Iterative Monomorphisation

Tanguy Bozec[1,2](✉) and Jasmin Blanchette[2]

[1] ENS Paris-Saclay, Université Paris-Saclay, France
[2] Ludwig-Maximilians-Universität München, Germany

Abstract. Monomorphisation can be used to extend monomorphic provers to support polymorphic logics. We describe a pragmatic iterative approach. We implemented it in the Zipperposition prover, where it is used to translate away polymorphism before invoking the monomorphic prover E as a backend. Our evaluation shows that this approach increases Zipperposition's success rate. Moreover, we find that iterative monomorphisation outperforms some native implementations of polymorphism.

Keywords: Polymorphism · monomorphism · automated reasoning

1 Introduction

Automatic theorem provers provide automation for proof assistant users. Many proof assistants, such as HOL4 [16], HOL Light [9] and Isabelle/HOL [13], support rank 1 polymorphism, where type quantification is allowed at the top level of formulae. By contrast, many automatic provers only work with monomorphic logics. One way to close this gap is to extend automatic provers to natively support polymorphism, as has been done in Vampire [1]. However, this entails a lot of work that must be redone for every prover. The alternative is to translate polymorphic problems to monomorphic problems.

To achieve this, one approach [2] is to encode a complete polymorphic type system, but this increases the size of the input problem substantially and slows down provers [2]. Another approach to encode polymorphism is based on iterative monomorphisation, as described by Böhme [5, Section 2.2.1]. Iterative monomorphisation heuristically instantiates the formulae's type variables with concrete types. However, any translation relying on a finite number of instantiations is inevitably incomplete. By a typed version of the compactness theorem, for any first order polymorphic formula φ, there exists an equisatisfiable finite set of monomorphic instances of φ, but it cannot be computed in general [4, Theorem 1].

Böhme's approach is implemented as part of Isabelle/HOL's SMT (satisfiability modulo theories) integration [5, Chapter 2]. This implementation is also used by Sledgehammer [6,14] to interface with superposition based automatic theorem provers. However, it is documented only superficially [5, Section 2.2.1]. An iterative monomorphisation approach is also described in the context of SMT-LIB [7]. Moreover, a similar algorithm appears to be implemented in the MESON tactic [8] of HOL Light, but it is undocumented.

In this paper, we present an algorithm based on our understanding of Böhme's description and implementation (Section 3). We also provide a more detailed description to help future implementers. In addition, we present some optimisations to curb the combinatorial explosions (Section 4).

The algorithm works as follows. The input problem is a set of formulae. All the problem's symbols are collected, and the polymorphic symbol instances are matched against the monomorphic ones. This yields new symbol instances, both polymorphic and monomorphic. The process is then iterated, making use of the newly generated instances. Consider the unary type constructor list. If a formula contains $\mathsf{list}(\alpha)$, where α is a type variable, the types $\mathsf{list}(\mathsf{int})$, $\mathsf{list}(\mathsf{list}(\mathsf{int}))$, etc., can be generated. However, because new types emerge through matching, $\mathsf{list}(\mathsf{list}(\mathsf{int}))$ can be obtained only once the $\mathsf{list}(\mathsf{int})$ instance has been generated.

To keep the number of generated formulae finite, we limit the number of iterations. After the iterations are completed, the new monomorphic symbol instances are used to instantiate the polymorphic symbols in the problem's formulae, generating new monomorphic formulae. Finally, because monomorphic provers support only nullary type constructors, types must be 'mangled'; for example, the compound type $\mathsf{list}(\mathsf{int})$ might be mangled to $\mathsf{list\_int}$.

We implemented iterative monomorphisation in Zipperposition [18], a higher order prover written in OCaml. Although Zipperposition is polymorphic, it uses the monomorphic prover E [15] as a backend. Thanks to our work, E can now be used with polymorphic problems. Moreover, our implementation can be used as a preprocessor for other stand-alone provers. Our source code is available online.[3] Our evaluation on the TPTP [17] tries to answer three questions (Section 6):

1. Is the new Zipperposition with the E backend more successful on polymorphic problems than Zipperposition without backend?
2. How competitive are monomorphic provers on monomorphised problems?
3. Is iterative monomorphisation more effective than the native polymorphism implemented in polymorphic provers?

2 Preliminaries

Our algorithm works independently of the structure of the problem's formulae. It relies exclusively on the formulae's monomorphic and polymorphic symbol instances. Type variables are assumed to be implicitly universally quantified at a formula's top level. The precise form of formulae is left unspecified. Due to this generality, iterative monomorphisation can be used with any standard variant of rank 1 polymorphic logic. In particular, it can work with the polymorphic first and higher order logics embodied by TPTP's TF1 and TH1 syntaxes [3,11].

Our abstract framework relies on the following basic definitions.

Definition 1. A (*polymorphic*) *type* τ is a type variable (e.g. α) or the application of an n-ary type constructor to n types (e.g. $\mathsf{list}(\alpha)$, $\mathsf{map}(\mathsf{int}, \mathsf{string})$). If $n = 0$, we omit the parentheses (e.g. int). A type is *monomorphic* if it contains no type variables.

[3] `https://github.com/nartannt/iterative_monomorphisation`

Definition 2. A (function or predicate) symbol f has a *type arity* that specifies the number of type arguments it takes. A *symbol instance* is a symbol applied to type arguments listed between angle brackets: $f\langle\tau_1,\ldots,\tau_n\rangle$, where each τ_i is a type. If $n = 0$, we omit the angle brackets (e.g. f).

Definition 3. A (*type*) *substitution* is a partial function mapping a finite number of type variables to corresponding types. Substitutions are written as $\sigma = \{\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n\}$. They are assumed to be lifted to formulae; thus, $\sigma(\varphi)$ yields the variant of φ in which each α_i is replaced by τ_i. Given two substitutions τ, υ, the successive application of τ and υ is denoted by $\upsilon \circ \tau$.

