EPiC Series in Computing

Volume 53, 2018, Pages 42–51

Vampire 2017. Proceedings of the 4th Vampire Workshop

Using Vampire with Support for Algebraic Datatypes in Type Soundness Proofs

Sylvia Grewe[1], André Pacak[1], and Mira Mezini[1]

TU Darmstadt, Germany

Abstract

In our ongoing project VeriTaS, we aim at automating soundness proofs for type systems of domain-specific languages. In the past, we successfully used previous Vampire versions for automatically discharging many intermediate proof obligations arising within standard soundness proofs for small type systems. With older Vampire versions, encoding the individual proof problems required manual encoding of algebraic datatypes via the theory of finite term algebras. One of the new Vampire versions now supports the direct specification of algebraic datatypes and integrates reasoning about term algebras into the internally used superposition calculus.

In this work, we investigate how many proof problems that typically arise within type soundness proofs different Vampire 4.1 versions can prove. Our test set consists of proof problems from a progress proof of a type system for a subset of SQL. We compare running Vampire 4.1 with our own encodings of algebraic datatypes (in untyped as well as in typed first-order logic) to running Vampire 4.1 with support for algebraic datatypes, which uses SMTLIB as input format. We observe that with our own encodings, Vampire 4.1 still proves more of our input problems. We discuss the differences between our own encoding of algebraic datatypes and the ones used within Vampire 4.1 with support for algebraic datatypes.

1 Introduction

VeriTaS [3, 6] is a Scala library under development for the specification and automated verification of statically typed domain-specific languages. VeriTaS shall achieve automated verification of type soundness by combining automated domain-specific proof strategies with existing automated theorem provers. To this end, VeriTaS will on the one hand provide concrete domain-specific proof strategies for proving progress and preservation properties of type systems [9] as well as an infrastructure for facilitating the development of further domain-specific proof strategies. On the other hand, VeriTaS already provides an infrastructure for structuring complicated proofs in several independent smaller proof problems, for encoding these proof problems into the input formats of automated theorem provers (ATPs) and SMT solvers, and for calling different ATPs to verify these individual proof steps.

Previous versions of VeriTaS translated a core specification language with algebraic datatypes and recursive functions to TPTP [11], encoding algebraic datatypes by axiomatizing term algebras. In previous work, we describe the details of this translation [5]. Our case studies so far

showed that Vampire [8] is well able to prove individual proof steps from progress and preservation proofs automatically [4], as well as proof goals for exploring language specifications [5], such as goals that test the concrete behavior of a type system for specific programs. On the other hand, we also frequently ran into seemingly trivial proof problems that previous Vampire versions could not prove, even with a high timeout.

One hypothetical reason for encountering such situations could be our external encoding of algebraic datatypes, which may not be optimal with regard to the search heuristics that Vampire 4.1 employs internally. The most recent version of Vampire 4.1 now includes *internal* support for algebraic datatypes [7], where reasoning about term algebras is integrated into the superposition calculus. This version allows using the SMTLIB format (version 2.6)[1] as input, which contains constructs for directly specifying algebraic datatypes. Kovács, Robillard and Voronkov [7] show that this new Vampire 4.1 variant can solve many proof problems with algebraic datatypes which previously, no other prover could solve. Hence, it may be that the new support for algebraic datatypes also proves more of the proof problems we are interested in.

To investigate whether the new support of Vampire 4.1 for algebraic datatypes proves more of our proof problems, we use a sample set of proof problems from a progress proof of a type system for a subset of SQL which we encoded in VeriTaS (Section 2). We implement an automated encoding of these proof problems into SMTLIB version 2.6. The translation is straightforward, since our current input format in VeriTaS for type system specifications is conceptionally very close to SMTLIB. Next, we run Vampire 4.1 with and without internal support for algebraic datatypes and compare the results (Section 3). Finally, we discuss our observations and our wishlist for future Vampire versions (Section 4).

