Monotonicity or How to Encode Polymorphic Types Safely and Efficiently

Jasmin Christian Blanchette[1,⋆], Sascha Böhme[1], and Nicholas Smallbone[2]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Dept. of CSE, Chalmers University of Technology, Gothenburg, Sweden

Abstract. Most automatic theorem provers are restricted to untyped or monomorphic logics, and existing translations from polymorphic logics are either bulky or unsound. Recent research shows how to exploit monotonicity to encode ground types efficiently: monotonic types can be safely erased, while nonmonotonic types must generally be encoded. We extend this work to rank-1 polymorphism and show how to eliminate even more clutter by also erasing most occurrences of nonmonotonic types, without sacrificing soundness or completeness. The new encodings are implemented in the Sledgehammer tool for Isabelle/HOL. Our evaluation finds them considerably superior to previous schemes.

1 Introduction

Specification languages and other theorem proving applications often rely on powerful type systems, with polymorphism and overloading, but most state-of-the-art automatic provers support only untyped or monomorphic formalisms. The various sound and complete translation schemes for polymorphic types proposed in the literature, whether they revolve around type guards (predicates) or tags (functions), produce their share of clutter, and lighter approaches are usually unsound (i.e. they do not preserve satisfiability). As a result, application authors face a painful choice between soundness and efficiency.

The third author, together with Claessen and Lilliestr̈om [10], designed sound, complete, and efficient translations from monomorphic to untyped first-order logic with equality. The key insight is that monotonic types (types whose domain can be extended with new elements while preserving satisfiability) can be simply erased, while the remaining types can be made monotonic by introducing guards or tags. Monotonicity is undecidable, but it can often be inferred in practice using suitable calculi [3, 10].

In this paper, we first generalise this approach to a rank-1 (ML-style) polymorphic logic, as embodied by the polymorphic TPTP typed first-order form (TFF1) [4]. Unfortunately, the presence of a single polymorphic literal[1] $X^{\alpha} = t$ will lead us to classify every type as potentially nonmonotonic and force the use of guards or tags everywhere, as in the traditional encodings. We solve this issue by our second main contribution,

⋆ Research supported by the Deutsche Forschungsgemeinschaft grant Ni 491/11-2.

[1] The notation t^{τ} stands for a term t with type τ, whereas α is a type variable.

a novel scheme that considerably reduces the clutter associated with nonmonotonic types, based on the observation that guards or tags are only required when translating the particular axioms that make a type nonmonotonic. Consider a two-valued *state* type axiomatised by $\forall S : \mathit{state}.\ S = \mathsf{on} \vee S = \mathsf{off}$ and $\forall S : \mathit{state}.\ \mathsf{toggle}(S) \neq S$. Claessen et al. would classify *state* as nonmonotonic and require systematic annotations with guards or tags, whereas our refined scheme detects that the second axiom is harmless and translates it to the untyped formula $\forall S.\ \mathsf{toggle}(S) \neq S$, simply erasing the types.

After a brief review of the traditional polymorphic type encodings (Section 2), we present the polymorphic monotonicity inference calculus and the related type encodings (Section 3). Although the focus is on sound encodings, we also consider unsound ones, both as evaluation yardsticks and because applications that certify external proofs can safely employ them for proof search. Furthermore, we explore incomplete versions of the type encodings based on monomorphisation (Section 4). The polymorphic encodings are proved sound and complete (Section 5).

The type encodings described here have been implemented in the popular Sledgehammer tool [2, 14], which provides a bridge between the interactive theorem prover Isabelle/HOL and a wide range of automatic provers. We evaluate the encodings' suitability for the resolution provers E [17], SPASS [21], and Vampire [16] and the SMT solver Z3 [15] (Section 6). Our comparison includes the traditional type encodings as well as the provers' native support for simple types (sorts) where available.

The exposition builds on the following running examples.

Example 1.1 (Monkey Village). Imagine a village of monkeys [10] where each monkey owns at least two bananas:

$$\forall M : \mathit{monkey}.\ \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))$$
$$\forall M : \mathit{monkey}.\ \mathsf{b}_1(M) \neq \mathsf{b}_2(M)$$
$$\forall M_1, M_2 : \mathit{monkey},\ B : \mathit{banana}.\ \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \longrightarrow M_1 = M_2$$

The predicate $\mathsf{owns} : \mathit{monkey} \times \mathit{banana} \to \mathrm{o}$ associates monkeys with bananas, and the functions $\mathsf{b}_1, \mathsf{b}_2 : \mathit{monkey} \to \mathit{banana}$ witness the existence of each monkey's minimum supply of bananas. The type *banana* is monotonic, because any model with κ bananas can be extended to a model with $\kappa' > \kappa$ bananas, where κ and κ' can be infinite cardinals. In contrast, *monkey* is nonmonotonic, because there can live at most n monkeys in a village with a finite supply of $2n$ bananas.

Example 1.2 (Algebraic Lists). The following axioms induce a minimalistic first-order theory of algebraic lists:

$$\forall X : \alpha,\ \mathit{Xs} : \mathit{list}(\alpha).\ \mathsf{nil} \neq \mathsf{cons}(X, \mathit{Xs})$$
$$\forall \mathit{Xs} : \mathit{list}(\alpha).\ \mathit{Xs} = \mathsf{nil} \vee (\exists Y : \alpha,\ \mathit{Ys} : \mathit{list}(\alpha).\ \mathit{Xs} = \mathsf{cons}(Y, \mathit{Ys}))$$
$$\forall X : \alpha,\ \mathit{Xs} : \mathit{list}(\alpha).\ \mathsf{head}(\mathsf{cons}(X, \mathit{Xs})) = X$$
$$\forall X : \alpha,\ \mathit{Xs} : \mathit{list}(\alpha).\ \mathsf{tail}(\mathsf{cons}(X, \mathit{Xs})) = \mathit{Xs}$$

We conjecture that cons is injective. Expressed negatively for an unknown but fixed type b, the conjecture becomes

$$\exists X, Y : b,\ \mathit{Xs}, \mathit{Ys} : \mathit{list}(b).\ \mathsf{cons}(X, \mathit{Xs}) = \mathsf{cons}(Y, \mathit{Ys}) \wedge (X \neq Y \vee \mathit{Xs} \neq \mathit{Ys})$$

Since the problem is unsatisfiable, all types are trivially monotonic.

2 Traditional Type Encodings

Encoding types in an untyped logic is an old problem, and many solutions have nearly folkloric status. They lay the foundation for our more efficient encodings.