Definition 4. Two substitutions $\{\alpha_1 \mapsto \tau_1, \ldots, \alpha_m \mapsto \tau_m\}$ and $\{\beta_1 \mapsto \upsilon_1, \ldots, \beta_n \mapsto \upsilon_n\}$ are said to be *compatible* if $\alpha_i = \beta_j$ implies $\tau_i = \upsilon_j$ for all i, j.

Definition 5. Given two types τ, υ, *matching* υ against τ either fails or yields a substitution σ such that $\sigma(\upsilon) = \tau$.

3 High level algorithm

The iterative monomorphisation algorithm takes a polymorphic problem as input and returns a monomorphic problem. It applies a bounded number of iterations, each taking a polymorphic problem as argument and returning a problem with new partially instantiated formulae. Once the iterations are completed, a final step discards all non-monomorphic formulae.

The initial phase of each iteration computes two maps, M and N, from the input problem Φ.

– Given a symbol f occurring in Φ, the set $M(f)$ consists of all monomorphic type argument tuples to which f is applied in Φ. For example, if $\mathsf{foldl}\langle\mathsf{nat},\mathsf{int}\rangle$ occurs in Φ, then $(\mathsf{nat},\mathsf{int}) \in M(\mathsf{foldl})$.
– Given a formula $\varphi \in \Phi$ and a symbol f occurring in φ, the set $N(\varphi)(f)$ consists of all type argument tuples to which f is applied in φ and which contain a type variable. For example, if $\mathsf{foldl}\langle\mathsf{nat},\mathsf{list}(\alpha)\rangle$ occurs in φ, then $(\mathsf{nat},\mathsf{list}(\alpha)) \in N(\varphi)(\mathsf{foldl})$.

N is parametrised with φ because type variables are implicitly quantified at the formula level. The formula indicates the scope of type variables. This is not necessary for M since all the types it contains are monomorphic. To avoid copying the monomorphic types for each formula, M and N are kept separate.

Once the maps M and N are initialised, each iteration performs the following steps to create new instances of formulae:

1. Create an empty set of formulae Φ'.
2. For each formula $\varphi \in \Phi$ and for each symbol f occurring in φ:
   2.1. For each $(\tau_1,\ldots,\tau_n) \in M(f)$ and $(\upsilon_1,\ldots,\upsilon_n) \in N(\varphi)(f)$ and for each i, match υ_i against τ_i, yielding the substitution σ_i in case of success.
   2.2. If all n matchings are successful and the substitutions σ_i are pairwise compatible, add the formula $(\sigma_1 \circ \cdots \circ \sigma_n)(\varphi)$ to Φ'.

3. Return $\Phi \cup \Phi'$.

The algorithm is trivially sound because the newly generated formulae are instances of the initial problem's formulae. However, it is not complete.

Example 6 Consider the following problem:

$\langle 1 \rangle$ $\mathsf{p}\langle\mathsf{int}\rangle(0)$
$\langle 2 \rangle$ $\forall a : \alpha,\ as : \mathsf{list}(\alpha).\ \mathsf{p}\langle\alpha\rangle(a) \rightarrow \mathsf{p}\langle\mathsf{list}(\alpha)\rangle(as)$

The first iteration matches α against int for p, generating the formula

$\langle 3 \rangle$ $\forall a : \mathsf{int},\ as : \mathsf{list}(\mathsf{int}).\ \mathsf{p}\langle\mathsf{int}\rangle(a) \rightarrow \mathsf{p}\langle\mathsf{list}(\mathsf{int})\rangle(as)$

The second iteration matches α against $\mathsf{list}(\mathsf{int})$, leading to the formula

$\langle 4 \rangle$ $\forall a : \mathsf{list}(\mathsf{int}),\ as : \mathsf{list}(\mathsf{list}(\mathsf{int})).\ \mathsf{p}\langle\mathsf{list}(\mathsf{int})\rangle(a) \rightarrow \mathsf{p}\langle\mathsf{list}(\mathsf{list}(\mathsf{int}))\rangle(as)$

Similarly the third iteration adds

$\langle 5 \rangle$ $\forall a : \mathsf{list}(\mathsf{list}(\mathsf{int})),\ as : \mathsf{list}(\mathsf{list}(\mathsf{list}(\mathsf{int}))).$
$\mathsf{p}\langle\mathsf{list}(\mathsf{list}(\mathsf{int}))\rangle(a) \rightarrow \mathsf{p}\langle\mathsf{list}(\mathsf{list}(\mathsf{list}(\mathsf{int})))\rangle(as)$

This example illustrates how an infinite number of new formulae can be generated from a simple initial problem. Any reasonable implementation must limit the number of new type arguments, substitutions and formulae.

4 Low level algorithm

The algorithm presented above is too naïve in practice. In this section, we present a lower level algorithm with the following features. First, numeric bounds are introduced to stop combinatorially explosive enumerations. Second, type argument tuples are separated into an old set and a new set to avoid re-computing some of the same matchings in successive iterations. Third, substitutions are directly applied to the type arguments instead of the formulae. This avoids having to re-extract the type arguments from the formulae at each iteration. New formulae are generated only once all iterations are completed, in a separate, final step.

The data structures used in the algorithm are based on the ones used in the high level description. Instead of a map M from symbols to monomorphic type argument tuples, we now have M_{old} and M_{new}, which play the same role whilst also distinguishing between those type argument tuples that have already been matched against and those that have not. Similarly, N_{old} and N_{new} replace the map N from formulae to symbols to non-monomorphic type argument tuples. Finally, we maintain a map S from formulae to the substitutions generated by the matchings. It is used to generate new formulae in the final phase.

