EasyChair Preprint

№ 13125

A Higher-Order Vampire (Short Paper)

Ahmed Bhayat and Martin Suda

April 28, 2024

A Higher-Order Vampire

Ahmed Bhayat[1] and Martin Suda[2]

[1] Independent Scholar, Leicester, UK
ahmed_bhayat@hotmail.com
[2] Czech Technical University in Prague, Czech Republic
martin.suda@cvut.cz

Abstract. The support for higher-order reasoning in the Vampire theorem prover has recently been completely reworked. This rework consists of new theoretical ideas, a new implementation, and a dedicated strategy schedule. The theoretical ideas are still under development, so we discuss them at a high level in this paper. We also describe the implementation of the calculus in the Vampire theorem prover, the strategy schedule construction and several empirical performance statistics.

Keywords: Vampire · Higher-Order · Strategy Scheduling.

1 Introduction

The Vampire prover [15] has supported higher-order reasoning since 2019 [7]. Until recently, this support was via a translation from higher-order logic (HOL) to polymorphic first-order logic using combinators. The approach had positives, specifically it avoided the need for higher-order unification. However, our experience suggested that for problems requiring complex unifiers, the approach was not competitive with calculi that do rely on higher-order unification. This intuition was supported by results at the CASC system competition [25].

Due to this, we recently devised an entirely new higher-order superposition calculus. This time we based our calculus on a standard presentation of HOL. The key idea behind our calculus is that rather than using full higher-order unification, we use a depth-bounded version. That is, when searching for higher-order unifiers, when some predefined number of projection and imitation steps have taken place, the search is backtracked. The crucial difference in our approach to similar approaches is that rather than failing on reaching the depth limit, we turn the set of remaining unification pairs into negative constraint literals which are returned along with the substitution formed until that point. This is similar to recent developments in the field of theory reasoning [6].

The new calculus has now been implemented in Vampire along with a dedicated strategy schedule. Together these developments propelled Vampire to first place in the THF division of the 2023 edition of the CASC competition.[3] As the completeness of the calculus is an open question which we are working on, we have to date not published a description of the calculus.

[3] https://tptp.org/CASC/29/WWWFiles/DivisionSummary1.html

In this paper, we describe the calculus, discuss its implementation in Vampire and also provide some details of the strategy schedule and its formation.

2 Preliminaries

We assume familiarity with higher-order logic and higher-order unification. Detailed presentations of these can be found in recent literature [5,3,29]

We work with a rank-1 polymorphic, clausal, higher-order logic. For the syntax of the logic we follow a more-or-less standard presentation such as that of Bentkamp et al. [3]. Higher-order applications such as $f\,a\,c$ contain subterms with no first-order equivalents such as f and $f\,a$. We refer to these as *prefix* subterms. We represent term variables with x, y, z, function symbols with f, g, h, and terms with s and t. To keep the presentation simple, we omit typing information from our terms.

A substitution is a mapping of variables to terms. Unification is the process of finding a substitution σ for terms t_1 and t_2 such that $t_1\sigma \approx t_2\sigma$ for some definition of equality ($\approx$) of interest. It is well known that first-order syntactic unification is decidable and unique most general unifiers exists. For the higher-order case, unification is not decidable, and the set of incomparable unifiers is potentially infinite. A commonly used higher-order unification procedure for enumerating unifiers is Huet's preunification routine [13]. Unlike full higher-order unification, preunification does not attempt to unify terms if both have variable head symbols. Thus, preunification does not require infinitely branching rules unlike full higher-order unification [29].

The two main rules that extend first-order unification in Huet's procedure are *projection* adn *imitation*. We provide a flavour of these via an example. Consider unifying terms $s = x\,a$ and $s' = a$. In searching for a suitable instantiation of the variable x, we can either attempt to copy the head symbol of s' leading to the substitution $x \to \lambda y.\, a$, or we can bring one of x's arguments to the head position leading to the substitution $x \to \lambda y.\, y$. The first is known as imitation and the second as projetcion.

