A Polymorphic Vampire
(Short Paper)

Ahmed Bhayat(✉) and Giles Reger(✉)

University of Manchester, Manchester, UK
ahmed_bhayat@hotmail.com, giles.reger@manchester.ac.uk

Abstract. We have modified the Vampire theorem prover to support rank-1 polymorphism. In this paper we discuss the changes required to enable this and compare the performance of polymorphic Vampire against other polymorphic provers. We also compare its performance on monomorphic problems against standard Vampire. Finally, we discuss how polymorphism can be used to support theory reasoning and present results related to this.

1 Introduction

Vampire is a well known automated theorem prover for first-order logic with equality [14]. For a long period, Vampire supported only untyped first-order logic. Around 2011 it was extended to support monomorphic first-order logic (FOL). As part of recent work on supporting higher-order logic reported on elsewhere [1], Vampire has been extended to support rank-1, polymorphic, first-order logic[1].

Polymorphic types have a number of advantages over their monomorphic counterparts. Firstly, they provide the user with a more succinct language for describing their problem. Secondly, they provide an elegant solution to dealing with theories. For example, when dealing with the theory of arrays, rather than having to provide separate sets of axioms for arrays of different sorts, polymorphism allows us to provide a single set of axioms. Thirdly, polymorphism also permits higher-order logic to be finitely axiomatizable in first-order logic by introducing polymorphic axioms for the **SK**-combinators.

There are several ways of encoding polymorphism. However, many of these are cumbersome and some even unsound [8]. Blanchette et al. [2] list a number of common translation methods including the use of type tags, type guards and type arguments. Of these, the last is unsound and the first two cumbersome. As the type tags and guards are ubiquitous in the literature, we provide a comparison between native handling of polymorphism and the use of these encodings in Sect. 5. Given issues with encodings, it makes sense to deal natively with polymorphism if possible. We are certainly not the first to attempt to do so. Bobot et

[1] At the time of publication this extension exists in a separate branch from the main Vampire development. See https://vprover.github.io/download.html.

Problem 1 Sample TF1 problem (truncated)

```
tff(map, type, map : ($tType * $tType) > $tType).
tff(lookup, type,
    lookup : !>[A : $tType, B : $tType]: ((map(A, B) * A) > B)).
tff(update, type,
    update : !>[A : $tType, B : $tType]:
           ((map(A, B) * A * B) > map(A, B))).
tff(lookup_update_same, axiom,
    ![A : $tType, B : $tType, M : map(A, B), K : A, V : B]:
        lookup(A, B, update(A, B, M, K, V), K) = V).
tff(update_idem, conjecture,
    ![A : $tType, B : $tType, M : map(A, B), K : A, V : B]:
        update(A, B, update(A, B, M, K, V), K, V) =
        update(A, B, M, K, V)).
```

al. [5] have introduced polymorphism into their SMT solver Alt-Ergo. Similarly, the first-order provers ZenonModulo [11] and Zipperposition [9] support some form of polymorphism. However, it remains the case that few first-order provers can handle polymorphism.

In this short paper we begin by describing the relatively modest changes that had to be made to Vampire to support polymorphism (Sect. 2). We then present experimental results demonstrating that these changes are useful (Sect. 3). Finally, we discuss work-in-progress to use these polymorphic extensions to improve theory reasoning in Vampire both in terms of proof search and implementation (Sect. 4).

Before this, we give a brief (and informal) reminder of what rank-1 polymorphism is. A polymorphic type is a type variable, or n-ary type constructor applied to n types. The type of all types is represented as `$tType` in TPTP syntax [17] which is used throughout this paper. Terms, in polymorphic FOL, are either a variable or a function symbol applied to m type arguments and n term arguments. Rank-1 polymorphism allows type and term variables to be quantified with the rule that an existentially quantified type may not occur underneath a universal term quantifier. On skolemisation such a construct would become a dependent type and require superposition into types.

2 Implementation

To support polymorphism modifications had to be carried out in three main areas. Firstly, changes had to be made to the concept of types in Vampire. Secondly, some inferences had to be modified slightly and finally preprocessing required consideration. We describe the work undertaken in this order.

Problem 1 is an example of a problem in TPTP TF1 syntax [3,17]. We use this problem to illustrate our implementation. The major change undertaken was to replace types with terms. In monomorphic Vampire each type in the input problem is stored as an unsigned integer. Function symbols are then assigned a type signature which is merely a list of unsigned integers representing the argument and return types. In polymorphic first-order logic, types have all the structure of terms. Therefore, it made sense to replace the types with terms. Type signatures then become a list of terms.

