Expressing Polymorphic Types in a Many-Sorted Language

François Bobot[1,2] and Andrei Paskevich[1,2]

[1] LRI, Université Paris-Sud 11, CNRS, Orsay F-91405
[2] INRIA Saclay-Île de France, ProVal, Orsay F-91893

Abstract. In this paper, we study translation from a first-order logic with polymorphic types à la ML (of which we give a formal description) to a many-sorted or one-sorted logic as accepted by mainstream automated theorem provers. We consider a three-stage scheme where the last stage eliminates polymorphic types while adding the necessary "annotations" to preserve soundness, and the first two stages serve to protect certain terms so that they can keep their original unannotated form. This protection allows us to make use of provers' built-in theories and operations. We present two existing translation procedures as sound and complete instances of this generic scheme. Our formulation generalizes over the previous ones by allowing us to protect terms of arbitrary monomorphic types. In particular, we can benefit from the built-in theory of arrays in SMT solvers such as Z3, CVC3, and Yices. The proposed methods are implemented in the Why3 tool and we compare their performance in combination with several automated provers.

1 Introduction

Polymorphic types are a means of abstraction over families of different types; a polymorphic definition or proposition stands for a potentially infinite number of its type-specific instances. Type systems employing polymorphism arise naturally in programming languages and they are a prominent feature of interactive proof assistants such as Coq [17] or Isabelle/HOL [16].

However, a proof task written in a language with polymorphic types is today a difficult subject for automation. This is not because polymorphism handling in a prover is complicated or inefficient *per se*. As was demonstrated by the Alt-Ergo project [3], this only requires a straightforward extension of the unification procedure and does not impose any significant overhead. The fact is, advanced type systems have not yet become mainstream in automated deduction: SMT solvers use many-sorted languages such as SMT-LIB [1], and TPTP provers are content with one-sorted first-order language. Thus, to apply a mainstream prover to a problem expressed in a polymorphic language, we have to translate it into an equivalent monomorphic or even one-sorted problem.

The challenge is not new and a number of solutions is known, ranging from adding per-variable "type guards" (also known as "relativisation of quantifiers", see [12] and [11, Sect. 3.0]), to throughout decoration of terms with their types

[9, 6], to various flavours of type erasure [10, 14, 11]. The latter method is logically unsound, though adding type annotations can prevent certain unsound inference steps (see [14, Sect. 2.5,2.6] and [11, Sect. 3.1]).

An important feature of a polymorphism encoding method is special treatment of types and operations that are directly handled by provers' built-in decision procedures, e.g., for linear arithmetic or bit-vectors. The idea is to prevent the terms that can be interpreted by a prover from being modified by translation, to preserve their original form [6, 11]. In what follows, we call this "type protection" to emphasize that we are interested in terms of particular types.

In this work, we aim to lift (or at least work around) several limitations we perceive in the previous approaches. Firstly, the existing type protection techniques only handle "simple" types, like integers or booleans, but not instances of polymorphic types, like lists of integers or arrays of reals. Yet decision procedures for such "complex" types are implemented in some SMT solvers; for example, Z3 [15], CVC3 [2], and Yices [7] have a built-in support for arrays. Secondly, type protection, as defined in [6], cannot be used to protect finite types such as booleans: given an axiom "every boolean is equal either to 'true' or to 'false'", one can derive that there are only two values in any encoded type, which can easily lead to a contradiction. Thirdly, while translation by type erasure with addition of type arguments to polymorphic symbols [11, Sect. 3.1] is less intrusive and more efficient than full term decoration [6], the former method is unsound and, according to [11], is only applicable in combination with provers that use trigger-based rather than unification-based instantiation. Such a requirement excludes the superposition-based provers and may be difficult to test when a third-party prover is used.

We begin with a formal presentation of first-order logic with polymorphic types (Section 2). In particular, we show that complete monomorphisation is undecidable, that is, we cannot effectively compute a finite set of monomorphic instances of a polymorphic formula F that is equisatisfiable to F. Then we introduce a generic three-stage scheme of polymorphism encoding (Section 3). In this scheme, we start by replacing interpreted polymorphic symbols (such as operations of access and update in arrays) with selected monomorphic instances. The translation proceeds then to type protection, which we consider as a separate transformation, and concludes with polymorphism elimination proper.

We present a sound and complete method of type instantiation with symbol discrimination for the first stage (Section 3.1). Furthermore, we give a generalized formulation of the type protection method from [6], free from the aforesaid restrictions (Section 3.2). As third-stage transformations, we consider full term decoration from [9, 6] (Section 3.3) and type erasure with added type annotations from [11, Sect. 3.1] (Section 3.4). We show the latter method to be sound on problems that admit models with infinite domains for every non-protected type and we discuss how this condition can be handled in practice.

We conclude by comparing the described techniques in combination with the SMT solvers Z3, CVC3, and Yices [7] on a set of about 4100 proof obligations in the Why3 tool [4] (Section 4).

2 First-Order Logic With Polymorphic Types

The logic $\mathbf{FOL}_\mathrm{T}$, presented below, is an extension of classical first-order many-sorted logic. In $\mathbf{FOL}_\mathrm{T}$, types are built from type constants (such as "integer"), type functions (such as "list of"), and type variables that stand for arbitrary monomorphic types. We do not admit quantifiers over type variables, neither in types, nor in formulas: a polymorphic formula is rather seen as a scheme, a potentially infinite conjunction of its monomorphic type instances. In other words, every type variable that occurs in a formula is bound by an implicit prenex universal quantifier. Basically, we use type polymorphism as a convenient way to write a set of polymorphic axioms — say, for lists or arrays — once, instead of copying them for every particular instance of these types.

The principal purpose of $\mathbf{FOL}_\mathrm{T}$ is to help specify and prove programs and its type system can be seen as the first-order fragment of the ML type system. The Why3 verification tool [4] is based upon $\mathbf{FOL}_\mathrm{T}$ with some extensions such as algebraic types. The papers [6] and [11] work in a similar setting, though the latter employs explicit quantifiers over type variables in logic formulas.

Syntax. We define types as syntactical expressions built from *type constructors* of fixed arity (denoted with capital sans-serif letters) and *type variables* (denoted α, β, γ). For example, β, I, $\mathsf{F}(\mathsf{I}, \gamma)$ are well-formed types. Type constructors of arity 0 are called *type constants*. A type that contains no type variables is called *monomorphic type* or *sort*. A vector of types $\langle T_1, \ldots, T_n \rangle$ is called *type signature*.

A *type substitution* is a mapping from type variables to types. A *monomorphic* type substitution maps every type variable either to itself or to a sort. A type T is said to *match* another type T' whenever there is a type substitution that instantiates T to T'. This notion is trivially extended to type signatures.

We use letters S and T for types, boldface letters $\mathbf{S}$, $\mathbf{T}$ for type signatures, and Greek letters τ, θ, and π for type substitutions. We denote the set of available type variables with $\mathbb{V}_\mathrm{T}$, the set of type constructors with $\mathbb{F}_\mathrm{T}$, the set of all types built from $\mathbb{V}_\mathrm{T}$ and $\mathbb{F}_\mathrm{T}$ with $\mathcal{T}(\mathbb{F}_\mathrm{T}, \mathbb{V}_\mathrm{T})$, and the set of all sorts with $\mathcal{T}(\mathbb{F}_\mathrm{T})$. We presume that $\mathbb{V}_\mathrm{T}$ is infinite and $\mathbb{F}_\mathrm{T}$ contains at least one type constant.

