Encoding Monomorphic and Polymorphic Types

Jasmin Christian Blanchette[1], Sascha Böhme[1],
Andrei Popescu[1,2], and Nicholas Smallbone[3]

[1] Fakultät für Informatik, Technische Universität München, Germany
[2] Institute of Mathematics Simion Stoilow of the Romanian Academy, Bucharest, Romania
[3] 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 [21, 26]. As a result, application authors face a difficult choice between soundness and efficiency.

The fourth author, together with Claessen and Lilliestrom [14], 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 1.1 (Monkey Village). Imagine a village of monkeys [14] where each monkey owns at least two bananas:

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

The predicate $\mathsf{owns} : \mathit{monkey} \times \mathit{banana} \rightarrow \mathrm{o}$ associates monkeys with bananas, and the functions $\mathsf{b}_1, \mathsf{b}_2 : \mathit{monkey} \rightarrow \mathit{banana}$ witness the existence of each monkey's minimum supply of bananas. The axioms are satisfiable.

In the monkey village of Example 1.1, the type *banana* is monotonic, because any model with b bananas can be extended to a model with $b' > b$ bananas. In contrast, *monkey* is nonmonotonic, because there can live at most $\lfloor b/2 \rfloor$ monkeys in a village with a finite supply of b bananas. Syntactically, the monotonicity of *banana* is inferable from the absence of a positive equality $B \approx t$ or $t \approx B$, where B is a variable of type *banana* and t is arbitrary; such a literal would be needed to make the type nonmonotonic.

The example can be encoded as follows, using the predicate g_{monkey} to guard against ill-typed instantiations of M, M_1, and M_2:

$$\exists M.\ \mathsf{g}_{monkey}(M)$$

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

(The first axiom witnesses the existence of a monkey.) Thanks to monotonicity, it is sound to omit all type information regarding bananas.

Monotonicity is not decidable, but it can often be inferred using suitable calculi. In this report, 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.

In addition, we generalise the monotonicity approach to a rank-1 polymorphic logic, as embodied by the TPTP typed first-order form TFF1 [6]. Unfortunately, the presence of a single equality literal $X \approx t$ or $t \approx X$, 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, as in the traditional encodings. A typical example is the list axiom $\forall X : \alpha,\ Xs : \mathit{list}(\alpha).\ \mathsf{hd}(\mathsf{cons}(X, Xs)) \approx X$. 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. This contribution improves the monomorphic case as well: for the monkey village example, our scheme detects that the first two axioms are harmless and translates them without the g_{monkey} guards. In fact, by relying on a relaxed notion of monotonicity, we can soundly eliminate all type information in the monkey village problem.

Encoding types in an untyped logic is an old problem, and several solutions have been proposed in the literature. We first review four main traditional approaches (Section 3), which prepare the ground for the more advanced encodings presented in this report. Next, we present known and novel monotonicity-based schemes that handle only ground types (Section 4); these are interesting in their own right and serve as stepping stones for the full-blown polymorphic encodings (Section 5). Besides the monotonicity-based encodings, we also present alternative schemes that aim at reducing the clutter associated with polymorphic symbols, based on the novel notion of "cover" (Section 6). Proofs of correctness accompany the descriptions of the new encodings. The proofs explicitly relate models of unencoded and encoded problems.

Figure 1 presents a brief overview of the main encodings. The traditional encodings are identified by single letters (`e`, `a`, `t`, `g`). The nontraditional encodings append a suffix to the letter: `?` (= monotonicity-based, lightweight), `??` (= monotonicity-based, feather-

	Traditional (Polymorphic)	Monotonicity-based Monomorphic	Monotonicity-based Polymorphic	Cover-based (Polymorphic)
Full type erasure	e (§ 3.1)			
Type arguments	a (§ 3.2)		$\mathsf{a}^{\mathsf{ctor}}$ (§ 5.1)	
Type tags	t (§ 3.3)	$\widetilde{\mathsf{t}}$?, $\widetilde{\mathsf{t}}$?? (§ 4.4)	t?, t?? (§ 5.4)	t@ (§ 6.1)
Type guards	g (§ 3.4)	$\widetilde{\mathsf{g}}$?, $\widetilde{\mathsf{g}}$?? (§ 4.5)	g?, g?? (§ 5.5)	g@ (§ 6.2)

Figure 1. Main encodings

weight), or @ (= cover-based). The decoration ~ identifies the monomorphic version of an encoding. Among the nontraditional schemes, $\widetilde{\mathsf{t}}$? and $\widetilde{\mathsf{g}}$? are due to Claessen et al. [14]; the other encodings are novel.

A formalisation [7] of the results in the proof assistant Isabelle/HOL [22] is under way; it currently covers all the monomorphic encodings. The encodings have been implemented in Sledgehammer [3, 21], which provides a bridge between Isabelle/HOL and several automatic theorem provers (Section 7). They were evaluated with E, iProver, SPASS, Vampire, and Z3 on a vast benchmark suite consisting of proof goals from existing Isabelle formalisations (Section 8). Our comparisons include the traditional encodings as well as the provers' native support for monomorphic types where it is available. Related work is considered at the end (Section 9).

2 Background: Logics

This report involves three versions of classical first-order logic with equality: polymorphic, monomorphic, and untyped. They correspond to the TPTP syntaxes TFF1, TFF0, and FOF, respectively, excluding interpreted arithmetic.

2.1 Polymorphic First-Order Logic

The source logic is a rank-1 polymorphic logic as provided by TFF1 [6].

Definition 2.1 (Syntax). Let $\mathcal{A}$ be a countable set of *type variables* with typical element α, and let $\mathcal{V}$ be a countable set of *term variables* with typical element X. A *polymorphic signature* is a triple $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$, where $\mathcal{K}$ is a finite set of n-ary type constructors k with their arities, $\mathcal{F}$ is a finite set of function symbol declarations, and $\mathcal{P}$ is a finite set of predicate symbol declarations. The types, declarations, terms, and formulas are defined below. Symbols may not be overloaded. A *problem* over Σ is a finite set of closed formulas over Σ.

Types:

$\sigma ::= k(\bar{\sigma})$		constructor type
	$\mid \alpha$	type variable

Declarations:

$$\mathsf{f} : \forall \bar{\alpha}.\, \sigma_1 \times \cdots \times \sigma_n \to \sigma \in \mathcal{F}$$
$$\mathsf{p} : \forall \bar{\alpha}.\, \sigma_1 \times \cdots \times \sigma_n \to \mathsf{o} \in \mathcal{P}$$

The type variables $\bar{\alpha}$ in a declaration must be distinct and comprise all type variables found in $\sigma_1, \ldots, \sigma_n$ and σ. Type variables α_i that do not occur in $\sigma_1, \ldots, \sigma_n$ or σ are allowed. The symbols $\times$, $\to$, and o are not type constructors but syntax. Both kinds of declaration are instances of the general syntax $\mathsf{s} : \forall \bar{\alpha}.\, \bar{\sigma} \to \varsigma$, where $\mathsf{s} \in \mathcal{F} \uplus \mathcal{P}$ and ς is either a type or o. An application of s requires $|\bar{\alpha}|$ type arguments in angle brackets and $|\bar{\sigma}|$ term arguments in parentheses. In examples, we usually omit type arguments that are irrelevant or clear from the context.

Terms:

$t ::= \mathsf{f}\langle\bar{\sigma}\rangle(\bar{t})$	function term
$\mid X$	term variable

Formulas:

$\varphi ::= \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t}) \mid \neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})$	predicate literal
$\mid t_1 \approx t_2 \mid t_1 \not\approx t_2$	equality literal
$\mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2$	binary connective
$\mid \forall X : \sigma.\, \varphi \mid \exists X : \sigma.\, \varphi$	term quantification
$\mid \forall \alpha.\, \varphi$	type quantification

Though it is not captured syntactically, a type quantifier may not appear underneath a term quantifier. When defining translations or reasoning about formulas, we assume they are expressed in negation normal form (NNF), with negation applied to equality or predicate atoms. In other contexts, we freely nest quantifiers and connectives and use implication $\to$. We assume that all type quantification is universal—type skolemisation will remove existential type quantifiers [6, § 5.2]. Finally, we adopt the convention that each type or term variable is bound only once in a formula.

