Encoding Monomorphic and Polymorphic Types

Jasmin Christian Blanchette[1], Sascha Böhme[1],
Andrei Popescu[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 logics, and existing translations from typed logics are bulky or unsound. Recent research proposes monotonicity as a means to remove some clutter. Here we pursue this approach systematically, analysing formally a variety of encodings that further improve on efficiency while retaining soundness and completeness. We extend the approach to rank-1 polymorphism and present alternative schemes that lighten the translation of polymorphic symbols based on the novel notion of "cover". The new encodings are implemented, and partly proved correct, in Isabelle/HOL. Our evaluation finds them vastly superior to previous schemes.

1 Introduction

Specification languages, proof assistants, and other theorem proving applications typically rely on polymorphic types, but state-of-the-art automatic provers support only untyped or monomorphic logics. The existing sound and complete translation schemes for polymorphic types, whether they revolve around functions (tags) or predicates (guards), produce clutter that severely hampers the proof search, and lighter approaches based on type arguments are unsound [13, 15]. As a result, application authors face a difficult choice between soundness and efficiency.

The fourth author, together with Claessen and Lillieström [10], designed a pair of 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 merged. The remaining types can be made monotonic by introducing protectors (tags or guards).

Example 1 (Monkey Village). Imagine a village of monkeys [10] where each monkey owns at least two bananas (b_1 and b_2):

$\forall M : \mathit{monkey}.\ \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))$
$\forall M : \mathit{monkey}.\ \mathsf{b}_1(M) \not\approx \mathsf{b}_2(M)$
$\forall M_1, M_2 : \mathit{monkey},\ B : \mathit{banana}.\ \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 \approx M_2$

The type *banana* is monotonic, whereas *monkey* is nonmonotonic because there can live at most $\lfloor b/2 \rfloor$ monkeys in a village with a finite supply of b bananas. Thanks to monotonicity, it is sound to omit all type information regarding *banana*s. The example can be encoded using a predicate $\mathsf{g}_{\mathit{monkey}}$ to guard against ill-typed *monkey* instantiations:

$\forall M.\ \mathsf{g}_{\mathit{monkey}}(M) \rightarrow \mathsf{owns}(M, \mathsf{b}_1(M)) \wedge \mathsf{owns}(M, \mathsf{b}_2(M))$
$\forall M.\ \mathsf{g}_{\mathit{monkey}}(M) \rightarrow \mathsf{b}_1(M) \not\approx \mathsf{b}_2(M)$
$\forall M_1, M_2, B.\ \mathsf{g}_{\mathit{monkey}}(M_1) \wedge \mathsf{g}_{\mathit{monkey}}(M_2) \wedge \mathsf{owns}(M_1, B) \wedge \mathsf{owns}(M_2, B) \rightarrow M_1 \approx M_2$

Monotonicity is not decidable, but it can often be inferred using suitable calculi. In this paper, we exploit this idea systematically, analysing a variety of encodings based on monotonicity: some are minor adaptations of existing ones, while others are novel encodings that further improve on the size of the translated formulas.

We also generalise the monotonicity approach to a rank-1 polymorphic logic, as embodied by TPTP TFF1 [3]. Unfortunately, the presence of a single equality literal $X^\alpha \approx t$ or $t \approx X^\alpha$, where X is a polymorphic variable of type α, will lead the analysis to classify all types as possibly nonmonotonic and force the use of protectors everywhere. We solve this issue through a novel scheme that reduces the clutter associated with nonmonotonic types, based on the observation that protectors are required only when translating the particular formulas that prevent a type from being inferred monotonic.

We first review four main traditional approaches (Sect. 2), which prepare the ground for the more advanced encodings. Next, we present known and novel monotonicity-based schemes that handle only ground types (Sect. 3); these are interesting in their own right and serve as stepping stones for the full-blown polymorphic encodings (Sect. 4). We also present alternative schemes that aim at reducing the clutter associated with polymorphic symbols, based on the novel notion of "cover" (Sect. 5). Proofs of correctness are included in a technical report [2].

A formalisation [4] of the results in the proof assistant Isabelle/HOL [14] is under way; it currently covers all the monomorphic encodings. The encodings have been implemented in Sledgehammer [13], which provides a bridge between Isabelle and automatic theorem provers. They were evaluated with E, iProver, SPASS, Vampire, and Z3 on a vast benchmark suite (Sect. 6).

2 Traditional Type Encodings

We assume that formulas are expressed in negation normal form (NNF), with negation applied to atoms, and that each variable is bound only once in a formula. Given a polymorphic signature $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ (with n-ary type constructors $\mathcal{K}$, function symbols $\mathcal{F}$, and predicate symbols $\mathcal{P}$, all three sets finite), symbols are declared as $\mathsf{s} : \forall \bar{\alpha}.\, \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$, where ς is either a type (for $\mathsf{s} \in \mathcal{F}$) or o (for $\mathsf{s} \in \mathcal{P}$). An application $\mathsf{s}\langle\bar{\tau}\rangle(\bar{t})$ of s requires $|\bar{\alpha}|$ type arguments in angle brackets and $|\bar{\sigma}|$ term arguments in parentheses. We often omit $\langle\bar{\tau}\rangle$ in examples. Σ is *monomorphic* if none of the symbols take type arguments and *untyped* if additionally $\mathcal{K} = \{\iota\}$, in which case we omit $\mathcal{K}$ and indicate arities by superscripts (s^n). A *problem* over Σ is a finite set of closed formulas over Σ.

The easiest way to translate a typed problem into an untyped logic is to erase all type information, omitting type arguments, type quantifiers, and types in term quantifiers.

Definition 2 (Full Erasure e). The *full type erasure* encoding e translates a polymorphic problem over Σ into an untyped problem over Σ', where the symbols in Σ' have the same term arities as in Σ (but without type arguments). It is defined as follows:

$$\begin{array}{lll} [\![\mathsf{f}\langle\bar{\sigma}\rangle(\bar{t})]\!]_\mathsf{e} = \mathsf{f}([\![\bar{t}]\!]_\mathsf{e}) & [\![\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]_\mathsf{e} = \mathsf{p}([\![\bar{t}]\!]_\mathsf{e}) & [\![\forall X\!:\!\sigma.\, \varphi]\!]_\mathsf{e} = \forall X.\, [\![\varphi]\!]_\mathsf{e} \\ [\![\forall \alpha.\, \varphi]\!]_\mathsf{e} = [\![\varphi]\!]_\mathsf{e} & [\![\neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]_\mathsf{e} = \neg\, \mathsf{p}([\![\bar{t}]\!]_\mathsf{e}) & [\![\exists X\!:\!\sigma.\, \varphi]\!]_\mathsf{e} = \exists X.\, [\![\varphi]\!]_\mathsf{e} \end{array}$$

Here and elsewhere, we omit the trivial cases where the function is simply applied to its subterms or subformulas, as in $[\![\varphi_1 \wedge \varphi_2]\!]_\mathsf{e} = [\![\varphi_1]\!]_\mathsf{e} \wedge [\![\varphi_2]\!]_\mathsf{e}$.

By way of composition, the e encoding lies at the heart of all the encodings presented in this paper. Given n encodings $x_1, \ldots, x_n$, we write $[\![\,]\!]_{x_1;\ldots;x_n}$ for $[\![\,]\!]_{x_n} \circ \cdots \circ [\![\,]\!]_{x_1}$.

Full type erasure is unsound in the presence of equality because equality can be used to encode cardinality constraints on domains. For example, $\forall U : \mathit{unit}.\ U \approx \mathsf{unity}$ forces the domain of *unit* to have only one element. Its erasure, $\forall U.\ U \approx \mathsf{unity}$, effectively restricts *all* types to one element. An additional issue is that erasure confuses distinct monomorphic instances of polymorphic symbols. The formula $\mathsf{q}\langle a\rangle(\mathsf{f}\langle a\rangle) \wedge \neg\, \mathsf{q}\langle b\rangle(\mathsf{f}\langle b\rangle)$ is satisfiable, but its type erasure $\mathsf{q}(\mathsf{f}) \wedge \neg\, \mathsf{q}(\mathsf{f})$ is unsatisfiable. A solution is to encode types as terms in the untyped logic: type variables α become term variables A, and type constructors k become function symbols k. A symbol with m type arguments is passed m additional term arguments. The example above is translated to $\mathsf{q}(\mathsf{a}, \mathsf{f}(\mathsf{a})) \wedge \neg\, \mathsf{q}(\mathsf{b}, \mathsf{f}(\mathsf{b}))$.