All sets referenced in the algorithm are finite. Moreover, the algorithm relies on primitives whose implementation depends on the specifics of the grammar and logic used. Functions computing the following are assumed to be available:

– *initialisation*(Φ), where Φ is a set of (polymorphic) formulae, extracts the initial type argument maps M and N from Φ.

```
Function $iterative\_monomorphisation(\Phi)$
    Data: set $\Phi$ of polymorphic formulae
    Result: set of monomorphic formulae
    $(M_{\text{old}}, N_{\text{old}}) \leftarrow (\emptyset, \emptyset)$
    $(M_{\text{new}}, N_{\text{new}}) \leftarrow initialisation(\Phi)$
    $S \leftarrow \emptyset$
    for $i = 1$ to num_loops do
        $(M_{\text{next}}, N_{\text{next}}) \leftarrow (\emptyset, \emptyset)$
        foreach $\varphi \in \Phi$ do
            $(M_\Delta, N_\Delta, S_\Delta) \leftarrow$
              $formula\_mono\_step(\varphi, M_{\text{old}}, M_{\text{new}}, N_{\text{old}}(\varphi), N_{\text{new}}(\varphi))$
            $S(\varphi) \leftarrow S(\varphi) \cup S_\Delta$
            $M_{\text{next}} \leftarrow M_{\text{next}} \cup M_\Delta$
            $N_{\text{next}}(\varphi) \leftarrow N_{\text{next}}(\varphi) \cup N_\Delta$
        $(M_{\text{old}}, M_{\text{new}}) \leftarrow (M_{\text{old}} \cup M_{\text{new}}, M_{\text{next}})$
        $(N_{\text{old}}, N_{\text{new}}) \leftarrow (N_{\text{old}} \cup N_{\text{new}}, N_{\text{next}})$
    return $mangle(generate\_mono\_formulae(\Phi, S))$
```

Fig. 1: Algorithm for iterative monomorphisation

- *type_vars*$(\tau_1, \dots, \tau_n)$, where $\tau_1, \dots, \tau_n$ are types, gathers all the type variables from each type $\tau_1, \dots, \tau_n$ into a set. This function is overloaded to accept a formula φ as input, in which case it returns the set of all type variables which occur in the formula.
- *match*(υ, τ), where υ and τ are types, matches υ against τ and either fails or returns *Some*(σ), where σ results from the matching. The algorithm matches only non-monomorphic types against monomorphic types.
- *domain*(σ) returns the set of type variables α such that $\sigma(\alpha) \neq \alpha$.
- *compatible*(σ_1, σ_2) tests the compatibility between σ_1 and σ_2.
- *mangle*(Φ), where Φ is a set of monomorphic formulae, returns the same set of formulae where all types have been mangled.

The iterative monomorphisation algorithm is given in Figure 1. It has three phases. The first phase applies a *monomorphisation step* to each formula in Φ until the user-set limit, num_loops, is reached. This limit is the only bound necessary for the algorithm to terminate. We use the colour blue to identify bounds and code related to bounds. At the end of each of these iterations, the old and new type argument maps are updated with newly generated types. No new formulae are generated at this stage, only new type arguments and substitutions. Once these iterations are completed, the first phase is complete and the substitutions used to create new type argument tuples are passed to *generate_mono_formulae* for the second phase. This is when the new formulae are generated. The third phase mangles the composite types of the newly monomorphised formulae. This allows targeting a simply typed logic with no support for n-ary type constructors.

```
Function $formula\_mono\_step(\varphi, M_{old}, M_{new}, N_{old}(\varphi), N_{new}(\varphi))$
  Data: polymorphic formula $\varphi$
        old and new monomorphic type argument maps $M_{\text{old}}, M_{\text{new}}$
        old and new non-monomorphic type argument maps $N_{\text{old}}(\varphi), N_{\text{new}}(\varphi)$
  Result: monomorphic type argument map
          non-monomorphic type argument map
          set of substitutions
  $\text{max\_mono\_args} \leftarrow$
    $min(max(\text{mono\_floor}, |M_{\text{old}} \cup M_{\text{new}}| \cdot \text{mono\_mult}), \text{mono\_cap})$
  $\text{max\_nonm\_args} \leftarrow$
    $min(max(\text{nonm\_floor}, |N_{\text{old}}(\varphi) \cup N_{\text{new}}(\varphi)| \cdot \text{nonm\_mult}), \text{nonm\_cap})$
  $S \leftarrow \emptyset$
  $S' \leftarrow matches(M_{\text{new}}, N_{\text{new}}(\varphi)) \cup matches(M_{\text{new}}, N_{\text{old}}(\varphi)) \cup$
    $matches(M_{\text{old}}, N_{\text{new}}(\varphi))$
  $(M_{\text{next}}, N_{\text{next}}(\varphi)) \leftarrow (\emptyset, \emptyset)$
  foreach $\sigma \in S'$ do
    foreach $(f \mapsto (\upsilon_1, \ldots, \upsilon_n)) \in N_{\text{old}}(\varphi) \cup N_{\text{new}}(\varphi)$ do
      if $type\_vars(\upsilon_1, \ldots, \upsilon_n) \subseteq subst\_dom(\sigma)$ then
        if $|M_{\text{next}}| < \text{max\_mono\_args}$ then
          $M_{\text{next}}(f) \leftarrow M_{\text{next}}(f) \cup \{(\sigma(\upsilon_1), \ldots, \sigma(\upsilon_n))$
          $S \leftarrow S \cup \{\sigma\}$
        else if $|N_{\text{next}}(\varphi)| \geq \text{max\_nonm\_args}$ then
          $M_{\text{next}} \leftarrow M_{\text{next}} \setminus (M_{\text{old}} \cup M_{\text{new}})$
          $N_{\text{next}}(\varphi) \leftarrow N_{\text{next}}(\varphi) \setminus (N_{\text{old}}(\varphi) \cup N_{\text{new}}(\varphi))$
          return $(M_{\text{next}}, N_{\text{next}}(\varphi), S)$
      else
        if $|N_{\text{next}}(\varphi)| < \text{max\_nonm\_args}$ then
          $N_{\text{next}}(\varphi)(f) \leftarrow N_{\text{next}}(\varphi)(f) \cup \{(\sigma(\upsilon_1), \ldots, \sigma(\upsilon_n))\}$
          $S \leftarrow S \cup \{\sigma\}$
  return $(M_{\text{next}} \setminus (M_{\text{old}} \cup M_{\text{new}}), N_{\text{next}}(\varphi) \setminus (N_{\text{old}}(\varphi) \cup N_{\text{new}}(\varphi)), S)$
```