We use the concept of a *depth$_n$ unifier*. We do not define the term formally, but provide an intuitive understanding. Consider a higher-order preunification algorithm. Any substitution formed by following a path of the unification tree, starting from the root, that contains exactly n imitation and projection steps, or reaches a leaf using fewer than n such steps, is a *depth$_n$ unifier*. For terms s and t, let $U_n(s,t)$ be the set of all depth$_n$ unifiers of s and t. Note that this set is finite as we are assuming preunification and hence the tree is finitely branching.

For terms s and t, for each depth$_n$ unifier $\sigma \in U_n(s,t)$, we associate a set of negative equality literals C_σ formed by turning the unification pairs that remain when the depth limit is reached into negative equalities. In the case σ is an *actual unifier* of s and t, C_σ is of course the empty set.

To make this clearer, consider the unification tree presented in Figure 2. There are two depth$_2$ unifiers labelled σ_1 and σ_2 in the figure. Related to these,

we have $C_{\sigma_1} = C_{\sigma_2} = \{x_2\, a\, b \not\approx b\}$. There are four depth$_3$ unifiers (not shown in the figure) and zero depth$_n$ unifiers for for $n > 3$.

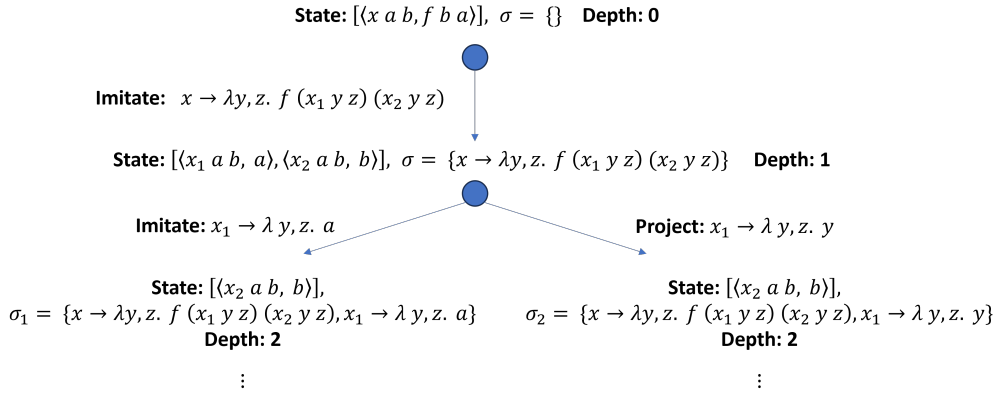

Fig. 1. Unification tree for terms $x\, a\, b$ and $f\, b\, a$

3 Calculus

Our calculus is parameterised by a selection function and an ordering $\succ$. Together these give rise to the concept of literals being (strictly) $\succ$-eligible with respect to a substitution σ [3]. When discussing eligibility we drop $\succ$ and σ and rely on the context to make these clear. We call a literal $s \not\approx t$, where both s and t have variable heads, a *flex-flex* literal. Such a literal is never selected in the calculus. We present the primary inference rule, Sup, below.

$$\frac{D' \vee t \approx t' \qquad C' \vee s\langle\, u \,\rangle \mathrel{\dot{\approx}} s'}{(C' \vee D' \vee s\langle\, t' \,\rangle \mathrel{\dot{\approx}} s' \vee C_\sigma)\sigma}\ \textsc{Sup}$$

In the rule above, we use $\mathrel{\dot{\approx}}$ to denote either a positive or negative equality. We use $s\langle\, u \,\rangle$ to denote that u is a *first-order* subterm of s. That is, a non-prefix subterm that is not below a lambda. The side conditions of the inference are $\sigma \in U_n(t, u)$, u is not a variable, $t \approx t'$ is strictly eligible in the left premise, $s\langle\, u \,\rangle \mathrel{\dot{\approx}} s'$ is eligible in the right premise, and the other standard ordering conditions. The remaining core inference rules are EqRes and EqFact.