We call this encoding a. It coincides with e for monomorphic problems and is unsound. Nonetheless, it forms the basis of all our sound polymorphic encodings in a slightly generalised version, called a^x below. First, let us fix a distinguished type ϑ (for encoded types) and two symbols $\mathsf{t} : \forall\alpha.\ \alpha \to \alpha$ (for tags) and $\mathsf{g} : \forall\alpha.\ \alpha \to \mathsf{o}$ (for guards).

Definition 3 (Type Argument Filter). Given a signature $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$, a *type argument filter* x maps any $\mathsf{s} : \forall\alpha_1, \ldots, \alpha_m.\ \bar{\sigma} \to \varsigma$ to a subset $x_\mathsf{s} = \{i_1, \ldots, i_{m'}\} \subseteq \{1, \ldots, m\}$ of its type argument indices. Given a list $\bar{z}$ of length m, $x_\mathsf{s}(\bar{z})$ denotes the sublist $z_{i_1}, \ldots, z_{i_{m'}}$, where $i_1 < \cdots < i_{m'}$. Filters are implicitly extended to $\{1\}$ for $\mathsf{t}, \mathsf{g} \notin \mathcal{F} \uplus \mathcal{P}$.

Definition 4 (Generic Arguments a^x). Given a type argument filter x, the *generic type arguments* encoding a^x translates a polymorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}, \mathcal{P}')$, where the symbols in $\mathcal{F}'$, $\mathcal{P}'$ are the same as those in $\mathcal{F}$, $\mathcal{P}$. For each symbol $\mathsf{s} : \forall\bar{\alpha}.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$, the arity of s in Σ' is $|x_\mathsf{s}| + |\bar{\sigma}|$. The encoding is defined as $[\![\,]\!]_{\mathsf{a}^x;\mathsf{e}}$, where the nontrivial cases are

$$[\![\mathsf{f}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathsf{a}^x} = \mathsf{f}\langle\bar{\sigma}\rangle(\langle\!\langle x_\mathsf{f}(\bar{\sigma})\rangle\!\rangle, [\![\bar{t}]\!]_{\mathsf{a}^x}) \qquad [\![\forall\alpha.\ \varphi]\!]_{\mathsf{a}^x} = \forall\alpha.\ \forall\langle\!\langle\alpha\rangle\!\rangle : \vartheta.\ [\![\varphi]\!]_{\mathsf{a}^x}$$
$$[\![\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathsf{a}^x} = \mathsf{p}\langle\bar{\sigma}\rangle(\langle\!\langle x_\mathsf{p}(\bar{\sigma})\rangle\!\rangle, [\![\bar{t}]\!]_{\mathsf{a}^x}) \qquad [\![\neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathsf{a}^x} = \neg\, [\![\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathsf{a}^x}$$

The auxiliary function $\langle\!\langle\sigma\rangle\!\rangle$ returns the *term encoding* of a type over $\mathcal{K}$ as a term over $(\{\vartheta\}, \mathcal{K})$ of the distinguished type ϑ, following the simple scheme described above.

An intuitive approach to encode type information soundly is to wrap each term and subterm with its type using type tags. For polymorphic type systems, this scheme relies on a distinguished binary function $\mathsf{t}(\langle\!\langle\sigma\rangle\!\rangle, t)$ that "annotates" each term t with its type σ. The tags make most type arguments superfluous. This encoding is defined as a two-stage process: the first stage adds tags $\mathsf{t}\langle\sigma\rangle(t)$ while preserving the polymorphism; the second stage encodes t's type argument as well as any phantom type arguments.

Definition 5 (Phantom Type Argument). Let $\mathsf{s} : \forall\alpha_1, \ldots, \alpha_m.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. The ith type argument is a *phantom* if α_i does not occur in $\bar{\sigma}$ or ς. Given a list $\bar{z} \equiv z_1, \ldots, z_m$, $\mathsf{phan}_\mathsf{s}(\bar{z})$ denotes the sublist $z_{i_1}, \ldots, z_{i_{m'}}$ corresponding to the phantom type arguments.

Definition 6 (Traditional Tags t). The *traditional type tags* encoding t translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K} \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for a^phan (i.e. a^x with $x = \mathsf{phan}$). It is defined as $[\![\,]\!]_{\mathsf{t};\mathsf{a}^\mathsf{phan};\mathsf{e}}$, i.e. the composition of $[\![\,]\!]_\mathsf{t}$, $[\![\,]\!]_{\mathsf{a}^\mathsf{phan}}$, and $[\![\,]\!]_\mathsf{e}$, where

$$[\![\mathsf{f}\langle\sigma\rangle(\bar{t})]\!]_\mathsf{t} = \lfloor \mathsf{f}\langle\sigma\rangle([\![\bar{t}]\!]_\mathsf{t}) \rfloor \qquad [\![X]\!]_\mathsf{t} = \lfloor X \rfloor \qquad \text{with } \lfloor t^\sigma \rfloor = \mathsf{t}\langle\sigma\rangle(t)$$

Example 7 (Algebraic Lists). The following axioms induce a minimalistic first-order theory of algebraic lists that will serve as our main running example:

$$\forall\alpha.\, \forall X\!:\!\alpha,\, Xs\!:\!\mathit{list}(\alpha).\ \mathsf{nil} \not\approx \mathsf{cons}(X, Xs)$$
$$\forall\alpha.\, \forall X\!:\!\alpha,\, Xs\!:\!\mathit{list}(\alpha).\ \mathsf{hd}(\mathsf{cons}(X, Xs)) \approx X \wedge \mathsf{tl}(\mathsf{cons}(X, Xs)) \approx Xs$$

We conjecture that cons is injective. The conjecture's negation can be expressed employing an unknown but fixed Skolem type b:

$$\exists X, Y\!:\!b,\, Xs, Ys\!:\!\mathit{list}(b).\ \mathsf{cons}(X, Xs) \approx \mathsf{cons}(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Because the hd and tl equations force injectivity of cons in both arguments, the problem is unsatisfiable: the unnegated conjecture is a consequence of the axioms. The t encoding translates the problem into

$$\forall A, X, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{nil}) \not\approx \mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs)))$$
$$\forall A, X, Xs.\ \mathsf{t}(A, \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs))))) \approx \mathsf{t}(A, X) \wedge \mathsf{t}(\mathsf{list}(A), \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs))))) \approx \mathsf{t}(\mathsf{list}(A), Xs)$$
$$\exists X, Y, Xs, Ys.\ \mathsf{t}(\mathsf{list}(\mathsf{b}), \mathsf{cons}(\mathsf{t}(\mathsf{b}, X), \mathsf{t}(\mathsf{list}(\mathsf{b}), Xs))) \approx \mathsf{t}(\mathsf{list}(\mathsf{b}), \mathsf{cons}(\mathsf{t}(\mathsf{b}, Y), \mathsf{t}(\mathsf{list}(\mathsf{b}), Ys))) \wedge (\mathsf{t}(\mathsf{b}, X) \not\approx \mathsf{t}(\mathsf{b}, Y) \vee \mathsf{t}(\mathsf{list}(\mathsf{b}), Xs) \not\approx \mathsf{t}(\mathsf{list}(\mathsf{b}), Ys))$$

Type tags heavily burden the terms. An alternative is to introduce type guards, which are predicates that restrict the range of variables. They take the form of a distinguished predicate $\mathsf{g}(\langle\!\langle\sigma\rangle\!\rangle, t)$ that checks whether t has type σ. With the type tags encoding, only phantom type arguments needed to be encoded; here, we must encode any type arguments that cannot be read off the types of the term arguments. Thus, the type argument is encoded for $\mathsf{nil}\langle\alpha\rangle$ but omitted for $\mathsf{cons}\langle\alpha\rangle(X, Xs)$, $\mathsf{hd}\langle\alpha\rangle(Xs)$, and $\mathsf{tl}\langle\alpha\rangle(Xs)$.

Definition 8 (Inferable Type Argument). Let $\mathsf{s} : \forall\alpha_1,\ldots,\alpha_m.\, \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. A type argument is *inferable* if it occurs in some of the term arguments' types. Given a list $\bar{z} \equiv z_1,\ldots,z_m$, $\mathsf{inf}_\mathsf{s}(\bar{z})$ denotes the sublist $z_{i_1},\ldots,z_{i_{m'}}$ corresponding to the inferable type arguments, and $\mathsf{ninf}_\mathsf{s}(\bar{z})$ denotes the sublist for noninferable type arguments.