Fig. 2: Algorithm for formula monomorphisation step

The formula monomorphisation algorithm is given in Figure 2. It forms the core of the process. Essentially, it computes new type argument tuples for a single formula. Type argument tuples are matched against each other to obtain a set of substitutions which is iterated over in the outermost loop. The separation of type argument tuples into old and new maps is used to ensure that only combinations involving at least one new map are considered. This avoids re-computing some matchings processed in previous iterations. New tuples are obtained by applying each substitution to each non-monomorphic type argument tuple such that at least one tuple component is instantiated by the substitution.

The total number of type argument tuples can increase cubically in the number of type argument tuples at each iteration and can therefore grow doubly exponentially in the number of iterations. We give a sketch of how such growth can occur for a single formula. If we assume that after k iterations, there are N_k

type argument tuples in total divided evenly between monomorphic and non-monomorphic type argument tuples, then there can be up to $\left(\frac{N_k}{2}\right)^2$ successful matches, yielding as many substitutions. Each substitution is then applied to each non-monomorphic type argument for a total of $\left(\frac{N_k}{2}\right)^3$ possible new type argument tuples. If half of these new type argument tuples are monomorphic and the other half are non-monomorphic, then the next iteration will begin with $N_{k+1} = N_k^3 \cdot 2^{-3}$ evenly split type argument tuples. Therefore, the total number of type argument tuples on the kth iteration can reach $N_k = N_0^{3^k} \cdot 2^{\frac{-3^{k+2}+3}{2}}$.

Depending on the shape and size of the input problem and the number of iterations performed, the doubly exponential growth may be problematic. Introducing bounds addresses this potential issue. The limit on the number of newly generated monomorphic type argument tuples is min(max(mono_mult $\cdot$ m, mono_floor), mono_cap), where m is the total number of monomorphic type argument tuples. The components of this limit are

1. mono_cap, a limit on the total number of new type argument tuples;
2. mono_mult, which is used to allow the total number of (monomorphic) type argument tuples to grow by a certain proportion of the current number m of monomorphic type argument tuples;
3. mono_floor, which balances out mono_mult, preventing mono_mult from inhibiting new type argument tuple generation if m is too low.

Similar bounds are used for the non-monomorphic type argument tuples: The limit on the number of new non-monomorphic type argument tuples is min(max(nonm_mult $\cdot$ n, nonm_floor), nonm_cap), where n is the number of non-monomorphic type argument tuples associated with the current formula. An important difference with the monomorphic case is that n depends on the current formula being processed whilst m does not. Both in the monomorphic and in the non-monomorphic case, the maximum number of newly generated type argument tuples is fixed per formula and per iteration.

The *matches* function, which computes the substitutions used for generating new type arguments, is given in Figure 3. Each symbol instance from $N(\varphi)$ is matched against all corresponding symbol instances from M. For two such symbol instances, the types from the non-monomorphic type argument tuple are matched component-wise against the types from the monomorphic type argument tuple. The resulting substitutions are composed if they are compatible. In the algorithm, compatibility is checked by making sure the **foreach** loop has successfully iterated over all elements of the type argument tuple. If any substitutions are incompatible, the matchings are discarded. Since the composition of two compatible substitutions is commutative, the order of composition is irrelevant. The total number of substitutions generated is limited by substitution_cap.

The various bounds presented here overlap to some extent. For instance, having at most substitution_cap substitutions generated by *matches* may be sufficient to curb the number of new type argument tuples, making the mono_cap, mono_mult, mono_floor triplet superfluous. Nonetheless every bound has uses. For example, problems that lead to few successful matches but many type ar-

```
Function $matches(M, N(\varphi))$
    Data: monomorphic type argument map $M$
          non-monomorphic type argument map $N(\varphi)$
    Result: set of substitutions
    $S \leftarrow \emptyset$
    foreach $f \mapsto (\upsilon_1, \ldots, \upsilon_n) \in N(\varphi)$ do
        foreach $(\tau_1, \ldots, \tau_n) \in M(f)$ do
            if $|S| >$ substitution_cap then
                return $S$
            if for all $0 \leq i \leq n$, $match(\upsilon_i, \tau_i) = Some(\sigma_i)$ then
                if $\sigma_1, \ldots, \sigma_n$ are compatible then
                    $\sigma \leftarrow \sigma_1 \circ \cdots \circ \sigma_n$
                    $S \leftarrow S \cup \{\sigma\}$
    return $S$
```

Fig. 3: Algorithm for match generation

```
Function $generate\_mono\_formulae(\Phi, S)$
    Data: set $\Phi$ of polymorphic formulae
          substitution map $S$
    Result: set of monomorphic formulae
    $\Psi \leftarrow \emptyset$
    foreach $\varphi \in \Phi$ s.t. $\varphi$ is non-monomorphic do
        foreach $\sigma \in mono\_substs(S(\varphi), type\_vars(\varphi), \emptyset, \{\})$ do
            if $|\Psi| <$ max_new_formulae then
                $\Psi \leftarrow \Psi \cup \{\sigma(\varphi)\}$
            else
                return $\Psi$
    return $\Psi$
```

Fig. 4: Algorithm for monomorphic formula generation

gument tuples may benefit from a limit on the number of new type argument tuples whilst problems for which substitution generation is more combinatorially explosive may benefit from a limit on the number of generated substitutions.