$$\frac{C' \vee t \approx t' \vee s \approx s'}{(C' \vee t' \not\approx s' \vee s \approx s' \vee C_\sigma)\sigma}\ \textsc{EqFact} \qquad \frac{C' \vee s \not\approx t'}{(C' \vee C_\sigma)\sigma}\ \textsc{EqRes}$$

For both rules, $\sigma \in U_n(t, s)$. For EqFact, $s \approx s'$ is eligble in the premise and for EqRes $s \not\approx s'$ is eligble. We also include inferences ArgCong (see [4]), and FlexFlexSimp which derives the empty clause, $\bot$, from a clause containing only flex-flex literals.

$$\frac{C' \vee s \approx s'}{C'\sigma \vee (s\sigma)\, x \approx (s'\sigma)\, x}\ \textsc{ArgCong} \qquad \frac{x_1\, \overline{s}_n \not\approx x_2\, \overline{r}_m \vee \cdots}{\bot}\ \textsc{FlexFlexSimp}$$

For ArgCong, $s \approx s'$ is eligible in the premise, σ is the type unifier of s and s' and x is a fresh variable. In our implementation, the depth parameter n is set via a user option. In the case it is set to 0, the following pair of inferences are added to the calculus.

$$\frac{C' \vee x\, \overline{s}_n \not\approx f\, \overline{t}_m}{(C' \vee x\, \overline{s}_n \not\approx f\, \overline{t}_m)\{x \to \lambda \overline{y}_n.\, f\, \overline{(z_j\, \overline{y}_n)}_m\}}\ \textsc{Imitate}$$

$$\frac{C' \vee x\, \overline{s}_n \not\approx f\, \overline{t}_m}{(C' \vee x\, \overline{s}_n \not\approx f\, \overline{t}_m)\{x \to \lambda \overline{y}_n.\, y_i\, \overline{(z_j\, \overline{y}_n)}_p\}}\ \textsc{Project}$$

Where j ranges from 1 to m in Imitate and 1 to p in Project, and each z_j is a fresh variable. The literals $x\, \overline{s}_n \not\approx f\, \overline{t}_m$ are eligible in the premises and p is the arity of y_i, the projected variable. The idea behind introducing these rules is to facilitate the instantiation of head variables with suitable lambda terms when this is not being done as part of unification. Our intuition is that by intertwining the unification and calculus rules in the spirit of the EP calculus [21], the need for explosive rules (such as FluidSup[3]) that simulate superposition underneath variables is removed. The examples we present below support this intuition. Besides the core inference rules, the calculus has a set of rules to handle reasoning about Boolean terms. These are similar to rules discussed in the literature [30,20]. Extensionality is supported either via an axiom or by using unification with abstraction as described by Bhayat [5]. Similarly, Hilbert choice can be supported via a lightweight inference in the manner of Leo-III [20] or via the addition of the Skolemized choice axiom. The calculus also contains various well-known simplification rules such as Demodulation and Subsumption.

Soundness and Completeness. The soundness of the calculus described above is relatively straightforward to show. On the other hand, the completeness of the calculus with respect to Henkin semantics is an open question. We hypothesise that given the right ordering, and with tuning of inference side conditions, the depth_0 variant of the calculus (with the Imitate and Project rules) is refutationally complete. A proof is unlikely to be straightforward due to the fact that we do not select flex-flex literals.