Full Type Erasure (e). The easiest way to translate a typed formula into an untyped logic is to omit, or erase, all type information. We call this encoding e. Type erasure is conspicuously unsound in a logic that interprets equality, because different cardinality constraints can be attached to different types; for example, the e encoding translates the exhaustion rule $\forall U : \mathit{unit}.\ U = \mathsf{unity}$ to $\forall U.\ U = \mathsf{unity}$, which forces a singleton universe. An expedient is to filter out all axioms of the form $\forall X : \tau.\ X = \mathsf{c}_1 \vee \cdots \vee X = \mathsf{c}_n$ [14, §2.8], but this generally does not suffice to prevent unsound cardinality reasoning.

Full type erasure is also unsound because it confuses distinct monomorphic instances of polymorphic symbols. Clearly, the axioms $\mathsf{p}(\mathsf{c}^a)$ and $\neg\, \mathsf{p}(\mathsf{c}^b)$ are satisfiable in a typed logic but become unsatisfiable if the types a and b are erased.

Type Arguments (a). To distinguish instances of polymorphic symbols, we can supply explicit type arguments, encoded as terms: type variables α become term variables A, and n-ary type constructors k become n-ary function symbols k. For the example above, we obtain $\mathsf{p}(\mathsf{a}, \mathsf{c}(\mathsf{a}))$ and $\neg\, \mathsf{p}(\mathsf{b}, \mathsf{c}(\mathsf{b}))$, and a fully polymorphic instance c^α would be mapped to $\mathsf{c}(A)$. More generally, we pass one type argument for each type variable occurring in the most general type for a symbol, e.g. $\mathsf{nil}(A)$; this suffices to reconstruct its type. We call this encoding a; it is unsound, although much less so than e.

Type Guards (g). Arguably the most intuitive approach to encode type information soundly and completely is to employ *type guards*—predicates that restrict the range of variables. For polymorphic type systems, they take the form of a binary predicate $\mathsf{g}(\tau, t)$ that checks whether t has type τ, where τ is encoded as a term. We call this encoding g.

Each variable in the generated clause normal form (CNF) problem is guarded by g, and type information is provided for the function symbols occurring in the problem. For full first-order form (FOF), we must guard bound variables as well, with $\longrightarrow$ as the connective for $\forall$ and $\wedge$ for $\exists$. The first axiom of Example 1.1 becomes

$$\forall M.\ \mathsf{g}(\mathsf{monkey}, M) \longrightarrow \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))$$

To witness the inhabitation of all types, we include $\forall A.\ \exists X.\ \mathsf{g}(A, X)$ as an axiom. We must also include type information for the function symbols occurring in the problem, as auxiliary typing axioms $\forall M.\ \mathsf{g}(\mathsf{banana}, \mathsf{b}_i(M))$. There is no need to guard M because the b_i's are always bananas irrespective of their argument.

This encoding gives a type to some ill-typed terms, such as $\mathsf{b}_1(\mathsf{b}_2(\mathsf{monkey}))$. Intuitively, this is safe because such terms cannot bring the proof forward (except to witness inhabitation, but even in that role they are redundant). On the other hand, well-typed terms must always be associated with their correct type, as they are in this encoding.

We must include type arguments that occur only in the result type of a function symbol, in order to distinguish instances of polymorphic function symbols, but all other type arguments can be omitted, since they can be deduced from the function's arguments. Thus, the type argument would be kept for nil but omitted for cons.

Type Tags (`t`). Type guards complicate the logical structure of formulas. An alternative is to encode all type information in the terms, by wrapping each term and subterm in suitable functions, which we call *type tags*. To support polymorphism and n-ary type constructors, the encoding relies on a single binary function $\mathsf{t}(\tau, t)$ that tags the term t with its type τ, where τ is encoded as a term. The tags make the type arguments entirely superfluous. We call this encoding `t`. The first axiom of Example 1.1 becomes

$$\forall M.\ \mathsf{owns}(\mathsf{t}(\mathsf{monkey}, M), \mathsf{t}(\mathsf{banana}, \mathsf{b}_1(\mathsf{t}(\mathsf{monkey}, M)))) \wedge \\ \mathsf{owns}(\mathsf{t}(\mathsf{monkey}, M), \mathsf{t}(\mathsf{banana}, \mathsf{b}_2(\mathsf{t}(\mathsf{monkey}, M))))$$

3 Sound Type Erasure via Monotonicity Inference

Type guards and tags significantly increase the size of the problems passed to the automatic provers, with a dramatic impact on their performance. Fortunately, most of the clutter can be removed by inferring monotonicity and (soundly) erasing type information based on the monotonicity analysis.

Polymorphic Monotonicity Inference. A type τ is *monotonic* in a formula φ if any model of φ where τ has cardinality κ can be extended into a model where it has cardinality κ', for any $\kappa' > \kappa$. Claessen et al. devised a simple calculus to infer monotonicity for monomorphic first-order logic [10, §2.3], based on the observation that a type τ must be monotonic if the problem contains no positive literal $X^\tau = t$ or $t = X^\tau$ (i.e. X does not occur *positively naked*).[2]

The calculus is easy to extend to polymorphism. Semantically, a polymorphic type is monotonic iff all of its ground instances are monotonic. The extended calculus computes the set of *possibly nonmonotonic* polymorphic types, which consists of all types τ such that there is a positively naked variable of type τ. Each nonmonotonic ground type is an instance of a type in this set. To infer that a polymorphic type τ is monotonic, we check that there is no possibly nonmonotonic type unifiable with τ. Annoyingly, a single occurrence of a positively naked variable of type α, such as X in the equation $\mathsf{head}(\mathsf{cons}(X, Xs)) = X$ from Example 1.2, is enough to completely flummox the analysis: since all types are instances of α, they are all possibly nonmonotonic.

Infinity Inference. We regain some precision by complementing the calculus with an infinity analysis: by the Löwenheim–Skolem theorem, all types with no finite models are monotonic. We call such types *infinite*. We could employ an approach similar to that implemented in Infinox [9] to automatically infer finite unsatisfiability of types; for Example 1.2, we would infer that $\mathit{list}(\alpha)$ is infinite because cons is injective but not surjective. However, in an interactive theorem prover, it is simpler to exploit metainformation available through introspection. Isabelle datatypes are registered with their constructors; if some of them are recursive, or take an argument of an infinite type, the datatype must be infinite. Combining infinity inference with the monotonicity inference calculus described above, we get the following rule for inferring monotonicity:

[2] Claessen et al. designed a second, more powerful calculus to detect predicates that act as fig leaves for naked variables. Whilst the calculus proved fairly successful on a subset of the TPTP benchmark suite [19], we assessed its suitability on about 1000 fairly large problems generated by Sledgehammer and found no improvement on the first calculus.