The type of a term of the form `func_sym(arg_1 ... arg_n)` can be calculated by substituting the type arguments that the head symbol is applied to into its result type. For example, the type of `update($int, $i, map, 1, X)` would be `map($int, $i)`. For a term `func_sym(arg_1 ... arg_n)`, the type of the ith term argument can be calculated in the same way. The one problem that arises is with two variable literals such as `X = Y`. In this case, the type of the terms `X` and `Y` have to be stored as a separate field in the literal.

The elegance of treating types as terms can be gauged when attention is turned to unification. Had types and terms been kept separate, unifying terms would have become an involved process requiring the unification of term and type arguments separately. As it is, type unification comes for 'free' with one caveat as shall be seen. Consider unifying the terms `update($int, $i, map, 1, X)` and `update(Y, Z, map, Z', a)`. The existing unification procedure in Vampire can handle this and return the type and term unifier $\{\texttt{Y} \rightarrow \texttt{\$int}, \texttt{Z} \rightarrow \texttt{\$i}, \texttt{Z'} \rightarrow 1, \texttt{X} \rightarrow \texttt{a}\}$. The one hitch occurs when unifying a term with a variable. As variables carry no type information, a second call must be made to the unification procedure to ensure that the type of the variable and the type of the term are unifiable.

As far as changes to inferences are concerned, no updates were required for inferences that do not work at subterms such as resolution or equality factoring. For inferences that work at subterms such as superposition and demodulation, we modified the iterators that return candidate subterms so that they do not return type arguments as superposition into types is unnecessary. We mentioned that the modifications required to support polymorphism were light. They also (in theory, see later experiment) add no overhead when dealing with monomorphic problems. In this case all types are constants and unifiability checking of types in the variable case degenerates to a syntactic equality check.

Finally, regarding preprocessing, implementing skolemisation posed a subtle issue. A skolem function must be applied to the free term and *type* variables in its context. For example, the skolemisation of `![X: $int, Y: $tType] : ?[Z : $i] : (func_sym(Y, X, Z))` would be `![X: $int, Y: $tType] : ?[Z : $i] : (func_sym(Y, X, sk(Y, Z)))`. This required us to update the notion of free variable within the code (e.g., when iterating over the free variables of a formula).

3 Results

To test our implementation we ran two experiments. All experiments were carried out with a CPU time limit of 300 s on StarExec [16] nodes equipped with four

Table 1. Problems proved theorem or unsat

	TF1 problems		TF0 problems	
	Solved	Uniques	Solved	Uniques
Leo-III 1.4	224	10	8,665	100
Vampire 4.4	–	–	**11,338**	**476**
Vampire-poly	**239**	**21**	10,641	88
ZenonModulo 0.4.2	80	1	2,926	14

2.40 GHz Intel Xeon CPUs. Our experimental results are publicly available.[2] All solver configurations used where taken from the last CASC in which that solver was entered. The portfolio of strategies for the two Vampire variants were the same.

Experiment 1. Firstly we ran Vampire on the set of 539 TF1 (rank-1 polymorphic) problems in the TPTP library. We compared the results against those of two other provers able to parse TPTP syntax and handle polymorphism that we are aware of, Leo-III [15] and ZenonModulo [11].[3] Vampire solved 15 more problems than Leo-III and 21 problems that neither Leo-III nor ZenonModulo could solve (see Table 1), although both solvers also solved problems Vampire was unable to solve. Vampire solves 7 previously unsolved rating 1.00 problems.

Experiment 2. We also wanted to ascertain how much overhead had been added for non-polymorphic problems, so we tested the polymorphic version of Vampire, Vampire-poly, against the previous version on the set of all 33,843 monomorphic or untyped first order problems in the TPTP library not containing arithmetic. Note that this simply tests whether we go from solving a problem to not solving it (or vice versa) and not the time taken to find a solution, i.e., we test the impact on proof search and whether any time overhead takes us past the given time limit.

The results (see Table 1) are interesting. For TF0 problems Vampire 4.4 does indeed outperform its polymorphic sibling. However, at the time of writing, there is a bug in the polymorphic parser that resulted in 324 problems being incorrectly rejected. Even taking this into account, the performance of Vampire-poly lags behind and the cause of this remains to be fully investigated, although is likely to be due to the fragile nature of proof search in Vampire. Note that Vampire-poly solves 88 problems unsolved by Vampire 4.4.