Example 1. Consider the following unsatisfiable clause set. Assume a depth of 1. Selected literals are underlined.

$$C = \underline{x\, a\, b \not\approx f\, b\, a} \vee x\, c\, d \not\approx f\, b\, a$$

An EqRes binds x to $\lambda y, z.f(x_1\,a\,b)(x_2\,a\,b)$ and results in $C_1 = \underline{f\,(x_1\,a\,b)(x_2\,a\,b) \not\approx f\,b\,a} \vee f\,(x_1\,c\,d)(x_2\,c\,d) \not\approx f\,b\,a$. An EqRes on C_1 binds x_1 to $\lambda y, z.b$ and results in $C_2 = \underline{x_2\,a\,b \not\approx a} \vee f\,b\,(x_2\,c\,d) \not\approx f\,b\,a$. A final EqRes on C_2 binds x_2 to $\lambda y, z.a$ and results in $\underline{f\,b\,a \not\approx f\,b\,a}$ from which it is trivial to obtain the empty clause $\bot$.

Example 2 (Example 1 of Bentkamp et al. [4]). Consider the following unsatisfiable clause set. Assume the depth_0 version of the calculus.

$$C_1 = f\,a \approx c \qquad C_2 = h\,(y\,b)\,(y\,a) \not\approx h\,(g\,(f\,b))\,(g\,c)$$

An EqRes inference on C_2 results in $C_3 = y\,b \not\approx g\,(f\,b) \vee y\,a \not\approx g\,c$. An Imitate inference on the first literal of C_3 followed by the application of the substitution and some β-reduction results in $C_4 = g\,(z\,b) \not\approx g\,(f\,b) \vee g\,(z\,a) \not\approx g\,c$. A further double application of EqRes gives us $C_5 = z\,b \not\approx f\,b \vee z\,a \not\approx c$. We again carry out Imitate on the first literal followed by an EqRes to leave us with $C_6 = x\,b \not\approx b \vee f\,(x\,a) \not\approx c$. We can now carry out a Sup inference between C_1 and C_6 resulting in $C_7 = x\,b \not\approx b \vee c \not\approx c \vee x\,a \not\approx a$ from which it is simple to derive $\bot$ via an application of Imitate on either the first or the third literal. Note, that the empty clause was derived without the need for an inference that simulates superposition underneath variables, unlike in [4].

4 Implementation

The calculus described above, along with a dedicated strategy schedule, has been implemented in the Vampire theorem prover.[4] Vampire natively supports rank-1 polymorphic first-order logic. Therefore, we translate higher-order terms into polymorphic first-order terms using the well known applicative encoding. Note, that we use the symbol $\mapsto$, in a first-order type, to separate the argument types from the return type. It should not be confused with the binary, higher-order function type constructor $\to$ that we assume to be in the type signature. Application is represented by a polymorphic symbol $app : \Pi\alpha_1, \alpha_2.(\alpha_1 \to \alpha_2 \times \alpha_1) \mapsto \alpha_2$. Lambda terms are stored internally using De Bruijn indices. A lambda is represented by a polymorphic symbol $lam : \Pi\alpha_1, \alpha_2.\,\alpha_2 \mapsto (\alpha_1 \to \alpha_2)$. De Bruijn indices are represented by a family of polymorphic symbols $d_i : \Pi\alpha.\,\alpha$ for $i \in \mathbb{N}$. Thus, the term $\lambda x : \tau.\,x$ is represented internally as $lam(\tau, \tau, d_0(\tau))$. The term $\lambda x.\,f(\lambda z.x)$ is represented internally (now ignoring type arguments) as $lam(app(f, lam(d_1)))$.

Some of the most important options available are: `hol_unif_depth` to control the depth unification proceeds to, `funx_ext` to control how function extensionality is handled, `cnf_on_the_fly` to control how eager or lazy the clausification algorithm is, and `applicative_unif` which replaces higher-order unification with (applicative) first-order unification. This is surprisingly helpful in some cases. Besides for the options listed above, there are many other higher-order specific options as well as options that impact both higher-order and first-order reasoning. These options can be viewed by building Vampire and running with `-help`.