> A polymorphic type is monotonic if, whenever it is unifiable with a possibly nonmonotonic type, the most general unifier is an instance of an infinite type.

Our rule is correct because if we infer a type to be monotonic, then all its ground instances either are infinite or can be inferred monotonic by the monotonicity calculus.

Type Erasure with Guards (g?, g??). Claessen et al. observed that monotonic types can be soundly erased when translating from a monomorphic logic to an untyped logic, whereas nonmonotonic types must be encoded, typically using guards or tags [10, §3.2]. In particular, type erasure as performed by the type argument encoding a is sound if all types are monotonic. We extend the approach to polymorphism and show how to eliminate even more type information than Claessen et al. in the monomorphic case.

We first focus on type guards. The following procedure soundly translates problems from polymorphic to untyped first-order logic:

1. Introduce type arguments to all polymorphic function and predicate symbols.
2. Add guards for the types that cannot be inferred monotonic, and add typing axioms.
3. Erase all the types.

We call the resulting encoding g?. In contrast to the traditional g encoding, g? generally requires type arguments to compensate for the incomplete type information.

Typing axioms are needed to discharge the guards. We could in principle simply generate typing axioms $\mathsf{g}(\tau, \mathsf{f}(\bar{X}))$ for every function symbol f and similarly for bound variables, as in the g encoding, but some of these axioms are needless. We reduce clutter in two ways. First, if τ is not unifiable with any of the possibly nonmonotonic types, the typing axiom will never be resolvable against a guard and can be omitted. Second, for infinite types τ, it suffices to generate an axiom $\mathsf{g}(\tau, X)$ that allows any term to be typed as τ; such an axiom is sound for any monotonic type τ, as we will prove in Section 5.

Example 3.1. For the algebraic list problem of Example 1.2, our monotonicity inference reports that α is possibly nonmonotonic, but $\mathit{list}(\alpha)$ is infinite. The g? encoding of the problem follows, including the negated conjecture and the typing axioms:

$$\forall A.\ \exists X.\ \mathsf{g}(A, X)$$
$$\forall A, Xs.\ \mathsf{g}(\mathsf{list}(A), Xs)$$
$$\forall A, Xs.\ \mathsf{g}(A, \mathsf{head}(A, Xs))$$

$$\forall A, X, Xs.\ \mathsf{g}(A, X) \longrightarrow \mathsf{nil}(A) \neq \mathsf{cons}(A, X, Xs)$$
$$\forall A, Xs.\ Xs = \mathsf{nil}(A) \vee (\exists Y, Ys.\ \mathsf{g}(A, Y) \wedge Xs = \mathsf{cons}(A, Y, Ys))$$
$$\forall A, X, Xs.\ \mathsf{g}(A, X) \longrightarrow \mathsf{head}(A, \mathsf{cons}(A, X, Xs)) = X$$
$$\forall A, X, Xs.\ \mathsf{g}(A, X) \longrightarrow \mathsf{tail}(A, \mathsf{cons}(A, X, Xs)) = Xs$$

$$\exists X, Y, Xs, Ys.\ \mathsf{g}(\mathsf{b}, X) \wedge \mathsf{g}(\mathsf{b}, Y) \wedge \mathsf{cons}(\mathsf{b}, X, Xs) = \mathsf{cons}(\mathsf{b}, Y, Ys) \wedge (X \neq Y \vee Xs \neq Ys)$$

The second typing axiom allows any term to be typed as $\mathit{list}(\alpha)$, which is sound because $\mathit{list}(\alpha)$ is infinite. We could also have provided separate axioms for nil, cons, and tail. Either way, the axioms are needed to discharge the $\mathsf{g}(A, X)$ guards in case the proof requires reasoning about $\mathit{list}(\mathit{list}(\alpha))$.

The g? encoding treats all variables of the same type uniformly. Hundreds of axioms can suffer because of one unhappy formula that uses a type nonmonotonically. A lighter encoding, called g??, is possible: if an essentially universal (i.e. nonskolemisable) var-

iable does not occur positively naked in a formula, we safely omit its guard. This is related to the observation that only paramodulation from or into a (positively naked) variable can cause ill-typed instantiations in a resolution prover [22, §4].

Example 3.2. The g?? encoding of Example 1.2 is identical to g? except that the $\mathsf{nil} \neq \mathsf{cons}$ and tail axioms do not need any guard.

Example 3.3. Let us return to the monkey village of Example 1.1. Encoded with g??, it requires only two guards, a clear improvement over g? and Claessen et al. [10, §2.3]:

$$\forall A.\ \exists X.\ \mathsf{g}(A, X)$$

$$\forall M.\ \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))$$
$$\forall M.\ \mathsf{b}_1(M) \neq \mathsf{b}_2(M)$$
$$\forall M_1, M_2, B.\ \mathsf{g}(\mathsf{monkey}, M_1) \longrightarrow \mathsf{g}(\mathsf{monkey}, M_2) \longrightarrow \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \longrightarrow M_1 = M_2$$

Type Erasure with Tags (t?, t??). The t? encoding, analogous to g?, tags all terms of a possibly nonmonotonic type that is not infinite. This can result in mismatches, e.g. if α is tagged but its instance $\mathit{list}(\alpha)$ is not. The solution is to generate an equation $\mathsf{t}(\tau, X) = X$ for each infinite type τ, which allows the prover to add or remove a tag. The lighter encoding t?? only annotates naked variables, whether positive or negative, and introduces equations $\mathsf{t}(\tau, \mathsf{f}(\overline{X})) = \mathsf{f}(\overline{X})$ to add or remove tags around each function symbol (or skolemisable variable) f of a possibly nonmonotonic type τ. It is not strictly necessary to tag negatively naked variables, but a uniform treatment of naked variables ensures that resolution can be directly applied on equality atoms.