[2] https://github.com/vprover/vampire_publications/tree/master/experimental_data/IJCAR-2020-POLY-VAMP - this contains the results themselves and a link to the Vampire executable that produced them. Polymorphism is not yet supported in the main branch of Vampire but is available in the `polymorphic_vampire` branch, which may be merged in the future.

[3] At a late stage, we realised that Zipperposition [10] can also parse TF1 syntax. Unfortunately, it was too late to incorporate it into the results.

4 Polymorphism and Theory Reasoning

Vampire has built-in support for the polymorphic theory of arrays [12] and the polymorphic theory of first-class tuples [13]. Here we briefly discuss work-in-progress to improve the implementation of these theories (and future similar theories) using polymorphism.

Both theories are supported by detecting instances of the polymorphic theory and adding the relevant instances of that theory's axioms to the input problem. For example, for the polymorphic theory of arrays, for each array sort `array(t1, t2)` detected in the input problem, we add instances of the axioms

```
![V:array(t1, t2), X: t1, Y: t2, Z:t2] : (Y = Z
    => select(store(V, Y, X), Z) = X)
![V:array(t1, t2), X: t1, Y: t2, Z:t2] : (Y != Z
    => select(store(V, Y, X), Z) = select(V, Z))
![V:array(t1, t2), X: array(t1, t2), Y: t1] :
    (select(V, Y) = select(X, Y) => V = X)
```

With support for polymorphism, as soon as we detect that arrays of any kind are used we can simply add the three polymorphic axioms

```
!>[T1: $tType, T2: $tType]:
  ( ![V:array(T1, T2), X: T1, Y: T2, Z:T2] :
     (Y = Z => select(store(V, Y, X), Z) = X))
!>[T1: $tType, T2: $tType]:
  ( ![V:array(T1, T2), X: T1, Y: T2, Z: T2] :
     (Y != Z  => select(store(V, Y, X), Z) = select(V, Z)))
!>[T1: $tType, T2: $tType]:
  ( ![V:array(T1, T2), X: array(T1, T2), Y: T1] :
     ((select(V, Y) = select(X, Y) => V = X))
```

This has a minor impact on proof search. Instead of adding $3n$ clauses when we have n different instances of the polymorphic theory, we only add 3 clauses. As n is usually small, this is unlikely to have a significant impact. At the same time, we should not see any negative impact, these polymorphic axioms will act in the same way as the $3n$ instances did.

The main impact is on the implementation of theories within Vampire. A non-trivial amount of complexity is required within the Vampire codebase to support the current mechanisms for the two supported polymorphic theories. Adding a new polymorphic theory of this kind involves a lot of boilerplate code and updating of various definitions. Replacing this with polymorphic theory axioms will simplify the code significantly. For example, if the SMT-LIB language is extended to support polymorphism in the future (this has been discussed, e.g., [6] but not implemented), internal support for polymorphism would make supporting the polymorphic theory of term algebras straightforward.

Moreover, not all polymorphic theories are supported by the mechanism described above; our current approach of adding instantiated axioms based on the input is complete for the theory of arrays, but cannot be complete in general as shown by Bobot and Paskevich [4]. For the theory of combinatory logic for example, no decision procedure can exist for selecting a set of monomorphic combinator axioms to add to a problem and ensure completeness (even though such a set must exist).

5 Related Work

The polymorphism of TPTP's TF1 language is inspired by ML-style polymorphism but differs in the use of type quantifiers. As pointed out by Blanchette et al. [3], ML-style polymorphism avoids explicit type quantifiers, choosing to determine type signatures by the types of arguments, results or additional annotations (which are sometimes needed to guide Hindley-Milner type inference). Comparatively, type checking is more straightforward in TF1 due to an explicit signature and explicit type quantifiers.

As mentioned earlier, there are two main methods for reasoning in polymorphic logic: natively or via translations. We discuss related work for each direction.

Zipperposition [9] was built using *explicit polymorphism* – types are explicitly represented in terms and inferences perform unification on both terms and types. The main difference with our approach is that we are 'retro-fitting' polymorphism into a monomorphic theorem prover. Additionally, our 'types as terms' internal representation (mostly) removes the additional book-keeping required when performing separate term and type unification.