[4] See `http://bit.ly/3vBQLi4` for the release, `https://bit.ly/3Hl3lES` for the code.

5 Strategies and the Schedule

We generally followed the Spider [27] methodology for strategy discovery and schedule creation. This starts with randomly sampling strategies to solve as-of-yet unsolved problems (or improve the best known time for problems already known to be solvable). Each newly discovered strategy is optimized with local search to work even better on the single problem which it just solved. This is done by trying out alternative values for each option, possibly in several rounds. A variant of the strategy that improves the solution time or at least uses a default value of an option is preferred. The final strategy is then evaluated on the totality of all considered problems and the process repeats.

In our case, we sought strategies to cover the 3914 TH0 problems of the TPTP library [24] version 8.1.2. The strategy space consisted of 87 options inherited from first-order Vampire and 26 dedicated higher-order options. To sample a random strategy, we considered each option separately and picked its value based on a guess of how useful each is. (E.g., for `applicative_unif` we used the relative frequencies of `on`: 3, `off`: 10.) During the strategy discovery process we adapted the maximum running time per problem, both for the random probes several times and for the final strategy evaluation: from the order of 1 s up to 100 s. In total, we collected 1158 strategies over the course of approximately two weeks of continuous 60 core CPU computation. The strategies cover 2804 unsatisfiable problems, including 50 problems of TPTP rating 1.0 (which means these problems were not officially solved by an ATP before).

Once a sufficiently rich set of strategies gets discovered and evaluated, schedule building can be posed as a constraint programming task in which one seeks to allot time slices to individual strategies to cover as many problems as possible while not exceeding a given overall time bound T [12,19]. We had a good experience with a weighted set cover formulation and applying a greedy algorithm [9]: starting from an empty schedule, at any point we decide to extend it by scheduling a strategy S for additional t units of time if this step is currently the best among all possible strategy extensions in terms of "the number of problems that will additionally get covered *divided by* t". This greedy approach does not guarantee an optimal result, but runs in polynomial time and gives a meaningful answer uniformly for any overall time bound T. (See [8] for more details).

Our final higher-order schedule tries to cover, in this greedy sense, as many problems as possible at several increasing time bounds: starting from 1 s, 5 s, and 10 s bounds relevant for the impatient users, all the way up to the CASC limit of 16 minutes (2 minutes on 8 cores) and further beyond. In the end, it makes use of 278 out of the 1158 available strategies and manages to cover all the known-to-be-solvable problems in a bit less than 1 hour of single core computation. We stress that our final schedule is a single monolithic sequence and does not branch based on any problems' characteristics or features.[5]

[5] One additional interesting aspect of our schedule building approach (see Appendix A for more details) is that we employ input shuffling and prover randomization [23]

Table 1. The most important options in terms of contribution to problem coverage

an option	default	# problems not solvable without non-default
`cnf_on_the_fly`	`eager`	102
`applicative_unif`	`off`	56
`equality_to_equiv`	`off`	24
`hol_unif_depth`	`2`	20
`func_ext`	`abstraction`	12

Most important options: In Table 1, we list the first five options sorted in descending order of "how many problems we would not be able to cover if the given option could not be varied in strategies." (In other words, as if the listed default value was "wired-in" to the prover code.)

Based on existing research [28], it is unsurprising to see that varying clausification has a large impact. Likewise, for varying the unification depth. What is perhaps more surprising is that replacing higher-order unification with applicative first-order unification can be beneficial. `equality_to_equiv` turns equality between Boolean terms into equivalence before the original clausification pass is carried out. The effectiveness of this option is also somewhat surprising.