Definition 9 (Traditional Guards g). The *traditional type guards* encoding g translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}, \mathcal{P}' \uplus \{\mathsf{g}^2\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ninf}}$. It is defined as $[\![\ ]\!]_{\mathsf{g};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$, where

$$[\![\forall X\!:\!\sigma.\, \varphi]\!]_\mathsf{g} = \forall X\!:\!\sigma.\, \mathsf{g}\langle\sigma\rangle(X) \to [\![\varphi]\!]_\mathsf{g} \qquad [\![\exists X\!:\!\sigma.\, \varphi]\!]_\mathsf{g} = \exists X\!:\!\sigma.\, \mathsf{g}\langle\sigma\rangle(X) \wedge [\![\varphi]\!]_\mathsf{g}$$

The translation of a problem is given by $[\![\Phi]\!]_\mathsf{g} = \mathrm{Ax} \cup \bigcup_{\varphi\in\Phi} [\![\varphi]\!]_\mathsf{g}$, where Ax consists of the following *typing axioms*:

$$\forall\bar{\alpha}.\, \bar{X}\!:\!\bar{\sigma}.\ \big(\textstyle\bigwedge_j \mathsf{g}\langle\sigma_j\rangle(X_j)\big) \to \mathsf{g}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})) \qquad \text{for } \mathsf{f} : \forall\bar{\alpha}.\, \bar{\sigma} \to \sigma \in \mathcal{F}$$
$$\forall\alpha.\, \exists X\!:\!\alpha.\, \mathsf{g}\langle\alpha\rangle(X)$$

The last axiom witnesses inhabitation of every type. It is necessary for completeness.

Example 10. The g encoding translates the algebraic list problem of Example 7 into

$$\forall A.\ \mathsf{g}(\mathsf{list}(A), \mathsf{nil}(A))$$
$$\forall A, X, Xs.\ \mathsf{g}(A, X) \wedge \mathsf{g}(\mathsf{list}(A), Xs) \to \mathsf{g}(\mathsf{list}(A), \mathsf{cons}(X, Xs))$$
$$\forall A, Xs.\ \mathsf{g}(\mathsf{list}(A), Xs) \to \mathsf{g}(A, \mathsf{hd}(Xs))$$
$$\forall A, Xs.\ \mathsf{g}(\mathsf{list}(A), Xs) \to \mathsf{g}(\mathsf{list}(A), \mathsf{tl}(Xs))$$
$$\forall A.\ \exists X.\ \mathsf{g}(A, X)$$

$$\forall A, X, Xs.\ \mathsf{g}(A, X) \wedge \mathsf{g}(\mathsf{list}(A), Xs) \rightarrow \mathsf{nil}(A) \not\approx \mathsf{cons}(X, Xs)$$
$$\forall A, X, Xs.\ \mathsf{g}(A, X) \wedge \mathsf{g}(\mathsf{list}(A), Xs) \rightarrow \mathsf{hd}(\mathsf{cons}(X, Xs)) \approx X \wedge \mathsf{tl}(\mathsf{cons}(X, Xs)) \approx Xs$$
$$\exists X, Y, Xs, Ys.\ \mathsf{g}(\mathsf{b}, X) \wedge \mathsf{g}(\mathsf{b}, Y) \wedge \mathsf{g}(\mathsf{list}(\mathsf{b}), Xs) \wedge \mathsf{g}(\mathsf{list}(\mathsf{b}), Ys) \wedge \mathsf{cons}(X, Xs) \approx \mathsf{cons}(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Bibliographical Notes. The earliest descriptions of type tags and type guards we are aware of are due to Enderton [11] and Stickel [15]. Wick and McCune [18] compare type arguments, tags, and guards in a monomorphic setting. Type arguments are reminiscent of System F; they are described by Meng and Paulson [13], who also consider full type erasure and polymorphic type tags. Urban [17] extended the untyped TPTP FOF syntax with dependent types to accommodate Mizar.

The intermediate verification language and tool Boogie 2 [12] 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 handle interpreted types [7, 12]. One of the Boogie translations [12] uses SMT triggers to prevent ill-typed instantiations. Bouillaguet et al. [8] showed that full type erasure is sound if all types can be assumed to have the same cardinality and exploit this in the verification system Jahob. An alternative to encoding polymorphic types 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].

3 Monotonicity-Based Type Encodings—The Monomorphic Case

Type tags and guards considerably increase the size of the problems passed to the automatic provers, with a dramatic impact on their performance. Most of the clutter can be removed by inferring monotonicity and soundly erasing type information based on the monotonicity analysis. Informally, a monotonic formula is one where, for any model of that formula, we can increase the size of the model while preserving satisfiability.

We focus on the monomorphic case, where the input problem contains no type variables or polymorphic symbols. Many of our definitions nonetheless handle the polymorphic case gracefully so that they can be reused in Section 4.

Before we start, let us define variants of the traditional $\mathtt{t}$ and $\mathtt{g}$ encodings that operate on monomorphic problems. The monomorphic encodings $\widetilde{\mathtt{t}}$ and $\widetilde{\mathtt{g}}$ coincide with $\mathtt{t}$ and $\mathtt{g}$ except that the polymorphic function $\mathtt{t}\langle\sigma\rangle(t)$ and predicate $\mathtt{g}\langle\sigma\rangle(t)$ are replaced by type-indexed families of unary functions $\mathtt{t}_\sigma(t)$ and predicates $\mathtt{g}_\sigma(t)$, as is customary in the literature [18].

Definition 11 (Monotonicity). Let S be a set of ground types and Φ be a problem. The types in S are (*infinitely*) *monotonic* in Φ if for all models $\mathcal{M}$ of Φ, there exists a model $\mathcal{M}'$ such that for all ground types σ, $[\![\sigma]\!]^{\mathcal{M}}$ is infinite if $\sigma \in S$ and $\big|[\![\sigma]\!]^{\mathcal{M}'}\big| = \big|[\![\sigma]\!]^{\mathcal{M}}\big|$ otherwise. A type σ is (*infinitely*) *monotonic* if $\{\sigma\}$ is monotonic. The problem Φ is (*infinitely*) *monotonic* if all its types, taken together, are monotonic.

Our criterion, infinite monotonicity, subsumes the finite monotonicity of Claessen et al. The set $\{\mathit{monkey}, \mathit{banana}\}$ is infinitely monotonic in Example 1, even though *banana* is not monotonic in the sense of Claessen et al. Another advantage of the new criterion is that it directly handles polymorphic signatures and infinitely many types.

Full type erasure is sound for monomorphic, monotonic problems. The intuition is that a model of such a problem can be extended into a model where all types are interpreted as sets of the same cardinality, which can be merged to yield an untyped model.

Claessen et al. introduced a simple calculus to infer finite monotonicity for monomorphic first-order logic [10]. The definition below generalises it from clause normal form to negation normal form. The calculus is based on the observation that a type σ must be monotonic if the problem expressed in NNF contains no positive literal of the form $X^\sigma \approx t$ or $t \approx X^\sigma$, where X is universal. We call such an occurrence of X a naked occurrence. Naked variables are the only way to express upper bounds on the cardinality of types in first-order logic.

Definition 12 (Naked Variable). The set of *naked variables* $\mathrm{NV}(\varphi)$ of a formula φ is defined as follows:

$$\begin{aligned}
\mathrm{NV}(\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})) &= \emptyset & \mathrm{NV}(t_1 \approx t_2) &= \{t_1, t_2\} \cap \mathcal{V} \\
\mathrm{NV}(\neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})) &= \emptyset & \mathrm{NV}(t_1 \not\approx t_2) &= \emptyset \\
\mathrm{NV}(\varphi_1 \wedge \varphi_2) &= \mathrm{NV}(\varphi_1) \cup \mathrm{NV}(\varphi_2) & \mathrm{NV}(\forall X{:}\sigma.\ \varphi) &= \mathrm{NV}(\varphi) \\
\mathrm{NV}(\varphi_1 \vee \varphi_2) &= \mathrm{NV}(\varphi_1) \cup \mathrm{NV}(\varphi_2) & \mathrm{NV}(\exists X{:}\sigma.\ \varphi) &= \mathrm{NV}(\varphi) - \{X\}
\end{aligned}$$

Variables of types other than σ are irrelevant when inferring whether σ is monotonic; a variable is problematic only if it occurs naked and has type σ. Annoyingly, a single naked variable of type σ will cause us to classify σ as possibly nonmonotonic.

We regain some precision by extending the calculus with an infinity analysis: trivially, all types with no finite models are monotonic. Abstracting over the specific analysis used to detect infinite types (e.g. Infinox [9]), we fix a set $\mathrm{Inf}(\Phi)$ of types whose interpretations are guaranteed to be infinite in all models of Φ. The monotonicity calculus takes $\mathrm{Inf}(\Phi)$ into account.