We use traditional first-order terms and formulas, built from variable symbols (denoted u, v, w), function symbols (denoted f, g, h), and predicate symbols (denoted p, q), with the following additions:

- Every term carries an explicit type, e.g.: $w : \mathsf{C}(\mathsf{I})$, $f(u : \alpha, v : \mathsf{L}(\alpha)) : \mathsf{L}(\alpha)$. We denote terms with letters s and t, and, by abuse of notation, we sometimes write the type of a term to the right of the letter: $s : T_1$, $t : T_2$, and so on.
- A *variable* is a variable symbol with a type, and we treat $w : \mathsf{C}(\mathsf{I})$ and $w : \mathsf{C}(\alpha)$ as two distinct variables even though they share the same variable symbol.
- To each function symbol of arity n we assign a type signature of length $n+1$. A term of the form $f(t_1 : T_1, \ldots, t_n : T_n) : T$ is well-formed if and only if the type signature of f matches $\langle T_1, \ldots, T_n, T \rangle$.
- To each predicate symbol of arity n we give a type signature of length n. An atomic formula of the form $p(t_1 : T_1, \ldots, t_n : T_n)$ is well-formed if and only if the type signature of p matches $\langle T_1, \ldots, T_n \rangle$.

– An atomic equality formula of the form $t_1 \approx t_2$ is well-formed if and only if the terms t_1, t_2 have the same type.
– Quantifiers bind variables, i.e., typed variable symbols: $\forall(u:\alpha)\, p(u:\alpha, u:\mathsf{C})$. Here, the first argument of p is bound, but the second one is free.

We treat equality ($\approx$), negation ($\neg$), conjunction ($\wedge$), and the universal quantifier ($\forall$) as logical symbols and we treat disjunction ($\vee$), implication ($\supset$), equivalence ($\equiv$), disequality ($\not\approx$), and the existential quantifier ($\exists$) as abbreviations.

We use letters x, y, z for variables, letters F, G, H for formulas, and Greek letters Γ, Δ for sets of formulas. We denote the (infinite) set of variable symbols with $\mathbb{V}$, the set of function symbols with $\mathbb{F}$, and the set of predicate symbols with $\mathbb{P}$. Given a term or a formula e, the set of type variables occurring in e is denoted $\mathrm{FV}_\mathrm{T}(e)$ and the set of free variables of e is denoted $\mathrm{FV}(e)$. If $\mathrm{FV}_\mathrm{T}(e)$ is empty, we call e *monomorphic*. If $\mathrm{FV}(e)$ is empty, we call e *closed* or *ground*.

Substitutions, denoted with letters σ and δ, apply to a term or a formula e, replacing free variables with terms of the same type (denoted $e\sigma$). The symbol $\circ$ denotes the composition of two (type) substitutions: $x(\sigma \circ \delta) \triangleq x\sigma\delta$.

Type substitutions apply only to closed formulas and ground terms; also, we require type instantiation to rename every bound variable symbol to some fresh variable symbol. In this way, we avoid variable collisions: for example, the type substitution $[\mathsf{I}/\alpha]$ would not instantiate the formula $\forall(u:\alpha)\forall(u:\mathsf{I})\, p(u:\alpha, u:\mathsf{I})$ to $\forall(u:\mathsf{I})\forall(u:\mathsf{I})\, p(u:\mathsf{I}, u:\mathsf{I})$, but to $\forall(u':\mathsf{I})\forall(u'':\mathsf{I})\, p(u':\mathsf{I}, u'':\mathsf{I})$. In our subsequent examples, we will not use a variable symbol in two different variables in the same formula to avoid confusion.

In what follows, we illustrate our transformations on the following simple polymorphic formula (for the sake of readability, we omit the most obvious type annotations): $\forall(m:\mathsf{M}(\alpha,\mathsf{I}))\,\forall(c:\alpha)\,\mathtt{get}(\mathtt{set}(m,c,6):\mathsf{M}(\alpha,\mathsf{I}),c):\mathsf{I}*7 \approx 42$. Here, the type I represents integers and the type $\mathsf{M}(\alpha,\beta)$ is that of polymorphic α-to-β maps. The function symbol `get` is of type signature $\langle \mathsf{M}(\alpha,\beta), \alpha, \beta\rangle$ and `set` is of type signature $\langle \mathsf{M}(\alpha,\beta), \alpha, \beta, \mathsf{M}(\alpha,\beta)\rangle$.

Satisfiability. Given sets $\mathbb{F}_\mathrm{T}$, $\mathbb{F}$, $\mathbb{P}$, an interpretation $\mathfrak{I}$ is defined by three maps:

– to each sort $S \in \mathcal{T}(\mathbb{F}_\mathrm{T})$, we assign a non-empty *domain* $\mathcal{D}^\mathfrak{I}_S$;
– to each symbol $f \in \mathbb{F}$ and each vector of sorts $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$ matched by the type signature of f, we assign a function $f^\mathfrak{I}_\mathbf{S} : \mathcal{D}^\mathfrak{I}_{S_1} \times \cdots \times \mathcal{D}^\mathfrak{I}_{S_n} \to \mathcal{D}^\mathfrak{I}_S$;
– to each symbol $p \in \mathbb{P}$ and each vector of sorts $\mathbf{S} = \langle S_1, \ldots, S_n\rangle$ matched by the type signature of p, we assign a function $p^\mathfrak{I}_\mathbf{S} : \mathcal{D}^\mathfrak{I}_{S_1} \times \cdots \times \mathcal{D}^\mathfrak{I}_{S_n} \to \{\top, \bot\}$, where $\top$ and $\bot$ stand for Boolean constants "true" and "false", respectively.

We call *type valuation* a type substitution that instantiates every type variable in $\mathbb{V}_\mathrm{T}$ to a sort. Given a type valuation π, we call *variable valuation under* π a function that maps every variable $u:T$ to some element of $\mathcal{D}^\mathfrak{I}_{T\pi}$. We simply say *variable valuation* when the implied type valuation is known from the context or in a purely monomorphic setting, where every type is already closed.

Let π be a type valuation and ξ be a variable valuation under π. We evaluate terms and formulas according to the following equalities, where $\mathbf{t}:\mathbf{T}$ stands for

a vector of terms $t_1 : T_1, \ldots, t_n : T_n$ and $\xi[u : T \mapsto a]$ is a valuation that coincides with ξ everywhere except $u : T$, which is mapped to a.

$$\begin{aligned}
\mathfrak{I}_{\pi,\xi}(u:T) &\triangleq \xi(u:T) & \mathfrak{I}_{\pi,\xi}(t_1 \approx t_2) &\triangleq (\mathfrak{I}_{\pi,\xi}(t_1) = \mathfrak{I}_{\pi,\xi}(t_2)) \\
\mathfrak{I}_{\pi,\xi}(f(\mathbf{t}:\mathbf{T}):T) &\triangleq f^{\mathfrak{I}}_{\langle \mathbf{T},T\rangle\pi}(\mathfrak{I}_{\pi,\xi}(\mathbf{t})) & \mathfrak{I}_{\pi,\xi}(\neg F) &\triangleq \neg\, \mathfrak{I}_{\pi,\xi}(F) \\
\mathfrak{I}_{\pi,\xi}(p(\mathbf{t}:\mathbf{T})) &\triangleq p^{\mathfrak{I}}_{\mathbf{T}\pi}(\mathfrak{I}_{\pi,\xi}(\mathbf{t})) & \mathfrak{I}_{\pi,\xi}(F \wedge G) &\triangleq \mathfrak{I}_{\pi,\xi}(F) \wedge \mathfrak{I}_{\pi,\xi}(G) \\
& & \mathfrak{I}_{\pi,\xi}(\forall(u:T)F) &\triangleq \bigwedge_{a \in \mathcal{D}^{\mathfrak{I}}_{T\pi}} \mathfrak{I}_{\pi,\xi[u:T \mapsto a]}(F)
\end{aligned}$$

It is easy to see that evaluation of a term or a formula e under $\mathfrak{I}_{\pi,\xi}$ does not depend on the values of π and ξ on (type) variables that do not occur in e. In what follows, when we evaluate closed or monomorphic expressions, we often omit the variable valuation or the type valuation, respectively.