Table 2. Number of problems solved by a single good higher-order strategy and our schedule at various time limit cutoffs. Run on the 3914 TH0 TPTP problems

	1 s	10 s	30 s	60 s	120 s	960 s
single strategy	1811	1949	2041	2094	–	–
our schedule	2067	2436	2584	2642	2691	2775

Performance statistics: It is long known [26,31] that a strategy schedule can improve over the performance of a single good strategy by large margin. Table 2 confirms this phenomenon for our case. For this comparison we selected one of the best performing (at the 60 s time limit mark) single strategies that we had previously evaluated. From the higher-order perspective, the strategy is interesting for setting `hol_unif_depth` to 4 and supporting choice reasoning via an inference rule (`choice_reasoning on`).[6]

Although our schedule has been developed on (and for) the TH0 TPTP problems, it helps the new higher-order Vampire solve more problems of other origin too. Of the Sledgehammer problems exported by Desharnais et al. in their last prover comparison [10], namely the 5000 problems denoted in their work

and thus treat our strategies as Las Vegas algorithms, whose running time or even success/failure may depend on chance.

[6] Otherwise, it uses Vampire's default setting, except for relying on an incomplete literal selection function [11] and using a relative high naming threshold [17], i.e., being reluctant to introduce new names for subformulas during clausification.

TH0$^-$, Vampire can now solve 2425 compared to 2179 obtained by Desharnais et al. with the previous Vampire version (both under 30 s per problem).[7]

We remark that we also developed a different schedule specifically adapted to Sledgehammer problems (in various TPTP dialects, i.e., not just TH0), which is now available to the Isabelle [16] users since the September 2023 release.

6 Related Work

The idea to intertwine superposition and unification appears in earlier work, particularly in the EP calculus implemented in Leo-III [21]. The main differences between our calculus and EP are:

1. We do not move first-order unification to the calculus level. Hence, there are no equivalents to the Triv, Bind and Decomp rules of EP.
2. Our Project and Imitate rules are instances of EP's FlexRigid rule. We do not include an equivalent to EP's FlexFlex rule since we never select flex-flex literals. Instead, we leave such literals until one of the head variables becomes instantiated, or the clause only contains flex-flex literals at which point FlexFlexSimp can be applied.
3. Our core inference rules are parameterised by a selection function and an ordering.
4. Whilst EP always applies unification lazily, our calculus can control how lazily unification is carried out by varying the depth bound.[8]

We also incorporate more recent work on higher-order superposition, mainly from the Matryoshka project [28,3]. Of course, the use of constraints in automated reasoning extends far beyond the realm of higher-order logic. They have been researched in the context of theory reasoning [18,14] and basic superposition [2].

7 Conclusion

In this paper, we have presented a new higher-order superposition calculus and discussed its implementation in Vampire. We have also described the new higher-order schedule created. The combination of calculus, implementation and schedule have already proven effective. However, we believe that there is great room for further exploration and improvement. On the theoretical side, we wish to prove refutational completeness of the calculus (or a variant thereof). On the practical side, we wish to refine the implementation, most notably by adding additional simplification rules.

[7] Our experiments were run on Intel®Xeon®Gold 6140 CPU @ 2.3 GHz, Desharnais et al. [10] used StarExec [22] with Intel®Xeon®CPU E5-2609 @ 2.4 GHz nodes.

[8] Our understanding is that the implementation of EP in Leo-III does make use of orderings as well as eager unification. However, eager unification does not return unification literals, instead failing once the depth bound is reached. See [20] for details.

Acknowledgments. The second author was supported by project CORESENSE no. 101070254 under the Horizon Europe programme and project RICAIP no. 857306 under the EU-H2020 programme.