Example 3.4. The t? encoding of Example 1.2 is as follows:

$$\forall A.\ \exists X.\ \mathsf{t}(A, X) = X$$
$$\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), Xs) = Xs$$

$$\forall A, X, Xs.\ \mathsf{nil}(A) \neq \mathsf{cons}(A, \mathsf{t}(A, X), Xs)$$
$$\forall A, Xs.\ Xs = \mathsf{nil}(A) \vee (\exists Y, Ys.\ Xs = \mathsf{cons}(A, \mathsf{t}(A, Y), Ys))$$
$$\forall A, X, Xs.\ \mathsf{t}(A, \mathsf{head}(A, \mathsf{cons}(A, \mathsf{t}(A, X), Xs))) = \mathsf{t}(A, X)$$
$$\forall A, X, Xs.\ \mathsf{tail}(A, \mathsf{cons}(A, \mathsf{t}(A, X), Xs)) = Xs$$

$$\exists X, Y, Xs, Ys.\ \mathsf{cons}(\mathsf{b}, \mathsf{t}(\mathsf{b}, X), Xs) = \mathsf{cons}(\mathsf{b}, \mathsf{t}(\mathsf{b}, Y), Ys) \wedge (\mathsf{t}(\mathsf{b}, X) \neq \mathsf{t}(\mathsf{b}, Y) \vee Xs \neq Ys)$$

Example 3.5. The t?? encoding of Example 1.2 requires fewer tags, at the cost of more type information (for head and some of the existential variables):

$$\forall A.\ \exists X.\ \mathsf{t}(A, X) = X$$
$$\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), Xs) = Xs$$
$$\forall A, Xs.\ \mathsf{t}(A, \mathsf{head}(A, Xs)) = \mathsf{head}(A, Xs)$$

$$\forall A, X, Xs.\ \mathsf{nil}(A) \neq \mathsf{cons}(A, X, Xs)$$
$$\forall A, Xs.\ Xs = \mathsf{nil}(A) \vee (\exists Y, Ys.\ \mathsf{t}(A, Y) = Y \wedge Xs = \mathsf{cons}(A, Y, Ys))$$
$$\forall A, X, Xs.\ \mathsf{head}(A, \mathsf{cons}(A, X, Xs)) = \mathsf{t}(A, X)$$
$$\forall A, X, Xs.\ \mathsf{tail}(A, \mathsf{cons}(A, X, Xs)) = Xs$$

$$\exists X, Y, Xs, Ys.\ \mathsf{t}(\mathsf{b}, X) = X \wedge \mathsf{t}(\mathsf{b}, Y) = Y \wedge \mathsf{cons}(\mathsf{b}, X, Xs) = \mathsf{cons}(\mathsf{b}, Y, Ys) \wedge (X \neq Y \vee Xs \neq Ys)$$

Finiteness Inference (g!, g!!, t!, t!!). A radical approach is to assume every type is infinite unless we have metainformation to the contrary. Only types that are clearly finite, such as *unit* and *pair*(*bool*, *bool*), are considered by the monotonicity inference calculus. This is of course unsound, but it eliminates even more clutter and can make sense if proof search is followed by certification. The encodings g!, g!!, t!, and t!! are defined analogously to g?, g??, t?, and t??.

4 Monomorphisation

Type variables give rise to term variables in encoded formulas. These variables dramatically increase the search space. In the context of Sledgehammer, a further complication is that axiomatic type class predicates must be included in the problem to restrict the variables' ranges [14, §2.1]. An alternative is to monomorphise the problem, i.e. to heuristically instantiate all type variables with ground types. Monomorphisation is necessarily incomplete [7, §2] and often overlooked or derided in the literature, but it was applied with much success in the Isabelle–SMT integration [2].

The Algorithm. Our monomorphisation algorithm involves three stages:

1. Separate the monomorphic and the polymorphic formulas, and collect all symbols occurring in the monomorphic formulas (the "mono-symbols").
2. For each polymorphic axiom, stepwise refine a set of substitutions, starting from the singleton set containing only the empty substitution, by matching known mono-symbols against their polymorphic counterparts. As long as new mono-symbols emerge, collect them and repeat this stage.
3. Apply the computed substitutions to the corresponding polymorphic formulas. Only keep fully monomorphic formulas.

To ensure termination, we limit the iterations performed in stage 2 to a configurable number K. To curb the exponential growth, we also enforce an upper bound Δ on the number of new formulas. Sledgehammer operates with $K = 3$ and $\Delta = 200$ by default, so that a problem with 500 axioms comprises at most 700 axioms after monomorphisation. Experiments found these values suitable. Increasing Δ sometimes helps solve more problems, but its potential for clutter is real.

Type Mangling and Native Types (∼, $\widetilde{\mathsf{n}}$). Monomorphisation is applicable in combination with all the encodings presented so far except e (which erases all types). Since all types are ground, we mangle them in the enclosing symbol names to lighten the translation; for example, $\mathsf{g}(\tau, t)$ becomes $\mathsf{g}_\tau(t)$. We decorate the letter denoting an encoding with ∼ to indicate monomorphisation.

The mangled type guard encoding $\widetilde{\mathsf{g}}$ also constitutes a suitable basis for generating typed problems in the monomorphic TPTP typed first-order form (TFF0) [20], a format supported natively by a growing number of provers, including Vampire and Z3. In $\widetilde{\mathsf{g}}$, each bound variable is guarded by a g_τ predicate; in the corresponding TFF0-based typed translation, which we call $\widetilde{\mathsf{n}}$ ("native"), the variable is declared with the type τ instead. Type declarations replace typing axioms. Since TFF0 forbids overloading, all type arguments must be kept and mangled in the symbols.

5 Soundness and Completeness

To cope with the variety of type encodings, we need a modular proof of correctness that isolates their various features. Obviously, we cannot prove the unsound encodings correct, but this still leaves `g?`, `g??`, `t?`, and `t??`, with and without monomorphisation (~).

We start by proving the monomorphic encodings sound and complete when applied to already monomorphic problems—i.e. they preserve satisfiability and unsatisfiability. (Monomorphisation in itself is obviously sound, although incomplete.) Then we proceed to lift the proof to polymorphic encodings.

5.1 The Monomorphic Case

To prove $\widetilde{\texttt{g}}\texttt{?}$, $\widetilde{\texttt{g}}\texttt{??}$, $\widetilde{\texttt{t}}\texttt{?}$, and $\widetilde{\texttt{t}}\texttt{??}$ correct, we follow a two-stage proof strategy: the first stage adds guards or tags without erasing any types, so that the formulas remain typed, and the second stage erases the types. We call the original problem A^τ, the intermediate problem Z^τ, and the final problem Z. (The τ superscripts stand for "typed".)