2 VeriTaS Infrastructure and Test Specification

VeriTaS The VeriTaS infrastructure is being implemented as a Scala library. Its main components are 1) an input format for type system specifications, 2) a compiler family which enables using different compilation strategies for translating from our input format for type system specifications to the input format of automated theorem provers, 3) a proof graph infrastructure conceptionally based on previous work on proof planning [10] for structuring and visualizing type soundness proofs, and 4) tactics and domain-specific strategies for automatically generating proof graphs from a type system specification (under development, and not in the scope of this paper). VeriTaS is available at github via `https://github.com/stg-tud/type-pragmatics/tree/master/Veritas`. The main work we describe in this paper is available in the dev-smt branch (`https://github.com/stg-tud/type-pragmatics/tree/dev-smt/Veritas`).

Our input format for type system specifications is currently implemented as an embedded DSL. This embedded DSL allows for specifying the syntax of a language via closed algebraic datatypes, the dynamic semantics (reduction semantics) of the language via simple recursive, immutable functions, and the static semantics (type system) as well as properties to prove via typing rules and formulas. In a previous publication, we thoroughly describe our input format and different variants of how we translate the format to untyped and typed first-order logic ([5]), where we encode algebraic datatypes ourselves by generating axioms which describe the respective term algebras.

We structure and visualize proofs via proof graphs, which is a variant of proof trees from Richardson et al. [10]. Proof graphs are directed, acyclic graphs (DAGs). We use DAGs instead

[1] `http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-draft-2017-06-05.pdf`

of trees to avoid redundant proof nodes, since some lemmas might be used in several parts of a type soundness proof. A node in a proof graph is an individual proof problem (consisting of a set of specification axioms and a single goal to be proven from the specification axioms). That is, each node can be translated separately into an input problem for an external prover. Each node either has no children (i.e. is a leaf node), which means that the corresponding proof problem should be directly provable from the given specification axioms, or it has one or more children, which are sub-goals that are required for proving the parent goal and will be included as lemmas into the specification of the input problem that is generated for the parent node. A root node (i.e. a node of a proof graph that does not have any parents) specifies a theorem. Each node is associated with a tactic that describes how the parent goal can be proven from its direct sub-goals (e.g. via induction, via case distinction). Leaves typically just have the basic "Solve" tactic associated, which simply sends the proof problem as is (i.e. without adding any additional lemmas) to a prover. A node is marked as verified if 1) all its children are marked verified and 2) the step from the children to the parent is marked verified.

SQL test specification For the present work, we use a type system specification described in one of our previous publications [5]: The language we model is a small subset of SQL, consisting of basic table values, simple `select ... from ... where ...` queries without joins and Cartesian products (i.e. combinations of table selection (of table rows) and projection (of table columns)), and queries for forming the union, intersection, and difference of two tables with the same table schema. We specify a small-step reduction semantics which reduces queries in small steps to table values (i.e. the final tables). The semantics may get stuck if it is has to perform an operation that is not possible with the currently given table, e.g. if it has to access a named column that does not exist within the current table. Furthermore, we augment the subset of SQL with a static type system, which type-checks a given query by checking whether the operations to be executed correspond to the typed table schemas of the original tables and all the intermediate tables implicitly created by a query.

For this type system, we prove *progress*, which is one of two standard soundness properties from the type system literature (e.g. Pierce [9]). Intuitively, progress says that if a query can be typed, it is either already a table value (with a well-typed table), or the reduction semantics can reduce the query at least one step further without getting stuck. We described the property and the progress proof in a previous publication at the Vampire workshop [4].