Definition 13 (Monotonicity Calculus $\rhd$). Let Φ be a monomorphic problem. A judgement $\sigma \rhd \varphi$ indicates that the ground type σ is inferred monotonic in $\varphi \in \Phi$. The *monotonicity calculus* consists of the following rules:

$$\frac{\sigma \in \mathrm{Inf}(\Phi)}{\sigma \rhd \varphi} \qquad \frac{\mathrm{NV}(\varphi) \cap \{X \mid X \text{ has type } \sigma\} = \emptyset}{\sigma \rhd \varphi}$$

Monotonic types can be soundly erased when translating from a monomorphic logic to an untyped logic. Nonmonotonic types in general cannot. Claessen et al. [10] point out that adding sufficiently many protectors to a nonmonotonic problem will make it monotonic, after which its types can be erased. Thus the following general two-stage procedure translates monomorphic problems to untyped first-order logic:

1. Introduce protectors (tags or guards) without erasing any types:
   (a) Introduce protectors for universal variables of possibly nonmonotonic types.
   (b) If necessary, generate *typing axioms* for any function symbol whose result type is possibly nonmonotonic, to make it possible to remove protectors.
2. Erase all the types.

The purpose of stage 1 is to make the problem monotonic while preserving satisfiability. This paves the way for the sound type erasure of stage 2.

The encoding $\widetilde{\mathsf{t}}?$, due to Claessen et al., specialises this procedure for tags. It is similar to the traditional encoding $\widetilde{\mathsf{t}}$ (the monomorphic t), except that it omits the tags for types that are inferred monotonic. By wrapping all naked variables (in fact, all terms) of possibly nonmonotonic types in a function term, stage 1 yields a monotonic problem.

Definition 14 (Lightweight Tags $\widetilde{\mathsf{t}}?$). The *monomorphic lightweight type tags* encoding $\widetilde{\mathsf{t}}?$ translates a monomorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \{\mathsf{t}^1_\sigma\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for e. It is defined as $[\![\,]\!]_{\widetilde{\mathsf{t}}?;\mathsf{e}}$, where

$$[\![\mathsf{f}(\bar{t})]\!]_{\widetilde{\mathsf{t}}?} = \lfloor \mathsf{f}([\![\bar{t}]\!]_{\widetilde{\mathsf{t}}?}) \rfloor \qquad [\![X]\!]_{\widetilde{\mathsf{t}}?} = \lfloor X \rfloor \qquad \text{with } \lfloor t^\sigma \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \\ \mathsf{t}_\sigma(t) & \text{otherwise} \end{cases}$$

Example 15. For a monomorphised version of Example 7, with α instantiated by b, the monomorphic type corresponding to $\mathit{list}(b)$ is monotonic by virtue of being infinite, whereas b cannot be inferred monotonic. The $\widetilde{\mathsf{t}}?$ encoding of the problem follows:

$$\forall X, Xs.\ \mathsf{nil}_b \not\approx \mathsf{cons}_b(\mathsf{t}_b(X), Xs)$$
$$\forall X, Xs.\ \mathsf{t}_b(\mathsf{hd}_b(\mathsf{cons}_b(\mathsf{t}_b(X), Xs))) \approx \mathsf{t}_b(X) \wedge \mathsf{tl}_b(\mathsf{cons}_b(\mathsf{t}_b(X), Xs)) \approx Xs$$
$$\exists X, Y, Xs, Ys.\ \mathsf{cons}_b(\mathsf{t}_b(X), Xs) \approx \mathsf{cons}_b(\mathsf{t}_b(Y), Ys) \wedge (\mathsf{t}_b(X) \not\approx \mathsf{t}_b(Y) \vee Xs \not\approx Ys)$$

The $\widetilde{\mathsf{t}}?$ encoding treats all variables of the same type uniformly. Hundreds of axioms can suffer because of one unhappy formula that uses a type nonmonotonically (or in a way that cannot be inferred monotonic). To address this, we introduce a lighter encoding: if a universal variable does not occur naked in a formula, its tag can safely be omitted.[1]

Our novel encoding $\widetilde{\mathsf{t}}??$ protects only naked variables and introduces equations $\mathsf{t}_\sigma(\mathsf{f}(\overline{X})^\sigma) \approx \mathsf{f}(\overline{X})$ to add or remove tags around each function symbol f whose result type σ is possibly nonmonotonic, and similarly for existential variables.

Definition 16 (Featherweight Tags $\widetilde{\mathsf{t}}??$). The *monomorphic featherweight type tags* encoding $\widetilde{\mathsf{t}}??$ translates a monomorphic problem Φ over Σ into an untyped problem over Σ', where Σ' is as for $\widetilde{\mathsf{t}}?$. It is defined as $[\![\,]\!]_{\widetilde{\mathsf{t}}??;\mathsf{e}}$, where

$$[\![t_1 \approx t_2]\!]_{\widetilde{\mathsf{t}}??} = \lfloor [\![t_1]\!]_{\widetilde{\mathsf{t}}??} \rfloor \approx \lfloor [\![t_2]\!]_{\widetilde{\mathsf{t}}??} \rfloor$$
$$[\![\exists X : \sigma.\ \varphi]\!]_{\widetilde{\mathsf{t}}??} = \exists X : \sigma.\ \begin{cases} [\![\varphi]\!]_{\widetilde{\mathsf{t}}??} & \text{if } \sigma \rhd \Phi \\ \mathsf{t}_\sigma(X) \approx X \wedge [\![\varphi]\!]_{\widetilde{\mathsf{t}}??} & \text{otherwise} \end{cases}$$

with

$$\lfloor t^\sigma \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \text{ or } t \text{ is not a universal variable} \\ \mathsf{t}_\sigma(t) & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$$\forall \bar{X} : \bar{\sigma}.\ \mathsf{t}_\sigma(\mathsf{f}(\bar{X})) \approx \mathsf{f}(\bar{X}) \qquad \text{for } \mathsf{f} : \bar{\sigma} \to \sigma \in \mathcal{F} \text{ such that } \sigma \not\rhd \Phi$$
$$\exists X : \sigma.\ \mathsf{t}_\sigma(X) \approx X \qquad \text{for } \sigma \not\rhd \Phi \text{ that is not the result type of a symbol in } \mathcal{F}$$

The side condition for the last axiom is a minor optimisation: it avoids asserting that σ is inhabited if the symbols in $\mathcal{F}$ already witness σ's inhabitation.

[1] This is related to the observation that only paramodulation from or into a variable can cause ill-typed instantiations in a resolution prover [18].

Example 17. The $\widetilde{\mathsf{t}}??$ encoding of Example 15 requires fewer tags than $\widetilde{\mathsf{t}}?$, at the cost of more type information (for hd and the existential variables of type b):

$$\forall Xs.\ \mathsf{t}_b(\mathsf{hd}_b(Xs)) \approx \mathsf{hd}_b(Xs)$$
$$\forall X, Xs.\ \mathsf{nil}_b \not\approx \mathsf{cons}_b(X, Xs)$$
$$\forall X, Xs.\ \mathsf{hd}_b(\mathsf{cons}_b(X, Xs)) \approx \mathsf{t}_b(X) \wedge \mathsf{tl}_b(\mathsf{cons}_b(X, Xs)) \approx Xs$$
$$\exists X, Y, Xs, Ys.\ \mathsf{t}_b(X) \approx X \wedge \mathsf{t}_b(Y) \approx Y \wedge \mathsf{cons}_b(X, Xs) \approx \mathsf{cons}_b(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

The $\widetilde{\mathsf{g}}?$ and $\widetilde{\mathsf{g}}??$ encodings are defined analogously to $\widetilde{\mathsf{t}}?$ and $\widetilde{\mathsf{t}}??$ but using type guards. The $\widetilde{\mathsf{g}}?$ encoding omits the guards for types that are inferred monotonic, whereas $\widetilde{\mathsf{g}}??$ omits more guards that are not needed to make the intermediate problem monotonic.