Lemma 1. *For any closed formula F and type valuation π, $\mathfrak{I}_\pi(F) = \mathfrak{I}(F\pi)$.*

As we said above, we treat type variables as implicitly universally quantified at the top of a polymorphic formula. Thus, a closed formula F is *satisfied* by $\mathfrak{I}$ if and only if $\mathfrak{I}_\pi(F)$ is true for every type valuation π. A closed formula is *satisfiable* if and only if it is satisfied by some interpretation, called a *model* of this formula. These definitions are trivially extended to sets of closed formulas. To prove a polymorphic formula G in a polymorphic context Γ, we take a type substitution τ that replaces all type variables in G with fresh type constants and show that the set $\Gamma, \neg G\tau$ is unsatisfiable. Generally speaking, the semantics of polymorphic formulas in $\mathbf{FOL}_{\mathrm{T}}$ is quite similar to that of first-order clauses, where the free variables are also implicitly universally quantified.

On monomorphic terms and formulas, our definitions correspond to the traditional many-sorted logic with disjoint sorts. Moreover, a trivial corollary of Lemma 1 is that F is satisfiable if and only if the set of all monomorphic type instances of F is satisfiable.

Computing monomorphic instances? A polymorphic formula can have infinitely many monomorphic type instances. But can't we find out, in finite time, all sorts that are potentially relevant to the problem and deal with a finite subset of instances, produced just with these sorts? On one hand, this resembles an attempt to pre-compute the relevant ground instances in a set of first-order formulas — a problem well known to be undecidable. On the other hand, type handling does not need to be as hard as proof search in general, and complete monomorphisation is often possible in programming languages (e.g., C++ templates).

Theorem 1. *There is no computable function that maps an arbitrary closed formula F to an equisatisfiable finite set of monomorphic type instances of F (notice that such a set always exists by compactness).*

Proof. It turns out that our type system is expressive enough to encode an undecidable theory, namely, combinatory logic. Consider the following signature:

$$\mathbb{F}_{\mathrm{T}} = \{\, \mathsf{A}(\cdot,\cdot),\, \mathsf{S},\, \mathsf{K} \,\} \quad \mathbb{F} = \{\, \mathtt{A} : \langle \alpha, \beta, \mathsf{A}(\alpha,\beta)\rangle,\, \mathtt{S} : \langle \mathsf{S}\rangle,\, \mathtt{K} : \langle \mathsf{K}\rangle \,\} \quad \mathbb{P} = \{\, \mathtt{R} : \langle \alpha, \beta\rangle \,\}$$

along with five axioms (for brevity, we omit some type annotations):

$$\forall(u:\alpha)\,\forall(v:\beta)\,\forall(w:\gamma)\;((\mathtt{R}(u,v)\wedge\mathtt{R}(v,w))\supset\mathtt{R}(u,w))$$

$$\forall(u:\alpha)\,\forall(v:\beta)\,\forall(w:\gamma)\;(\mathtt{R}(u,v)\supset\mathtt{R}(\mathtt{A}(u,w):\mathsf{A}(\alpha,\gamma),\mathtt{A}(v,w):\mathsf{A}(\beta,\gamma)))$$

$$\forall(u:\alpha)\,\forall(v:\beta)\,\forall(w:\gamma)\;(\mathtt{R}(u,v)\supset\mathtt{R}(\mathtt{A}(w,u):\mathsf{A}(\gamma,\alpha),\mathtt{A}(w,v):\mathsf{A}(\gamma,\beta)))$$

$$\forall(u:\alpha)\,\forall(v:\beta)\;\mathtt{R}(\mathtt{A}(\mathtt{A}(\mathsf{K},u),v):\mathsf{A}(\mathsf{A}(\mathsf{K},\alpha),\beta),\;u:\alpha)$$

$$\forall(u:\alpha)\,\forall(v:\beta)\,\forall(w:\gamma)\;\mathtt{R}(\mathtt{A}(\mathtt{A}(\mathtt{A}(\mathsf{S},u),v),w):\mathsf{A}(\mathsf{A}(\mathsf{A}(\mathsf{S},\alpha),\beta),\gamma),\;\mathtt{A}(\mathtt{A}(u,w),\mathtt{A}(v,w)):\mathsf{A}(\mathsf{A}(\alpha,\gamma),\mathsf{A}(\beta,\gamma)))$$

Here the binary function symbol $\mathtt{A}$ stands for term application, and the binary predicate symbol $\mathtt{R}$ for CL-reducibility. Notice that every ground combinatory term is reflected in its type.

Now, if we were able to compute a finite set of potentially relevant *closed types* for an arbitrary reducibility problem in this theory, this would readily let us decide the problem itself, as we would thus obtain the set of potentially relevant *ground terms*. Since ground reducibility in CL is undecidable, complete monomorphisation in $\mathbf{FOL}_{\mathrm{T}}$ is undecidable, too. □

3 Eliminating Polymorphic Types

Being unable to select just a relevant monomorphic subset of a polymorphic problem, we have to resort to some form of encoding, converting the polymorphic problem to an equisatisfiable monomorphic one. Such conversion inevitably implies merging many types into few sorts or just a single sort. This is undesirable if we target an automated prover equipped with special techniques (decision procedures, unification modulo, etc.) for particular types, such as integers, booleans or arrays. These types ought to be separated from the rest, protected against this "type fusion", expelled from polymorphism in the problem.

To this purpose, we slightly extend our language in order to be able to select the terms that will keep their (monomorphic) type through polymorphism elimination. To every sort S in $\mathcal{T}(\mathbb{F}_{\mathrm{T}})$ we associate a new *protected sort* $\bar{S}$. The use of protected sorts is restricted: a protected sort can appear in the type signature of a symbol or as a type of a term, but it cannot occur under a type constructor or in the range of a type substitution. In other words, the only type that matches a protected sort $\bar{S}$ is $\bar{S}$ itself.

For example, $\mathtt{get}(v:\overline{\mathsf{M}(\mathsf{I},\mathsf{I})},c:\mathsf{I}):\mathsf{I}$ is a malformed term, since the type signature of $\mathtt{get}$ is $\langle\mathsf{M}(\alpha,\beta),\alpha,\beta\rangle$ and $\mathsf{M}(\alpha,\beta)$ does not match $\overline{\mathsf{M}(\mathsf{I},\mathsf{I})}$. Similarly, the term $\mathtt{get}(v:\mathsf{M}(\mathsf{I},\bar{\mathsf{I}}),c:\mathsf{I}):\bar{\mathsf{I}}$ is malformed, because β does not match $\bar{\mathsf{I}}$ and also because $\mathsf{M}(\mathsf{I},\bar{\mathsf{I}})$ is a malformed type expression. However, if we consider a "protected specialization" of $\mathtt{get}$, denoted $\overline{\mathtt{get}}$, with the type signature $\langle\overline{\mathsf{M}(\mathsf{I},\mathsf{I})},\bar{\mathsf{I}},\bar{\mathsf{I}}\rangle$, the application $\overline{\mathtt{get}}(v:\overline{\mathsf{M}(\mathsf{I},\mathsf{I})},c:\bar{\mathsf{I}}):\bar{\mathsf{I}}$ is a well-formed term.