The following result, due to Claessen et al. [10, §2.2], plays a key role in the proof:

Lemma 5.1 (Monotonic Type Erasure). *Let Φ^τ be a monomorphic problem. If Φ^τ is monotonic (i.e. all of its types are monotonic), then Φ^τ is equisatisfiable to its type-erased variant Φ.*

Proof. Let $\mathcal{M}$ be a model of Φ^τ. By monotonicity, there exists a model $\mathcal{N}$ where all the domains have the cardinality of the largest domain in $\mathcal{M}$. From $\mathcal{N}$, we construct a model of Φ by identifying all the domains. Conversely, from a model $\mathcal{N}$ of Φ we construct a model of Φ^τ with the same interpretations of functions and predicates as in $\mathcal{N}$ and with $\mathcal{N}$'s unique domain as the domain for every type. □

Corollary 5.2 (Equisatisfiability Conditions). *The problems A^τ and Z are equisatisfiable if the following conditions hold:*

MONO: *Z^τ is monotonic.*
SOUND: *If A^τ is satisfiable, then so is Z^τ.*
COMPLETE: *If Z^τ is satisfiable, then so is A^τ.*

We show the conditions of Corollary 5.2 separately for guards and tags. The proofs rely on the following lemma:

Lemma 5.3 (Domain Restriction). *Let $\mathcal{M}$ be a model of Φ, and let $\mathcal{M}'$ be an interpretation constructed from $\mathcal{M}$ by deleting some domain elements while leaving the interpretations of functions and predicates intact. This $\mathcal{M}'$ is a model of Φ provided that*

(a) *we do not make any domain empty;*
(b) *we do not delete any domain element that is in the image of a function;*
(c) *we do not delete any witness for an existential variable.*

Proof. For simplicity, suppose the problem is expressed in CNF, in which case (b) subsumes (c). Conditions (a) and (b) ensure that $\mathcal{M}'$ is well-defined and that ground clauses are interpreted as in $\mathcal{M}$. Since every domain element of $\mathcal{M}'$ is also in $\mathcal{M}$, all clauses that are satisfied in $\mathcal{M}$ are also satisfied in $\mathcal{M}'$. □

Theorem 5.4 (Correctness of Monomorphic Guards). *The encodings* $\widetilde{\mathtt{g}}?$ *and* $\widetilde{\mathtt{g}}??$ *are sound and complete for monomorphic problems.*

Proof. It suffices to show that the three conditions of Corollary 5.2 are fulfilled.

MONO: Infinite types are necessarily monotonic. The other types are monotonic if all positively naked variables of their types are guarded [10, §2.4]. Both $\widetilde{\mathtt{g}}?$ and $\widetilde{\mathtt{g}}??$ guard all such variables—$\widetilde{\mathtt{g}}??$ guards exactly those variables, while $\widetilde{\mathtt{g}}?$ guards more.

SOUND: Given a model of A^τ, we extend it to a model of Z^τ by giving an interpretation to the type guards. To do this, we simply interpret all type guards by the true predicate (the predicate that is true everywhere).

COMPLETE: A model of Z^τ is *canonical* if all guards are interpreted by the true predicate. From a canonical model, we obtain a model of A^τ by the converse construction to SOUND. It then suffices to prove that whenever there exists a model of Z^τ, there exists a canonical model. We appeal to Lemma 5.3 to remove the domain elements that do not satisfy their guard predicate. For this to work, (a) each predicate must be satisfied by at least one element, (b) each function must satisfy its predicate, and (c) each existential variable must satisfy its predicate; this is exactly what our typing axioms ensure. □

Theorem 5.5 (Correctness of Monomorphic Tags). *The encodings* $\widetilde{\mathtt{t}}?$ *and* $\widetilde{\mathtt{t}}??$ *are sound and complete for monomorphic problems.*

Proof. The proof for tags is analogous to that for guards, so we leave out the details. A model of Z^τ is *canonical* if the type tags are interpreted by the identity function. We construct a canonical model by deleting all the domain elements for which the type tag is not the identity. The typing axioms ensure that this gives us a model. □

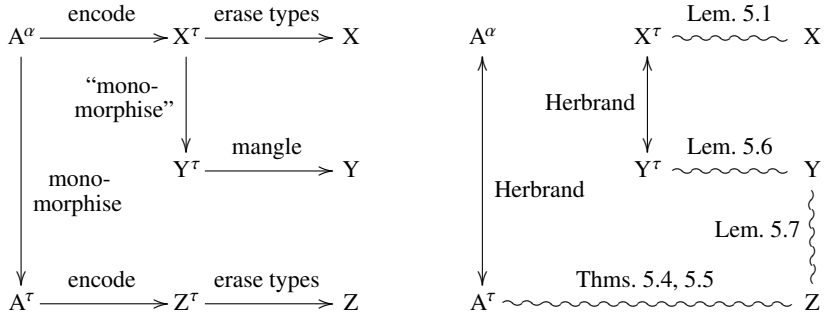
The above proof goes through even if we tag more terms than are necessary to ensure monotonicity. Hence, it is sound to tag negatively naked variables. We may also add further typing axioms to Z^τ—for example, equations $\mathtt{f}(\overline{U}, \mathtt{t}_\tau(X), \overline{V}) = \mathtt{f}(\overline{U}, X, \overline{V})$ to add or remove tags around well-typed arguments of a function symbol $\mathtt{f}$, or instances of the idempotence law $\mathtt{t}_\tau(\mathtt{t}_\tau(X)) = \mathtt{t}_\tau(X)$—provided that they hold for canonical models (where the type tag is the identity function) and preserve monotonicity.

5.2 Extension to Polymorphism

The next step is to lift the argument to polymorphic encodings and polymorphic problems. Regrettably, it is not possible to adjust the two-stage proof of Section 5.1 to polymorphism: without dependent types, neither the binary $\mathtt{g}$ predicate nor the binary $\mathtt{t}$ function can be typed, preventing us from constructing the polymorphic intermediate problem corresponding to Z^τ. Instead, we *reduce* the general, polymorphic case to the already proved monomorphic case. Our Herbrandian motto is,

> A polymorphic formula is equivalent to the set of its monomorphic instances, which in general will be an infinite set.

This complete form of monomorphisation is not to be confused with the finitary, heuristic monomorphisation algorithm presented in Section 4. Our proof exploits a form of commutativity between our encodings and complete monomorphisation.



(a) Transformations **(b)** Equisatisfiability

Figure 1. Relationships between problems

More specifically, given a polymorphic problem A^α, the following two routes (among others) are possible.