References

1. CASC design and organization. `https://www.tptp.org/CASC/29/Design.html`, Accessed: January 2024
2. Bachmair, L., Ganzinger, H., Lynch, C., Snyder, W.: Basic paramodulation and superposition. In: CADE. LNAI, vol. 607. Springer (1992)
3. Bentkamp, A., Blanchette, J., Tourret, S., Vukmirović, P.: Superposition for higher-order logic. Journal of Automated Reasoning **67**(1) (2023)
4. Bentkamp, A., Blanchette, J.C., Tourret, S., Vukmirović, P., Waldmann, U.: Superposition with lambdas. In: CADE. LNAI, vol. 11716, pp. 55–73. Springer (2019)
5. Bhayat, A.: Automated Theorem Proving in Higher-Order Logic. Ph.D. thesis (2015)
6. Bhayat, A., Korovin, K., Kovács, L., Schoisswohl, J.: Refining unification with abstraction. In: LPAR. pp. 36–47 (2023)
7. Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: IJCAR. LNAI, vol. 12166, pp. 278–296. Springer (2020)
8. Bártek, F., Chvalovský, K., Suda, M.: Regularization in spider-style strategy discovery and schedule construction. In: IJCAR (2024), accepted
9. Chvátal, V.: A greedy heuristic for the set-covering problem. Math. Oper. Res. **4**(3), 233–235 (1979)
10. Desharnais, M., Vukmirović, P., Blanchette, J., Wenzel, M.: Seventeen provers under the hammer. In: ITP. LIPIcs, vol. 237, pp. 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022)
11. Hoder, K., Reger, G., Suda, M., Voronkov, A.: Selecting the selection. In: Olivetti, N., Tiwari, A. (eds.) IJCAR. LNCS, vol. 9706, pp. 313–329. Springer (2016)
12. Holden, E.K., Korovin, K.: Heterogeneous heuristic optimisation and scheduling for first-order theorem proving. In: CICM. LNCS, vol. 12833, pp. 107–123. Springer (2021)
13. Huet, G.P.: A unification algorithm for typed λ-calculus. Theoretical Computer Science **1**(1), 27–57 (1975)
14. Korovin, K., Kovács, L., Reger, G., Schoisswohl, J., Voronkov, A.: ALASCA: Reasoning in quantified linear arithmetic. In: TACAS. LNCS, vol. 13993, pp. 647–665. Springer (2023)
15. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: CAV. LNCS, vol. 8044, pp. 1–35. Springer (2013)
16. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
17. Reger, G., Suda, M., Voronkov, A.: New techniques in clausal form generation. In: GCAI. EPiC Series in Computing, vol. 41, pp. 11–23. EasyChair (2016)
18. Reger, G., Suda, M., Voronkov, A.: Unification with abstraction and theory instantiation in saturation-based reasoning. In: TACAS. LNCS, vol. 10805, pp. 3–22. Springer (2018)
19. Schurr, H.: Optimal strategy schedules for everyone. In: PAAR. CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022)

20. Steen, A.: Extensional paramodulation for higher-order logic and its effective implementation Leo-III. Ph.D. thesis (2018)
21. Steen, A., Benzmüller, C.: Extensional higher-order paramodulation in Leo-III. Journal of Automated Reasoning **65**(6), 775–807 (2021)
22. Stump, A., Sutcliffe, G., Tinelli, C.: Starexec: A cross-community infrastructure for logic solving. In: IJCAR. LNCS, vol. 8562, pp. 367–373. Springer (2014)
23. Suda, M.: Vampire getting noisy: Will random bits help conquer chaos? (system description). In: IJCAR. LNCS, vol. 13385, pp. 659–667. Springer (2022)
24. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. Journal of Automated Reasoning **59**(4), 483–502 (2017)
25. Sutcliffe, G., Suttner, C.: The state of CASC. AI Communications pp. 35–48 (2006)
26. Tammet, T.: Towards efficient subsumption. In: Kirchner, C., Kirchner, H. (eds.) CADE. LNCS, vol. 1421, pp. 427–441. Springer (1998)
27. Voronkov, A.: Spider: Learning in the sea of options. In: Vampire23: The 7th Vampire Workshop (2023), `https://easychair.org/smart-program/Vampire23/2023-07-05.html#talk:223833`, to appear.
28. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. In: CADE. LNAI, vol. 12699, pp. 415–432. Springer (2021)
29. Vukmirović, P., Bentkamp, A., Nummelin, V.: Efficient full higher-order unification. Logical Methods in Computer Science **17** (2021)
30. Vukmirović, P., Nummelin, V.: Boolean reasoning in a higher-order superposition prover. In: PAAR. pp. 148–166 (2020)
31. Wolf, A., Letz, R.: Strategy parallelism in automated theorem proving. In: Cook, D.J. (ed.) FLAIRS. pp. 142–146. AAAI Press (1998)