The typing rules and semantics of the logic are modelled after those of TFF1 [6, § 3]. Briefly, the type arguments completely determine the types of the term arguments and, for functions, of the result. Polymorphic symbols are interpreted as families of functions or predicates indexed by ground types. All types are inhabited (nonempty).

Definition 2.2 (Typing Rules). Let Γ be a *type context*, a function that maps every variable to its type. A judgement $\Gamma \vdash t : \sigma$ expresses that the term t is *well-typed* and has type σ in context Γ. A judgement $\Gamma \vdash \varphi : \mathrm{o}$ expresses that the formula φ is *well-typed* in Γ. The *typing rules* of polymorphic first-order logic are given below:

$$\frac{}{\Gamma \vdash X : \Gamma(X)} \qquad \frac{\mathsf{f} : \forall \bar{\alpha}.\, \bar{\sigma} \to \sigma \qquad \Gamma \vdash t_j : \sigma_j \rho \ \text{ for all } j}{\Gamma \vdash \mathsf{f}\langle\bar{\alpha}\rho\rangle(\bar{t}) : \sigma\rho}$$

$$\frac{\mathsf{p} : \forall \bar{\alpha}.\, \bar{\sigma} \to \mathrm{o} \qquad \Gamma \vdash t_j : \sigma_j \rho \ \text{ for all } j}{\Gamma \vdash \mathsf{p}\langle\bar{\alpha}\rho\rangle(\bar{t}) : \mathrm{o}} \qquad \frac{\mathsf{p} : \forall \bar{\alpha}.\, \bar{\sigma} \to \mathrm{o} \qquad \Gamma \vdash t_j : \sigma_j \rho \ \text{ for all } j}{\Gamma \vdash \neg\, \mathsf{p}\langle\bar{\alpha}\rho\rangle(\bar{t}) : \mathrm{o}}$$

$$\frac{\Gamma \vdash t_1 : \sigma \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 \approx t_2 : \mathrm{o}} \qquad \frac{\Gamma \vdash t_1 : \sigma \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 \not\approx t_2 : \mathrm{o}} \qquad \frac{\Gamma \vdash \varphi_1 : \mathrm{o} \quad \Gamma \vdash \varphi_2 : \mathrm{o}}{\Gamma \vdash \varphi_1 \wedge \varphi_2 : \mathrm{o}}$$

$$\frac{\Gamma \vdash \varphi_1 : \mathrm{o} \quad \Gamma \vdash \varphi_2 : \mathrm{o}}{\Gamma \vdash \varphi_1 \vee \varphi_2 : \mathrm{o}} \qquad \frac{\Gamma[X \mapsto \sigma] \vdash \varphi : \mathrm{o}}{\Gamma \vdash \forall X : \sigma.\, \varphi : \mathrm{o}} \qquad \frac{\Gamma[X \mapsto \sigma] \vdash \varphi : \mathrm{o}}{\Gamma \vdash \exists X : \sigma.\, \varphi : \mathrm{o}} \qquad \frac{\Gamma \vdash \varphi : \mathrm{o}}{\Gamma \vdash \forall \alpha.\, \varphi : \mathrm{o}}$$

Superscripts attach type annotations to terms; for example, t^σ indicates that t has type σ. These annotations are a notational device and not part of the syntax.

Definition 2.3 (Semantics). Let $\mathbb{D}$ be a fixed nonempty collection of nonempty sets (the *domains*), and let $\mathbb{U} = \bigcup \mathbb{D}$ (the *universe*). Let $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ be a polymorphic signature. A *structure* $\mathcal{S}$ for Σ is a triple of families:

- $(k^{\mathcal{S}})_{k \in \mathcal{K}} : \mathbb{D}^n \to \mathbb{D}$;
- $(\mathsf{f}^{\mathcal{S}})_{\mathsf{f}:\forall \bar{\alpha}.\, \bar{\sigma} \to \sigma \in \mathcal{F}} : \mathbb{D}^{|\bar{\alpha}|} \times \mathbb{U}^{|\bar{\sigma}|} \to \mathbb{U}$;
- $(\mathsf{p}^{\mathcal{S}})_{\mathsf{p}:\forall \bar{\alpha}.\, \bar{\sigma} \to \mathsf{o} \in \mathcal{P}} \subseteq \mathbb{D}^{|\bar{\alpha}|} \times \mathbb{U}^{|\bar{\sigma}|}$.

Given a *type variable valuation* $\theta : \mathcal{A} \to \mathbb{D}$, the *interpretation* of types $[\![\ ]\!]^{\mathcal{S}}_{\theta}$ is defined by the equations

$$[\![k(\bar{\sigma})]\!]^{\mathcal{S}}_{\theta} = k^{\mathcal{S}}([\![\bar{\sigma}]\!]^{\mathcal{S}}_{\theta}) \qquad\qquad [\![\alpha]\!]^{\mathcal{S}}_{\theta} = \theta(\alpha)$$

The $\mathsf{p}^{\mathcal{S}}$ component of a structure is required to map any tuple of $|\bar{\alpha}|$ domains D_i and $|\bar{\sigma}|$ universe elements $a_j \in [\![\sigma_j]\!]^{\mathcal{S}}_{\theta}$ to an element of $[\![\sigma]\!]^{\mathcal{S}}_{\theta}$, where θ maps each α_i to D_i.

Given a type variable valuation θ and a *term variable valuation* $\xi : \mathcal{V} \to \mathbb{U}$, the *interpretation* of terms and formulas by the structure $\mathcal{S}$ is as follows:

$$\begin{aligned}
[\![\mathsf{f}\langle\bar{\sigma}\rangle(\bar{t})]\!]^{\mathcal{S}}_{\theta,\xi} &= f^{\mathcal{S}}([\![\bar{\sigma}]\!]^{\mathcal{S}}_{\theta}, [\![\bar{t}]\!]^{\mathcal{S}}_{\theta,\xi}) & [\![X]\!]^{\mathcal{S}}_{\theta,\xi} &= \xi(X)\\
[\![\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]^{\mathcal{S}}_{\theta,\xi} &= p^{\mathcal{S}}([\![\bar{\sigma}]\!]^{\mathcal{S}}_{\theta}, [\![\bar{t}]\!]^{\mathcal{S}}_{\theta,\xi}) & [\![t_1 \approx t_2]\!]^{\mathcal{S}}_{\theta,\xi} &= ([\![t_1]\!]^{\mathcal{S}}_{\theta,\xi} = [\![t_2]\!]^{\mathcal{S}}_{\theta,\xi})\\
[\![\neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]^{\mathcal{S}}_{\theta,\xi} &= \neg\, p^{\mathcal{S}}([\![\bar{\sigma}]\!]^{\mathcal{S}}_{\theta}, [\![\bar{t}]\!]^{\mathcal{S}}_{\theta,\xi}) & [\![t_1 \not\approx t_2]\!]^{\mathcal{S}}_{\theta,\xi} &= ([\![t_1]\!]^{\mathcal{S}}_{\theta,\xi} \neq [\![t_2]\!]^{\mathcal{S}}_{\theta,\xi})\\
[\![\varphi_1 \wedge \varphi_2]\!]^{\mathcal{S}}_{\theta,\xi} &= [\![\varphi_1]\!]^{\mathcal{S}}_{\theta,\xi} \wedge [\![\varphi_2]\!]^{\mathcal{S}}_{\theta,\xi} & [\![\forall X\!:\!\sigma.\, \varphi]\!]^{\mathcal{S}}_{\theta,\xi} &= \forall a \in [\![\sigma]\!]^{\mathcal{S}}_{\theta}.\, [\![\varphi]\!]^{\mathcal{S}}_{\theta,\xi[X \mapsto a]}\\
[\![\varphi_1 \vee \varphi_2]\!]^{\mathcal{S}}_{\theta,\xi} &= [\![\varphi_1]\!]^{\mathcal{S}}_{\theta,\xi} \vee [\![\varphi_2]\!]^{\mathcal{S}}_{\theta,\xi} & [\![\exists X\!:\!\sigma.\, \varphi]\!]^{\mathcal{S}}_{\theta,\xi} &= \exists a \in [\![\sigma]\!]^{\mathcal{S}}_{\theta}.\, [\![\varphi]\!]^{\mathcal{S}}_{\theta,\xi[X \mapsto a]}\\
[\![\forall \alpha.\, \varphi]\!]^{\mathcal{S}}_{\theta,\xi} &= \forall D \in \mathbb{D}.\, [\![\varphi]\!]^{\mathcal{S}}_{\theta[\alpha \mapsto D],\xi}
\end{aligned}$$

We omit irrelevant subscripts to $[\![\ ]\!]$, writing $[\![\sigma]\!]^{\mathcal{S}}$ if σ is ground and $[\![\varphi]\!]^{\mathcal{S}}$ if φ is closed.