There are three main approaches to translation - type tags, type guards, and type arguments. The purpose of both the type tag and type guard encoding is to ensure that unsound inferences that violate typing constraints cannot occur in the untyped problem. We do not provide details of the encodings here, but refer readers to [2] with a further example given by Brown et al. [7] in their work translating between different TPTP formalisms. Consider the following satisfiable polymorphic problem with a polymorphic predicate `p`:

```
tff(a,type, p : !> [X : $tType] : X > $o).
tff(b, conjecture, ?[X:$i, Y:$int] : p($i,X) => p($int,Y)).
```

The negated conjecture becomes the two clauses `~p($i,X)` and `p($int,Y)`. Clearly, if we drop the types (i.e. via type erasure) then this satisfiable problem becomes unsatisfiable as we can no longer differentiate between the two versions of `p`. Using type tags we would get `~p(ti(X, $i))` and `p(ti(Y, $int))` and with type guards we would get `~isi(X) | ~p (X)` and `~isint(Y) | p(Y)` – both prevent the unsatisfiability from type erasure at the expense of introducing extra functions or predicates. We achieve the same through type inference and unification. The type argument translation looks similar to our internal representation of types, e.g. types are encoded as terms. However, without being aware of the type of equalities where at least one side is a variable (as we are in our translation) this encoding can be unsound as equalities can capture cardinality constraints between types.

6 Conclusion

We have successfully extended a state-of-the-art first-order prover to support prenex polymorphism and shown that the difficulty in doing so is not as great as may be expected. We hope to encourage other researchers to do the same.

Theoretically, extending Vampire to polymorphic FOL should be graceful in the sense that no degradation of performance should be seen on non-polymorphic problems. Our experimental results do not bear this out. In future work, we hope to achieve two objectives. Firstly, to fix and refine our implementation of polymorphism such that no degradation on monomorphic or untyped problems is experienced. Secondly, as outlined above, to utilise polymorphism to simplify and extend theory reasoning in Vampire for polymorphic theories such as arrays.

References

1. Bhayat, A., Reger, G.: A combinator-based superposition calculus for higher-order logic. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020. LNCS, vol. 12166, pp. 278–296. Springer, Heidelberg (2020)
2. Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 493–507. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36742-7_34
3. Blanchette, J.C., Paskevich, A.: TFF1: the TPTP typed first-order form with rank-1 polymorphism. In: Bonacina, M.P. (ed.) CADE 2013. LNCS (LNAI), vol. 7898, pp. 414–420. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38574-2_29
4. Bobot, F., Paskevich, A.: Expressing polymorphic types in a many-sorted language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS (LNAI), vol. 6989, pp. 87–102. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24364-6_7
5. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: ACM International Conference Proceeding Series, pp. 1–5 (2008)
6. Bonichon, R., Déharbe, D., Tavares, C.: Extending SMT-LIB v2 with λ-terms and polymorphism. In: SMT, pp. 53–62. Citeseer (2014)
7. Brown, C.E., Gauthier, T., Kaliszyk, C., Sutcliffe, G., Urban, J.: GRUNGE: a grand unified ATP challenge. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 123–141. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_8
8. Couchot, J.-F., Lescuyer, S.: Handling polymorphism in automated deduction. In: Pfenning, F. (ed.) CADE 2007. LNCS (LNAI), vol. 4603, pp. 263–278. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73595-3_18
9. Cruanes, S.: Extending superposition with integer arithmetic, structural induction, and beyond. Ph.D. thesis, INRIA (2015)
10. Cruanes, S.: Superposition with structural induction. In: Dixon, C., Finger, M. (eds.) FroCoS 2017. LNCS (LNAI), vol. 10483, pp. 172–188. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66167-4_10

11. Delahaye, D., Doligez, D., Gilbert, F., Halmagrand, P., Hermant, O.: Zenon Modulo: when achilles outruns the tortoise using deduction modulo. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) LPAR 2013. LNCS, vol. 8312, pp. 274–290. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-45221-5_20
12. Kotelnikov, E., Kovács, L., Reger, G., Voronkov, A.: The vampire and the FOOL. In: Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, pp. 37–48 (2016)
13. Kotelnikov, E., Kovács, L., Voronkov, A.: A FOOLish encoding of the next state relations of imperative programs. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 405–421. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_27
14. Kovács, L., Voronkov, A.: First-order theorem proving and Vampire. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 1–35. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_1
15. Steen, A., Benzmüller, C.: The higher-order prover Leo-III. In: Galmiche, D., Schulz, S., Sebastiani, R. (eds.) IJCAR 2018. LNCS (LNAI), vol. 10900, pp. 108–116. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-94205-6_8
16. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a cross-community infrastructure for logic solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) IJCAR 2014. LNCS (LNAI), vol. 8562, pp. 367–373. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08587-6_28
17. Sutcliffe, G.: The TPTP problem library and associated infrastructure. J. Autom. Reasoning **43**(4), 337–362 (2009). https://doi.org/10.1007/s10817-009-9143-8