–1737, 2005.
9. Marcin Mostowski. Computational semantics for monadic quantifiers. *Journal of Applied Non-Classical Logics*, 8:107–121, 1998.
10. Marcin Mostowski and Jakub Szymanik. Semantic bounds for everyday language. *Semiotica*, 188(1-4):363–372, 2012.
11. Stanley Peters and Dag Westerståhl. *Quantifiers in Language and Logic.* Clarendon Press, Oxford, 2006.
12. Shane Steinert-Threlkeld and III Icard, ThomasF. Iterating semantic automata. *Linguistics and Philosophy*, 36(2):151–173, 2013.
13. Jakub Szymanik. A comment on a neuroimaging study of natural language quantifier comprehension. *Neuropsychologia*, 45:2158–2160, 2007.
14. Jakub Szymanik. *Quantifiers in TIME and SPACE. Computational Complexity of Generalized Quantifiers in Natural Language.* PhD thesis, University of Amsterdam, Amsterdam, 2009.
15. Jakub Szymanik. Computational complexity of polyadic lifts of generalized quantifiers in natural language. *Linguistics and Philosophy*, 33(3):215–250, 2010.
16. Jakub Szymanik. *Quantifiers and Cognition.* Studies in Linguistics and Philosophy. Springer, 2015.
17. Jakub Szymanik and Marcin Zajenkowski. Comprehension of simple quantifiers. Empirical evaluation of a computational model. *Cognitive Science: A Multidisciplinary Journal*, 34(3):521–532, 2010.
18. Jakub Szymanik and Marcin Zajenkowski. Quantifiers and working memory. In M. Aloni and K. Schulz, editors, *Amsterdam Colloquium 2009, Lecture Notes In Artificial Intelligence 6042*, pages 456–464. Springer, 2010.
19. Camilo Thorne and Jakub Szymanik. Semantic complexity of quantifiers and their distribution in corpora. In *Proceedings of the International Conference on Computational Semantics*, 2015.
20. Marcin Zajenkowski, Rafał Styła, and Jakub Szymanik. A computational approach to quantifiers as an explanation for some language impairments in schizophrenia. *Journal of Communication Disorders*, 44(6):595 – 600, 2011.