A structure $\mathcal{M}$ is a *model* of a problem Φ if $[\![\varphi]\!]^{\mathcal{M}}$ is true for every $\varphi \in \Phi$. A problem that has a model is *satisfiable*.

Example 2.4 (Algebraic Lists). The following axioms induce a minimalistic first-order theory of algebraic lists that will serve as our main running example:

$$\begin{aligned}
&\forall \alpha.\, \forall X\!:\!\alpha,\, Xs\!:\!\mathit{list}(\alpha).\ \mathsf{nil} \not\approx \mathsf{cons}(X, Xs)\\
&\forall \alpha.\, \forall Xs\!:\!\mathit{list}(\alpha).\ Xs \approx \mathsf{nil} \vee (\exists Y\!:\!\alpha,\, Ys\!:\!\mathit{list}(\alpha).\ Xs \approx \mathsf{cons}(Y, Ys))\\
&\forall \alpha.\, \forall X\!:\!\alpha,\, Xs\!:\!\mathit{list}(\alpha).\ \mathsf{hd}(\mathsf{cons}(X, Xs)) \approx X \wedge \mathsf{tl}(\mathsf{cons}(X, Xs)) \approx Xs
\end{aligned}$$

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.

Central to this report are the notions of soundness and completeness of a translation function between problems.

Definition 2.5 (Correctness). Assume a function that translates problems Φ over Σ to problems Φ' over Σ'. The function is *sound* if satisfiability of Φ implies satisfiability of Φ'; it is *complete* if satisfiability of Φ' implies satisfiability of Φ; it is *correct* if it is both sound and complete (i.e. Φ and Φ' are equisatisfiable).

2.2 Monomorphic First-Order Logic

Monomorphic first-order logic constitutes a special case of polymorphic first-order logic. Type constructors are nullary, symbol declarations have the form $\mathsf{s} : \bar{\sigma} \to \varsigma$, and formulas contain no type variables.

Example 2.6. A monomorphised version of the algebraic list problem of Example 2.4, with α instantiated by b, follows:

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

2.3 Untyped First-Order Logic

Our target logic is an untyped logic coinciding with the TPTP first-order form FOF [27]. This is the logic traditionally implemented in automatic theorem provers. An *untyped signature* is a pair $\Sigma = (\mathcal{F}, \mathcal{P})$, where $\mathcal{F}$ and $\mathcal{P}$ associate symbols with their arities (indicated by superscripts). The untyped syntax is identical to that of the monomorphic logic, except that quantification is written $\forall X.\ \varphi$ and $\exists X.\ \varphi$.

3 Traditional Type Encodings

There are four main traditional approaches to encoding polymorphic types: full type erasure, type arguments, type tags, and type guards [17, 21, 26, 32].

3.1 Full Type Erasure

The easiest way to translate a typed problem into an untyped logic is to erase all its type information, which means omitting all type arguments, type quantifiers, and types in term quantifiers. We call this encoding `e`.

Definition 3.1 (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 on terms and formulas by the following structurally recursive function $[\![\ ]\!]_{\mathsf{e}}$:

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

Here and elsewhere, we omit the trivial cases where the function is simply applied to its subterms or subformulas, as in $[\![\varphi_1 \wedge \varphi_2]\!]_{\mathsf{e}} = [\![\varphi_1]\!]_{\mathsf{e}} \wedge [\![\varphi_2]\!]_{\mathsf{e}}$. The translation of a problem Φ is the union of the translations its formulas: $[\![\Phi]\!]_{\mathsf{e}} = \bigcup_{\varphi\in\Phi} [\![\varphi]\!]_{\mathsf{e}}$.

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

Example 3.2. Encoded using e, the monkey village axioms of Example 1.1 become

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

Like the original axioms, the encoded axioms are satisfiable: the requirement that each monkey possesses two bananas of its own can be met by taking an infinite domain (since $2k = k$ for any infinite cardinal k).

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; from any disequality $t \not\approx u$ or any pair of clauses $\mathsf{p}(\bar{t})$ and $\neg\ \mathsf{p}(\bar{u})$, we can derive a contradiction. An expedient proposed by Meng and Paulson [21, § 2.8] and implemented in Sledgehammer is to filter out all axioms of the form $\forall X\!:\!\sigma.\ X \approx \mathsf{a}_1 \vee \cdots \vee X \approx \mathsf{a}_n$, but this makes the translation incomplete and generally does not suffice to prevent unsound cardinality reasoning.

An additional issue with full type erasure is that it 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 less contrived example is $N \not\approx 0 \rightarrow N > 0$, which we would expect to hold for the natural number versions of 0 and $>$ but not for integers or real numbers.

Nonetheless, full type erasure is complete, and this property will be useful later.

Theorem 3.3 (Completeness of e). *Full type erasure is complete.*

Proof. From a model $\mathcal{M}$ of $[\![\Phi]\!]_{\mathsf{e}}$, we construct a structure for Φ by taking the same domain for all types and interpreting all instances of each polymorphic symbol in the same way as $\mathcal{M}$. This construction clearly yields a model of Φ. □

3.2 Type Arguments

A way to prevent the confusion arising with full type erasure is to encode types as terms in the untyped logic: type variables α become term variables A, and n-ary type constructors k become n-ary function symbols k. A polymorphic symbol with m type arguments is passed m additional term arguments. The example given in the previous subsection is translated to $\mathsf{q}(\mathsf{a}, \mathsf{f}(\mathsf{a})) \wedge \neg\, \mathsf{q}(\mathsf{b}, \mathsf{f}(\mathsf{b}))$, and a fully polymorphic instance $\mathsf{f}\langle\alpha\rangle$ would be encoded as $\mathsf{f}(A)$. We call this encoding $\mathtt{a}$.

Definition 3.4 (Term Encoding of Types). Let $\mathcal{K}$ be a finite set of n-ary type constructors, and let $\vartheta \notin \mathcal{K}$ be a distinguished nullary type constructor. For each type variable α, let $\mathcal{V}(\alpha)$ be a fresh term variable. The *term encoding* of a polymorphic type over $\mathcal{K}$ is a term over $(\{\vartheta\}, \mathcal{K})$, where $\mathsf{k} : \vartheta^n \to \vartheta$ for each n-ary type constructor k. The encoding is specified by the following equations:

$$\langle\!\langle k(\bar{\sigma})\rangle\!\rangle = \mathsf{k}(\langle\!\langle\bar{\sigma}\rangle\!\rangle) \qquad\qquad \langle\!\langle\alpha\rangle\!\rangle = \mathcal{V}(\alpha)$$

Definition 3.5 (Traditional Arguments $\mathtt{a}$). The *traditional type arguments* encoding $\mathtt{a}$ 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 $|\bar{\alpha}| + |\bar{\sigma}|$. The encoding is defined as $[\![\ ]\!]_{\mathtt{a};\mathtt{e}}$, where

$$\begin{aligned}
[\![\mathsf{f}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathtt{a}} &= \mathsf{f}\langle\bar{\sigma}\rangle(\langle\!\langle\bar{\sigma}\rangle\!\rangle, [\![\bar{t}]\!]_{\mathtt{a}}) \\
[\![\mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathtt{a}} &= \mathsf{p}\langle\bar{\sigma}\rangle(\langle\!\langle\bar{\sigma}\rangle\!\rangle, [\![\bar{t}]\!]_{\mathtt{a}}) \qquad\qquad [\![\forall\alpha.\, \varphi]\!]_{\mathtt{a}} = \forall\alpha.\, \forall\langle\!\langle\alpha\rangle\!\rangle : \vartheta.\, [\![\varphi]\!]_{\mathtt{a}} \\
[\![\neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\bar{t})]\!]_{\mathtt{a}} &= \neg\, \mathsf{p}\langle\bar{\sigma}\rangle(\langle\!\langle\bar{\sigma}\rangle\!\rangle, [\![\bar{t}]\!]_{\mathtt{a}})
\end{aligned}$$

(Again, we omit the trivial cases, e.g. $[\![\forall X : \sigma.\, \varphi]\!]_{\mathtt{a}} = \forall X : \sigma.\, [\![\varphi]\!]_{\mathtt{a}}$.)