3 Comparison of Different Vampire Versions

3.1 Setup

The SQL progress proof overall consists of 32 goals which we transform into the corresponding proof problems. For this transformation, we apply various steps successively, as described in our previous work [5]. Here, we briefly highlight one of them: Firstly, we eliminate universally quantified variables that appear in premises of implications and are refined by equations by inlining them. For example, a formula of the shape $\forall v\ a.\ v = f(a) \implies p(v)$ becomes $\forall a.\ p(f(a))$. Our previous empirical study [5] showed that this often increases the probability of proving our particular input problems. For the generated input problems we try to find proofs with the provers Vampire 4.1 using TPTP [11] FOF encoding (untyped first-order logic), Vampire 4.1 using TPTP TFF encoding (typed first-order logic), Vampire 4.1 with support for reasoning about term algebras (-tar option) and VampireZ3 4.1 with the -tar option. For the latter two Vampire versions, we translate our input problems to SMTLIB 2.6. The main

difference between the problems encoded in TPTP and the problems encoded in SMTLIB is the encoding of closed algebraic datatypes: For the TPTP encodings, we add axioms that describe the corresponding term algebras for each closed algebraic datatype. These axioms are mostly just like the ones also used internally in Vampire 4.1 with the -tar option [7]. When using SMTLIB, we simply translate our original closed algebraic datatypes syntactically to corresponding constructs in SMTLIB. We discuss some of the finer differences in Section 4. We call all provers in casc mode and additionally Vampire 4.1 and VampireZ3 with support for term algebraic reasoning with the -tar option and with --input_syntax smtlib2 to support the usage of SMTLIB as an input format. We leave the acyclicity axioms off, since we do not require them within our proof problems. We call the provers with increasing timeouts (5s, 10s, 30s, 90s, 120s) for each input problem. If a proof is found, we stop looking at other timeouts for the current combination of input problem and prover.

We execute all prover calls on a MacBook Pro with an Intel Core i7, 2,9 GHz and 8GB RAM.

3.2 Results

Table 1 lists the full results for all 4 prover variants and the 32 input problems. The name of each problem reveals the role of each input problem within the overall progress proof: The top-level goal is called `SQL-Progress`. We prove it via structural induction. Whenever we prove a goal via induction, we attach `-icaseN` to its name, where `N` is the number of the induction case. If we prove an input problem via case distinction, we further attach `-caseM` to the name of the input problem for each case `M`. We visualize the corresponding proof graph in Figure2.

We call different Vampire 4.1 versions for each of these proof problems. For each call, we list the status of the prover in Table 1. If an input problem was proven, we also list the time reported by the prover. Note that one has to take the prover times with a grain of salt, since the prover calls were not executed in a benchmarking environment. However, executing the same calls several times does not yield significantly different prover times.

Out of the 32 input problems, Vampire 4.1 without support for reasoning about term algebras (Vampire 4.1 FOF and Vampire 4.1 TFF) found 23 proofs. Vampire 4.1 with support for reasoning about term algebras (Vampire 4.1 tar SMTLIB) as well as VampireZ3 (VampireZ3 4.1 tar SMTLIB) found 15 proofs for the given maximum timeout of 120 seconds. An interesting observation is that whenever one of Vampire 4.1 FOF and Vampire 4.1 TFF found a proof, the other version was also able to find a proof within 120 seconds. The same holds for Vampire 4.1 tar SMTLIB and VampireZ3 tar SMTLIB.

The first general observation we can make is that the provers with support for reasoning about term algebras are almost always slower when trying to find a proof for a given input problem than the other provers which have no support for reasoning about term algebras.[2] Next, we can say that if provers without reasoning about term algebras could not find a proof for the maximum timeout of 120 seconds the provers with support about term algebras could not find a proof as well.

An interesting observation is that there are input problems for which the prover times of Vampire 4.1 FOF and Vampire 4.1 TFF differ considerably: For the input problems texttttSQL-Progress-icase2-case0, `SQL-Progress-icase3-case0`, and `SQL-Progress-icase4-case0`, Vam-

[2] In our very first experiments for this work, the SMTLIB encodings lacked one group of axioms compared to the TPTP encodings (these axioms had nothing to do with the axioms generated for algebraic datatypes in the TPTP encodings). With this difference, we first occasionally observed faster prover times with the Vampire 4.1 tar support. Later, we noticed this difference and added the missing set of axioms to the SMTLIB encodings, which yielded the current results.