1. *Encode, then "monomorphise"*: Generate an untyped problem X from A^α using a polymorphic encoding; then generate all possible "monomorphic" instances of the problem's formulas by instantiating the encoded type variables with all possible "types" and mangle the resulting (generally infinite) set to obtain the problem Y. According to our motto, X and Y are equisatisfiable.
2. *Monomorphise, then encode*: Compute the set of monomorphic formulas A^τ by instantiating all type variables in A^α with all possible ground types; then translate A^τ to Z using the monomorphic variant of the chosen encoding. A^α and Z are equisatisfiable by Section 5.1 and our motto.

As in the monomorphic case, where we distinguished between the intermediate, typed problem Z^τ and the final, untyped Z, we find it useful to oppose X^τ to X and Y^τ to Y. Although the protectors g and t cannot be typed polymorphically, a crude typing is possible, with encoded types assigned the type ϑ and all other terms assigned ι, avoiding mixing types and terms in the encoded problems. Figure 1(a) summarises the situation.

Intuitively, the problems Y and Z obtained by taking routes 1 and 2 should be very similar. If we can show that they are in fact equisatisfiable, the desired equisatisfiability of A^α and X follows by transitivity. Figure 1(b) sketches the equisatisfiability proof. The missing equisatisfiabilities $\mathrm{Y}^\tau \sim \mathrm{Y} \sim \mathrm{Z}$ are proved below.

Lemma 5.6 (Correctness of Mangling). *The problems* Y^τ *and* Y *are equisatisfiable.*

Proof. The difference between Y^τ and Y is that the former has ground arguments of type ϑ, while the latter mangles them into the symbol names, e.g. $\mathsf{p}(\mathsf{b}^\vartheta, X^\iota)$ vs. $\mathsf{p}_b(X)$. Mangling is generally incomplete in an untyped logic; for example, the formula $X = Y \wedge \mathsf{q}(\mathsf{a}, U) \wedge \neg\, \mathsf{q}(\mathsf{b}, V)$ is unsatisfiable (since it implies $\mathsf{a} = \mathsf{b}$), but its mangled variant $X = Y \wedge \mathsf{q}_a(U) \wedge \neg\, \mathsf{q}_b(V)$ is satisfiable. In our two-typed setting, since there are no equations relating ϑ terms, mangling is easy to prove correct by considering (equality) Herbrand interpretations of the non-mangled and mangled formulas. □

Lemma 5.7 (Commutativity of Encoding and Monomorphisation). *The problems* Y *and* Z *are equisatisfiable.*

Proof. We start with an example that illustrates the reasoning behind the proof. As polymorphic problem A^{α}, we simply take the polymorphic list axiom

$$\forall X : \alpha,\, Xs : \mathit{list}(\alpha).\ \mathsf{head}(\mathsf{cons}(X, Xs)) = X$$

from Example 1.2. We suppose that $\mathit{list}(\alpha)$ is infinite (and hence monotonic) but the base type b is possibly nonmonotonic.

Following route 1, we apply the two-typed variant of $\mathsf{t}??$ directly to the polymorphic formula A^{α}. This yields the set X^{τ}, where the second axiom below repairs mismatches between tagged and untagged terms with the infinite type $\mathit{list}(\alpha)$:

$$\forall A : \vartheta,\, X : \iota,\, Xs : \iota.\ \mathsf{head}(A, \mathsf{cons}(A, X, Xs)) = \mathsf{t}(A, X)$$
$$\forall A : \vartheta,\, Xs : \iota.\ \mathsf{t}(\mathsf{list}(A), Xs) = Xs$$

This set would also contain a typing axiom for head, which we omit here. The constant b and the unary function list are the only function symbols with a result of type ϑ. Next, we instantiate the variables A with all ground terms of type ϑ, yielding Y^{τ}. Finally, we mangle Y^{τ}, transforming $\mathsf{head}(\mathsf{b}, t)$ into $\mathsf{head}_b(t)$ and so on. This gives Y:

$$\forall X,\, Xs.\ \mathsf{head}_b(\mathsf{cons}_b(X, Xs)) = \mathsf{t}_b(X) \qquad \forall Xs.\ \mathsf{t}_{\mathit{list}(b)}(Xs) = Xs$$
$$\forall X,\, Xs.\ \mathsf{head}_{\mathit{list}(b)}(\mathsf{cons}_{\mathit{list}(b)}(X, Xs)) = \mathsf{t}_{\mathit{list}(b)}(X) \qquad \forall Xs.\ \mathsf{t}_{\mathit{list}(\mathit{list}(b))}(Xs) = Xs$$
$$\vdots \qquad\qquad \vdots$$

In contrast, with route 2 we fully monomorphise A^{α} to A^{τ}. Then we use a monomorphic encoding, say, $\widetilde{\mathsf{t}}??$, to translate it into a set Z of untyped formulas

$$\forall X,\, Xs.\ \mathsf{head}_b(\mathsf{cons}_b(X, Xs)) = \mathsf{t}_b(X)$$
$$\forall X,\, Xs.\ \mathsf{head}_{\mathit{list}(b)}(\mathsf{cons}_{\mathit{list}(b)}(X, Xs)) = X$$
$$\vdots$$

Notice that the treatment of X in the right-hand sides above differs, since b is possibly nonmonotonic but $\mathit{list}(b)$ is infinite.

Are Y and Z equisatisfiable? The first formula of Z is also the first member of Y. The second formula of Z, however, does not appear in Y: the second formula of Y is the closest but its right-hand side is $\mathsf{t}_{\mathsf{list}(b)}(X)$ instead of X. Fortunately, Y also contains the axiom $\forall Xs.\ \mathsf{t}_{\mathit{list}(b)}(Xs) = Xs$, so Y must imply the second formula of Z. Conversely, Z does not mention the symbol $\mathsf{t}_{\mathit{list}(\tau)}$ for any τ, so we can add, for all ground types τ, the axiom $\forall Xs.\ \mathsf{t}_{\mathit{list}(\tau)}(Xs) = Xs$ to Z while preserving satisfiability. This new set implies all members of Y, including the second formula, so Y and Z are equisatisfiable.