Example 3.6. The $\mathtt{a}$ encoding translates the algebraic list problem of Example 2.4 into the following untyped problem:

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

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

Definition 3.7 (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 the distinguished symbols $\mathsf{t}, \mathsf{g} \notin \mathcal{F} \uplus \mathcal{P}$.

Definition 3.8 (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

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

The e and a encodings correspond to the special cases of a^x where x returns none or all of its arguments, respectively.

Theorem 3.9 (Completeness of a^x). *The type arguments encoding a^x is complete.*

Proof. By Theorem 3.3, it suffices to construct a model $\mathcal{M}'$ of Φ from a model $\mathcal{M}$ of $[\![\Phi]\!]_{\mathsf{a}^x}$. We let $\mathcal{M}'$ interpret each type constructor in $\mathcal{K}$ in the same way as $\mathcal{M}$. For each symbol $\mathsf{s} \in \mathcal{F} \uplus \mathcal{P}$, the entry for $\mathsf{s}\langle\bar{\sigma}\rangle(\bar{a})$ in $\mathcal{M}'$ is given by the interpretation of $\mathsf{s}\langle\bar{\sigma}\rangle(\langle\!\langle x_\mathsf{s}(\bar{\sigma})\rangle\!\rangle, \bar{a})$ in $\mathcal{M}$. This construction obviously yields a model of Φ. □

3.3 Type Tags

An intuitive approach to encode type information soundly (as well as completely) 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. We call this scheme t, after the tag function of the same name. It 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 3.10 (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 3.11 (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 3.12. The t encoding translates the algebraic list problem of Example 2.4 into the following equisatisfiable untyped problem:

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

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

Since there are no phantoms in this example, all type information is carried by the t function's first argument.

3.4 Type Guards

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 $\mathsf{nil}\langle\alpha\rangle$ (which has no term arguments) but omitted for $\mathsf{cons}\langle\alpha\rangle(X, Xs)$, $\mathsf{hd}\langle\alpha\rangle(Xs)$, and $\mathsf{tl}\langle\alpha\rangle(Xs)$.

Definition 3.13 (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 3.14 (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*:

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

The last axiom witnesses inhabitation of every type. It is necessary for completeness, in case some of the types do not appear in the result type of any function symbol.

Example 3.15. The g encoding translates the algebraic list problem of Example 2.4 into the following:

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

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

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

In this and later examples, we push guards inside past quantifiers and group them in a conjunction to enhance readability.

The typing axioms must in general be guarded. Without the guard, any cons, hd, or tl term could be typed with any type, defeating the purpose of the guard predicates.

4 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 5.

Before we start, let us define variants of the traditional t and g encodings that operate on monomorphic problems. The monomorphic encodings $\widetilde{\mathsf{t}}$ and $\widetilde{\mathsf{g}}$ coincide with t and g except that the polymorphic function $\mathsf{t}\langle\sigma\rangle(t)$ and predicate $\mathsf{g}\langle\sigma\rangle(t)$ are replaced by type-indexed families of unary functions $\mathsf{t}_\sigma(t)$ and predicates $\mathsf{g}_\sigma(t)$, as is customary in the literature [32, § 4].

4.1 Monotonicity

The concept of monotonicity used by Claessen et al. [14, § 2.2] is defined below.

Definition 4.1 (Finite Monotonicity). Let σ be a ground type and Φ be a problem. The type σ is *finitely monotonic* in Φ if for all models $\mathcal{M}$ of Φ such that $[\![\sigma]\!]^{\mathcal{M}}$ is finite, there exists a model $\mathcal{M}'$ where $\big|[\![\sigma]\!]^{\mathcal{M}'}\big| = \big|[\![\sigma]\!]^{\mathcal{M}}\big| + 1$ and $\big|[\![\tau]\!]^{\mathcal{M}'}\big| = \big|[\![\tau]\!]^{\mathcal{M}}\big|$ for all $\tau \neq \sigma$. The problem Φ is *finitely monotonic* if all its types are monotonic.

We propose a stronger criterion, infinite monotonicity, that considers sets of types.

Definition 4.2 (Infinite Monotonicity). Let S be a set of ground types and Φ be a problem. The set S is (*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.

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.

Example 4.3. The monkey village of Example 1.1 is infinitely monotonic because any model with finitely many bananas can be extended to a model with infinitely many, and any model with infinitely many bananas and finitely many monkeys can be extended to one where monkeys and bananas have the same infinite cardinality (cf. Example 3.2).

The next result allows us to focus on infinite monotonicity.

Theorem 4.4 (Subsumption of Finite Monotonicity). *Let σ be ground type and Φ be a problem. If σ is finitely monotonic in Φ, then σ is infinitely monotonic in Φ.*

We need to introduce a few lemmas before we can reestablish the key result of Claessen et al. for our notion of monotonicity.[1]

Lemma 4.5 (Same-Cardinality Erasure). *Let Φ be a monomorphic problem. If Φ has a model where all domains have cardinality k, $[\![\Phi]\!]_{\mathsf{e}}$ has a model where the unique domain has cardinality k.*

Proof. See either Theorem 1 in Bouillaguet et al. [12, § 4] or Lemma 1 in Claessen et al. [14, § 1]. □

Lemma 4.6 (Downward Löwenheim–Skolem). *If a problem Φ has a model where all domains are infinite, it also has a model where all domains are countably infinite.*

Theorem 4.7 (Monotonic Erasure). *Full type erasure is sound for monotonic monomorphic problems.*

Proof. Let Φ be such a problem. If Φ is satisfiable, Φ has a model where all domains are infinite by definition of monotonicity. By Lemma 4.6, it also has a model where all domains are countably infinite. Hence, $[\![\Phi]\!]_{\mathsf{e}}$ is satisfiable by Lemma 4.5. □

The definition in terms of sets of types is what makes infinite monotonicity more powerful than finite monotonicity, but it is often more convenient to focus on single types and combine the results. The following lemma permit precisely such combinations.

Lemma 4.8 (Monotonicity Preservation by Union). *Let $\mathcal{S}$ be a set of sets of ground types and Φ be a problem. If every $S \in \mathcal{S}$ is monotonic in Φ, then $\bigcup \mathcal{S}$ is monotonic in Φ.*

4.2 Monotonicity Inference

Claessen et al. introduced a simple calculus to infer finite monotonicity for monomorphic first-order logic [14, § 2.3]. The definition below generalises it from clause normal form to negation normal form. The generalisation is straightforward; we present it because we later adapt it to polymorphism. 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.

[1] Another reason for reproving the result is that their proof relies on a flawed statement of the Löwenheim–Skolem theorem for monomorphic first-order logic (their Lemma 3).

Definition 4.9 (Naked Variable). The set of *naked variables* $\mathrm{NV}(\varphi)$ of a formula φ is defined as follows:

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

Notice the equation for the $\exists$ case: existential variables are never naked.

Claessen et al. designed a second, more powerful calculus to detect predicates that act as guards for naked variables. Whilst the calculus proved successful on a subset of the TPTP benchmarks [27], we assessed its suitability on about 1000 problems generated by Sledgehammer and found no improvement on the simple calculus. We restrict our attention to the first calculus.

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, such as X on the right-hand side of the equation $\mathsf{hd}_b(\mathsf{cons}_b(X, Xs)) \approx X$ from Example 2.6, will cause us to classify σ as possibly non-monotonic.

We regain some precision by extending the calculus with an infinity analysis, as suggested by Claessen et al. as future work: trivially, all types with no finite models are monotonic. Abstracting over the specific analysis used to detect infinite types (e.g. Infinox [13]), 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 4.10 (Monotonicity Calculus $\rhd$). Let Φ be a monomorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$. 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:

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

$\mathrm{Inf}(\Phi)$ is a set of ground types over $\mathcal{K}$ (the *infinite types*) which all models of Φ interpret as infinite sets; We write $\sigma \rhd \varphi$ to indicate that the judgement is derivable and $\sigma \not\rhd \varphi$ if it is not derivable. These notations are lifted to problems Φ, with $\sigma \rhd \Phi$ if and only if $\sigma \rhd \varphi$ for all $\varphi \in \Phi$.