Once all monomorphisation iterations have been completed, we are left with a set of the substitutions that have been used to generate new type arguments. The last phase uses this set to instantiate the type variables in the input problem's non-monomorphic formulae. In the presence of bounds, the order in which the elements of S are traversed affects the formulae resulting from the last phase.

The *generate_mono_formulae* function is given in Figure 4. It generates monomorphising substitutions and applies them to the polymorphic formulae of the input problem that they instantiate. A substitution σ is *monomorphising* for a formula φ if $\sigma(\varphi)$ is monomorphic. Since the substitutions are monomorphising relative to the formula they are applied to, the resulting formulae will

```
Function $mono\_substs(S, V, S_{res}, \sigma)$
    Data: set $S$ of substitutions
          set $V$ of type variables
          set $S_{\mathrm{res}}$ of substitutions
          substitution $\sigma$
    Result: set of substitutions
    if $V = \emptyset$ then
        return $S_{\mathrm{res}} \cup \{\sigma\}$
    else
        let $\alpha$ s.t. $\alpha \in V$
        foreach $\sigma_\Delta \in S$ s.t. $\alpha \in domain(\sigma_\Delta)$ and $compatible(\sigma, \sigma_\Delta)$
         do
            if $|S_{\mathrm{res}}| <$ max_substs then
                $S_{\mathrm{res}} \leftarrow$
                  $mono\_substs(S, V \setminus domain(\sigma_\Delta), S_{\mathrm{res}}, \sigma_\Delta \circ \sigma)$
            else
                return $S_{\mathrm{res}}$
        return $S_{\mathrm{res}}$
```

Fig. 5: Algorithm for monomorphising substitution generation

be monomorphic. The max_new_formulae bound is used to control the total number of new formulae. It overlaps with max_substs but can be useful to set an absolute limit on the size of the final problem.

To monomorphise a polymorphic formula, we first compute its monomorphising substitutions using the *mono_substs* function given in Figure 5. Such substitutions are computed using a recursive function. Given a set V of type variables and a set $S(\varphi)$ of substitutions, it selects a substitution σ_Δ from $S(\varphi)$ that instantiates at least one of the type variables in V. It is important that σ_Δ be compatible with σ so that they can be composed and the function recursively called to instantiate the remaining type variables.

The Zipperposition implementation of the *mono_substs* function uses a map from type variables to substitutions instead of a set to filter the relevant substitutions from S efficiently. The max_substs bound exists for two main reasons:

1. The iterative monomorphisation algorithm can generate up to max_substs new monomorphic formulae per initial polymorphic formula. Generating an excessive number of new formulae can overwhelm the prover. The final number of output formulae is limited to at most $|\Phi| \cdot$ max_substs.
2. The *mono_substs* function is the algorithm's most combinatorially explosive part. For a formula φ, if $S(\varphi)$ contains n substitutions that each instantiate exactly one of v type variables, up to n^v monomorphising substitutions may be generated. Recall that the total number of type argument tuples used to generate $S(\varphi)$ can be doubly exponential in the number of loop iterations. The starting n may therefore already be very large.

5 Detailed example

To illustrate the low level version of the iterative algorithm, we consider the following (admittedly contrived) initial problem:

$\langle 1\rangle$ $\mathsf{p}\langle \mathsf{int}, \mathsf{nat}\rangle(-1, 3) \wedge \mathsf{p}\langle \mathsf{int}, \mathsf{int}\rangle(-1, -2)$
$\langle 2\rangle$ $\forall x : \alpha,\, y : \mathsf{list}(\alpha),\, z : \beta.\ \mathsf{p}\langle \mathsf{list}(\alpha), \alpha\rangle(y, x) \wedge \mathsf{p}\langle \alpha, \alpha\rangle(x, x) \wedge \mathsf{p}\langle \alpha, \beta\rangle(x, z)$

M_{new} is initialised with $\{\mathsf{p} \mapsto \{(\mathsf{int}, \mathsf{nat}), (\mathsf{int}, \mathsf{int})\}$ and N_{new} with $\{\langle 2\rangle \mapsto \{\mathsf{p} \mapsto \{(\mathsf{list}(\alpha), \alpha), (\alpha, \alpha), (\alpha, \beta)\}\}\}$. Then we enter the main loop (assuming num_loops is at least 1). We iterate over each formula and call *formula_mono_step* for each of them. Nothing happens for formula $\langle 1\rangle$ because it contains no type variables. For formula $\langle 2\rangle$, M_{new} and $N_{\text{new}}(\langle 2\rangle)$ are passed as arguments to *matches*.

The three non-monomorphic type argument tuples are matched against their monomorphic counterparts in M_{new}:

- $(\mathsf{list}(\alpha), \alpha)$ fails in both cases because $\mathsf{list}(\alpha)$ fails to match against int.
- (α, α) fails to match against $(\mathsf{int}, \mathsf{nat})$ because the substitutions resulting from the match of the first and second element of the tuple are incompatible. The second match is successful and yields the substitution $\sigma_1 = \{\alpha \mapsto \mathsf{int}\}$
- (α, β) succeeds in both cases and generates the substitutions $\sigma_2 = \{\alpha \mapsto \mathsf{int}, \beta \mapsto \mathsf{int}\}$ and $\sigma_3 = \{\alpha \mapsto \mathsf{int}, \beta \mapsto \mathsf{nat}\}$.

Then *matches* returns, and the substitutions are used to generate new type argument tuples. Each substitution is applied to the type arguments of the function symbols of formula $\langle 2\rangle$ because they are the only function symbols with non-monomorphic type arguments. The table below summarises the situation:

	σ_1	σ_2	σ_3
$(\mathsf{list}(\alpha), \alpha)$	$(\mathsf{list}(\mathsf{int}), \mathsf{int})$	$(\mathsf{list}(\mathsf{int}), \mathsf{int})$	$(\mathsf{list}(\mathsf{int}), \mathsf{int})$
(α, α)	$(\mathsf{int}, \mathsf{int})$	$(\mathsf{int}, \mathsf{int})$	$(\mathsf{int}, \mathsf{int})$
(α, β)	(int, β)	$(\mathsf{int}, \mathsf{int})$	$(\mathsf{int}, \mathsf{nat})$

With a simple two-formula initial problem, there are already up to nine new type arguments tuples generated at this step in the first iteration alone, although only four are unique. Once the new type argument tuples are added to their respective maps, a new iteration is begun. We only consider one iteration and continue to the next phase of the algorithm.