Concerning interpretation, every protected sort $\bar{S}$ has its proper non-empty domain $\mathcal{D}^{\mathfrak{I}}_{\bar{S}}$. As with any type substitution, we restrict type valuations to non-protected sorts. Thus, a set $\{\forall(u:\alpha)\,\forall(v:\alpha)\,u \approx v,\ \exists(a:\bar{\mathrm{I}})\,\exists(b:\bar{\mathrm{I}})\,a \not\approx b\}$ is satisfiable. Indeed, the first formula requires the domain of every non-protected sort to be a singleton, but does not constrain the domains of protected sorts.

Using protected sorts, we can define a general three-stage scheme of encoding of polymorphic formulas, explained below from the end to the beginning.

The final, "type-fusing" stage takes a set of polymorphic formulas with protected sorts and converts it into an equisatisfiable set of monomorphic formulas. A common requirement to the methods on this stage is preservation of terms with protected types: monomorphic protected fragments of the problem, e.g., arithmetic expressions, must be sent to a prover as is. We present two "type-fusing" transformations, Dec and Exp, in Sections 3.3 and 3.4. Both methods have been previously described in the literature [9, 10, 14, 6, 11]. Our presentation is more general in that it permits to protect arbitrarily complex monomorphic types, such as "list of integers" or "integer-to-real map". The ability to preserve such sorts is of more than purely theoretical interest: as we have already mentioned, Z3, CVC3, and Yices provide built-in support for access and update operations on integer-indexed arrays.

The intermediate, "type-protecting" stage takes a set of polymorphic formulas without protected sorts and converts it into an equisatisfiable set of formulas with protection. The methods on this stage take as a parameter the set of sorts that we wish to protect; we expect them to put protection over every occurrence of every sort from this set in the problem. We present a type-protecting transformation called Tw in Section 3.2. This method was introduced in [6]; we generalize it to complex sorts.

The first stage can be figuratively called "type-revealing". Even if our type-protecting and type-fusing transformations are not limited to sort constants and can protect arbitrarily complex sorts, say, arrays of integers, we cannot readily benefit from this capacity. In an initial $\mathbf{FOL}_{\mathrm{T}}$-problem, arrays are most probably formalized as a polymorphic type, with premises that apply to arrays of any type and with polymorphic function symbols for access and update. In order to produce interpreted monomorphic operations for Z3, CVC3, or Yices in the end, we must start by replacing, wherever possible, these function symbols with their monomorphic specializations. This is the purpose of the Dis transformation, presented in Section 3.1. We show in Section 4 that this "type revealing" brings a considerable improvement to provers' results.

To fit the page limit, we omit the proofs of our theorems. The reader is referred to the extended technical report [5].

3.1 Symbol Discrimination

The Dis transformation involves producing a sufficient number of type instances of formulas in an initial problem Γ with subsequent discrimination of function and predicate symbols.

Let f be a function symbol of type signature $\mathbf{S}$ in the initial problem Γ. Let τ be a type substitution in the type variables of $\mathbf{S}$. A fresh function symbol f_τ with the type signature $\mathbf{S}\tau$ is called a *specialization* of f. We call f_τ a *monomorphic specialization* if $\mathbf{S}\tau$ is monomorphic. Specializations of predicate symbols are defined in the same way.

Let W be a set of monomorphic specializations of function and predicate symbols in Γ. These are the instances that we want to "reveal" in the problem. The set W is fixed for the rest of this section; the Dis transformation is implicitly parametrized by it.

First of all, the Dis transformation modifies the signature of Γ:

1. For every variable symbol u and type T, we add a new variable symbol u_T.
2. We add every function and predicate symbol from W.

Given an arbitrary type substitution θ, the discriminating transformation Dis_θ instantiates and converts terms and formulas into the new signature:

1. Given a variable $u:T$, $\text{Dis}_\theta(u:T) \triangleq u_T:T\theta$.
2. Consider a term $t = f(t_1:T_1,\ldots,t_n:T_n):T$. Let τ be the type substitution that instantiates the type signature of the symbol f to $\langle T_1\theta,\ldots,T_n\theta,T\theta\rangle$. If f_τ belongs to W, then $\text{Dis}_\theta(t) \triangleq f_\tau(\text{Dis}_\theta(t_1):T_1\theta,\ldots,\text{Dis}_\theta(t_n):T_n\theta):T\theta$. Otherwise, $\text{Dis}_\theta(t) \triangleq f(\text{Dis}_\theta(t_1):T_1\theta,\ldots,\text{Dis}_\theta(t_n):T_n\theta):T\theta$.
3. Consider an atomic formula $F = p(t_1:T_1,\ldots,t_n:T_n)$. Let τ be the type substitution that instantiates the type signature of p to $\langle T_1\theta,\ldots,T_n\theta\rangle$. If p_τ is in W, then $\text{Dis}_\theta(F) \triangleq p_\tau(\text{Dis}_\theta(t_1):T_1\theta,\ldots,\text{Dis}_\theta(t_n):T_n\theta)$. Otherwise, $\text{Dis}_\theta(F) \triangleq p(\text{Dis}_\theta(t_1):T_1\theta,\ldots,\text{Dis}_\theta(t_n):T_n\theta)$.

Equalities and complex formulas are converted in a natural way:

$$\text{Dis}_\theta(t_1 \approx t_2) \triangleq \text{Dis}_\theta(t_1) \approx \text{Dis}_\theta(t_2) \qquad \text{Dis}_\theta(F \wedge G) \triangleq \text{Dis}_\theta(F) \wedge \text{Dis}_\theta(G)$$
$$\text{Dis}_\theta(\neg F) \triangleq \neg\,\text{Dis}_\theta(F) \qquad \text{Dis}_\theta(\forall x\,F) \triangleq \forall(\text{Dis}_\theta(x))\,\text{Dis}_\theta(F)$$

Now, let F be a closed formula. The set of monomorphic type substitutions $\Theta(F)$ is defined as follows:

$$\begin{aligned}\Theta(F) \triangleq\ & \{\,\theta \mid F \text{ contains a term } f(t_1:T_1,\ldots,t_n:T_n):T \text{ such that}\\ & \quad \theta \text{ only instantiates the variables of } \mathbf{T} = \langle T_1,\ldots,T_n,T\rangle \text{ and}\\ & \quad W \text{ contains a specialization of } f \text{ with the type signature } \mathbf{T}\theta\,\}\\ \cup\ & \{\,\theta \mid F \text{ contains an atomic formula } p(t_1:T_1,\ldots,t_n:T_n) \text{ such that}\\ & \quad \theta \text{ only instantiates the variables of } \mathbf{T} = \langle T_1,\ldots,T_n\rangle \text{ and}\\ & \quad W \text{ contains a specialization of } p \text{ with the type signature } \mathbf{T}\theta\,\}\end{aligned}$$

We call two monomorphic type substitutions *compatible* if they do not substitute two different sorts for the same type variable. The *union* of two compatible monomorphic type substitutions is their composition (the order is irrelevant). We define $\Theta^\star(F)$ as the closure of $\Theta(F)$ with respect to finite unions of compatible type substitutions. The empty union, i.e., the identity type substitution, also belongs to $\Theta^\star(F)$.

Finally, Dis translates a closed formula F into a set of formulas:

$$\textsc{Dis}(F) \triangleq \{\, \textsc{Dis}_\theta(F) \mid \theta \in \Theta^\star(F) \,\}$$

On our running example, assuming $W = \{\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}, \mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}\}$, Dis produces the following two formulas:

$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\alpha,\mathsf{I}))\, \forall(c_\alpha : \alpha)\, \mathtt{get}(\mathtt{set}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, 6), c_\alpha) * 7 \approx 42$$
$$\forall(m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{M}(\mathsf{I},\mathsf{I}))\, \forall(c_\alpha : \mathsf{I})\, \mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(\mathtt{set}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}(m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, 6), c_\alpha) * 7 \approx 42$$

The new symbol $\mathtt{get}_{[\mathsf{I}/\alpha,\mathsf{I}/\beta]}$ has the monomorphic type signature $\langle \mathsf{M}(\mathsf{I},\mathsf{I}), \mathsf{I}, \mathsf{I}\rangle$.