Goalname	Vampire 4.1 FOF	Vampire 4.1 TFF	Vampire 4.1 tar SMTLIB	VampireZ3 4.1 tar SMTLIB
SQL-Progress-icase0	Proved (0.033s).	Proved (0.026s).	Proved (0.044s).	Proved (0.045s).
SQL-Progress-icase1	Inconclusive.	Inconclusive.	Inconclusive.	Inconclusive.
successful-lookup-icase0	Proved (0.011s).	Proved (0.024s).	Proved (0.081s).	Proved (0.081s).
successful-lookup-icase1	Proved (0.024s).	Proved (0.499s).	Inconclusive.	Inconclusive.
welltyped-lookup-icase0	Proved (0.01s).	Proved (0.023s).	Proved (0.032s).	Proved (0.032s).
welltyped-lookup-icase1	Proved (0.019s).	Proved (2.717s).	Inconclusive.	Inconclusive.
filter-preserves-type	Proved (0.692s).	Proved (0.058s).	Proved (3.137s).	Proved (3.131s).
filterRows-preserves-table-icase0	Proved (0.029s).	Proved (0.024s).	Proved (0.045s).	Proved (0.037s).
filterRows-preserves-table-icase1	Proved (5.518s).	Proved (0.033s).	Inconclusive.	Inconclusive.
projectTable-progress	Proved (10.82s).	Proved (11.183s).	Inconclusive.	Inconclusive.
projectCols-progress-icase0	Proved (0.01s).	Proved (0.026s).	Proved (0.046s).	Proved (0.044s).
projectCols-progress-icase1	Proved (63.326s).	Proved (0.182s).	Inconclusive.	Inconclusive.
projectType-implies-findCol-icase0	Proved (0.011s).	Proved (0.024s).	Proved (0.027s).	Proved (0.027s).
projectType-implies-findCol-icase1	Inconclusive.	Inconclusive.	Inconclusive.	Inconclusive.
findColType-implies-findCol-icase0	Proved (0.011s).	Proved (0.026s).	Proved (0.065s).	Proved (0.071s).
findColType-implies-findCol-icase1	Inconclusive.	Inconclusive.	Inconclusive.	Inconclusive.
dropFirstColRaw-preserves-welltypedRaw-icase0	Proved (0.024s).	Proved (0.021s).	Proved (0.033s).	Proved (0.026s).
dropFirstColRaw-preserves-welltypedRaw-icase1	Proved (6.172s).	Proved (0.216s).	Proved (16.693s).	Proved (10.778s).
projectTypeAttrL-implies-findAllColType-icase0	Proved (0.009s).	Proved (0.021s).	Proved (0.023s).	Proved (0.022s).
projectTypeAttrL-implies-findAllColType-icase1	Proved (5.559s).	Proved (0.058s).	Proved (7.087s).	Proved (7.085s).
SQL-Progress-icase2	Proved (0.01s).	Proved (0.033s).	Proved (0.245s).	Proved (0.243s).
SQL-Progress-icase2-case0	Proved (1.073s).	Proved (63.652s).	Inconclusive.	Inconclusive.
SQL-Progress-icase2-case1	Inconclusive.	Inconclusive.	Inconclusive.	Inconclusive.
SQL-Progress-icase2-case2	Inconclusive.	Inconclusive.	Inconclusive.	Inconclusive.
SQL-Progress-icase3	Proved (0.014s).	Proved (0.035s).	Proved (0.244s).	Proved (0.248s).
SQL-Progress-icase3-case0	Proved (1.058s).	Proved (64.548s).	Inconclusive.	Inconclusive.
SQL-Progress-icase3-case1	Inconclusive.	Inconclusive.	Inconclusive.	Inconclusive.
SQL-Progress-icase3-case2	Inconclusive.	Inconclusive.	Inconclusive.	Inconclusive.
SQL-Progress-icase4	Proved (0.016s).	Proved (0.036s).	Proved (0.257s).	Proved (0.279s).
SQL-Progress-icase4-case0	Proved (1.112s).	Proved (64.915s).	Inconclusive.	Inconclusive.
SQL-Progress-icase4-case1	Inconclusive.	Inconclusive.	Inconclusive.	Inconclusive.
SQL-Progress-icase4-case2	Inconclusive.	Inconclusive.	Inconclusive.	Inconclusive.