Definition 18 (Lightweight Guards $\widetilde{\mathsf{g}}?$). The *monomorphic lightweight type guards* encoding $\widetilde{\mathsf{g}}?$ translates a monomorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}', \mathcal{P}' \uplus \{\mathsf{g}^1_\sigma\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for e. It is defined as $[\![\ ]\!]_{\widetilde{\mathsf{g}}?;\mathsf{e}}$, where

$$[\![\forall X : \sigma.\ \varphi]\!]_{\widetilde{\mathsf{g}}?} = \forall X : \sigma.\ \begin{cases} [\![\varphi]\!]_{\widetilde{\mathsf{g}}?} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}_\sigma(X) \to [\![\varphi]\!]_{\widetilde{\mathsf{g}}?} & \text{otherwise} \end{cases}$$
$$[\![\exists X : \sigma.\ \varphi]\!]_{\widetilde{\mathsf{g}}?} = \exists X : \sigma.\ \begin{cases} [\![\varphi]\!]_{\widetilde{\mathsf{g}}?} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}_\sigma(X) \wedge [\![\varphi]\!]_{\widetilde{\mathsf{g}}?} & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$$\forall \bar{X} : \bar{\sigma}.\ \mathsf{g}_\sigma(\mathsf{f}(\bar{X})) \qquad \text{for } \mathsf{f} : \bar{\sigma} \to \sigma \in \mathcal{F} \text{ such that } \sigma \not\rhd \Phi$$
$$\exists X : \sigma.\ \mathsf{g}_\sigma(X) \qquad \text{for } \sigma \not\rhd \Phi \text{ that is not the result type of a symbol in } \mathcal{F}$$

Example 19. The $\widetilde{\mathsf{g}}?$ encoding of Example 15 is as follows:

$$\forall Xs.\ \mathsf{g}_b(\mathsf{hd}_b(Xs))$$
$$\forall X, Xs.\ \mathsf{g}_b(X) \to \mathsf{nil}_b \not\approx \mathsf{cons}_b(X, Xs)$$
$$\forall X : b, Xs.\ \mathsf{g}_b(X) \to \mathsf{hd}_b(\mathsf{cons}_b(X, Xs)) \approx X \wedge \mathsf{tl}_b(\mathsf{cons}_b(X, Xs)) \approx Xs$$
$$\exists X, Y, Xs, Ys.\ \mathsf{g}_b(X) \wedge \mathsf{g}_b(Y) \wedge \mathsf{cons}_b(X, Xs) \approx \mathsf{cons}_b(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$$

Our novel encoding $\widetilde{\mathsf{g}}??$ omits the guards for variables that do no occur naked, regardless of whether they are of a monotonic type.

Definition 20 (Featherweight Guards $\widetilde{\mathsf{g}}??$). The *monomorphic featherweight type guards* encoding $\widetilde{\mathsf{g}}??$ is identical to the lightweight encoding $\widetilde{\mathsf{g}}?$ except that the condition "if $\sigma \rhd \Phi$" in the $\forall$ case is weakened to "if $\sigma \rhd \Phi$ or $X \notin \mathrm{NV}(\varphi)$".

Example 21. The $\widetilde{\mathsf{g}}??$ encoding of the algebraic list problem is identical to $\widetilde{\mathsf{g}}?$ except that the $\mathsf{nil}_b \not\approx \mathsf{cons}_b$ axiom does not have any guard.

Theorem 22 (Soundness and Completeness). *Let Φ be a monomorphic problem, and let $x \in \{\widetilde{\mathsf{t}}?, \widetilde{\mathsf{t}}??, \widetilde{\mathsf{g}}?, \widetilde{\mathsf{g}}??\}$. The problems Φ and $[\![\Phi]\!]_{x;\mathsf{e}}$ are equisatisfiable.*

Section 4 will show how to translate polymorphic types soundly and completely. If we are willing to sacrifice completeness, an easy way to extend $\widetilde{\mathsf{t}}?$, $\widetilde{\mathsf{t}}??$, $\widetilde{\mathsf{g}}?$, and $\widetilde{\mathsf{g}}??$ to polymorphism is to perform *finite monomorphisation*: heuristically instantiate all type variables with suitable ground types, taking as many copies of the formulas as desired. Finite monomorphisation is generally incomplete [7], but by eliminating type variables it considerably simplifies the generated formulas, leading to very efficient encodings.

4 Complete Monotonicity-Based Encoding of Polymorphism

Finite monomorphisation is simple and effective, but its incompleteness can be a cause for worry, and its nonmodular nature makes it unsuitable for some applications that need to export an entire polymorphic theory independently of any conjecture. Here we adapt the monotonicity calculus and the monomorphic encodings to a polymorphic setting.

We start with a brief digression. With monotonicity-based encoding schemes, type arguments are needed to distinguish instances of polymorphic symbols. These additional arguments introduce clutter, which we can eliminate in some cases. The result is an optimised variant $\mathsf{a}^{\mathsf{ctor}}$ of the type arguments encoding a, which will serve as the foundation for $\mathsf{t?}$, $\mathsf{t??}$, $\mathsf{g?}$, and $\mathsf{g??}$. Consider a type $\mathsf{sum}(\alpha,\beta)$ that is axiomatised to be freely constructed by $\mathsf{inl} : \alpha \to \mathsf{sum}(\alpha,\beta)$ and $\mathsf{inr} : \beta \to \mathsf{sum}(\alpha,\beta)$. Regardless of β, inl must be interpreted as an injection from α to $\mathsf{sum}(\alpha,\beta)$. For a fixed α, its interpretations for different β instances are isomorphic. As a result, it is safe to omit the type argument for β when encoding $\mathsf{inl}\langle\alpha,\beta\rangle$ and that for α in $\mathsf{inr}\langle\alpha,\beta\rangle$ and $\mathsf{nil}\langle\alpha\rangle : \mathit{list}(\alpha)$. In general, the type arguments that can be omitted for constructors are precisely those that are noninferable in the sense of Definition 8. We call this encoding $\mathsf{a}^{\mathsf{ctor}}$. The encodings presented below exploit the fact that $[\![\Phi]\!]_{\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$ is equisatisfiable to Φ if Φ is monotonic.

The polymorphic version of the monotonicity calculus captures the insight that a polymorphic type is monotonic if each of its common instances with the type of any naked variable is an instance of an infinite type.

Definition 23 (Monotonicity Calculus $\rhd$). Let Φ be a polymorphic problem. The *monotonicity calculus* consists of the single rule

$$\frac{\forall X^\tau \in \mathrm{NV}(\varphi).\ \mathrm{mgu}(\sigma,\tau) \in \mathrm{Inf}^*(\varphi)}{\sigma \rhd \varphi}$$

where $\mathrm{mgu}(\sigma,\tau)$ is the most general unifier of σ and τ, and $\mathrm{Inf}^*(\varphi)$ consists of all instances of all types in $\mathrm{Inf}(\varphi)$.

The polymorphic $\mathsf{t?}$ encoding can be seen as a hybrid between traditional tags (t) and monomorphic lightweight tags ($\widetilde{\mathsf{t}}\mathsf{?}$): as in t, tags take the form of a function $\mathsf{t}\langle\sigma\rangle(t)$; as in $\widetilde{\mathsf{t}}\mathsf{?}$, tags are omitted for types that are inferred monotonic.

The main novelty concerns the typing axioms. The $\widetilde{\mathsf{t}}\mathsf{?}$ encoding omits all typing axioms for infinite types. In the polymorphic case, the infinite type σ might be an instance of a more general, potentially finite type for which tags are generated. For example, if α is tagged (because it is possibly nonmonotonic) but its instance $\mathit{list}(\alpha)$ is not (because it is infinite), there will be mismatches between tagged and untagged terms. Our solution is to add the typing axiom $\mathsf{t}\langle\mathit{list}(\alpha)\rangle(\mathit{Xs}) \approx \mathit{Xs}$, which allows the prover to add or remove a tag for the infinite type $\mathit{list}(\alpha)$. Such an axiom is sound for any monotonic type.

Definition 24 (Lightweight Tags $\mathsf{t?}$). The *polymorphic lightweight type tags* encoding $\mathsf{t?}$ translates a polymorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ctor}}$. It is defined as $[\![\ ]\!]_{\mathsf{t?};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

$$[\![\mathsf{f}\langle\sigma\rangle(\bar{t})^\sigma]\!]_{\mathsf{t?}} = \lfloor \mathsf{f}\langle\sigma\rangle([\![\bar{t}]\!]_{\mathsf{t?}})\rfloor \quad [\![X^\sigma]\!]_{\mathsf{t?}} = \lfloor X\rfloor \quad \text{with } \lfloor t^\sigma\rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \\ \mathsf{t}\langle\sigma\rangle(t) & \text{otherwise} \end{cases}$$

The encoding is complemented by the following typing axioms, where ρ is a type substitution and $\mathrm{TV}(\sigma\rho)$ denotes the type variables of $\sigma\rho$:

$$\forall \mathrm{TV}(\sigma\rho).\, \forall X : \sigma\rho.\, \mathsf{t}\langle\sigma\rho\rangle(X) \approx X \qquad \text{for } \sigma\rho \in \mathrm{Inf}(\Phi) \text{ such that } \sigma \not\rhd \Phi$$

The lighter encoding t?? protects only naked variables and introduces equations of the form $\mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\overline{X})) \approx \mathsf{f}\langle\bar{\alpha}\rangle(\overline{X})$ to add or remove tags around each function symbol f of a possibly nonmonotonic type σ, and similarly for existential variables.