Lemma 2. *Let θ be a type substitution, t a term of type T, and F a formula. Then $\textsc{Dis}_\theta(t)$ is a well-formed term of type $T\theta$ and $\textsc{Dis}_\theta(F)$ is a well-formed formula such that $\mathrm{FV}(\textsc{Dis}_\theta(F)) = \{\textsc{Dis}_\theta(x) \mid x \in \mathrm{FV}(F)\}$ and $\mathrm{FV}_\mathrm{T}(\textsc{Dis}_\theta(F)) = \mathrm{FV}_\mathrm{T}(F) \backslash \mathrm{dom}(\theta)$.*

Theorem 2. *A set of closed formulas Γ is equisatisfiable to $\textsc{Dis}(\Gamma)$.*

The definition of Dis can be generalized to a case where W admits polymorphic specializations. This requires W to be closed with respect to unification of type signatures, so that we can always choose the most refined specialization symbol during discrimination. The substitutions in the set $\Theta(F)$ must be considered modulo renaming of type variables in the signatures of specialization symbols. Finally, the union of two substitutions would be their most general common refinement. However, since our transformations target monomorphic theorem provers, we find this generalization of lesser practical interest and do not pursue it in this paper.

3.2 Twin Sorts

The Tw transformation converts a set of formulas into an equisatisfiable set with protected sorts. It applies a pair of conversion functions to pass, wherever necessary, from a protected sort to a non-protected one and vice versa.

Let U be a set of sorts that we want to preserve across our type-eliminating transformations. The set U is fixed for the rest of the section and the Tw transformation is parametrized by it. Given a type T, the transformed type $[T]$ is $\overline{T}$ if $T \in U$, and T otherwise. Then Tw modifies the signature of a transformed theory as follows:

1. We replace every function symbol f of type signature $\langle S_1, \ldots, S_n, S\rangle$ with a symbol $\bar{f}$ of type signature $\langle [S_1], \ldots, [S_n], [S]\rangle$.
2. We replace every predicate symbol p of type signature $\langle S_1, \ldots, S_n\rangle$ with a symbol $\bar{p}$ of type signature $\langle [S_1], \ldots, [S_n]\rangle$.
3. For every sort $T \in U$, we add a pair of "bridge" function symbols $\mathtt{to}_T : \langle \overline{T}, T\rangle$ and $\mathtt{from}_T : \langle T, \overline{T}\rangle$.

Then we convert terms and atomic formulas into the new signature. Our aim is to forbid a polymorphic type in a symbol's type signature being instantiated into a type from U. Whenever such instantiation takes place, a bridge function is applied. In more precise terms:

1. Given a variable $u : T$, $\textsc{Tw}(u : T) \triangleq u : [T]$.
2. Consider a term $t = f(t_1 : T_1, \ldots, t_n : T_n) : T$ and let $\langle S_1, \ldots, S_n, S \rangle$ be the type signature of f.

$$\text{For every } t_i,\ t'_i \triangleq \begin{cases} \mathtt{to}_{T_i}(\textsc{Tw}(t_i)) : T_i & \text{if } T_i \in U \text{ and } S_i \notin U, \\ \textsc{Tw}(t_i) & \text{if } T_i \notin U \text{ or } S_i \in U. \end{cases}$$

$$\text{Then } \textsc{Tw}(t) \triangleq \begin{cases} \bar{f}(t'_1, \ldots, t'_n) : [T] & \text{if } T \notin U \text{ or } S \in U, \\ \mathtt{from}_T(\bar{f}(t'_1, \ldots, t'_n) : T) : \overline{T} & \text{if } T \in U \text{ and } S \notin U. \end{cases}$$

3. Consider a formula $p(t_1 : T_1, \ldots, t_n : T_n)$ and let $\langle S_1, \ldots, S_n \rangle$ be the type signature of p. For every argument t_i, we define t'_i as in the previous case. Then, $\textsc{Tw}(p(t_1 : T_1, \ldots, t_n : T_n)) \triangleq \bar{p}(t'_1, \ldots, t'_n)$.

Equalities and complex formulas are converted in a natural way:

$$\textsc{Tw}(t_1 \approx t_2) \triangleq \textsc{Tw}(t_1) \approx \textsc{Tw}(t_2) \qquad \textsc{Tw}(F \wedge G) \triangleq \textsc{Tw}(F) \wedge \textsc{Tw}(G)$$
$$\textsc{Tw}(\neg F) \triangleq \neg\, \textsc{Tw}(F) \qquad \textsc{Tw}(\forall x\, F) \triangleq \forall (\textsc{Tw}(x))\, \textsc{Tw}(F)$$

Finally, we convert the formulas in Γ and add axioms for the bridge functions:

$$\begin{aligned} \textsc{Tw}(\Gamma) \triangleq & \{\, \textsc{Tw}(F) \mid F \in \Gamma \,\} \\ & \cup \{\, \forall (v : \overline{T})\ \mathtt{from}_T(\mathtt{to}_T(v : \overline{T})) \approx v : \overline{T} \mid T \in U \,\} \\ & \cup \{\, \forall (u : T)\ \mathtt{to}_T(\mathtt{from}_T(u : T)) \approx u : T \mid T \in U \,\} \end{aligned}$$

Assuming $U = \{\mathsf{I}\}$, the running example is transformed as follows. Notice that $\mathtt{6}$, $\mathtt{7}$, and $\mathtt{42}$ have the type $\bar{\mathsf{I}}$ and the type signature of $*$ is $\langle \bar{\mathsf{I}}, \bar{\mathsf{I}}, \bar{\mathsf{I}} \rangle$.

$$\forall (m : \mathsf{M}(\alpha, \mathsf{I}))\, \forall (c : \alpha)\, \mathtt{from}_{\mathsf{I}}(\mathtt{get}(\mathtt{set}(m, c, \mathtt{to}_{\mathsf{I}}(\mathtt{6}) : \mathsf{I}), c) : \mathsf{I}) * \mathtt{7} \approx \mathtt{42}$$

Lemma 3. *For every term t of type T, $\textsc{Tw}(t)$ is a well-formed term of type $[T]$, and for every formula F, $\textsc{Tw}(F)$ is a well-formed formula with the same free variables (modulo conversion of their types) and type variables.*

Theorem 3. *A set of closed formulas Γ is equisatisfiable to $\textsc{Tw}(\Gamma)$.*

3.3 Decorated Terms

The Dec transformation converts a polymorphic problem with protected sorts into an equisatisfiable monomorphic problem. Roughly speaking, in order to preserve type information, it decorates every term with its type, which itself is transformed to a term of a special sort.