Theorem 4.11 (Soundness of $\rhd$). *Let Φ be a monomorphic problem. If $\sigma \rhd \Phi$, then σ is monotonic in Φ.*

Proof. Given a model $\mathcal{M}$ where σ is interpreted as D, we construct a model $\mathcal{M}'$ where σ is interpreted as the infinite set $D \uplus D'$. If $D' \neq \emptyset$, we choose an arbitrary representative element $a \in D$ and extend the interpretations of all functions and predicates so that they coincide on a and each $a' \in D'$. For problems with no equality over σ, $\mathcal{M}'$ is obviously a model. Equality atoms between non-variables $\mathsf{f}(\ldots) \approx \mathsf{g}(\ldots)$ do not compromise the construction, because the new element a' cannot be in the functions' ranges. Nor do

negative equality literals pose any difficulties, because although the cases $a \not\approx a'$, $a' \not\approx a$, and $a' \not\approx a'$ are possible by instantiating naked variables with a', they are weaker than the cases where the same variables are instantiated with a, which themselves must be true for $\mathcal{M}$ to be a model. □

Even though monotonicity is defined on sets of types, we allow ourselves to write that σ is monotonic if $\sigma \rhd \Phi$ and possibly nonmonotonic if $\sigma \not\rhd \Phi$.

4.3 Encoding Nonmonotonic Types

Monotonic types can be soundly erased when translating to untyped first-order logic, by Theorem 4.7. Nonmonotonic types in general cannot. Claessen et al. [14, § 3.2] 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) Infer monotonicity to identify the possibly nonmonotonic types in the problem.
   (b) Introduce protectors for universal variables of possibly nonmonotonic types.
   (c) 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 following lemmas will help us prove such two-stage encodings correct.

Lemma 4.12 (Correctness Conditions). *Let Φ be a monomorphic problem, and let x be a monomorphic encoding. The problems Φ and $[\![\Phi]\!]_{x;\mathsf{e}}$ are equisatisfiable provided the following conditions hold:*

MONO: *$[\![\Phi]\!]_x$ is monotonic.*

SOUND: *If Φ is satisfiable, so is $[\![\Phi]\!]_x$.*

COMPLETE: *If $[\![\Phi]\!]_x$ is satisfiable, so is Φ.*

Proof. Immediate from Theorems 3.3 and 4.7. □

Lemma 4.13 (Submodel). *Let Φ be a problem, let $\mathcal{M}$ be a model of Φ, and let $\mathcal{M}'$ be a substructure of $\mathcal{M}$ (i.e. a structure constructed from $\mathcal{M}$ by removing some domain elements while leaving the interpretations of functions and predicates intact for the remaining elements). This $\mathcal{M}'$ is a model of Φ provided that it does not remove any witness for an existential variable.*

4.4 Monotonicity-Based Type Tags

The monotonicity-based encoding $\widetilde{\mathtt{t}}?$ specialises this procedure for tags. It is similar to the traditional encoding $\widetilde{\mathtt{t}}$ (the monomorphic version of $\mathtt{t}$), 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 4.14 (Lightweight Tags $\widetilde{\mathtt{t}}?$). The *monomorphic lightweight type tags* encoding $\widetilde{\mathtt{t}}?$ translates a monomorphic problem Φ over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \{\mathtt{t}^1_\sigma \mid \sigma \text{ is ground over } \mathcal{K}\}, \mathcal{P}')$, where $\mathcal{F}', \mathcal{P}'$ are as for $\mathtt{e}$. It is defined as $[\![\,]\!]_{\widetilde{\mathtt{t}}?;\mathtt{e}}$, where

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

Example 4.15. For the algebraic list problem of Example 2.6, the type *list_b* is monotonic by virtue of being infinite, whereas b cannot be inferred monotonic. The $\widetilde{\mathtt{t}}?$ encoding of the problem follows:

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

The $\widetilde{\mathtt{t}}?$ 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.[2]

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

Definition 4.16 (Featherweight Tags $\widetilde{\mathtt{t}}??$). The *monomorphic featherweight type tags* encoding $\widetilde{\mathtt{t}}??$ translates a monomorphic problem Φ over Σ into an untyped problem over Σ', where Σ' is as for $\widetilde{\mathtt{t}}?$. It is defined as $[\![\,]\!]_{\widetilde{\mathtt{t}}??;\mathtt{e}}$, where

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

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

with

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

The encoding is complemented by typing axioms:

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

[2] This is related to the observation that only paramodulation from or into a variable can cause ill-typed instantiations in a resolution prover [32, § 4].

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.

Example 4.17. The $\widetilde{\mathtt{t}}??$ encoding of Example 2.6 requires fewer tags than $\widetilde{\mathtt{t}}?$, at the cost of more type information (for hd and the existential variables of type b):

$$\forall Xs.\ \mathtt{t}_b(\mathsf{hd}_b(Xs)) \approx \mathsf{hd}_b(Xs)$$

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

Theorem 4.18 (Correctness of $\widetilde{\mathtt{t}}?$, $\widetilde{\mathtt{t}}??$). *The monomorphic type tags encodings* $\widetilde{\mathtt{t}}?$ *and* $\widetilde{\mathtt{t}}??$ *are correct.*

Proof. It suffices to show the three conditions of Lemma 4.12.

MONO: By Theorem 4.11, types are monotonic unless they are possibly finite and variables of their types occur naked in the original problem. Both encodings protect all such variables—$\widetilde{\mathtt{t}}??$ tags exactly these variables, while $\widetilde{\mathtt{t}}?$ tags even more terms. The typing axioms contain no naked variables. Since all types are monotonic, the encoded problem is monotonic by Lemma 4.8.

SOUND: Given a model of Φ, we extend it to a model of $[\![\Phi]\!]_x$ by interpreting all type tags as the identity.

COMPLETE: A model of $[\![\Phi]\!]_x$ is *canonical* if all tag functions $\mathtt{t}_\sigma$ are interpreted as the identity. From a canonical model, we obtain a model of Φ by leaving out $\mathtt{t}_\sigma$. It then suffices to prove that whenever there exists a model $\mathcal{M}$ of $[\![\Phi]\!]_x$, there exists a canonical model $\mathcal{M}'$.

For $\widetilde{\mathtt{t}}?$, values of a tagged type σ are systematically accessed through $\mathtt{t}_\sigma$. Hence, we can safely permute the entries of the function table of each $\mathtt{t}_\sigma$ so that it is the identity for the values in its range. We then construct $\mathcal{M}'$ by removing the domain elements for which $\mathtt{t}_\sigma$ is not the identity. It is a model by Lemma 4.13.

For $\widetilde{\mathtt{t}}??$, the construction must take possibly nonmonotonic types into account. No permutation is necessary for these thanks to the typing axioms, which ensure that the tag functions are the identity for well-typed terms. For each $\sigma \not\vdash \Phi$, we remove the model elements for which $\mathtt{t}_\sigma$ is not the identity. The typing axioms ensure that the substructure is well-defined: each tag function is the identity for at least one element and also for each element within the range of a non-tag function. The equations $\mathtt{t}_\sigma(X) \approx X$ generated for existential variables ensure that witnesses are preserved, as required by Lemma 4.13.

□

The above proof goes through even if the generated problems contain more tags than are necessary to ensure monotonicity. Moreover, we may add further typing axioms to $[\![\Phi]\!]_x$—for example, equations $\mathsf{f}(\overline{U}, \mathtt{t}_\sigma(X), \overline{V}) \approx \mathsf{f}(\overline{U}, X, \overline{V})$ to add or remove tags around well-typed arguments of a symbol f, or the idempotence law $\mathtt{t}_\sigma(\mathtt{t}_\sigma(X)) \approx \mathtt{t}_\sigma(X)$—provided that they hold for canonical models and preserve monotonicity.

4.5 Monotonicity-Based Type Guards

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 4.19 (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 $\Sigma' = (\mathcal{F}', \mathcal{P}' \uplus \{\mathsf{g}^1_\sigma \mid \sigma \text{ is ground over } \mathcal{K}\})$, where $\mathcal{F}'$, $\mathcal{P}'$ are as for $\mathtt{e}$. It is defined as $[\![\,]\!]_{\widetilde{\mathsf{g}}?;\mathtt{e}}$, 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:

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

Example 4.20. The $\widetilde{\mathsf{g}}?$ encoding of Example 2.6 is as follows:

$$\forall Xs.\ \mathsf{g}_b(\mathsf{hd}_b(Xs))$$

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

Notice that the tl_b equation is needlessly in the scope of the guard. The encoding is more precise if the problem is clausified.