A Note on Probabilistic Schedule Building

CASC organizers randomly shuffle the input problems to make sure that "no system receives an advantage or disadvantage due to the specific presentation" [1]. At the same time, it is well known that with a saturation-based prover even such small changes may have dramatic effect on strategy's running time [23]. To create a schedule resilient to input shuffling, we actually sampled our strategies with Vampire's internal shuffling enabled, ready to evaluate a strategy on a problem more than once (under different random seeds) to establish an estimate of its runtime distribution.

We then adapted the greedy algorithm to seek to cover problems "in expectation". By this we mean that a strategy can score a fractional point for solving a problem (if it solves it, e.g., in 50 % of its runs), where appropriately smaller fractions are awarded for problems already partially covered.

Our final schedule is then also executed under fresh random seeds and its performance, therefore, slightly varies depending on chance.

The estimate of a runtime distribution of a strategy on a given problem is explained on an example in Table 3. Note that we work with two possible failure modes: a deliberate giving up, which may happen with incomplete strategies, and termination through a timeout. The difference in interpretation is that a timeout could later have turned into a success if we had waited longer.

Table 3. Four independent example runs of a strategy on a given problem. The probability estimate for solving the given problem at intervals $I_1, \ldots, I_5$ changes between time moments $t_1, \ldots, t_4$ where one of the runs changes status from running (r) to either success (i.e, solved), timeout (solution interrupted), or gave up (premature failure).

time points and intervals	run_1	run_2	run_3	run_4	probability
I_1	r	r	r	r	0/4
t_1	success	r	r	r	
I_2		r	r	r	1/4
t_2		gave up	r	r	
I_3			r	r	1/4
t_3			timeout	r	
I_4				r	1/3
t_4				success	
I_5					2/3

We initially evaluate each strategy only once, as described in the main text. We then run the greedy schedule construction algorithm multiple times and iteratively reevaluate strategies on problems which the greedy algorithm reports are getting covered by them, thus getting gradually better probability estimates at points where they matter.

Covering problems in expectation uses the assumption of strategy independence. For example, knowing that the current schedule solves problem P with

a probability of 0.6 and a new strategy S added for t units of time solves P with the probability 0.8, adding S to the schedule will improve the coverage of P from 0.6 to 0.92, that is, by $r = (1 - 0.6) \cdot 0.8$. When computing the weight for the greedy covering, the strategy will accrue these r points for contributing to solving problem P.

The performance impact of the probabilistic approach can be observed in Figure 2. We can see that there is a substantial difference in the lower time limit bracket (around 90 problems at 1 s on TPTP and 426 problems on the Sledgehammer problems), which becomes less pronounced with higher time limits (50 problems at 30 s on TPTP and 31 on Sledgehammer there). The gap completely closes at around 300 s per problem on the Sledgehammer problems, from which point the deterministic schedule even mildly dominates. We currently do not have a good explanation for this last observation.

We conclude that investing into developing a probabilistic schedule on higher-order problems does pay off, especially at low time limits desired in applications with an impatient user, and carries over to in-training-unseen problems.

Fig. 2. A "cactus" plot comparing the performance of the probabilistic schedule with a deterministic one. Run on (upper) the 3914 TH0 problems from the TPTP library version 8.1.2 (training data) and on (lower) the 5000 TH0$^-$ problems from Desharnais et al. [10] (unseen during schedule construction). The time limit was 960 s per problem