Reuse of Introduced Symbols in Automatic Theorem Provers

Michael Rawson[1], Martin Suda[2], Petra Hozzová[1] and Giles Reger[3]

[1] *TU Wien*
[2] *Czech Institute of Informatics, Robotics, and Cybernetics*
[3] *University of Manchester*

Abstract

Automatic theorem provers may introduce fresh function or predicate symbols for various reasons. Sometimes, such symbols can be reused. We describe a simple form of symbol reuse in the first-order system Vampire, investigate its practical effect, and propose future schemes for more aggressive reuse.

1. Introduction

An automatic theorem prover might introduce a fresh symbol — that is, a symbol it has not previously seen — at any time between the start and end of its execution. During traditional preprocessing [1, 2], the two most well-known examples are *Skolemization* to eliminate ∃-binders, and some variation of the *Tseitin transformation*[1] to avoid excessive duplication of subformulae. Fresh symbols may also be used in preprocessing to eliminate exotic input features [3], or to optimise inputs with respect to some search algorithm [4]. New symbols may even be introduced during proof-search proper, such as for clause splitting [5] or induction [6]. In some cases, these symbols can be reused. We examine reuse of fresh symbols in the Vampire [7] automatic theorem prover, but results could easily generalise to other similar systems.

For example, consider Skolemizing

$$\exists x.P(x) \vee Q(x)$$
$$\exists x.P(x) \vee Q(x)$$

Plainly, these are the same formula and could employ the same Skolem constant, but both of Vampire's normal-form routines produce two constants — and therefore two clauses — instead of the possible one:

$$P(s_1) \vee Q(s_1)$$
$$P(s_2) \vee Q(s_2)$$

PAAR'22: 8th Workshop on Practical Aspects of Automated Reasoning, August 11–12, 2022, Haifa, Israel

[1] aka *naming*, *definition introduction*

Vampire could in principle notice that these are the same formula, then only process it once. A natural way to achieve this is by *formula sharing* [8], but this would be quite a large change to Vampire and has some issues of its own. The example given here is very simple, but symbol reuse can be employed extensively: see §4 for some ideas.

1.1. Motivation

It may appear that reusing introduced symbols is a clear win for system performance. In the case of definition introduction, removing duplicate definitions reduces the total number of clauses, which is considered likely to improve performance. However, reused symbols allow inference between parts of the search space that do not otherwise interact: in some cases this might shorten proofs or eliminate redundancy, but in others it expands the search space needlessly.

There is cause for optimism: Vampire uses an increasing amount of SAT/SMT technology to organise and perform ground reasoning tasks, such as AVATAR [9], AVATAR modulo theories [10], the InstGen calculus and global subsumption [11], MACE-style finite model building [4], theory instantiation [12], and more [13]. Reusing symbols in this context reduces the SAT/SMT model space, which is likely to improve both ground and first-order performance. Additionally, concurrent proof attempts [14] can share information over a common signature: reusing symbols consistently extends this common signature to introduced symbols.

1.2. Justification

When is it possible to reuse a symbol? Informally, introduced symbols often *represent* or *stand for* some formula. Then, an existing symbol representing the same formula may be reused. This sleight of hand is most alluring with definition introduction, where a fresh predicate P is defined to be equivalent to F: P stands for F. Skolemization, when encountering $\exists x.F$, introduces a function s with semantics roughly "some s such that F": again, s stands for F.

It is possible to relax the requirement for identical formulae to merely *equivalent* formulae. Consider introducing a symbol s to represent F: naturally enough, s could also be used to refer to G if $F \equiv G$. First-order logic is sufficiently expressive to even allow $s(x)$ to stand for $F[x]$, and then use $s(t)$ for $F[t]$. Such relaxations can be powerful, see §4 for more on this.