Figure 1: Prover status and time for a given input problem and prover

pire 4.1 FOF proves each problem within approximately one second and Vampire 4.1 TFF requires more than 60 seconds. For the proof problem `projectCols-progress-icase1` it is the other way around: Vampire 4.1 TFF needs less than a second to find a proof while Vampire 4.1 FOF can only find a proof after about 63 seconds.

Figures 2 and 3 show the visualized proof graphs after trying to find a proof for every input problem generated by the goals contained within the proof graph. The boxed nodes represent the individual proof goals of the problem, the diamond-shaped ones the tactics associated with each proof problem (i.e. the individual proof step for proving a parent node, assuming that all child goals are true). We visualize the prover stati from Figure 1 via border colors: A green border represents a proved problem, a red one represents an inconclusive status. For

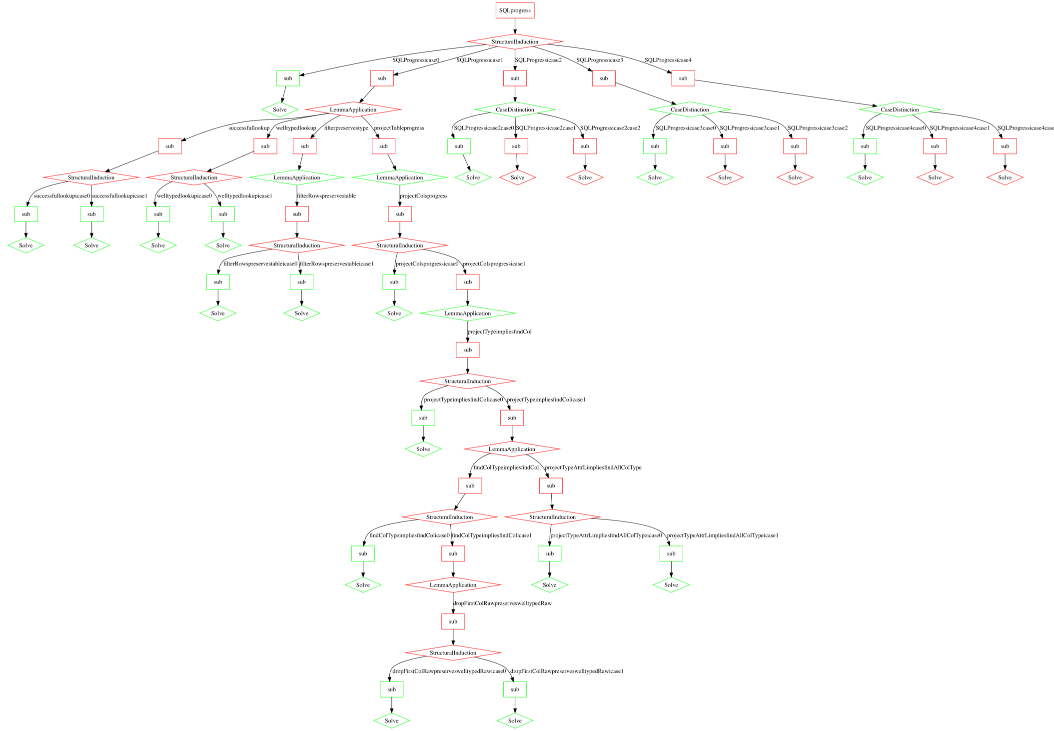

Figure 2: Visualization of proof graph after proving the single steps with Vampire 4.1 FOF/TFF.

our purposes here, the colors of the diamond-shaped nodes are the interesting ones - they directly correspond to the stati from Figure 1. Note that in our evaluation, we ignored the nodes arising from the diamond-shaped nodes labeled `StructuralInduction`: These input problems would require verifying the application of a scheme for structural induction, which Vampire cannot do so far (induction being a higher-order reasoning technique, not a first-order reasoning technique).