First of all, we introduce three fresh sort constants U, D, and T. The first one is assigned to undecorated terms, the second one to decorated terms, and the third one to the terms representing types. To transform the type signatures of function and predicate symbols, we use the following operations on types:

$$[T]^- \triangleq \begin{cases} T & \text{if } T \text{ is protected,} \\ \mathsf{D} & \text{otherwise} \end{cases} \qquad [T]^+ \triangleq \begin{cases} T & \text{if } T \text{ is protected,} \\ \mathsf{U} & \text{otherwise} \end{cases}$$

Now, the signature of the resulting theory is defined as follows:

1. The set of type constructors is extended with U, D, T.
2. We replace every function symbol f of type signature $\langle S_1, \ldots, S_n, S\rangle$ with a symbol $\hat{f}$ with the monomorphic type signature $\langle [S_1]^-, \ldots, [S_n]^-, [S]^+\rangle$.
3. We replace every predicate symbol p of type signature $\langle S_1, \ldots, S_n\rangle$ with a symbol $\hat{p}$ with the monomorphic type signature $\langle [S_1]^-, \ldots, [S_n]^-\rangle$.
4. For every variable symbol u and type T, we add a new variable symbol u_T.
5. For every type variable $\alpha \in \mathbb{V}_\mathrm{T}$, we add a new variable symbol v^α.
6. For every type constructor $\mathsf{F} \in \mathbb{F}_\mathrm{T}$, we add a new function symbol $\mathtt{F}$ of the same arity and with type signature $\langle \mathsf{T}, \ldots, \mathsf{T}, \mathsf{T}\rangle$.
7. We add a new "decoration" function symbol $\mathtt{deco} : \langle \mathsf{T}, \mathsf{U}, \mathsf{D}\rangle$.

The Dec transformation applies to non-protected types, translating them to terms of type T:

$$\textsc{Dec}(\alpha) \triangleq v^\alpha : \mathsf{T} \qquad \textsc{Dec}(\mathsf{F}(T_1, \ldots, T_n)) \triangleq \mathtt{F}(\textsc{Dec}(T_1), \ldots, \textsc{Dec}(T_n)) : \mathsf{T}$$

In the next definition, $\mathbf{t}$ stands for a vector of terms, $\bar{S}$ for a protected sort, and T for a non-protected type. The Dec transformation applies to terms:

$$\begin{aligned} \textsc{Dec}(u : \bar{S}) &\triangleq u_{\overline{S}} : \bar{S} \\ \textsc{Dec}(u : T) &\triangleq \mathtt{deco}(\textsc{Dec}(T), u_T : \mathsf{U}) : \mathsf{D} \\ \textsc{Dec}(f(\mathbf{t}) : \bar{S}) &\triangleq \hat{f}(\textsc{Dec}(\mathbf{t})) : \bar{S} \\ \textsc{Dec}(f(\mathbf{t}) : T) &\triangleq \mathtt{deco}(\textsc{Dec}(T), \hat{f}(\textsc{Dec}(\mathbf{t})) : \mathsf{U}) : \mathsf{D} \end{aligned}$$

and formulas (here, $\{\alpha_1, \ldots, \alpha_m\} = \mathrm{FV}_\mathrm{T}(H)$):

$$\begin{aligned} \textsc{Dec}(p(\mathbf{t})) &\triangleq \hat{p}(\textsc{Dec}(\mathbf{t})) & \textsc{Dec}(\neg F) &\triangleq \neg\, \textsc{Dec}(F) \\ \textsc{Dec}(t_1 \approx t_2) &\triangleq \textsc{Dec}(t_1) \approx \textsc{Dec}(t_2) & \textsc{Dec}(\forall (u : \bar{S}) F) &\triangleq \forall (u_{\overline{S}} : \bar{S})\, \textsc{Dec}(F) \\ \textsc{Dec}(F \wedge G) &\triangleq \textsc{Dec}(F) \wedge \textsc{Dec}(G) & \textsc{Dec}(\forall (u : T) F) &\triangleq \forall (u_T : \mathsf{U})\, \textsc{Dec}(F) \end{aligned}$$

$$\textsc{Dec}^\circ(H) \triangleq \forall (v^{\alpha_1} : \mathsf{T}) \ldots \forall (v^{\alpha_m} : \mathsf{T})\, \textsc{Dec}(H)$$

On the running example, assuming $U = \{\mathsf{I}\}$, the transformations Tw and $\textsc{Dec}^\circ$ produce the following monomorphic formula:

$$\begin{aligned} \forall (v^\alpha : \mathsf{T})\, \forall (m_{\mathsf{M}(\alpha,\mathsf{I})} : \mathsf{U})\, \forall (c_\alpha : \mathsf{U})\, \mathtt{from}_\mathsf{I}\big(\mathtt{deco}\big(\mathtt{I}, \mathtt{get}(\mathtt{deco}(\mathtt{M}(v^\alpha, \mathtt{I}), \\ \mathtt{set}(\mathtt{deco}(\mathtt{M}(v^\alpha, \mathtt{I}), m_{\mathsf{M}(\alpha,\mathsf{I})}), \mathtt{deco}(v^\alpha, c_\alpha), \mathtt{deco}(\mathtt{I}, \mathtt{to}_\mathsf{I}(6)))), \\ \mathtt{deco}(v^\alpha, c_\alpha))\big)\big) * 7 \approx 42 \end{aligned}$$

The second axiom of bridge functions $\mathtt{to_I}$ and $\mathtt{from_I}$ becomes

$$\forall(u_\mathsf{I} : \mathsf{U})\, \mathtt{deco}(\mathtt{I}, \mathtt{to_I}(\mathtt{from_I}(\mathtt{deco}(\mathtt{I}, u_\mathsf{I})))) \approx \mathtt{deco}(\mathtt{I}, u_\mathsf{I})$$

Due to the outer application of `deco` on the both sides of equality, our translation is sound even when we protect finite types, such as booleans. Without this additional decoration (as in [6, Eq. (8)]), the finiteness of a protected sort implies the finiteness of the whole sort U.

Lemma 4. *For every term t of type T, $\textsc{Dec}(t)$ is a well-formed monomorphic term of type $[T]^-$. For every formula F, $\textsc{Dec}(F)$ is a well-formed monomorphic formula. Also, $\mathrm{FV}(\textsc{Dec}(t)) = \{u_T : [T]^+ \mid u : T \in \mathrm{FV}(t)\} \cup \{v^\alpha : \mathsf{T} \mid \alpha \in \mathrm{FV_T}(t)\}$ and $\mathrm{FV}(\textsc{Dec}(F)) = \{u_T : [T]^+ \mid u : T \in \mathrm{FV}(F)\} \cup \{v^\alpha : \mathsf{T} \mid \alpha \in \mathrm{FV_T}(F)\}$. For every closed formula F, $\textsc{Dec}^\circ(F)$ is a well-formed closed monomorphic formula.*

Theorem 4. *A set of closed formulas with protected sorts Γ is satisfiable if and only if $\textsc{Dec}^\circ(\Gamma)$ is satisfiable.*

3.4 Explicit Polymorphism

The Exp transformation is similar to Dec except that instead of attaching an explicit type annotation to every term, we add type-representing arguments to polymorphic symbols. This allows for much lighter modifications in the original problem. However, the method is only sound on problems that admit a model where every non-protected sort has an infinite domain.

We introduce fresh sort constants U and T. The first one replaces non-protected types and the second one, as in Dec, is the sort of type-representing terms. For any type T, we define $[T]$ to be T if T is protected, and U otherwise. Then Exp modifies the signature of a transformed theory in the following way:

1. The set of type constructors is extended with U and T.
2. Let f be a function symbol of signature $\mathbf{S} = \langle S_1, \ldots, S_n, S\rangle$ and $\alpha_1, \ldots, \alpha_r$ be the free type variables of $\mathbf{S}$. We replace f with a function symbol $\hat{f}$ of arity $r + n$ with monomorphic type signature $\langle \mathsf{T}, \ldots, \mathsf{T}, [S_1], \ldots, [S_n], [S]\rangle$.
3. Let p be a predicate symbol of signature $\mathbf{S} = \langle S_1, \ldots, S_n\rangle$ and $\alpha_1, \ldots, \alpha_r$ be the free type variables of $\mathbf{S}$. We replace p with a predicate symbol $\hat{p}$ of arity $r + n$ with monomorphic type signature $\langle \mathsf{T}, \ldots, \mathsf{T}, [S_1], \ldots, [S_n]\rangle$.
4. For every variable symbol u and type T, we add a new variable symbol u_T.
5. For every type variable $\alpha \in \mathbb{V}_\mathrm{T}$, we add a new variable symbol v^α.
6. For every type constructor $\mathsf{F} \in \mathbb{F}_\mathrm{T}$, we add a new function symbol $\mathtt{F}$ of the same arity and with type signature $\langle \mathsf{T}, \ldots, \mathsf{T}, \mathsf{T}\rangle$.