1.3. Setting

Vampire [7] is a state-of-the-art automatic theorem prover for first-order logic, with extensions to support various *theories*, induction [15, 6], rank-1 polymorphism [16], first-class booleans [3], and higher-order logic [17]. It implements many different techniques for reasoning efficiently despite these very hard problem domains. These techniques are not equally effective on all problems and have many tunable parameters, which can make the tool tricky for end-users: therefore, pre-tuned *strategy schedules* are provided that run a series of pre-tuned options.

VAMPIRE operates exclusively within a suitably-extended clause normal form. All formulae not in this form are translated into it before proof search begins, using VAMPIRE's advanced preprocessing routines [8]. This preprocessing frequently introduces fresh symbols.

2. Implementation and Pitfalls

We implement symbol reuse for Skolemization and definition introduction in VAMPIRE's preprocessing routines. Formulae are identified up to α-equivalence for this initial work, and symbol reuse for Skolem symbols and definitions can be switched on and off separately. We also describe some unpleasant bugs we encountered during testing (subsections beginning *Pitfall*) which we hope the reader can now avoid in their own implementation.

2.1. Key Functions

The basic implementation idea is to define a "key" function k from a formula to some reasonable key type, such as a formula, string or code sequence. This key can then be used to lookup and reuse symbols that have been used for the same key before. Suppose we need a symbol to stand for some formula F. If we have an existing symbol for $k(F)$, then we may re-use this symbol. Otherwise, we create a fresh symbol and store it with $k(F)$ so that we may reuse it later. In this work we consider a relatively simple key function (see below), but the general setting allows more complex key functions in future (§4).

2.2. Detecting α-equivalence

The simplest key function is the identity function k_{id}, but this will only reuse symbols for really trivial cases, and even a renaming of variables will defeat it. Consider introducing a symbol for

$$F \equiv \forall x \forall y.\ H[x, y]$$

versus

$$G \equiv \forall y \forall x.\ H[y, x]$$

for example: $k_{\mathrm{id}}(F) \neq k_{\mathrm{id}}(G)$. We implement a slightly more sophisticated key function k_α to detect α-equivalence, which canonically renames formulae left-to-right using a sequence x_i for each new variable encountered. Then both the above formulae map to the same key,

$$k_\alpha(F) = k_\alpha(G) = \forall x_0 \forall x_1 . H[x_0, x_1]$$

allowing more generous symbol reuse. This still does not handle more complex cases, such as identifying $\forall x \forall y.\ H[x, y]$ and $\forall x \forall y.\ H[y, x]$. We note that for our purposes this is equivalent in power to more-complex schemes like de Bruijn indices, as we only wish to identify strictly α-equivalent formulae. We do not need other operations such as capture-avoiding substitution.

2.3. Pitfall: Free Variables

Introduced symbols must usually close over the free variables of the formula they represent. For example, consider Skolemizing $\forall x \forall y \exists z. P(x, y, z)$. The outer two universal quantifiers are removed, and we obtain $P(x, y, s(x, y))$, where s is a fresh symbol applied to the now-free variables x and y. The reuse key in this case is

$$k_\alpha(\exists z. P(x, y, z)) = \exists x_0. P(x_1, x_2, x_0).$$

If we were now to additionally Skolemize $\forall x \forall y \exists z. P(y, x, z)$, then the key is the same and $P(y, x, s(y, x))$ is a legitimate instance of symbol reuse. Here we draw the reader's attention to the order of variables in the Skolem term. However, Vampire's existing Skolemization routine created Skolem terms with variables in the order of *binding*, not of *occurrence*, and therefore produced $P(y, x, s(x, y))$, which is unsound. To see this, consider the satisfiable set of formulae

$$\begin{aligned} \forall x \forall y \exists z.\ z &= f(x, y) \\ \forall x \forall y \exists z.\ z &= f(y, x) \\ f(a, b) &\neq f(b, a) \end{aligned}$$

Implementors must ensure that terms are created with variables in the order they occur in subformulae (or at least in a *consistent* order with respect to keys), lest they fall prey to this mis-reuse.