The next function to be called is *generate_formulae*. It iterates over all non-monomorphic formulae of the initial problem; in our case, this will only be $\langle 2\rangle$. The set of type variable tuples of $\langle 2\rangle$ is passed to *mono_substs* along with the set of all previously generated substitutions.

First, we instantiate α, the first type variable of $\langle 2\rangle$. If σ_1 is selected to instantiate α, both σ_2 and σ_3 will in turn be selected to instantiate β. This will generate two monomorphising substitutions, $\sigma_1 \circ \sigma_2 = \sigma_2$ and $\sigma_1 \circ \sigma_3 = \sigma_3$, which are added to the set of monomorphising substitutions S_{res}. Now σ_2 is selected. It simultaneously instantiates α and β. Here, σ_2 is already in S_{res} and is therefore ignored, and σ_3 is treated similarly. No more than substitution_cap monomorphising substitutions can be generated in this way.

Now, the monomorphising substitutions have been generated. We only need to apply them to the formula they monomorphise. This is repeated for all non-monomorphic formulae or until max_new_formulae is reached. The order of formulae will impact the output problem in the latter case. For some larger input problems, *mono_substs* could be sufficiently explosive to only allow a handful of different formulae to be monomorphised. The ability to set substitution_cap independently can help avoid this issue.

Finally, if the target language supports only nullary type construtors, the types of all monomorphic formulae, both new and old, are mangled. Assuming that mangling is not necessary, the algorithm outputs

$\langle 1\rangle$ $\mathsf{p}\langle\mathsf{int},\mathsf{nat}\rangle(-1,3) \wedge \mathsf{p}\langle\mathsf{int},\mathsf{int}\rangle(-1,-2)$
$\langle 3\rangle$ $\forall x : \mathsf{int},\, y : \mathsf{list}(\mathsf{int}),\, z : \mathsf{int}.\ \mathsf{p}\langle\mathsf{list}(\mathsf{int}),\mathsf{int}\rangle(y,x) \wedge \mathsf{p}\langle\mathsf{int},\mathsf{int}\rangle(x,x) \wedge \mathsf{p}\langle\mathsf{int},\mathsf{int}\rangle(x,z)$
$\langle 4\rangle$ $\forall x : \mathsf{int},\, y : \mathsf{list}(\mathsf{int}),\, z : \mathsf{nat}.\ \mathsf{p}\langle\mathsf{list}(\mathsf{int}),\mathsf{int}\rangle(y,x) \wedge \mathsf{p}\langle\mathsf{int},\mathsf{int}\rangle(x,x) \wedge \mathsf{p}\langle\mathsf{int},\mathsf{nat}\rangle(x,z)$

In this example, two new monomorphic formulae are generated.

6 Evaluation

The monomorphisation algorithm is parametrised by many bounds. The first part of the evaluation process seeks appropriate values for these bounds. The second part compares the performance of Zipperposition without E and with the new monormorphising E backend on polymorphic problems. The third part compares the performance of different provers on polymorphic problems and their monomorphised counterparts.

The benchmarks are taken from version 8.2.0 of the TPTP library [17]. The library contains 1765 problems in TF1 and TH1, corresponding respectively to first and higher order logic with rank 1 polymorphism. Because Zipperposition does not support reasoning with real numbers, we removed all problems that include them. In total, our benchmark suite contains 1534 polymorphic problems. We chose as a measure of success for a given prover (or prover configuration) the number of problems that could be solved by the prover in at most 30 s per problem with a single thread. Our raw evaluation data is available online.[4]

Parameter optimisation. Each bound of the monomorphisation process represents a tradeoff: a higher bound allows for a more exhaustive instantiation of type variables but takes more time. Since we cannot test all possible combinations of values for all bounds to find the best compromise between completeness and speed, we group closely related bounds together and test combinations of values for the bounds in these groups. Once we find the best performing set of values for a group, we assign these values to the corresponding bounds as we

[4] `https://zenodo.org/records/14881532`

begin the search for the next group. If several groups result in the same number of proved problems, we select the most constraining values. Winning entries are shown in bold in Tables 1 to 6.

To guard against overfitting took place, we carried out the part of the evaluation related to parameter optimisation and all preliminary evaluations on 500 randomly chosen problems out of the 1534 selected problems. We carried out the rest of the evaluations on the remaining 1034 problems.

Before finding values to assign to the bounds of the monomorphisation algorithm, we must choose which base options to run Zipperposition with. Because the space of possible base configurations is too large to evaluate exhaustively, we evaluated, in a preliminary experiment, all pre-existing portfolio configurations that called E against our benchmark suite of 500 problems. Since E could not treat these non-monomorphic problems, it was disabled. The preliminary evaluation found that the `40_b.comb` configuration performed best by proving 131 problems. This configuration became the base configuration, which we used as a basis to evaluate the monomorphisation options.

We conducted additional informal evaluations on the 500 problems to find appropriate default values and test ranges for the monomorphisation bounds. We started the option evaluation process with the base configuration and the following default values for monomorphisation bounds and parameters for E:

- nonm_cap: ∞
- nonm_mult: 1
- nonm_floor: 50
- substitution order: separation
- substitution_cap: ∞
- max_substs: 10
- max_new_formulae: 2000
- new formulae limit multiplier: 0
- monomorphisation timeout: 20
- num_loops: 4
- E timeout: 30
- E call point: 0

The initial values for the mono_cap, mono_mult and mono_floor options are irrelevant because the options' values are set when computing Table 1.