As said previously, we could not observe any differences regarding prover stati between Vampire 4.1 FOF and Vampire 4.1 TFF, or between Vampire 4.1 tar SMTLIB and VampireZ3 4.1 tar SMTLIB, hence the respective proof graphs look alike and we depict only one graph for Vampire 4.1 FOF/TFF (Figure 2) and one for Vampire/VampireZ3 4.1 with support for term algebra reasoning (Figure 3).

We observe that for most of the induction cases arising from structural induction, Vampire 4.1 without support for reasoning about term algebras could prove the base cases (`-icase0`, except for top-level induction) as well as the step cases (`-icase1`, except for top-level induction). Vampire 4.1 with support for reasoning about term algebras found only proofs for the base cases. We can see examples of this observation in lemmas `successfulllookup`, `welltypedlookup`, and `filterRowsPreservestable` (left side of the proof graphs).

When looking at the proofs of the structurally similar induction cases for the three set queries (`SQL-Progress-icase2 - SQL-Progress-icase4`), we can observe that Vampire 4.1 without support for reasoning about term algebras could find proofs for the subcases within the set queries (`SQL-Progress-icaseN-case0`, N from {2,3,4}), while the two Vampire 4.1 versions with support for reasoning about term algebra were not able to find proofs for these

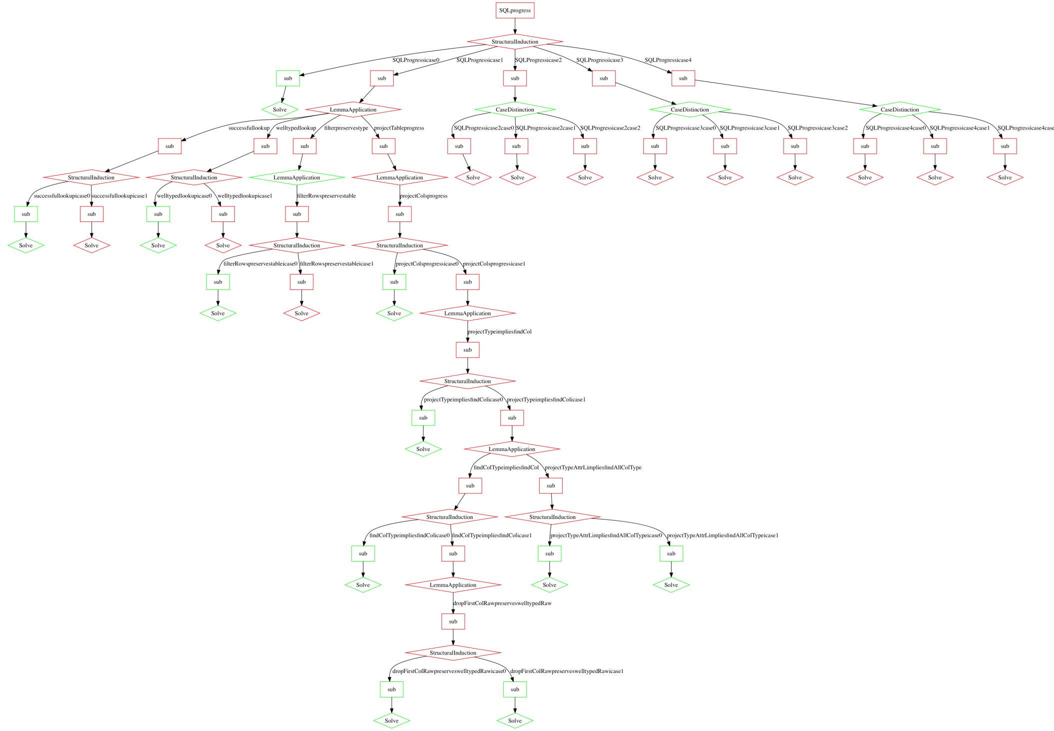

Figure 3: Visualization of proof graph after proving the single steps with Vampire/VampireZ3 4.1 with support for term algebra reasoning.