By leaving Xs unconstrained, the typing axiom gives a type to some ill-typed terms, such as $\mathsf{hd}_b(0^{nat})$. Intuitively, this is safe because such terms cannot be used to prove anything useful that could not be proved by a well-typed typed. What matters is that well-typed terms are associated with their correct type.

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 4.21 (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 4.22. 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 4.23 (Correctness of $\widetilde{\mathsf{g}}?$, $\widetilde{\mathsf{g}}??$). *The monomorphic type guards encodings* $\widetilde{\mathsf{g}}?$ *and* $\widetilde{\mathsf{g}}??$ *are correct.*

Proof. It suffices to show the three conditions of Lemma 4.12.

MONO: By Theorem 4.11, types are monotonic unless they are possibly finite and variables of their types occur naked in the original problem. Both encodings guard all such variables—$\widetilde{\mathsf{g}}??$ guards exactly those variables, while $\widetilde{\mathsf{g}}?$ guards more. The typing axioms contain no naked variables. We cannot use Theorem 4.11 directly, because guarding a naked variable does not make it less naked, but we can generalise the proof slightly to exploit the guards. Given a model $\mathcal{M}$ of $[\![\Phi]\!]_x$ where each σ is interpreted as D_σ, we construct a model $\mathcal{M}'$ of $[\![\Phi]\!]_x$ where σ is interpreted as $D_\sigma \uplus D'_\sigma$. We choose a representative $a \in D_\sigma$ and extend the interpretations of all symbols so that they coincide on a and each $a' \in D'_\sigma$ except that $\mathsf{g}_\sigma(a')$ is interpreted as false. This is compatible with the typing axioms (since a' never occurs in the range of a function) and effectively prevents a' from arising naked.

SOUND: Given a model of Φ, we extend it to a model of $[\![\Phi]\!]_x$ by interpreting all type guards as the true predicate (the predicate that is true everywhere).

COMPLETE: A model of $[\![\Phi]\!]_x$ is *canonical* if all guards are interpreted as the true predicate. From a canonical model, we obtain a model of Φ by leaving out g_σ. It then suffices to prove that whenever there exists a model $\mathcal{M}$ of $[\![\Phi]\!]_x$, there exists a canonical model $\mathcal{M}'$. We construct $\mathcal{M}'$ by removing the domain elements from $\mathcal{M}$ that do not satisfy their guard predicate. The typing axioms ensure that the substructure is well-defined: each guard predicate is satisfied by at least one element and each function is satisfy its predicate. The guards $\mathsf{g}_\sigma(X)$ generated for existential variables ensure that witnesses are preserved, as required by Lemma 4.13. □

A simpler but less instructive way to prove MONO is to observe that the second monotonicity calculus by Claessen et al. [14, § 2.4] can infer monotonicity of all problems generated by $\widetilde{\mathsf{g}}$, $\widetilde{\mathsf{g}}?$, and $\widetilde{\mathsf{g}}??$.

4.6 Finite Monomorphisation

Section 5 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* on the polymorphic problem:

1. Heuristically instantiate all type variables with suitable ground types, taking as many copies of the formulas as desired.
2. Map each ground occurrence $\mathsf{s}\langle\bar{\alpha}\rho\rangle$ of a polymorphic symbol $\mathsf{s} : \forall\bar{\alpha}.\, \bar{\sigma} \to \sigma$ to a fresh monomorphic symbol $\mathsf{s}_{\bar{\alpha}\rho} : \lfloor\bar{\sigma}\rho\rfloor \to \lfloor\sigma\rho\rfloor$, where ρ is a ground type substitution (a function from type variables to ground types) and $\lfloor\ \rfloor$ is an injection from ground types to nullary type constructors (e.g. $\{b \mapsto b, \mathit{list}(b) \mapsto \mathit{list\_b}\}$).

Finite monomorphisation is generally incomplete [10, § 2] and often overlooked in the literature, but by eliminating type variables it considerably simplifies the generated formulas, leading to very efficient encodings. It also provides a simple and effective way to exploit the native support for monomorphic types in some automatic provers.

5 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, the type encodings, and the proofs from the previous section to a polymorphic setting.

5.1 Type Arguments to Constructors

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 3.13. 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.

Definition 5.1 (Constructor). Given a polymorphic problem Φ over $\Sigma = (\mathcal{K},\mathcal{F},\mathcal{P})$, a set of *constructors* for Φ consists of symbols $\mathsf{c} : \forall\bar{\alpha}.\ \bar{\sigma} \to k(\bar{\alpha}) \in \mathcal{F}$ such that the injectivity law

$$\forall \bar{X}, \bar{Y}.\ \mathsf{c}(\bar{X}) \approx \mathsf{c}(\bar{Y}) \to \bar{X} \approx \bar{Y}$$

holds in all models of Φ, and for each pair of distinct constructors c, d whose result type involves the same type constructor, distinctness also hold:

$$\forall \bar{X}, \bar{Y}.\ \mathsf{c}(\bar{X}) \not\approx \mathsf{d}(\bar{Y})$$

Abstracting over the specific analysis used to detect constructors, we fix a set of constructors $\mathrm{Ctor}(\Phi) \subseteq \mathcal{F}$.

Definition 5.2 (Constructor-Needed Type Argument). Let Φ be a polymorphic problem over $\Sigma = (\mathcal{K},\mathcal{F},\mathcal{P})$, and let $\mathsf{s} : \forall\alpha_1,\ldots,\alpha_m.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. The ith type argument is *constructor-needed* if either $\mathsf{s} \notin \mathrm{Ctor}(\Phi)$ or α_i is inferable. Given a list $\bar{z}$, $\mathsf{ctor}_{\mathsf{s}}(\bar{z})$ denotes the sublist corresponding to the constructor-needed type arguments.

The notion of constructor-needed type arguments induces, via Definition 3.8, a type encoding $\mathsf{a}^{\mathsf{ctor}}$ that is lighter than a and that enjoys interesting properties.

Example 5.3. The $\mathsf{a}^{\mathsf{ctor}}$ encoding of the polymorphic algebraic list problem of Example 2.4 is identical to the a encoding (Example 3.6) except that nil replaces $\mathsf{nil}(A)$.

5.2 Monotonicity

Definition 4.2 and Lemmas 4.6, 4.13, and 4.8 cater for polymorphic problems. Other results from Section 4 must be adapted to the polymorphic case.

Definition 5.4 (Monotonicity of Polymorphic Type). A polymorphic type is (*infinitely*) *monotonic* if all its ground instances are monotonic.

Theorem 5.5 (Monotonic Erasure). *The constructor-needed type arguments encoding* $\mathsf{a}^{\mathsf{ctor}}$ *is sound for monotonic polymorphic problems.*

Proof. Let Φ be such a problem, and let $\mathcal{M}$ be a model of Φ. By Lemma 4.6, we may assume that all types in $\mathcal{M}$ are interpreted by the same infinite domain D. We first adjust the model $\mathcal{M}$ to synchronise the interpretation of the constructors, so that constructor instances whose inferable type arguments coincide have semantics that coincide. The definition of inferable type arguments, together with injectivity, ensures that the considered interpretations are isomorphic. From there, we can permute each type's domain elements to obtain true coincidence, proceeding one constructor at a time and relying on the distinctness property to ensure that $\mathcal{M}$ remains a model.

From this adjusted model $\mathcal{M}$, we construct a model $\mathcal{M}'$ of $[\![\Phi]\!]_{\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$ that interprets the encoded types as distinct elements of D. The function and predicate tables for $\mathcal{M}'$ are based on those from $\mathcal{M}$, with encoded type arguments corresponding to actual type arguments. For constructors, some type arguments may be missing in $\mathcal{M}'$, but they are irrelevant since the interpretations in $\mathcal{M}$ coincide. □

Lemma 5.6 (Correctness Conditions). *Let* Φ *be a polymorphic problem, and let* x *be a polymorphic encoding. The problems* Φ *and* $[\![\Phi]\!]_{x;\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$ *are equisatisfiable provided the following conditions hold:*

MONO: $[\![\Phi]\!]_x$ *is monotonic.*

SOUND: *If* Φ *is satisfiable, so is* $[\![\Phi]\!]_x$.