2.4. Pitfall: Sorts and ad-hoc Polymorphism

Some symbols are usually assumed to be ad-hoc polymorphic, even in monomorphic many-sorted systems. A good example is the equality predicate, but Vampire also considers e.g. arithmetic operators to be ad-hoc polymorphic over the integers, rationals, and reals. The problem here is that the reuse key must differentiate between the types of arguments the overloaded symbol is applied to. Consider Skolemizing the input problem

$$\begin{aligned} \forall x : \sigma.\ \forall y : \sigma.\ \exists z : \rho.\ P(z) \vee x = y \\ \forall x : \tau.\ \forall y : \tau.\ \exists z : \rho.\ P(z) \vee x = y \end{aligned}$$

where σ, τ, ρ are distinct sorts. The reuse key in both cases is

$$k_\alpha(\exists z : \rho.\ P(z) \vee x = y) = \exists x_0 : \rho.\ P(x_0) \vee x_1 = x_2$$

But the symbol cannot be reused in this case as it would be ill-typed. One solution (which we implement) is to include the types of free variables in the reuse key, so that the keys are

$$\begin{aligned} x_1 : \sigma,\ x_2 : \sigma \vdash \exists x_0 : \rho.\ P(x_0) \vee x_1 = x_2 \\ x_1 : \tau,\ x_2 : \tau \vdash \exists x_0 : \rho.\ P(x_0) \vee x_1 = x_2 \end{aligned}$$

Another solution, perhaps neater, is annotating ad-hoc overloaded symbols with their argument types, resulting in keys

$$\exists x_0 : \rho.\ P(x_0) \vee x_1 =_\sigma x_2$$
$$\exists x_0 : \rho.\ P(x_0) \vee x_1 =_\tau x_2$$

Yet another is to have the reused symbol be polymorphic over the ad-hoc type variables in the reuse key, although this does require rank-1 polymorphism. We implement the first solution for this work, although at some point in the future expect to switch to another.

2.5. Summary of Implemented Technique

To recap, we define a key function $k_\alpha(F) = \Gamma \vdash F'$, where Γ is an ordered list of pairs $x : \tau$ giving the types of free variables in F', and F' is a canonically-renamed copy of F. If two formulae F and G have the same key, we may use the same introduced symbol s in terms/predicates representing F or G. We reuse symbols in this way while performing definition introduction or Skolemization. The term for a given formula F is constructed from a possibly-reused symbol s and the free variables of F in order of their occurrence.

2.6. Compromise: Quantifier Blocks and Skolemization

While developing the technique, we noticed that on certain extreme problems attempting to reuse Skolem symbols leads to a degradation of performance. A careful look revealed a prohibitive quadratic complexity of repetitive key computation for deep existential quantifier blocks. For instance, on a formula $\exists x_1 x_2 \ldots x_n.\ F$ we would need to compute $k_\alpha(\exists x_1 x_2 \ldots x_n.\ F)$, $k_\alpha(\exists x_2 \ldots x_n.\ F)$, $k_\alpha(\exists x_3 \ldots x_n.\ F) \ldots$, which becomes too expensive for a large n and F.

We decided to avoid this expensive case by storing Skolem symbols for reuse on *per-quantifier-block* basis. This way, we only need one call to the key function for each (existential) quantifier block and nominally store a whole vector of Skolems with the key. In the actual implementation, however, only one symbol is stored, because we can assume the remaining ones occupy consecutive slots in the symbol table. As a mild concession, this trick gives up on the ability to reuse symbols within blocks: for example, when $s_1, s_2 \ldots, s_n$ get jointly stored at $k_\alpha(\exists x_1 x_2 \ldots x_n.\ F)$, we are not able to later selectively retrieve, e.g., s_n alone for $k_\alpha(\exists x_n.\ F)$. However, in practice, this seems to be a reasonable compromise.