cases. Provers with and without support for term algebra reasoning were able to find proofs for the top-level set case distinction input problems (proving `SQL-Progress-icaseN` with N from {2,3,4}, from assuming the sub-cases).

4 Discussion

The only crucial difference between our own encoding of algebraic datatypes to untyped logic and our own encoding of algebraic datatypes to typed logic lies in the presence of the *domain closure axiom*: Suppose we have a closed algebraic datatype for lists containing elements of type `A`:

```
datatype AList = nil | cons(head: A, tail: AList)
```

To axiomatize this datatype, we would firstly add an axiom stating `nil` is never equal to any term of the form `cons(h, t)` (*distinctness axiom*). Secondly, we would add an axiom stating that all terms of the form `cons(h, t)` are equal to each other if the corresponding arguments are equal (*injectivity axiom*) [7, 5]. All of our own encodings of algebraic datatypes as well as the encoding used internally by Vampire 4.1 with support for term algebraic reasoning contain these axioms.

For completeness, you would need to add a *domain closure axiom*, which is also added internally by Vampire 4.1 with the -tar option [7]:

$$\forall al : AList.\ al = nil \lor (\exists a : A, ar : AList.\ al = cons(a, ar))$$

We add domain closure axioms for all datatypes when encoding our input problems to typed logic, but not when encoding to untyped logic. There, we omit the domain closure axioms since we apply full type erasure. Full type erasure is sound in our setting, since all of the types we use either always have an infinite domain or, in the case of underspecified datatypes, are explicitly axiomatized to have an infinite domain (see our own previous work [5] for further explanation on this, which is based on work by Claessen et al. [2] and Blanchette et al. [1]). The domain closure axiom in our example then becomes

$$\forall al.\ al = nil \lor (\exists a, ar.\ al = cons(a, ar))$$

Although this axiom is not unsound, we judged it to be unintuitive and hence removed it from our input problems. Removing the axiom cannot introduce any inconsistencies, but could theoretically lead to incompleteness: Without the domain closure axiom, the prover might not be able to create certain new clauses during the proof search.

However, in our particular proof problems, we so far could not detect any major differences between including or not including domain closure axioms regarding whether a prover is able to prove an input problem or not (see also the first two result columns of Figure 1, which confirms this for our set of test problems). This may be since coincidentally, the particular problems we used do not require domain closure axioms to be proven. Having fewer axioms over all in the proof problem could be a valid explanation for the differences we observed regarding Vampire 4.1 with untyped logic and Vampire 4.1 with support for term algebraic reasoning. Still, we emphasize that our encoding to typed logic does include domain closure axioms just like the ones internally used by Vampire 4.1 -tar, but we nevertheless observed that Vampire 4.1 using *typed* logic with our own encoding of algebraic datatypes proved more proof problems than Vampire 4.1 with support for term algebraic reasoning. Hence, it seems unlikely that including or not including domain closure axioms explains the differences we observed.

Another more likely explanation for the differences we observed may be the strategy portfolio used within the different Vampire versions: The strategies and options used on input problems are in general "hidden" from the outside user who typically simply employs the casc mode. However, which strategies and options are used first or used at all severely influences whether the prover finds a proof to a problem or not. For instance, older Vampire versions such as Vampire 3.0 would prove even more of our input problems than Vampire 4.1 (e.g. the remaining set cases `SQL-Progress-icase2-case1`, `SQL-Progress-icase2-case2` etc.) were proven by Vampire 3.0 [4]). The most likely explanation here is that the casc mode of Vampire 3.0 contained a different strategy portfolio than the one of Vampire 4.1. Similarly, the portfolio used by Vampire 4.1 may not (yet) work as well with the internal support for reasoning on term algebras. Finally, the differences we observed may also arise from the computation of the final proof, which we typically query as well.