We now generalise the above argument. Y contains axioms of the form $\mathsf{g}_{\tau}(X)$ or $\mathsf{t}_{\tau}(X) = X$ for each infinite type τ, whereas Z does not mention g_{τ} or t_{τ} for these types because they are monotonic; we can add the corresponding axioms to Z while preserving satisfiability. Otherwise, Y and Z contain the same formulas, except when A^{α} quantifies over a variable X of a possibly nonmonotonic type with an infinite instance τ. Z will not protect the instances of X that have type τ, but Y might; however, since τ is infinite, Y must contain $\mathsf{g}_{\tau}(X)$ or $\mathsf{t}_{\tau}(X) = X$, allowing us to remove the guard or tag. Hence, the two sets of formulas are equisatisfiable. □

Theorem 5.8 (Correctness of Polymorphic Encodings). *The encodings* `g?`*,* `g??`*,* `t?`*, and* `t??` *are sound and complete.*

Proof. This follows from Lemmas 5.1, 5.6, and 5.7, Theorems 5.4 and 5.5, and Herbrand's theorem (for terms and for types), as depicted in Figure 1(b). The application of Lemma 5.1 to erase ϑ and ι in X^τ requires X^τ to be monotonic; this can be proved either in the style of MONO in the proof of Theorem 5.4 or by observing that monotonicity is preserved along the equisatisfiability chain $\mathrm{Z}^\tau \sim \mathrm{Z} \sim \mathrm{Y} \sim \mathrm{Y}^\tau \sim \mathrm{X}^\tau$. □

6 Evaluation

To evaluate the type encodings described in this paper, we put together two sets of exactly 1000 polymorphic first-order problems originating from 10 Isabelle theories, translated with Sledgehammer's help (100 problems per theory).[3] Nine of the theories are the same as in a previous evaluation [2]; the tenth one is an optimality proof for Huffman's algorithm.[4] The problems in the first benchmark set include about 150 heuristically selected axioms (before monomorphisation); that number is increased to 500 for the second set, to reveal how well the encodings scale with the problem size.

We evaluated each type encoding with four modern automatic theorem provers: the resolution provers E 1.4, SPASS 3.7, and Vampire 1.8, and the SMT solver Z3 3.0. Each prover was invoked with the set of options we had previously determined worked best for Sledgehammer.[5] The provers were granted 20 seconds of CPU time per problem on one core of a 3.06 GHz Dual-Core Intel Xeon processor. Most proofs were found within a few seconds; a higher time limit would have had little impact on the success rate [8].

Figures 2 and 3 give, for each combination of prover and encoding, the number of solved problems from each problem set. Rows marked with ∼ concern the monomorphic encodings. To avoid giving the unsound encodings an unfair advantage, proof search is followed by a certification phase that tries to re-find the proof using a combination of sound encodings. For the second benchmark set, Figure 4 presents the average number of clauses, literals per clause, symbols per atom, and symbols for clausified problems (using E's clausifier), to give an idea of each encoding's overhead.

The monomorphic versions of our more advanced scheme, especially $\widetilde{\mathtt{g}}$`!!` and $\widetilde{\mathtt{g}}$`??`, performed best overall. This confirms the intuition that clutter (whether type arguments, guards, or tags) slows down automatic provers. Surprisingly, some of our monomorphic encodings outperformed Vampire's and Z3's native support for simple types ($\widetilde{\mathtt{n}}$), partly because the type support in Vampire 1.8 is unsound (leading to many rejected proofs) and interferes with the prover's internal strategy scheduling. Polymorphic encodings lag behind, but our approach nonetheless constitutes a substantial improvement over the traditional polymorphic schemes. The best unsound encodings performed very slightly better than the best sound ones.

[3] The TPTP benchmark suite [19], which is customarily used for evaluating theorem provers, has just begun collecting polymorphic (TFF1) problems [4, §6].

[4] Our test data are available at `http://www21.in.tum.de/~blanchet/ijcar2012-data.tgz`.

[5] The setup for E was suggested by Stephan Schulz and includes the little known "symbol offset" weight function. We passed `-Auto`, `-SOS=1`, `-Splits=0`, `-VarWeight=3`, and `-FullRed=0` to SPASS. We ran Vampire in CASC mode and Z3 with model-based quantifier instantiation.

	UNSOUND						SOUND						
	e	a	g!!	t!!	g!	t!	g??	t??	g?	t?	g	t	n
E 1.4	316	362	350	353	343	338	344	351	345	302	255	295	–
~	–	344	**390**	389	382	372	388	**390**	386	373	355	334	–
SPASS 3.7	275	324	305	308	309	293	291	309	290	242	247	267	–
~	–	267	**337**	334	334	322	**344**	341	340	333	321	311	–
VAMPIRE 1.8	291	376	328	326	333	333	331	313	325	294	240	211	–
~	–	357	**385**	368	376	379	**381**	374	365	364	303	238	376
Z3 3.0	295	365	345	347	328	313	329	333	307	260	253	319	–
~	–	339	**366**	360	362	362	353	352	356	348	349	314	**361**

Figure 2. Number of solved problems with 150 axioms

	UNSOUND						SOUND						
	e	a	g!!	t!!	g!	t!	g??	t??	g?	t?	g	t	n
E 1.4	193	339	309	308	312	300	313	310	304	245	199	216	–
~	–	309	**368**	365	363	354	364	**366**	364	350	314	308	–
SPASS 3.7	186	306	287	285	284	274	274	294	264	204	198	243	–
~	–	247	310	304	**312**	284	**308**	306	300	276	239	231	–
VAMPIRE 1.8	180	350	282	292	298	286	282	288	283	279	185	148	–
~	–	315	344	344	**354**	334	339	342	339	331	251	171	**346**
Z3 3.0	175	339	319	323	311	296	306	303	262	222	200	219	–
~	–	302	343	343	**350**	337	343	340	**344**	327	329	254	337

Figure 3. Number of solved problems with 500 axioms

	UNSOUND						SOUND					
AVG. NUM.	e	a	g!!	t!!	g!	t!	g??	t??	g?	t?	g	t
CLAUSES	**749**	808	896	835	896	835	974	867	974	867	1107	**827**
~	–	1080	1139	1105	1139	1105	1176	1105	1176	1105	1649	1105
LITERALS	2.32	2.55	2.70	2.60	3.00	2.55	2.84	2.59	3.88	2.49	5.41	2.56
PER CLAUSE ~	–	2.28	2.35	**2.22**	2.35	2.29	2.57	**2.15**	2.57	2.29	4.74	2.29
SYMBOLS	**6.7**	9.3	15.8	16.6	14.5	18.3	14.8	16.4	11.9	25.6	6.3	28.1
PER ATOM ~	–	7.2	6.9	7.4	7.0	8.0	6.3	7.6	6.5	8.3	**4.4**	14.0
SYMBOLS	**11.7**	19.2	38.2	38.6	39.0	38.9	40.9	41.5	45.1	55.2	38.0	59.5
('000) ~	–	17.8	18.5	18.6	18.7	20.1	**19.0**	19.2	19.6	21.0	34.4	35.3

Figure 4. Average size of clausified problems with 500 axioms

From both a conceptual and an implementation point of view, the encodings are all instances of a general framework, in which mostly orthogonal features can be combined in various ways. Defining such a large number of encodings makes it possible to select the most appropriate scheme for each automatic prover, based on empirical evidence. In fact, using time slicing or parallelism, it pays off to have each prover employ a combination of encodings with complementary strengths.