Definition 25 (Featherweight Tags t??). The *polymorphic featherweight type tags* encoding t?? translates a polymorphic problem Φ over Σ into an untyped problem over Σ', where Σ' is as for t?. It is defined as $[\![\ ]\!]_{\mathsf{t??};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

$$[\![t_1 \approx t_2]\!]_{\mathsf{t??}} = \lfloor [\![t_1]\!]_{\mathsf{t??}} \rfloor \approx \lfloor [\![t_2]\!]_{\mathsf{t??}} \rfloor$$

$$[\![\exists X : \sigma.\ \varphi]\!]_{\mathsf{t??}} = \exists X : \sigma.\ \begin{cases} [\![\varphi]\!]_{\mathsf{t??}} & \text{if } \sigma \rhd \Phi \\ \mathsf{t}\langle\sigma\rangle(X) \approx X \wedge [\![\varphi]\!]_{\mathsf{t??}} & \text{otherwise} \end{cases}$$

with

$$\lfloor t^{\sigma} \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \text{ or } t \text{ is not a universal variable} \\ \mathsf{t}\langle\sigma\rangle(t) & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$\forall \bar{\alpha}.\, \forall \bar{X} : \bar{\sigma}.\, \mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})) \approx \mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})$	for $\mathsf{f} : \forall \bar{\alpha}.\, \bar{\sigma} \to \sigma \in \mathcal{F}$ such that $\exists \rho.\, \sigma\rho \not\rhd \Phi$
$\forall \mathrm{TV}(\sigma\rho).\, \forall X : \sigma\rho.\, \mathsf{t}\langle\sigma\rho\rangle(X) \approx X$	for $\sigma\rho \in \mathrm{Inf}(\Phi)$ such that $\sigma \not\rhd \Phi$
$\forall \mathrm{TV}(\sigma).\, \exists X : \sigma.\, \mathsf{t}\langle\sigma\rangle(X) \approx X$	for $\sigma \not\rhd \Phi$ that is not an instance of the result type of $\mathsf{f} \in \mathcal{F}$ or a proper instance of $\tau \not\rhd \Phi$

Example 26. In Example 7, $\mathit{list}(\alpha)$ is infinite and hence monotonic, whereas α and its instance b cannot be inferred monotonic. The t?? encoding of the problem follows:

$$\begin{array}{l} \forall A, Xs.\ \mathsf{t}(A, \mathsf{hd}(A, Xs)) \approx \mathsf{hd}(A, Xs) \\ \forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), Xs) \approx Xs \\ \forall A.\, \exists X.\ \mathsf{t}(A, X) \approx X \\ \forall A, X, Xs.\ \mathsf{nil} \not\approx \mathsf{cons}(A, X, Xs) \\ \forall A, X, Xs.\ \mathsf{hd}(A, \mathsf{cons}(A, X, Xs)) \approx \mathsf{t}(A, X) \wedge \mathsf{tl}(A, \mathsf{cons}(A, X, Xs)) \approx Xs \\ \exists X, Y, Xs, Ys.\ \mathsf{t}(\mathsf{b}, X) \approx X \wedge \mathsf{t}(\mathsf{b}, Y) \approx Y \wedge \\ \qquad\qquad \mathsf{cons}(\mathsf{b}, X, Xs) \approx \mathsf{cons}(\mathsf{b}, Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys) \end{array}$$

Analogously to t?, the g? encoding is best understood as a hybrid between traditional guards (g) and monomorphic lightweight guards ($\widetilde{\mathsf{g}}$?): as in g, guards take the form of a predicate $\mathsf{g}\langle\sigma\rangle(t)$; as in $\widetilde{\mathsf{g}}$?, guards are omitted for types that are inferred monotonic.

Once again, the main novelty concerns the typing axioms. The $\widetilde{\mathsf{g}}$? encoding omits all typing axioms for infinite types. In the polymorphic case, the infinite type σ might be an instance of a more general, potentially finite type for which guards are generated. Our solution is to add the typing axiom $\mathsf{g}\langle\sigma\rangle(X)$, which allows the prover to discharge any guard for the infinite type σ.

Definition 27 (Lightweight Guards g?). The *polymorphic lightweight type guards* encoding g? translates a polymorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}', \mathcal{P}' \uplus \{\mathsf{g}^2\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ctor}}$. It is defined as $[\![\ ]\!]_{\mathsf{g?};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

$$[\![\forall X\!:\!\sigma.\ \varphi]\!]_{\mathsf{g?}} = \forall X\!:\!\sigma.\ \begin{cases} [\![\varphi]\!]_{\mathsf{g?}} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}\langle\sigma\rangle(X) \to [\![\varphi]\!]_{\mathsf{g?}} & \text{otherwise} \end{cases}$$

$$[\![\exists X\!:\!\sigma.\ \varphi]\!]_{\mathsf{g?}} = \exists X\!:\!\sigma.\ \begin{cases} [\![\varphi]\!]_{\mathsf{g?}} & \text{if } \sigma \rhd \Phi \\ \mathsf{g}\langle\sigma\rangle(X) \wedge [\![\varphi]\!]_{\mathsf{g?}} & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$\forall\bar{\alpha}.\ \forall\bar{X}\!:\!\bar{\sigma}.\ \mathsf{g}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\bar{X}))$	for $\mathsf{f} : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma \in \mathcal{F}$ such that $\exists\rho.\ \sigma\rho \not\rhd \Phi$
$\forall \mathrm{TV}(\sigma\rho).\ \forall X\!:\!\bar{\sigma}\rho.\ \mathsf{g}\langle\sigma\rho\rangle(X)$	for $\sigma\rho \in \mathrm{Inf}(\Phi)$ such that $\sigma \not\rhd \Phi$
$\forall \mathrm{TV}(\sigma).\ \exists X\!:\!\sigma.\ \mathsf{g}\langle\sigma\rangle(X)$	for $\sigma \not\rhd \Phi$ that is not an instance of the result type of $\mathsf{f} \in \mathcal{F}$ or a proper instance of $\tau \not\rhd \Phi$

The featherweight cousin is a straightforward generalisation of g?.

Definition 28 (Featherweight Guards g??). The *polymorphic featherweight type guards* encoding g?? is identical to the lightweight encoding g? except that the condition "if $\sigma \rhd \Phi$" in the $\forall$ case is weakened to "if $\sigma \rhd \Phi$ or $X \notin \mathrm{NV}(\varphi)$".

Example 29. The g?? encoding of Example 7 follows:

$\forall A, Xs.\ \mathsf{g}(A, \mathsf{hd}(A, Xs))$
$\forall A, Xs.\ \mathsf{g}(\mathsf{list}(A), Xs)$
$\forall A, X, Xs.\ \mathsf{nil} \not\approx \mathsf{cons}(A, X, Xs)$
$\forall A, X, Xs.\ \mathsf{g}(A, X) \to \mathsf{hd}(A, \mathsf{cons}(A, X, Xs)) \approx X \wedge \mathsf{tl}(A, \mathsf{cons}(A, X, Xs)) \approx Xs$
$\exists X, Y, Xs, Ys.\ \mathsf{g}(\mathsf{b}, X) \wedge \mathsf{g}(\mathsf{b}, Y) \wedge \mathsf{cons}(\mathsf{b}, X, Xs) \approx \mathsf{cons}(\mathsf{b}, Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)$

Theorem 30 (Soundness and Completeness). *Let Φ be a polymorphic problem, and let $\mathsf{x} \in \{\mathsf{t?}, \mathsf{t??}, \mathsf{g?}, \mathsf{g??}\}$. The problems Φ and $[\![\Phi]\!]_{\mathsf{x};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$ are equisatisfiable.*

5 Alternative, Cover-Based Encoding of Polymorphism

An issue with t?, t??, g?, and g?? is that they clutter the generated problem with type arguments. In that respect, the traditional t and g encodings are superior—t omits all non-phantom type arguments, and g omits all inferable type arguments. This would be unsound for the monotonicity-based encodings, because these leave out many of the protectors that implicitly "carry", or "cover", the type arguments in the traditional encodings. Nonetheless, an alternative is possible: by keeping more protectors around, we can omit inferable type arguments.