COMPLETE: *If* $[\![\Phi]\!]_x$ *is satisfiable, so is* Φ.

Proof. Immediate from Theorems 3.9 and 5.5. □

5.3 Monotonicity Inference

The monotonicity inference of Section 4.2 must be adapted to the polymorphic setting. The calculus presented below 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 5.7 (Monotonicity Calculus $\rhd$). Let Φ be a polymorphic problem. A judgement $\sigma \rhd \varphi$ indicates that the type σ is inferred monotonic in $\varphi \in \Phi$. 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 $\mathrm{Inf}(\varphi)$.

Remark. Although type variables occurring in declarations and formulas are bound by a $\forall$ quantifier, for monotonicity inference polymorphic types $\sigma[\bar{\alpha}]$, where $\bar{\alpha}$ are the type variables occurring in σ, are viewed as being implicitly bound (i.e. "$\forall\bar{\alpha}.\,\sigma[\bar{\alpha}]$" to abuse notation). Type variables are assumed to be fresh in distinct types, comparison between types is modulo α-renaming, and substitution is capture-avoiding. Thus, we have $\mathit{list}(\alpha) = \mathit{list}(\beta)$ (since "$\forall\alpha.\,\mathit{list}(\alpha) = \forall\beta.\,\mathit{list}(\beta)$"), and $\mathit{map}(\alpha,d)$ can be unified with $\mathit{map}(c,\alpha)$ by renaming the second α to β and applying $\rho = \{\alpha \mapsto c, \beta \mapsto d\}$.

Our proof strategy is to reduce the polymorphic case to the already proved monomorphic case. Our Herbrandian motto is,

> A polymorphic formula is equisatisfiable to the (generally infinite) set of its monomorphic instances.

This complete form of monomorphisation is not to be confused with the finitary, heuristic monomorphisation algorithm presented in Section 4.6.

Theorem 5.8 (Soundness of $\rhd$). *Let Φ be a polymorphic problem. If $\sigma \rhd \Phi$, then σ is monotonic in Φ.*

Proof. We consider an unknown but fixed monomorphic instance of the rule and justify it by one of the two rules of the monomorphic version of the calculus, proved sound in Theorem 4.11. This is sufficient because the rule honours the abstract view of polymorphic types as sets of ground types, without inspecting their concrete structure. For monomorphic σ and φ, suppose that the polymorphic monotonicity calculus judges that $\sigma \rhd \varphi$. By Definition 5.7, there is no $X^\tau \in \mathrm{NV}(\varphi)$ such that σ and τ are unifiable to $\beta \notin \mathrm{Inf}^*(\varphi)$. Since both σ and φ are monomorphic, this states that for all $X^\sigma \in \mathrm{NV}(\varphi)$, $\sigma \in \mathrm{Inf}(\varphi)$, i.e. either $\mathrm{NV}(\varphi)$ contains no variables of type σ, or $\sigma \in \mathrm{Inf}(\varphi)$. In both cases, a rule of the monomorphic monotonicity calculus applies. □

5.4 Monotonicity-Based Type Tags

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

The main novelty concerns the typing axioms. The $\widetilde{\mathtt{t}?}$ 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 $\mathtt{t}\langle\mathit{list}(\alpha)\rangle(\mathit{Xs}) \approx \mathit{Xs}$, 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 5.9 (Lightweight Tags $\mathtt{t?}$). The *polymorphic lightweight type tags* encoding $\mathtt{t?}$ 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 $[\![\,]\!]_{\mathtt{t?};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

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

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 $\sigma\rho$:

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

Example 5.10. For the algebraic list problem of Example 2.4, $\mathit{list}(\alpha)$ is monotonic by virtue of being infinite, whereas α and its instance b cannot be inferred monotonic. The $\mathtt{t?}$ encoding of the problem follows:

$$\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), Xs) \approx Xs$$

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

The typing axiom allows any term to be typed as $\mathit{list}(\alpha)$, which is sound because $\mathit{list}(\alpha)$ is infinite. It would have been equally correct to provide separate axioms for nil, cons, and tl. Either way, the axioms are needed to remove the $\mathsf{t}(A, X)$ tags in case the proof requires reasoning about $\mathit{list}(\mathit{list}(\alpha))$.

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

Definition 5.11 (Featherweight Tags $\mathtt{t??}$). The *polymorphic featherweight type tags* encoding $\mathtt{t??}$ translates a polymorphic problem Φ over Σ into an untyped problem over Σ', where Σ' is as for $\mathtt{t?}$. It is defined as $[\![\,]\!]_{\mathtt{t??};\mathsf{a}^{\mathsf{ctor}};\mathsf{e}}$, where

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

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

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:

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

Example 5.12. The t?? encoding of Example 2.4 requires fewer tags than $\widetilde{\mathsf{t}}?$, at the cost of more type information:

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

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

Theorem 5.13 (Correctness of t?, t??). *The polymorphic type tags encodings* t? *and* t?? *are correct.*

Proof. It suffices to show the three conditions of Lemma 5.6. The proof is analogous to that of Theorem 4.18.

MONO: Both encodings tag any naked variables that occur in the original problem and that could obstruct the monotonicity calculus. The typing axioms $\mathsf{t}\langle\sigma\rangle(X) \approx X$ contain a naked variable, but they are inferred monotonic because σ must be infinite. Monotonicity follows by Theorem 5.8 and Lemma 4.8.

SOUND: Given a model of Φ, we extend it to a model of $[\![\Phi]\!]_x$ by interpreting the type tag function $\mathsf{t}\langle\sigma\rangle$ as the identity.

COMPLETE: We construct a canonical model of $[\![\Phi]\!]_x$, from which we obtain a model of Φ by leaving out $\mathsf{t}\langle\sigma\rangle$. The ground types σ occurring in Φ are partitioned into three classes:

(a) The possibly nonmonotonic types ($\sigma \not\rhd \Phi$) are treated differently by t? and t??. In t?, these types are systematically accessed through $\mathsf{t}\langle\sigma\rangle$, and we simply permute the entries of the function table of $\mathsf{t}\langle\sigma\rangle$ as in the proof of Theorem 4.18. In t??, the typing axioms $\exists X : \sigma.\ \mathsf{t}\langle\sigma\rangle(X) \approx X$ and $\mathsf{t}\langle\sigma\rangle(\mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})) \approx \mathsf{f}\langle\bar{\alpha}\rangle(\bar{X})$ ensure that $\mathsf{t}\langle\sigma\rangle$ is the identity for at least one element and for all well-typed terms; furthermore, the equations $\mathsf{t}\langle\sigma\rangle(X) \approx X$ generated for existential variables ensures that $\mathsf{t}\langle\sigma\rangle$ is the identity for witnesses, as required to apply Lemma 4.13.

(b) For the infinite instances of possibly nonmonotonic types (types $\sigma\rho \in \mathrm{Inf}(\Phi)$ such that $\sigma \not\rhd \Phi$), the typing axiom $\mathsf{t}\langle\sigma\rangle(X) \approx X$ ensures that $\mathsf{t}\langle\sigma\rangle$ is the identity.

(c) Values of the remaining monotonic types σ (such that $\sigma \rhd \Phi$) are never accessed through a tag function $\mathsf{t}\langle\tau\rangle$—such a tag is generated only if $\tau \not\rhd \Phi$, but in that case σ must be infinite for $\sigma \rhd \Phi$ to be derivable.

The canonical model is obtained by removing the domain elements for which $\mathsf{t}\langle\sigma\rangle$ is not the identity, appealing to Lemma 4.13. □

5.5 Monotonicity-Based Type Guards

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 5.14 (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? along the lines of the generalisation of $\widetilde{\mathsf{g}}$? into $\widetilde{\mathsf{g}}$??.

Definition 5.15 (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 5.16. The g?? encoding of Example 2.4 follows:

$\forall A, Xs.\ \mathsf{g}(A, \mathsf{hd}(A, Xs))$
$\forall A, Xs.\ \mathsf{g}(\mathsf{list}(A), Xs)$

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

Theorem 5.17 (Correctness of g?, g??). *The polymorphic type guards encodings* g? *and* g?? *are correct.*

Proof. It suffices to show the three conditions of Lemma 5.6. The proof is analogous to that of Theorem 4.23.