3. Practical Effect

With the change described in §2, the example from §1 now works as expected. Practical effectiveness now depends upon two factors: whether benchmarks actually contain α-equivalent (sub)formulae for which Vampire can reuse introduced symbols, and whether this reuse is beneficial for proof search in Vampire (or downstream users of Vampire's clausification routines).

Table 1
Number of TPTP problems solved by Vampire's default strategy under a 50 billion instruction limit, with formula definition reuse (dr), Skolem symbol reuse (skr), and both at once (skr+dr).

	solved	uniquely
default	4290	11
dr	4297	14
skr	4300	12
skr+dr	4305	15

3.1. Reusable Symbols

We consider the untyped first-order ("FOF") benchmarks of the TPTP [18] problem library, version 7.5.0, which amounts to 9091 problems over a moderate number of domains. Vampire processed these problems using its default clausification routine and attempted to reuse definition and Skolem symbols up to α-equivalence. Of these 9091 problems, Vampire could reuse at least one symbol for 4311 problems. In some cases, a large number of symbols could be reused many times: in the case of `ITP024+4`, Vampire was able to reuse one of 3978 introduced symbols on 19442 separate occasions, with the most commonly-reused symbol reused in 184 different locations. We consider this relatively strong evidence that attempting to reuse introduced symbols may be worthwhile, even for smaller systems that have otherwise simple preprocessing.

3.2. Effect on Proof Search

To measure the effect of symbol reuse on proof search, we ran Vampire[2] (version 4.6.1) in its default (single-strategy) setting and its variations using symbol reuse on the mentioned 9091 FOF TPTP problems. The experiment was run on a server with Intel® Xeon® Gold 6140 CPUs clocked at 2.3 GHz with 500 GB RAM. To utilise the parallelism of our server while maintaining stability of the experiment we limited the runs using an instruction limit (rather than the more usual time limit).[3] More specifically, we used a limit of 50 billion instructions, which approximately corresponds to 10 seconds on the machine.

The result of our experiment are shown in Table 1. We can see that there is a consistent (although modest) improvement by both symbol reuse applications and that there is a combined benefit of using then jointly. The column with unique solutions reminds us that neither of the techniques is beneficial universally. However, as a sign of complementarity, unique solutions indicate that the newly available symbol reuse options will be useful at constructing more powerful strategy schedules.

To get a more robust version of the results, resistant to possible influence of statistical noise, we rerun the whole experiment in a shuffling mode, randomizing the prover at various don't-care non-deterministic call sites and averaging the results over many independently seeded runs (please check out our previous work [19] for the exact description of the

[2] https://github.com/vprover/vampire
[3] See appendix A of our previous work [19] for more details on instruction limiting.

Table 2
An analogue of Table 1 for problems *solved on average* (based on the methodology from [19]).

	av. solved	sigma
default	4281.3	10.4
dr	4281.2	10.5
skr	4287.2	10.4
skr+dr	4289.2	10.5

method). Table 2 reports on the computed averages as well as the estimated standard deviation (sigma). We can see that the positive effect of definition reuse on its own could not be confirmed, but the combined strategy using both definition reuse and Skolem symbol reuse is still the best one. All the averages are slightly lower than the values in Table 1 due to the overhead of shuffling (in accord with a remark in [19]).

We could identify a single TPTP problem as a *robust gain* of symbol reuse, namely the problem `LCL667+1.010`, which means the probability of solving this problem by *default* was 0.0 and by *skr+dr* 1.0, and no robust loss. Vampire reused 344 Skolem symbols on `LCL667+1.010`.

3.3. Bonus: Induction

We were pleasantly surprised to find that Vampire can now also reuse symbols generated *during* proof search with inductive reasoning. For inductive reasoning, Vampire uses many different inference rules [6, 15, 20, 21]. The simplest is of the form

$$\frac{\neg L[t] \vee C}{\texttt{cnf}(F \to \forall x.L[x])}$$

where t is a ground term, L is a ground literal, C is a clause, $\texttt{cnf}(\cdot)$ denotes a translation to clausal normal form, and $F \to \forall x.L[x]$ is a valid induction schema. The schema can be instantiated with various induction axioms, such as structural induction axiom for inductive datatypes or induction axiom with symbolic bounds for integers [6].