In summary, even after careful analysis we could not find any difference between our own encodings and the one used internally by Vampire 4.1 with support for reasoning about term algebras which would explain the differences we observed. The differences may arise from different strategy portfolios. Comparing more input problems and further work on the encodings may shed some light on the issue.

Wishlist for future Vampire versions Overall, from our perspective as Vampire users, we have three items on our personal wishlist which arose from the experiments conducted

for this article as well as our previous experience with Vampire: Firstly, we believe it would be very helpful if Vampire users were able to train their custom modes using a set of input problems instead of relying on the casc mode. Secondly, we think it would be very useful if the support for reasoning with term algebras would be further improved so that more input problems can be solved. We think that internally supporting reasoning with term algebras is very useful for external Vampire users, notably from the programming languages community, who should not have to consider "low-level" issues such as how exactly an algebraic datatype should be axiomatized and whether a domain closure axiom is needed in their case or not (which might heavily influence prover performance). Thirdly, it would be very useful if Vampire also internally supported simple structural induction on algebraic datatypes. Since we sometimes observed that Vampire was well able to prove all induction cases of a lemma whose proof requires structural induction, the only missing step for automatically proving the entire lemma here would be to also apply structural induction automatically first. Hence, adding structural induction along with appropriate heuristics when to apply it could further increase the number of problems that Vampire can solve.

Acknowledgements

We thank Simon Robillard for insightful discussions regarding type erasure and the role of the domain closure axiom. This work has been supported in part by the European Research Council, grant No. 321217.

References

[1] Jasmin Christian Blanchette, Sascha Böhme, Andrei Popescu, and Nicholas Smallbone. Encoding monomorphic and polymorphic types. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 493–507. Springer, 2013.

[2] Koen Claessen, Ann Lillieström, and Nicholas Smallbone. Sort it out with monotonicity: Translating between many-sorted and unsorted first-order logic. In *Proceedings of International Conference on Automated Deduction (CADE)*, pages 207–221. Springer, 2011.

[3] Sylvia Grewe. Veritas: Verification of type system specifications: Mechanizing domain knowledge about progress and preservation proofs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016*, pages 12–14, 2016.

[4] Sylvia Grewe, Sebastian Erdweg, and Mira Mezini. Automating proof steps of progress proofs: Comparing vampire and dafny. In *Vampire@IJCAR 2016. Proceedings of the 3rd Vampire Workshop, Coimbra, Portugal, July 2, 2016.*, pages 33–45, 2016.

[5] Sylvia Grewe, Sebastian Erdweg, Michael Raulf, and Mira Mezini. Exploration of language specifications by compilation to first-order logic. In *Proceedings of International Symposium on Principles and Practice of Declarative Programming (PPDP)*, pages 104–117, 2016.

[6] Sylvia Grewe, Sebastian Erdweg, Pascal Wittmann, and Mira Mezini. Type systems for the masses: Deriving soundness proofs and efficient checkers. In *Proceedings of International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (ONWARD)*, pages 137–150. ACM, 2015.

[7] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to terms with quantified reasoning. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 260–270, 2017.

[8] Laura Kovács and Andrei Voronkov. First-order theorem proving and Vampire. In *Proceedings of International Conference on Computer Aided Verification (CAV)*, pages 1–35. Springer, 2013.

[9] Benjamin C. Pierce. *Types and programming languages*. MIT press, 2002.

[10] Julian Richardson and Alan Bundy. Proof planning methods as schemas. *J. Symbolic Computation*, 11:1–000, 1999.

[11] G. Sutcliffe. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.