Table 1 groups bounds that control the maximum number of newly generated monomorphic type arguments per formula and per iteration. The limit on newly generated type arguments is determined by three components that form a natural group of bounds.

The table shows that generating no new monomorphic type arguments seems to be the best approach. This result may seem counterintuitive, but it is possible to monomorphise formulae without generating monomorphic type arguments. This is because non-monomorphic type argument generation can produce substitutions that instantiate one or more type variables, and it is these substitutions that are used to monomorphise formulae.

Table 2 is similar to Table 1 except that it evaluates the bounds limiting the number of new *non*-monomorphic type arguments. The bound values are lower because we found non-monomorphic type arguments to be combinatorially explosive in preliminary evaluations. The table confirms that non-monomorphic type argument generation drives the creation of useful non-monomorphic formulae. This is indicated by the very low number of problems solved when no new non-

Table 1: Evaluation of bounds for monomorphic type argument generation

	cap											
	500				1000				∞			
	floor											
mult	0	50	100	200	0	50	100	200	0	50	100	200
0	**178**	161	161	156	178	160	160	156	178	161	160	156
1	155	155	155	158	153	154	154	156	154	154	155	155
2	154	154	153	154	153	153	154	152	154	153	154	154
∞	153	154	153	155	155	153	154	156	159	160	161	161

Table 2: Evaluation of bounds for non-monomorphic type argument generation

	cap											
	500				1000				∞			
	floor											
mult	0	10	50	100	0	10	50	100	0	10	50	100
0	125	**184**	182	177	125	184	182	177	125	184	182	177
0.5	176	184	182	177	176	184	182	177	176	184	182	177
1	182	181	178	177	182	181	178	177	182	181	178	177
∞	173	174	174	174	174	174	173	173	125	125	125	125

monomorphic type arguments are allowed. Performance of the monomorphisation algorithm seems to plateau for some ranges of values and drops off beyond.

The substitution generation phase occurs once all type arguments have been generated. The bound limiting the number of monomorphising substitutions per formula is directly related to the order which dictates how such monomorphising substitutions are generated. The heuristic greatly affects monomorphising substitution generation. The 'age' order of substitution orders substitutions generated in earlier iterations first. The 'random' order randomly shuffles substitutions. Finally, the 'separation' order separates the substitutions into groups of substitutions generated in the same iteration and generates monomorphising substitutions from each of these groups independently. Table 3 shows that the values of bounds limiting the number of monomorphising substitutions seem to affect performance only when the 'separation' heuristic is used.

Table 4 groups the bounds related to the size of the output problem to be passed to the E prover. The absolute limit is the maximum number of formulae passed to E. The multiplier limits the total number of newly generated formulae based on the problem's initial number of formulae. We find that the E prover tends to perform better when given a limited number of formulae.

For larger problems, the monomorphisation algorithm may time out despite the bounds. In these cases, neither Zipperposition nor E will have had a chance to try to solve the problem. To avoid this, a timer can interrupt the mono-

Table 3: Evaluation of bounds for substitution generation

	substitution order		
mono subst	age	random	separation
2	161	178	175
5	161	178	180
7	161	178	182
10	161	178	**184**

Table 4: Evaluation of bounds directly related to the size of the output problem

	formula multiplier			
formula cap	1	2	3	∞
500	**184**	184	184	183
2000	184	184	184	184
∞	168	178	183	125

Table 5: Evaluation of parameters related to the depth of monomorphisation

	mono time			
num. loops	5	10	20	30
1	183	184	183	183
2	**186**	186	186	185
3	186	186	186	185
4	186	186	186	185
5	185	185	185	184

morphisation algorithm, after which Zipperposition resumes normal operation. Table 5 tests the amount of time that is allocated to monomorphisation against the number of iterations of the monomorphisation algorithm. Neither parameter seems to substantially affect the algorithm's performance.

Table 6 shows the impact of the options with which we call the E prover. The point at which Zipperposition interrupts its normal operation and begins the monomorphisation process is determined by the 'E call point' parameter. It is expressed as a fraction of the total time allotted to Zipperposition. The E timeout (in seconds) limits how long E is run before being interrupted, at which point Zipperposition resumes normal operation. The longer Zipperposition runs before E is invoked, the more formulae are generated, and the more combinatorially explosive iterative monomorphisation is. This likely explains the poor performance for greater values of the E call point.

E as a Zipperposition backend. We compare the performance of two instances of Zipperposition. By default, Zipperposition may call E as a backend

Table 6: Evaluation of parameters related to the E prover

	E call point			
E timeout	0	0.1	0.2	0.3
2	180	143	132	124
5	**185**	142	134	125
10	184	143	132	125
20	184	137	133	125
30	182	133	134	125

Table 7: Evaluation of Zipperposition without E vs. with E

	without E	with E	union
500 problem suite	160	198	207
1034 problem suite	337	410	434

Table 8: Evaluation of native polymorphism vs. monomorphisation

	Native	Mono	Union
E	–	340	340
Leo-III with E	157	231	274
Zipperposition	339	351	404

when given a monomorphic problem [18]. The first instance is run in the portfolio mode `portfolio.sequential.py`, which attempts to prove the problem with various configurations tried in succession. Since all given problems are non-monomorphic, these configurations can never invoke E as a backend, this instance is therefore labeled 'without E'. The second instance is a modification of the `portfolio.sequential.py` file where each configuration is modified analogously to `40_b.comb` with the options obtained in the previous evaluation phase. This instance can successfully call E as a backend because it is able to provide E with a monomorphised problem. Each of the two instances is evaluated on the set of the 500 previously used problems, and the set of the remaining 1034 problems is evaluated separately. The proportion of problems successfully solved on the 500 and 1034 problem suites are similar, suggesting that no overfitting took place during the option optimisation phase.