All clauses from the clausified conclusion of the rule contain the literal $L[x]$. After adding these clauses to the search space, Vampire immediately resolves them against the premise of the rule, $\neg L[t] \vee C$, obtaining

$$\texttt{cnf}(\neg F) \vee C. \tag{1}$$

Since the resulting clauses (1) do not contain the term t from $\neg L[t] \vee C$, the result of applying induction on $\neg L[t] \vee C$ is the same as on $\neg L[t'] \vee C$ for different t and t'. Vampire therefore reduces redundancy by applying the induction rule on a premise $\neg L[t'] \vee C$ only if it did not already apply induction with the same axiom on some $\neg L[t] \vee C$. However, this redundancy-avoiding heuristic does not work for some induction axioms having a more complex consequent, used in more-complex induction inference rules. One such case is the *upward integer induction axiom with default bound* [6]:

$$L[0] \wedge \forall y.(y \geq 0 \wedge L[y] \implies L[y+1]) \implies \forall x.(x \geq 0 \implies L[x]) \tag{2}$$

Consider applying the induction rule with the axiom (2) on the premise $\neg L[t]$. The result of clausifying and Skolemizing the axiom is:

$$\begin{gathered} \neg L[0] \vee s \geq 0 \vee x < 0 \vee L[x] \\ \neg L[0] \vee L[s] \vee x < 0 \vee L[x] \\ \neg L[0] \vee \neg L[s+1] \vee x < 0 \vee L[x] \end{gathered} \tag{3}$$

where s is a Skolem constant corresponding to y in (2). Resolving the clauses (3) with the premise $\neg L[t]$ results in clauses containing t:

$$\begin{gathered} \neg L[0] \vee s \geq 0 \vee t < 0 \\ \neg L[0] \vee L[s] \vee t < 0 \\ \neg L[0] \vee \neg L[s+1] \vee t < 0 \end{gathered} \tag{4}$$

If Vampire later encounters $\neg L[t']$, it applies induction with the same axiom (2) on it, since the resulting clauses will be different — they will contain t' instead of t. However, to do that, the negated antecedent of (2) needs to be clausified and Skolemized twice. Vampire can therefore apply symbol reuse to obtain the clauses:

$$\begin{gathered} \neg L[0] \vee s \geq 0 \vee t' < 0 \\ \neg L[0] \vee L[s] \vee t' < 0 \\ \neg L[0] \vee \neg L[s+1] \vee t' < 0 \end{gathered}$$

with the same Skolem constant s as in (4). In this way, symbol reuse has the potential to reduce work when using some integer induction rules. For example, on one benchmark[4] from the inductive benchmark set [22], Vampire can reuse 6 Skolem constants, and generates only 1430 clauses to find a proof compared to 1737 without symbol reuse.

4. Future Schemes

If $F \equiv G$, then the same symbol can be used to stand for F or G. However, since this criterion is undecidable for first-order logic, we suggest some more-pragmatic schemes below that we have not yet implemented. If, on the other hand, reusing symbols is of paramount importance, sub-formulae are typically small and the criterion easy compared to the main problem, then determining $F \equiv G$ with the theorem prover itself may be an option.

4.1. Normalised Formulae and AC Operators

Our existing scheme to detect α-equivalence (§2.2) still cannot identify even some trivially-equivalent formulae: consider e.g. $F \wedge G \equiv G \wedge F$. By a suitable choice of key function

[4] https://github.com/vprover/inductive_benchmarks/blob/master/benchmarks/int/val/smt2/declared_axall_conjall_valconst_unint.smt2

it would be possible to identify some such cases with little computational effort. More aggressive schemes are possible in exchange for more computation, such as with the key "sorted conjunctive normal form". We note that even more care must be taken with free variables (refer §2.3) as the order of occurrence of variables may be changed in the resulting keys. We suspect this scheme may be particularly useful when considering clausification *during* proof search [23].