The Exp transformation applies to non-protected types, translating them to terms of type T, exactly as Dec:

$$\textsc{Exp}(\alpha) \triangleq v^\alpha : \mathsf{T} \qquad \textsc{Exp}(\mathsf{F}(T_1, \ldots, T_n)) \triangleq \mathtt{F}(\textsc{Exp}(T_1), \ldots, \textsc{Exp}(T_n)) : \mathsf{T}$$

The Exp transformation applies to terms and formulas. In the definition below, $\mathbf{t}$ stands for a list of terms; $\alpha_1, \ldots, \alpha_r$ are the type variables of the type signature of f and p; the type signature of f and p is instantiated with a type substitution τ; and $\beta_1, \ldots, \beta_m$ are the type variables of H:

$$\begin{aligned}
\textsc{Exp}(u\!:\!T) &\triangleq u_T\!:\![T] \\
\textsc{Exp}(f(\mathbf{t})\!:\!T) &\triangleq \hat{f}(\textsc{Exp}(\alpha_1\tau), \ldots, \textsc{Exp}(\alpha_r\tau), \textsc{Exp}(\mathbf{t}))\!:\![T] \\
\textsc{Exp}(p(\mathbf{t})) &\triangleq \hat{p}(\textsc{Exp}(\alpha_1\tau), \ldots, \textsc{Exp}(\alpha_r\tau), \textsc{Exp}(\mathbf{t})) \\
\textsc{Exp}(t_1 \approx t_2) &\triangleq \textsc{Exp}(t_1) \approx \textsc{Exp}(t_2) \\
\textsc{Exp}(\neg F) &\triangleq \neg\, \textsc{Exp}(F) \\
\textsc{Exp}(F \wedge G) &\triangleq \textsc{Exp}(F) \wedge \textsc{Exp}(G) \\
\textsc{Exp}(\forall x\, F) &\triangleq \forall(\textsc{Exp}(x))\, \textsc{Exp}(F) \\
\textsc{Exp}^\circ(H) &\triangleq \forall(v^{\beta_1}\!:\!\mathsf{T}) \ldots \forall(v^{\beta_m}\!:\!\mathsf{T})\, \textsc{Exp}(H)
\end{aligned}$$

On the running example, assuming $U = \{\mathsf{I}\}$, the transformations Tw and $\textsc{Exp}^\circ$ produce the following formula:

$$\forall(v^\alpha\!:\!\mathsf{T})\, \forall(m_{\mathsf{M}(\alpha,\mathsf{I})}\!:\!\mathsf{U})\, \forall(c_\alpha\!:\!\mathsf{U})\, \mathtt{from}_\mathsf{I}(\mathtt{get}(v^\alpha, \mathtt{I}, \mathtt{set}(v^\alpha, \mathtt{I}, m_{\mathsf{M}(\alpha,\mathsf{I})}, c_\alpha, \mathtt{to}_\mathsf{I}(\mathsf{6})), c_\alpha)) * \mathsf{7} \approx \mathsf{42}$$

Lemma 5. *For every term t of type T, $\textsc{Exp}(t)$ is a well-formed monomorphic term of type $[T]$. For every formula F, $\textsc{Exp}(F)$ is a well-formed monomorphic formula. Also, $\mathrm{FV}(\textsc{Exp}(t)) = \{u_T\!:\![T] \,|\, u\!:\!T \in \mathrm{FV}(t)\} \cup \{v^\alpha\!:\!\mathsf{T} \,|\, \alpha \in \mathrm{FV}_\mathrm{T}(t)\}$ and $\mathrm{FV}(\textsc{Exp}(F)) = \{u_T\!:\![T] \,|\, u\!:\!T \in \mathrm{FV}(F)\} \cup \{v^\alpha\!:\!\mathsf{T} \,|\, \alpha \in \mathrm{FV}_\mathrm{T}(F)\}$. Finally, for every closed formula F, $\textsc{Exp}^\circ(F)$ is a well-formed closed monomorphic formula.*

Theorem 5. *Let Γ be a set of closed formulas with protected sorts. If Γ is satisfiable so that every non-protected sort has an infinite domain in the model, then $\textsc{Exp}^\circ(\Gamma)$ is satisfiable. Conversely, if $\textsc{Exp}^\circ(\Gamma)$ is satisfiable then Γ is satisfiable.*

From a practical point of view, the soundness part of Theorem 5 is not comforting. Given a $\mathbf{FOL}_\mathrm{T}$-problem Γ, we cannot effectively decide which sorts admit infinite models and which do not (one can postulate a bijection between a given sort and the domain of a partial-recursive function). A practical way out could consist in a small language extension: for every type/sort, we specify explicitly whether it is finite or infinite. We proceed from the assumption that the author of any given problem knows the intended model of every type.

In Why3 [4], every type is declared either as abstract or algebraic (i.e., a sum of products). We postulate that abstract types are all infinite and we analyse the definitions of algebraic types to find out which of their monomorphic instances admit infinite models. For example, given the standard algebraic definitions of booleans (B), lists ($\mathsf{L}(\alpha)$), and pairs ($\mathsf{P}(\alpha, \beta)$), we can conclude that the sorts B and $\mathsf{P}(\mathsf{B}, \mathsf{B})$ are finite and I, $\mathsf{L}(\mathsf{B})$, and $\mathsf{P}(\mathsf{I}, \mathsf{B})$ are infinite.

Once we know the finite sorts, can we transform a problem to eliminate them, so that Exp (or a similar method) can be applied? Meng and Paulson propose

to filter out the premises implying the finiteness of sorts [14, Sect. 2.8]; however, we need an infallible filter to ensure the soundness of type erasure. We have implemented an alternative solution which consists in putting a special "projection" function $\mathtt{proj}_T$ over every variable and function symbol of a finite type T. Thus, the premise $\forall(x:\mathsf{B})(x \approx \mathtt{True} \vee x \approx \mathtt{False})$ becomes $\forall(x:\mathsf{B})(\mathtt{proj}_{\mathsf{B}}(x) \approx \mathtt{proj}_{\mathsf{B}}(\mathtt{True}) \vee \mathtt{proj}_{\mathsf{B}}(x) \approx \mathtt{proj}_{\mathsf{B}}(\mathtt{False}))$ and the domain of B does not need to be finite anymore, as we confine ourselves to the range of $\mathtt{proj}_{\mathsf{B}}$. This method is still potentially unsound, as one can state the finiteness of a sort with a polymorphic axiom, where no projection would apply. Precisely, let $\mathtt{isUnit}:\langle\alpha\rangle$ be a unary predicate. Then the formulas $\forall(x:\alpha)(\mathtt{isUnit}(x) \supset \forall(y:\alpha)(y \approx x))$ and $\forall(x:\mathsf{A})\,\mathtt{isUnit}(x)$ imply that the sort A has a single inhabitant. Today, we know of no way to use Exp soundly on polymorphic problems with finite sorts.

The last remark to make is that Exp provides a path towards one-sorted languages. Indeed, in a monomorphic setting, Theorem 5 comes to: "if every sort admits an infinite domain, then we can safely erase the sort annotations". Thus, if we want to use a TPTP prover such as Vampire or SPASS, we start by translating a proof task to the many-sorted language, using any of the methods described above. Then we eliminate the protected finite sorts (if any) using projections; in absence of polymorphism, this is a sound and complete transformation. And finally, we apply Exp assuming that all types are non-protected, which amounts to simply erasing all sorts.