Definition 31 (Cover). Let $\mathsf{s} : \forall\bar{\alpha}.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. A (*type argument*) *cover* $C \subseteq \{1, \ldots, |\bar{\sigma}|\}$ for s is a set of term argument indices such that any inferable type argument can be inferred from a term argument whose index is in C. We let $\mathrm{Cover}_{\mathsf{s}}$ denote an arbitrary but fixed minimal cover of s.

For example, $\{1\}$ and $\{2\}$ are minimal covers for $\mathsf{cons} : \forall\alpha.\ \alpha \times \mathit{list}(\alpha) \to \mathit{list}(\alpha)$, and $\{1,2\}$ is also a cover. As canonical cover, we arbitrarily choose $\mathrm{Cover}_{\mathsf{cons}} = \{1\}$.

The encodings $\mathtt{t@}$ and $\mathtt{g@}$ introduced below ensure that each argument that is part of its enclosing function or predicate's cover has a unique type, from which the omitted type arguments can be inferred. For example, $\mathtt{t@}$ translates the term $\mathsf{cons}\langle\alpha\rangle(X, Xs)$ to $\mathsf{cons}(\mathsf{t}(A, X), Xs)$ with a type tag around X, effectively preventing an ill-typed instantiation of X that would result in the wrong type argument being inferred. We call variables that occur in their enclosing symbol's cover "undercover variables". They can be seen as a generalisation of naked variables to arbitrary predicate and function symbols.

Definition 32 (Undercover Variable). The set of *undercover variables* $\mathrm{UV}(\varphi)$ of a formula φ is defined by the equations

$$\begin{aligned}
\mathrm{UV}(\mathsf{f}\langle\bar\sigma\rangle(\bar t)) &= \lfloor\bar t\rfloor_{\mathsf{f}} \cup \mathrm{UV}(\bar t) & \mathrm{UV}(X) &= \emptyset\\
\mathrm{UV}(\mathsf{p}\langle\bar\sigma\rangle(\bar t)) &= \lfloor\bar t\rfloor_{\mathsf{p}} \cup \mathrm{UV}(\bar t) & \mathrm{UV}(t_1 \approx t_2) &= (\{t_1, t_2\} \cap \mathcal{V}) \cup \mathrm{UV}(t_1, t_2)\\
\mathrm{UV}(\neg\, \mathsf{p}\langle\bar\sigma\rangle(\bar t)) &= \lfloor\bar t\rfloor_{\mathsf{p}} \cup \mathrm{UV}(\bar t) & \mathrm{UV}(t_1 \not\approx t_2) &= \mathrm{UV}(t_1, t_2)\\
\mathrm{UV}(\varphi_1 \wedge \varphi_2) &= \mathrm{UV}(\varphi_1, \varphi_2) & \mathrm{UV}(\forall X\colon\sigma.\ \varphi) &= \mathrm{UV}(\varphi)\\
\mathrm{UV}(\varphi_1 \vee \varphi_2) &= \mathrm{UV}(\varphi_1, \varphi_2) & \mathrm{UV}(\exists X\colon\sigma.\ \varphi) &= \mathrm{UV}(\varphi) - \{X\}
\end{aligned}$$

where $\lfloor\bar t\rfloor_{\mathsf{s}} = \{t_j \mid j \in \mathrm{Cover}_{\mathsf{s}}\} \cap \mathcal{V}$ and $\mathrm{UV}(\bar t) = \bigcup_j \mathrm{UV}(t_j)$.

The cover-based encoding $\mathtt{t@}$ is similar to the traditional encoding $\mathtt{t}$, except that it tags only undercover occurrences of variables and requires typing axioms.

Definition 33 (Cover Tags $\mathtt{t@}$). The *polymorphic cover-based type tags* encoding $\mathtt{t@}$ translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K} \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ninf}}$. It is defined as $[\![\ ]\!]_{\mathtt{t@};\mathsf{a}^{\mathsf{ninf}};\mathtt{e}}$, where

$$\begin{aligned}
[\![\mathsf{f}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathtt{t@}} &= \mathsf{f}\langle\bar\sigma\rangle(\lfloor[\![\bar t]\!]_{\mathtt{t@}}\rfloor_{\mathsf{f}}) & [\![t_1 \approx t_2]\!]_{\mathtt{t@}} &= \lfloor[\![t_1]\!]_{\mathtt{t@}}\rfloor_{\approx} \approx \lfloor[\![t_2]\!]_{\mathtt{t@}}\rfloor_{\approx}\\
[\![\mathsf{p}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathtt{t@}} &= \mathsf{p}\langle\bar\sigma\rangle(\lfloor[\![\bar t]\!]_{\mathtt{t@}}\rfloor_{\mathsf{p}}) & [\![\exists X\colon\sigma.\ \varphi]\!]_{\mathtt{t@}} &= \exists X\colon\sigma.\ \mathsf{t}\langle\sigma\rangle(X) \approx X \wedge [\![\varphi]\!]_{\mathtt{t@}}\\
[\![\neg\, \mathsf{p}\langle\bar\sigma\rangle(\bar t)]\!]_{\mathtt{t@}} &= \neg\, \mathsf{p}\langle\bar\sigma\rangle(\lfloor[\![\bar t]\!]_{\mathtt{t@}}\rfloor_{\mathsf{p}}) & &
\end{aligned}$$

The auxiliary function $\lfloor(t_1^{\sigma_1}, \ldots, t_n^{\sigma_n})\rfloor_{\mathsf{s}}$ returns a vector $(u_1, \ldots, u_n)$ such that

$$u_j = \begin{cases} t_j & \text{if } j \notin \mathrm{Cover}_{\mathsf{s}} \text{ or } t_j \text{ is not a universal variable}\\ \mathsf{t}\langle\sigma_j\rangle(t_j) & \text{otherwise}\end{cases}$$

taking $\mathrm{Cover}_{\approx} = \{1, 2\}$. The encoding is complemented by typing axioms:

$$\begin{aligned}
&\forall\bar\alpha.\ \forall\bar X\colon\bar\sigma.\ \mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar\alpha\rangle(\lfloor\bar X\rfloor_{\mathsf{f}})) \approx \mathsf{f}\langle\bar\alpha\rangle(\lfloor\bar X\rfloor_{\mathsf{f}}) \qquad \text{for } \mathsf{f}\colon\forall\bar\alpha.\ \bar\sigma \to \sigma \in \mathcal{F}\\
&\forall\alpha.\ \exists X\colon\alpha.\ \mathsf{t}\langle\alpha\rangle(X) \approx X
\end{aligned}$$

Example 34. The $\mathtt{t@}$ encoding of Example 7 is as follows:

$$\begin{aligned}
&\forall A.\ \mathsf{t}(\mathsf{list}(A), \mathsf{nil}(A)) \approx \mathsf{nil}(A)\\
&\forall A, X, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), Xs)) \approx \mathsf{cons}(\mathsf{t}(A, X), Xs)\\
&\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), Xs))) \approx \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), Xs))\\
&\forall A, Xs.\ \mathsf{t}(A, \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), Xs))) \approx \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), Xs))\\
&\forall A, X, Xs.\ \mathsf{nil}(A) \not\approx \mathsf{cons}(\mathsf{t}(A, X), Xs)\\
&\forall A, X, Xs.\ \mathsf{hd}(\mathsf{cons}(\mathsf{t}(A, X), Xs)) \approx \mathsf{t}(A, X) \wedge \mathsf{tl}(\mathsf{cons}(\mathsf{t}(A, X), Xs)) \approx \mathsf{t}(\mathsf{list}(A), Xs)\\
&\exists X, Y, Xs, Ys.\ \mathsf{t}(\mathsf{b}, X) \approx X \wedge \mathsf{t}(\mathsf{b}, Y) \approx Y \wedge \mathsf{t}(\mathsf{list}(\mathsf{b}), Xs) \approx Xs \wedge \mathsf{t}(\mathsf{list}(\mathsf{b}), Ys) \approx Ys \wedge\\
&\qquad\qquad \mathsf{cons}(X, Xs) \approx \mathsf{cons}(Y, Ys) \wedge (X \not\approx Y \vee Xs \not\approx Ys)
\end{aligned}$$

Definition 35 (Cover Guards g@). The *polymorphic cover-based type guards* encoding g@ is identical to the traditional g encoding except for the $\forall$ case:

$$[\![\forall X\!:\!\sigma.\ \varphi]\!]_{\mathsf{g@}} = \forall X\!:\!\sigma.\ \begin{cases} [\![\varphi]\!]_{\mathsf{g@}} & \text{if } X \notin \mathrm{UV}(\varphi) \\ \mathsf{g}\langle\sigma\rangle(X) \to [\![\varphi]\!]_{\mathsf{g@}} & \text{otherwise} \end{cases}$$