4.2. Generalisation and Instantiation

The schemes discussed so far cannot reuse symbols for the following scenario. Suppose that we have an often-repeated formula $F[t_i]$ containing a series of different terms t_i: $F[t_1]$, $F[t_2]$ and so on, and that we wish to introduce symbols to represent these formulae. Such formulae are quite common in common-sense reasoning over large knowledge bases, such as those exported from SUMO [24].

To fix this shortcoming, we can generalise $F[t_i]$ to $F[x]$ for some fresh variable x, introduce a single symbol $s(x)$ and use $s(t_i)$ to represent each $F[t_i]$. However, producing candidate generalisations from a set of formulae is not easy to implement efficiently, and application of this technique would be necessarily heuristic since it is not clear which generalisation(s) are best for proof search. An alternative is to *maximally* generalise $k_{\text{gen}}(F)$ such that no non-variable terms occur in it. For example, consider

$$F \equiv P(c, g(d)) \wedge \neg Q(c, x)$$

Then, the reuse key is

$$k_{\text{gen}}(F) = P(x_0, x_1) \wedge \neg Q(x_2, x_3)$$

and the term $s(c, g(d), c, x)$ can be used to represent F. While the potential for symbol reuse is very high with this approach (particularly when combined with §4.1), this technique also increases symbol arity enormously, which is generally viewed negatively from the perspective of system performance.

5. Related Work

Computing normal forms [1] optimised for consumption [2] by automated theorem provers is well-studied. However, there is less work that aims to reduce the number of introduced *symbols* specifically, rather than reducing number or size of clauses. Other techniques in Vampire have achieved such a reduction as a side-effect [8, 15]. In the context of Spass [25], the technique of *generalized renaming* [26] is effective at reducing the number of introduced definitions. Generalized renaming is similar in spirit to §4.2 but with a greater degree of sophistication.

6. Conclusion

Reusing symbols is often overlooked as a topic for preprocessing, but can be effective, especially for systems using some amount of ground reasoning. We show that even a

simple form of reuse can be practically effective in Vampire and suggest some future schemes for reusing a greater number of symbols. The techniques proposed and evaluated here are relatively easy to implement, despite the hazards that we highlight, and are widely-applicable to other ATP systems.

Acknowledgments

Petra Hozzová and Michael Rawson were supported by the ERC CoG ARTIST 101002685 and the FWF grant LogiCS W1255-N23. Martin Suda was supported by the Czech Science Foundation project no. 20-06390Y and the project RICAIP no. 857306 under the EU-H2020 programme.

References

[1] M. Baaz, U. Egly, A. Leitsch, Normal form transformations, in: Handbook of automated reasoning, Elsevier, 2001, pp. 273–333.

[2] A. Nonnengart, C. Weidenbach, Computing small clause normal forms, in: Handbook of automated reasoning, Elsevier, 2001, pp. 335–367.

[3] E. Kotelnikov, L. Kovács, G. Reger, A. Voronkov, The Vampire and the FOOL, in: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, ACM, 2016, pp. 37–48.

[4] K. Claessen, N. Sörensson, New techniques that improve MACE-style finite model finding, in: Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications, Citeseer, 2003, pp. 11–27.

[5] A. Riazanov, A. Voronkov, Splitting without backtracking, in: B. Nebel (Ed.), Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Morgan Kaufmann, 2001, pp. 611–617.

[6] P. Hozzová, L. Kovács, A. Voronkov, Integer induction in saturation, in: Proceedings of CADE, volume 12699 of *LNCS*, Springer, 2021, pp. 361–377.

[7] L. Kovács, A. Voronkov, First-order theorem proving and Vampire, in: International Conference on Computer Aided Verification, Springer, 2013, pp. 1–35.