Table 7 shows that the use of E as a backend markedly improves the performance of Zipperposition. It is not a strict improvement, since some problems are solved without E and not with E.

Monomorphisation as a preprocessor. To evaluate the usefulness of iterative monomorphisation as an alternative to native polymorphism, two competitive higher order polymorphic provers were run on the 1034 problem suite. We chose Leo-III and Zipperposition. Unfortunately we needed to exclude Vampire because of parsing issues. The fix for these issues was unavailable for Vampire's higher order branch at the time of the evaluation.

Table 8 shows the results. The monomorphisation approach is evaluated in two steps. First, each problem is monomorphised using the options obtained from the first evaluation phase except for the monomorphisation timeout option, which is increased to 30 s. For 149 problems, monomorphisation times out. Second, each prover is run on the remaining monomorphised problems, and the results are tallied in the 'Mono' column. In addition to the polymorphic provers used in the 'Native' tests, the monomorphic prover E is run on the monomorphised problems to provide an additional point of comparison. Instead of running each prover for 30 s on the monomorphised problems, the monomorphisation time (rounded up to the nearest second) is subtracted to compare fairly against the 'Native' column, which does not have a similar preprocessing phase.

Despite the monomorphisation timeouts, monomorphisation is more effective than Leo-III's and Zipperposition's native polymorphism on the benchmarks.

7 Conclusion

We described a translation algorithm that iteratively instantiates polymorphic types to produce monomorphic problems. Our primary motivation was to improve the success rate of Zipperposition and its monomorphic E backend, and indeed our evaluation shows a clear improvement. We also saw that even with automatic provers that support polymorphism, iterative monomorphisation is a better alternative in practice.

We see the following avenues for future work. First, iterative monomorphisation blindly enumerates candidate instantiations, without exploiting any knowledge about the logical structure of the formulae in which symbols occur. For example, a lemma $\mathsf{p}\langle\alpha\rangle$ cannot be used to prove the conjecture $\neg\mathsf{p}\langle\mathsf{nat}\rangle$ because of the incompatible polarities, but our algorithm instantiates α with nat regardless. Second, some automatic provers as well as tools such as Sledgehammer include relevance filters that heuristically select a subset of the available axioms; filters such as MePo [12] and SInE [10] are iterative and could be interleaved with monomorphisation. Third, although one would expect native implementations of polymorphism to outperform any preprocessor, currently this is not the case, suggesting that there is considerable room for improvement on the native front.

Acknowledgements. We thank Sascha Böhme for fruitful discussions. We thank Jannis Limperg, Mark Summerfield, and the peer reviewers for suggesting textual improvements. This research is co-funded by the European Union (ERC, Nekoka, 101083038). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

1. Bhayat, A., Reger, G.: A polymorphic Vampire (short paper). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) IJCAR 2020, Part II. LNCS, vol. 12167, pp. 361–368. Springer (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 (2013)
3. Blanchette, J.C., Paskevich, A.: TFF1: The TPTP typed first-order form with rank-1 polymorphism. In: Bonacina, M.P. (ed.) CADE-24. LNCS, vol. 7898, pp. 414–420. Springer (2013)
4. Bobot, F., Paskevich, A.: Expressing polymorphic types in a many-sorted language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNCS, vol. 6989, pp. 87–102. Springer (2011)
5. Böhme, S.: Proving Theorems of Higher-Order Logic with SMT Solvers. PhD thesis, Technische Universität München (2012)
6. Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS, vol. 6173, pp. 107–121. Springer (2010)
7. Bonichon, R., Déharbe, D., Tavares, C.: Extending SMT-LIB v2 with λ-terms and polymorphism. In: Rümmer, P., Wintersteiger, C.M. (eds.) SMT 2014. CEUR Workshop Proceedings, vol. 1163, pp. 53–62. CEUR-WS.org (2014)
8. Harrison, J.: Optimizing proof search in model elimination. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE-13. LNCS, vol. 1104, pp. 313–327. Springer (1996)
9. Harrison, J.: HOL Light: An overview. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 60–66. Springer (2009)
10. Hoder, K., Voronkov, A.: Sine qua non for large theory reasoning. In: Bjørner, N.S., Sofronie-Stokkermans, V. (eds.) CADE-23. LNCS, vol. 6803, pp. 299–314. Springer (2011)
11. Kaliszyk, C., Sutcliffe, G., Rabe, F.: TH1: The TPTP typed higher-order form with rank-1 polymorphism. In: Fontaine, P., Schulz, S., Urban, J. (eds.) PAAR 2016. CEUR Workshop Proceedings, vol. 1635, pp. 41–55. CEUR-WS.org (2016)
12. Meng, J., Paulson, L.C.: Lightweight relevance filtering for machine-generated resolution problems. J. Appl. Log. **7**(1), 41–57 (2009)
13. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
14. Paulson, L.C., Blanchette, J.C.: Three years of experience with Sledgehammer, a practical link between automatic and interactive theorem provers. In: Sutcliffe, G., Schulz, S., Ternovska, E. (eds.) IWIL-2010. EPiC, vol. 2, pp. 1–11. EasyChair (2012)
15. Schulz, S.: E – a brainiac theorem prover. AI Commun. **15**(2–3), 111–126 (2002)
16. Slind, K., Norrish, M.: A brief overview of HOL4. In: Mohamed, O.A., Muñoz, C.A., Tahar, S. (eds.) TPHOLs 2008. LNCS, vol. 5170, pp. 28–32. Springer (2008)
17. Sutcliffe, G.: The TPTP problem library and associated infrastructure – from CNF to TH0, TPTP v8.2.0. J. Autom. Reason. **59**(4), 483–502 (2017)
18. Vukmirović, P., Bentkamp, A., Blanchette, J., Cruanes, S., Nummelin, V., Tourret, S.: Making higher-order superposition work. In: Platzer, A., Sutcliffe, G. (eds.) CADE-28. LNCS, vol. 12699, pp. 415–432. Springer (2021)