4 Experiments and Conclusion

In our experiments, we wanted to compare the impact of different "paths" of polymorphism encoding on the performance of three well-established SMT solvers. We add the classical type encoding technique with per-variable "type guards" [12]. Our implementation of this method (denoted Grd below) closely follows the description given in [11, Sect. 3.0].

We run our tests on 4123 verification conditions generated by the Why platform from 166 programs, which originate from Caduceus [8], Jessie [13], or directly from Why. Translated tasks were sent to Z3, CVC3, and Yices with a time limit of 60 seconds. On the whole, 3993 proof obligations were proved by at least one prover. The initial Why3 files and our results are available at `http://why3.lri.fr/download/polyfol_encoding.tar.gz`.

We have tested the encodings Tw+Dec, Tw+Exp, and Tw+Grd, both with and without Dis. In the latter case, these methods correspond to what is described in [6] and [11]; the set U of sorts to protect in Tw is set to contain only integers and reals, which are natively supported by the three provers. In presence of Dis, we put in the set W every monomorphic specialization that occurs in the goal formula along with the specializations of access and update operations on every monomorphic array type in the goal; we also protect every sort in the goal (as well as integers and reals). This configuration of Dis and Tw gives better results comparing to other configurations that we tried, e.g., collect the specializations and the sorts to protect from the whole proof task.

Z3 (3809)	Tw+Grd		Tw+Exp		Tw+Dec		Dis+Tw+Grd		Dis+Tw+Exp	
Dis+Tw+Dec	+203	−36	+20	−49	+66	−37	+18	−5	+26	−30
Dis+Tw+Exp	+191	−20	+13	−38	+63	−30	+35	−18		
Dis+Tw+Grd	+195	−41	+11	−53	+59	−43				
Tw+Dec	+157	−19	+15	−73						
Tw+Exp	+211	−15								

CVC3 (3756)	Tw+Grd		Tw+Exp		Tw+Dec		Dis+Tw+Grd		Dis+Tw+Exp	
Dis+Tw+Dec	+269	−20	+0	−26	+84	−19	+66	−4	+0	−6
Dis+Tw+Exp	+272	−17	+0	−20	+88	−17	+69	−1		
Dis+Tw+Grd	+204	−17	+1	−89	+46	−43				
Tw+Dec	+188	−4	+0	−91						
Tw+Exp	+275	−0								

Yices (3717)	Tw+Grd		Tw+Exp		Tw+Dec		Dis+Tw+Grd		Dis+Tw+Exp	
Dis+Tw+Dec	+882	−6	+13	−276	+379	−79	+204	−2	+3	−272
Dis+Tw+Exp	+1149	−4	+39	−33	+574	−5	+472	−1		
Dis+Tw+Grd	+684	−10	+6	−471	+241	−143				
Tw+Dec	+577	−1	+5	−568						
Tw+Exp	+1140	−1								

Our results are given in the table above. To the right of the prover's name, we put the number of goals proved by at least one encoding method. In every cell we specify the number of goals proved by one encoding but not by the other one. For example, with CVC3, the encoding by Dis+Tw+Dec allows us to prove 84 goals that were not proved by Tw+Dec. On the other hand, with Tw+Dec, CVC3 proves 19 goals that were not proved with Dis+Tw+Dec.

On the average, symbol discrimination increases the number of premises by a factor of 1.8 (ranging from 1 to 10 on some examples). Nevertheless, adding the Dis phase allows us to prove more goals in every case except for Z3 and CVC3 with Exp. In particular, the Grd transformation is remarkably helped by Dis. Apart from the possibility to use the built-in support for arrays, the effectiveness of Dis is also explained by the fact that we protect the sorts that occur in the selected monomorphic specializations. Thus, the new premises generated by Dis are not only instantiated to the relevant sorts, they are also liberated from decorations imposed by the third, type-fusing, stage. This effect is less important in the case of Exp, because this transformation, unlike Dec and Grd, adds very little clutter to the encoded formulas in the first place.

Also notice that type protection, Tw, is crucially important: if we protect no types at all, we prevent provers from using their built-in theories, and the total number of goals proved (using only Exp, Dec, or Grd) drops to 1861.

The comparison between Exp, Dec, and Grd shows that Exp is generally more efficient than Dec which in its turn is more efficient than Grd. This is quite different from the results given in [11], where Exp and Grd have roughly the same performance. We have not yet identified whether this discrepancy comes from the difference in our test cases or in our implementations.

Conclusion. In the present paper, we described first-order logic with polymorphic types and introduced generic notions to define and reason about practical methods of polymorphism elimination. Using these notions, we generalized and proved two translation techniques known from literature. We also proposed to combine type protection with symbol discrimination. As our experiments show, this improves the performance of automated proof search and allows us to use built-in theories of complex types, such as arrays, in SMT solvers. One interesting problem we would like to resolve in the future is protection of polymorphic types, allowing to merge all monomorphic instances of a given complex type in a single protected sort. We also believe that better heuristics to choose the sets W and U can be devised, and further experiments are in order.

References

1. Barrett, C., Stump, A., Tinelli, C.: The SMT-LIB Standard: Version 2.0. Tech. rep., Department of Computer Science, The University of Iowa (2010)
2. Barrett, C., Tinelli, C.: CVC3. In: CAV'07. LNCS, vol. 4590, pp. 298–302. Springer (2007)
3. Bobot, F., Conchon, S., Contejean, E., Lescuyer, S.: Implementing polymorphism in SMT solvers. In: SMT'08. ACM ICPS, vol. 367, pp. 1–5 (2008)
4. Bobot, F., Filliâtre, J.C., Marché, C., Paskevich, A.: Why3: Shepherd your herd of provers. In: Boogie 2011 (co-loc. with CADE-23). Wrocław, Poland (August 2011)
5. Bobot, F., Paskevich, A.: Expressing polymorphic types in a many-sorted language (extended version) (Jul 2011), `http://hal.inria.fr/inria-00591414/en/`
6. Couchot, J.F., Lescuyer, S.: Handling polymorphism in automated deduction. In: CADE-21. LNAI, vol. 4603, pp. 263–278. Springer (2007)
7. Dutertre, B., de Moura, L.: The YICES SMT solver. Tech. rep., SRI International (2006)
8. Filliâtre, J.C., Marché, C.: Multi-prover verification of C programs. In: ICFEM'04. LNCS, vol. 3308, pp. 15–29. Springer (2004)
9. Hurd, J.: An LCF-style interface between HOL and first-order logic. In: CADE-18. LNAI, vol. 2392, pp. 134–138. Springer (2002)
10. Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: Design and Application of Strategies/Tactics in Higher Order Logics. pp. 56–68. NASA Technical Report NASA/CP-2003-212448 (2003)
11. Leino, K.R.M., Rümmer, P.: A polymorphic intermediate verification language: Design and logical encoding. In: TACAS'10. LNCS, vol. 6015, pp. 312–327. Springer (2010)
12. Manzano, M.: Extensions of First-Order Logic, Cambridge Tracts in Theoretical Computer Science, vol. 19. Cambridge University Press (1996)
13. Marché, C., Moy, Y.: Jessie plug-in (2010), `http://frama-c.com/jessie.html`
14. Meng, J., Paulson, L.C.: Translating higher-order clauses to first-order clauses. Journal of Automated Reasoning 40(1), 35–60 (2008)
15. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS'08. LNCS, vol. 4963, pp. 337–340. Springer (2008)
16. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
17. The Coq Development Team: The Coq Proof Assistant Reference Manual – Version V8.0 (2004), `http://coq.inria.fr`