The encoding is complemented by typing axioms:

$$\forall \bar{\alpha}.\ \bar{X}\!:\!\bar{\sigma}.\ \big(\bigwedge_{j\in \mathrm{Cover}_{\mathsf{f}}} \mathsf{g}\langle\sigma_j\rangle(X_j)\big) \to \mathsf{g}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})) \qquad \text{for } \mathsf{f}\!:\!\forall\bar{\alpha}.\ \bar{\sigma}\to\sigma \in \mathcal{F}$$
$$\forall \alpha.\ \exists X\!:\!\alpha.\ \mathsf{g}\langle\alpha\rangle(X)$$

Example 36. The g@ encoding of the algebraic list problem is identical to the g encoding (Example 10), except that the guard on *Xs* is omitted in two of the axioms:

$$\forall A, X, Xs.\ \mathsf{g}(A, X) \to \mathsf{g}(\mathsf{list}(A), \mathsf{cons}(X, Xs))$$
$$\forall A, X, Xs.\ \mathsf{g}(A, X) \to \mathsf{nil}(A) \not\approx \mathsf{cons}(X, Xs)$$

Theorem 37 (Soundness and Completeness). *Let Φ be a polymorphic problem, and let $\bar{x} \in \{\mathsf{t@};\mathsf{a}^{\mathsf{ninf}}, \mathsf{g@};\mathsf{a}^{\mathsf{ninf}}\}$. The problems Φ and $[\![\Phi]\!]_{\bar{x};\mathsf{e}}$ are equisatisfiable.*

6 Evaluation

To evaluate the type encodings described in this paper, we put together a set of 1000 polymorphic first-order problems originating from 10 existing Isabelle theories, translated with Sledgehammer's help. Our test data are publicly available [1].

The problems include up to 500 heuristically selected facts. We evaluated each type encoding with five modern automatic provers: E 1.6, iProver 0.99, SPASS 3.8ds, Vampire 2.6, and Z3 4.0. To make the evaluation more informative, we also tested the provers' native support for monomorphic types where it is available; it is referred to as $\widetilde{\mathsf{n}}$. Each prover was invoked with the set of options we had previously determined worked best for Sledgehammer.[2] The provers were granted 20 seconds of CPU time per problem on one core of a 3.06 GHz Dual-Core Intel Xeon processor. To avoid giving the unsound encodings an unfair advantage, for these proof search was followed by a certification phase that attempted to re-find the proof using a combination of sound encodings, based on its referenced facts. This phase slightly penalises the unsound encodings by rejecting a few sound proofs, but such is the price of unsoundness.

Figure 1 gives, for each combination of prover and encoding, the number of solved problems. Rows marked with ~ concern the monomorphic encodings. The encodings $\widetilde{\mathsf{a}}$, $\widetilde{\mathsf{a}}^{\mathsf{ctor}}$, $\widetilde{\mathsf{t@}}$, and $\widetilde{\mathsf{g@}}$ are omitted; the first two coincide with $\widetilde{\mathsf{e}}$, whereas $\widetilde{\mathsf{t@}}$ and $\widetilde{\mathsf{g@}}$ are identical to versions of $\widetilde{\mathsf{t}}$?? and $\widetilde{\mathsf{g}}$?? that treat all types as possibly nonmonotonic. Among the encodings to untyped first-order logic, the monomorphic featherweight encoding $\widetilde{\mathsf{g}}$?? performed best overall. It even outperformed Vampire's recently added native types ($\widetilde{\mathsf{n}}$). Among the polymorphic encodings, our novel monotonicity-based and cover-based encodings (t?, t??, t@, g?, g??, and g@), with the exception of t@, constitute a substantial improvement over the traditional sound schemes (t and g).

[2] The setup for E was suggested by Stephan Schulz and includes the little known "symbol offset" weight function. We ran iProver with the default setup, SPASS in Isabelle mode, Vampire in CASC mode, and Z3 in TPTP mode with model-based quantifier instantiation enabled.

		e	a	t	t?	t??	t@	g	g?	g??	g@	n
E		116	**361**	263	275	347	228	216	344	349	262	–
	~	393	–	328	390	397	–	337	393	**401**	–	–
iProver		243	212	231	202	**262**	135	140	242	257	169	–
	~	210	–	243	246	245	–	180	**247**	241	–	–
SPASS		131	292	262	245	**299**	164	164	283	296	208	–
	~	331	–	293	326	330	–	237	320	334	–	**356**
Vampire		120	**341**	277	281	314	212	171	271	299	241	–
	~	393	–	309	379	382	–	265	390	**403**	–	372
Z3		281	**355**	250	238	350	279	213	291	351	268	–
	~	354	–	268	343	346	–	328	**355**	349	–	350

Figure 1. Number of solved problems

The new type encodings also made an impact at the 2012 edition of CASC, the annual automatic prover competition [16]. Isabelle competes against LEO-II, Satallax, and TPS in the higher-order division. Largely thanks to the new schemes (but also to improvements in the underlying first-order provers), Isabelle moved from the third place it had occupied since 2009 to the first place.

7 Conclusion

This paper introduced a family of translations from polymorphic into untyped first-order logic, with a focus on efficiency. Our monotonicity-based encodings soundly erase all types that are inferred monotonic, as well as most occurrences of the remaining types. The best translations outperform the traditional encoding schemes.

We implemented the new translations in the Sledgehammer tool for Isabelle/HOL and the companion proof method *metis*, 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 Monotonox [10]. Applications such as Boogie [12] and Why3 [6] also stand to gain from lighter encodings.

The TPTP family recently welcomed the addition of TFF1 [3], an extension of the monomorphic TFF0 logic with rank-1 polymorphism. Equipped with a concrete syntax and translation tools, we can 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.

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.

Acknowledgement. Koen Claessen and Tobias Nipkow made this collaboration possible. Lukas Bulwahn, Peter Lammich, Rustan Leino, Tobias Nipkow, Mark Summerfield, Tjark Weber, and several anonymous reviewers suggested dozens of textual improvements. We thank them all. The first author's research was supported by the Deutsche Forschungsgemeinschaft (grants Ni 491/11-2 and Ni 491/14-1). The third author's research was supported by the DFG project Ni 491/13-2, part of the priority program RS^3. The authors are listed alphabetically regardless of contributions or seniority.

References

[1] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Empirical data associated with this paper. `http://www21.in.tum.de/~blanchet/enc_types_data.tar.gz`, 2012.

[2] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. Tech. report, `http://www21.in.tum.de/~blanchet/enc_types_report.pdf`, 2012.

[3] J. C. Blanchette and A. Paskevich. TFF1: The TPTP typed first-order form with rank-1 polymorphism. Tech. report, `http://www21.in.tum.de/~blanchet/tff1spec.pdf`, 2012.

[4] J. C. Blanchette and A. Popescu. Formal development associated with this paper. `http://www21.in.tum.de/~popescua/enc_types_devel.zip`, 2012.

[5] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In C. Barrett and L. de Moura, editors, *SMT 2008*, 2008.

[6] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In K. R. M. Leino and M. Moskal, editors, *Boogie 2011*, pages 53–64, 2011.

[7] F. Bobot and A. Paskevich. Expressing polymorphic types in a many-sorted language. In C. Tinelli and V. Sofronie-Stokkermans, editors, *FroCoS 2011*, volume 6989 of *LNCS*, pages 87–102. Springer, 2011.

[8] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. Using first-order theorem provers in the Jahob data structure verification system. In B. Cook and A. Podelski, editors, *VMCAI 2007*, volume 4349 of *LNCS*, pages 74–88. Springer, 2007.

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

[10] K. Claessen, A. Lilliestr̈om, and N. Smallbone. Sort it out with monotonicity—Translating between many-sorted and unsorted first-order logic. In N. Bjørner and V. Sofronie-Stokkermans, editors, *CADE-23*, volume 6803 of *LNAI*, pages 207–221. Springer, 2011.

[11] H. B. Enderton. *A Mathematical Introduction to Logic*. Academic Press, 1972.

[12] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In J. Esparza and R. Majumdar, editors, *TACAS 2010*, volume 6015 of *LNCS*, pages 312–327. Springer, 2010.

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

[14] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

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

[16] G. Sutcliffe. Proceedings of the 6th IJCAR ATP system competition (CASC-J6). In G. Sutcliffe, editor, *CASC-J6*, volume 11 of *EPiC*, pages 1–50. EasyChair, 2012.

[17] J. Urban. MPTP 0.2: Design, implementation, and initial experiments. *J. Autom. Reasoning*, 37(1-2):21–43, 2006.

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