[8] G. Reger, M. Suda, A. Voronkov, New techniques in clausal form generation, in: GCAI 2016, volume 41 of *EPiC Series in Computing*, EasyChair, 2016, pp. 11–23.

[9] A. Voronkov, AVATAR: The architecture for first-order theorem provers, in: International Conference on Computer Aided Verification, Springer, 2014, pp. 696–710.

[10] G. Reger, N. S. Bjørner, M. Suda, A. Voronkov, AVATAR modulo theories, in: GCAI 2016, volume 41 of *EPiC Series in Computing*, EasyChair, 2016, pp. 39–52.

[11] K. Korovin, Instantiation-based automated reasoning: From theory to practice, in: International Conference on Automated Deduction, Springer, 2009, pp. 163–166.

[12] G. Reger, M. Suda, A. Voronkov, Unification with abstraction and theory instantiation in saturation-based reasoning, in: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer, 2018, pp. 3–22.

[13] G. Reger, M. Suda, The uses of SAT solvers in Vampire, in: Proceedings of the 1st and 2nd Vampire Workshops, volume 38 of *EPiC Series in Computing*, EasyChair, 2015, pp. 63–69.
[14] M. Rawson, G. Reger, A multithreaded Vampire with shared persistent grounding, in: FMCAD 2021, IEEE, 2021, p. 280.
[15] G. Reger, A. Voronkov, Induction in Saturation-Based Proof Search, in: P. Fontaine (Ed.), Proceedings of CADE, volume 11716 of *LNCS*, Springer, 2019, pp. 477–494.
[16] A. Bhayat, G. Reger, A polymorphic Vampire, in: International Joint Conference on Automated Reasoning, Springer, 2020, pp. 361–368.
[17] A. Bhayat, G. Reger, A combinator-based superposition calculus for higher-order logic, in: International Joint Conference on Automated Reasoning, Springer, 2020, pp. 278–296.
[18] G. Sutcliffe, The TPTP problem library and associated infrastructure, Journal of Automated Reasoning 43 (2009) 337–362.
[19] M. Suda, Vampire getting noisy: Will random bits help conquer chaos? (system description), EasyChair Preprint no. 7719, EasyChair, 2022.
[20] M. Hajdú, P. Hozzová, L. Kovács, J. Schoisswohl, A. Voronkov, Induction with Generalization in Superposition Reasoning, in: C. Benzmüller, B. Miller (Eds.), Proc. of CICM, volume 12236 of *LNCS*, Springer, 2020, pp. 123–137. doi:doi:10.1007/978-3-030-53518-6_8.
[21] M. Hajdu, P. Hozzová, L. Kovács, A. Voronkov, Induction with recursive definitions in superposition, in: 2021 Formal Methods in Computer Aided Design (FMCAD), IEEE, 2021, pp. 1–10.
[22] M. Hajdu, P. Hozzová, L. Kovács, J. Schoisswohl, A. Voronkov, Inductive benchmarks for automated reasoning, in: F. Kamareddine, C. Sacerdoti Coen (Eds.), Intelligent Computer Mathematics, Springer International Publishing, Cham, 2021, pp. 124–129.
[23] V. Nummelin, A. Bentkamp, S. Tourret, P. Vukmirovic, Superposition with first-class booleans and inprocessing clausification, in: Proceedings of CADE, volume 12699 of *LNCS*, Springer, 2021, pp. 378–395.
[24] A. Pease, G. Sutcliffe, N. Siegel, S. Trac, Large theory reasoning with SUMO at CASC, AI communications 23 (2010) 137–144.
[25] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, P. Wischnewski, Spass version 3.5, in: International Conference on Automated Deduction, Springer, 2009, pp. 140–145.
[26] N. Azmy, C. Weidenbach, Computing tiny clause normal forms, in: Proceedings of CADE, volume 7898 of *LNCS*, Springer, 2013, pp. 109–125.