7 Related Work

The earliest descriptions of type guards and type tags we are aware of are due to Enderton [12, §4.3] and Stickel [18, p. 99]. Wick and McCune [22, §4] compare full type erasure, guards, and tags. Type arguments are reminiscent of System F; they are described by Meng and Paulson [14], who also present a translation of axiomatic type classes.

The intermediate verification language and tool Boogie 2 [13] supports a restricted form of higher-rank polymorphism (with polymorphic maps), and its cousin Why3 [6] provides rank-1 polymorphism. Both define translations to a monomorphic logic and rely on proxies to handle interpreted types [7,11,13]. One of the Boogie translations [13, §3.1] uses SMT triggers to prevent ill-typed instantiations in conjunction with type arguments; however, this approach is risky in the absence of a semantics for triggers.

An alternative to encoding polymorphic types or monomorphising them away is to support them natively in the prover. This is ubiquitous in interactive theorem provers, but perhaps the only automatic prover that supports polymorphism is Alt-Ergo [5].

Blanchette and Krauss [3] studied monotonicity inferences for higher-order logic without polymorphism. Claessen et al. [10] were first to apply them to type erasure.

8 Conclusion

This paper introduced a family of efficient translations from polymorphic into untyped first-order logic. Our approach soundly erases all types that are inferred monotonic, as well as most occurrences of the remaining types. The new translations outperform the traditional sound encoding schemes as well as common unsound schemes.

We implemented the new translations in the Sledgehammer tool for Isabelle/HOL [2, 14], thereby addressing a recurring user complaint. Although Isabelle certifies external proofs, unsound proofs are annoying and often conceal sound proofs. The same translation module forms the core of Isabelle's TPTP exporter tool, which makes entire theorem libraries available to first-order reasoners. Our refinements to the monomorphic case have made their way into the Monotonox translator [10]. Applications such as Boogie [13], Why3 [6], and LEO-II [1] also stand to gain from a lighter translation.

The TPTP family recently welcomed the addition of TFF1 [4], an extension of TFF0 [20] with rank-1 polymorphism. Equipped with a concrete syntax and translation tools, we could turn any popular automatic theorem prover into an efficient polymorphic prover. Translating the untyped proof back into a typed proof is usually straightforward, but there are important corner cases that call for more research. It should also be possible to extend our approach to integrate interpreted arithmetic types and symbols.

A promising direction for future research would be to look into strengthening the monotonicity analysis. Type arguments severely clutter our polymorphic translations; they can often be omitted soundly, but we lack an inference to find out precisely when.

Acknowledgement. Koen Claessen and Tobias Nipkow made this collaboration possible. Andrei Popescu helped us tackle the correctness proof. Peter Lammich, Rustan Leino, Tobias Nipkow, Mark Summerfield, Tjark Weber, and an anonymous reviewer suggested several textual improvements. We thank them all.

References

[1] Benzmüller, C., Paulson, L.C., Theiss, F., Fietzke, A.: LEO-II—A cooperative automatic theorem prover for higher-order logic. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNAI, vol. 5195, pp. 162–170. Springer (2008)

[2] Blanchette, J.C., Böhme, S., Paulson, L.C.: Extending Sledgehammer with SMT solvers. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNAI, vol. 6803, pp. 207–221. Springer (2011)

[3] Blanchette, J.C., Krauss, A.: Monotonicity inference for higher-order formulas. J. Autom. Reasoning 47(4), 369–398 (2011)

[4] Blanchette, J.C., Paskevich, A.: TFF1: The TPTP typed first-order form with rank-1 polymorphism—Version 1.0. Submitted to IJCAR 2012

[5] Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: Barrett, C., de Moura, L. (eds.) SMT 2008 (2008)

[6] Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Leino, K.R.M., Moskal, M. (eds.) Boogie 2011. pp. 53–64 (2011)

[7] Bobot, F., Paskevich, A.: Expressing polymorphic types in a many-sorted language. In: Tinelli, C., Sofronie-Stokkermans, V. (eds.) FroCoS 2011. LNAI, vol. 6989, pp. 87–102. Springer (2011)

[8] Böhme, S., Nipkow, T.: Sledgehammer: Judgement Day. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNAI, vol. 6173, pp. 107–121. Springer (2010)

[9] Claessen, K., Lilliesträm, A.: Automated inference of finite unsatisfiability. J. Autom. Reasoning 47(2), 111–132 (2011)

[10] Claessen, K., Lilliesträm, A., Smallbone, N.: Sort it out with monotonicity—Translating between many-sorted and unsorted first-order logic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE-23. LNAI, vol. 6803, pp. 207–221. Springer (2011)

[11] Couchot, J.F., Lescuyer, S.: Handling polymorphism in automated deduction. In: Pfenning, F. (ed.) CADE-21. LNAI, vol. 4603, pp. 263–278. Springer (2007)

[12] Enderton, H.B.: A Mathematical Introduction to Logic. Academic Press (1972)

[13] Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 312–327. Springer (2010)

[14] Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. J. Autom. Reasoning 40(1), 35–60 (2008)

[15] de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer (2008)

[16] Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Comm. 15(2-3), 91–110 (2002)

[17] Schulz, S.: System description: E 0.81. In: Basin, D., Rusinowitch, M. (eds.) IJCAR 2004. LNAI, vol. 3097, pp. 223–228. Springer (2004)

[18] Stickel, M.E.: Schubert's steamroller problem: Formulations and solutions. J. Autom. Reasoning 2(1), 89–101 (1986)

[19] Sutcliffe, G.: The TPTP problem library and associated infrastructure—The FOF and CNF parts, v3.5.0. J. Autom. Reasoning 43(4), 337–362 (2009)

[20] Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Bjørner, N., Voronkov, A. (eds.) LPAR-18. LNCS, vol. 7180, pp. 406–419. Springer (2012)

[21] Weidenbach, C.: Combining superposition, sorts and splitting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning. pp. 1965–2013. Elsevier (2001)

[22] Wick, C.A., McCune, W.W.: Automated reasoning about elementary point-set topology. J. Autom. Reasoning 5(2), 239–255 (1989)