MONO: Both encodings guard any naked variables that occur in the original problem and could prevent the problem from being monotonic. Given a model $\mathcal{M}$ of $[\![\Phi]\!]_x$ where each σ is interpreted as D_σ, we construct a model $\mathcal{M}'$ of $[\![\Phi]\!]_x$ where σ is interpreted as $D_\sigma \uplus D'_\sigma$. We choose a representative $a \in D_\sigma$ and extend the interpretations of all symbols so that they coincide on a and each $a' \in D'_\sigma$ except for $\mathtt{g}\langle\sigma\rangle(a')$, which is set to true if σ is an instance of a type in $\mathrm{Inf}(\Phi)$ and false otherwise. This is compatible with the typing axioms (including $\mathtt{g}\langle\sigma\rangle(X)$ for infinite σ) and effectively prevents a' from arising naked if σ is not infinite.

SOUND: Given a model of Φ, we extend it to a model of $[\![\Phi]\!]_x$ by interpreting the type tag function $\mathtt{t}\langle\sigma\rangle$ as the identity.

COMPLETE: The ground types σ occurring in Φ are partitioned into three classes, as in the proof of Theorem 5.13. The canonical model is obtained by removing the domain elements for which $\mathtt{g}\langle\sigma\rangle$ is false, appealing to Lemma 4.13. □

6 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. Let us first rigorously define this notion of term arguments "covering" type arguments.

Definition 6.1 (Cover). Let $\mathtt{s} : \forall\bar{\alpha}.\, \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. A (*type argument*) *cover* $C \subseteq \{1,\ldots,|\bar{\sigma}|\}$ for $\mathtt{s}$ 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. A cover C of $\mathtt{s}$ is *minimal* if no proper subset of C is a cover for $\mathtt{s}$. We let $\mathrm{Cover}_{\mathtt{s}}$ denote an arbitrary but fixed minimal cover of $\mathtt{s}$ and extend the notion to the distinguished symbols $\mathtt{t}, \mathtt{g}$ by taking $\mathrm{Cover}_{\mathtt{t}} = \mathrm{Cover}_{\mathtt{g}} = \emptyset$ (cf. Definition 3.7).

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 cover-based encodings `t@` and `g@` introduced below ensure that each term argument position that is part of its enclosing function or predicate's cover has a unique type associated with it, from which the omitted type arguments can be inferred. For example, `t@` translates the term $\mathsf{cons}\langle\alpha\rangle(X, Xs)$ to $\mathsf{cons}(\mathtt{t}(A, X), Xs)$ with a type tag around X, effectively preventing an ill-typed instantiation of X that would result in the wrong type argument being inferred. But we do not need to protect the second argument, Xs—it is sufficient to introduce enough protectors to "cover" all the inferable type arguments. We call variables that occur in their enclosing symbol's cover (and hence that "carry" some type arguments) "undercover variables".

Definition 6.2 (Undercover Variable). The set of *undercover variables* $\mathrm{UV}(\varphi)$ of a formula φ is defined by the equations

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

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

Undercover variables are reminiscent of naked variables. This is no coincidence: naked variables effectively carry the implicit type argument of $\approx : \forall\alpha.\ \alpha \times \alpha \to \mathrm{o}$, and undercover variables generalise this to arbitrary predicate and function symbols. Equality is special in two respects, though: its only sound cover is effectively $\{1,2\}$ because of the equality axioms (which are built into the provers, without any protectors), and the negative case is optimised to take advantage of disequality's fixed semantics.

6.1 Cover-Based Type Tags

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 to add or remove tags around function terms.

Definition 6.3 (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

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

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:

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

Example 6.4. The $\mathtt{t@}$ encoding of Example 2.4 is as follows:

$$\forall A.\ \mathsf{t}(\mathsf{list}(A), \mathsf{nil}(A)) \approx \mathsf{nil}(A)$$
$$\forall A, X, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), Xs)) \approx \mathsf{cons}(\mathsf{t}(A, X), Xs)$$
$$\forall A, Xs.\ \mathsf{t}(\mathsf{list}(A), \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), Xs))) \approx \mathsf{hd}(\mathsf{t}(\mathsf{list}(A), Xs))$$
$$\forall A, Xs.\ \mathsf{t}(A, \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), Xs))) \approx \mathsf{tl}(\mathsf{t}(\mathsf{list}(A), Xs))$$

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

In Section 5, we showed that it is sound to omit noninferable type arguments to constructors for monotonicity-based encodings, yielding nil instead of $\mathsf{nil}(A)$ in the translation. For cover-based encodings, this would be unsound, as shown by the following counterexample: $\mathsf{q}\langle a\rangle(\mathsf{nil}\langle a\rangle) \wedge \neg\, \mathsf{q}\langle b\rangle(\mathsf{nil}\langle b\rangle)$ is satisfiable, but the naive translation to $\mathsf{q}(\mathsf{nil}) \wedge \neg\, \mathsf{q}(\mathsf{nil})$ is unsatisfiable. The counterexample does not apply to monotonicity-based encodings, because q would be passed an encoded type argument: $\mathsf{q}(\mathsf{a}, \mathsf{nil}) \wedge \neg\, \mathsf{q}(\mathsf{b}, \mathsf{nil})$ is satisfiable.

Theorem 6.5 (Correctness of t@). *The cover-based type tags encoding* t@ *is correct.*

Proof. SOUND: Given a model of Φ, we extend it to a model $\mathcal{M}$ of $\Phi' = [\![\Phi]\!]_{\mathsf{t@}}$ by interpreting $\mathsf{t}\langle\sigma\rangle$ as the identity. We assume the domains of $\mathcal{M}$ are disjoint and choose from each domain a representative element. We then construct a model $\mathcal{M}'$ of $[\![\Phi']\!]_{\mathsf{a}^{\mathsf{phan}};\mathsf{e}}$ from $\mathcal{M}$ as follows. The domain for $\mathcal{M}'$ is the disjoint union of the domains for $\mathcal{M}$ and of the term universe of encoded ground types over $\mathcal{K}$. For each symbol $\mathsf{s} : \forall\bar{\alpha}.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P} \uplus \{\mathsf{t}\}$, the entry for $\mathsf{s}(\langle\!\langle\bar{\tau}\rangle\!\rangle, \bar{a})$ in $\mathcal{M}'$ is set as follows:

1. If $\mathsf{s}\langle\bar{\sigma}\rangle(\bar{a})$ is defined in $\mathcal{M}$ (i.e. the $\bar{a}$'s lie within the interpretations of their respective domains) for some $\bar{\sigma}$ such that $\mathsf{phan}_{\mathsf{s}}(\bar{\sigma}) = \bar{\tau}$, set the entry for $\mathsf{s}(\langle\!\langle\bar{\tau}\rangle\!\rangle, \bar{a})$ to the interpretation of $\mathsf{s}\langle\bar{\sigma}\rangle(\bar{a})$ by $\mathcal{M}$. (The domain disjointness assumption, together with the condition $\mathsf{phan}_{\mathsf{s}}(\bar{\sigma}) = \bar{\tau}$, ensures that there exists at most one vector $\bar{\sigma}$ such that $\mathsf{s}\langle\bar{\sigma}\rangle(\bar{a})$ is defined.)
2. Otherwise, attempt to repair the vector $\bar{a}$ by replacing any a_j that is not in its domain by a "well-typed" representative, while keeping all a_j's such that $j \in \mathrm{Cover}_{\mathsf{s}}$ unchanged. If this construction succeeds, proceed as in step 1.
3. Otherwise, choose some arbitrary domain element (if s is a function) or truth value (if s is a predicate).

We must show that $\mathcal{M}'$ is a model of $[\![\Phi']\!]_{\mathsf{a}^{\mathsf{phan}};\mathsf{e}}$. Thanks to step 1, the semantics of the problems' formulas (including the typing axioms) coincide if universal variables are instantiated by "well-typed" values. If all tagged variables in a formula are instantiated by "well-typed" values but some of the untagged variables are not, the semantics of the formula coincides with, or is weaker than (cf. the proof of Theorem 4.18), that of an already considered "well-typed" instance thanks to step 2, since "ill-typed" values are interpreted the same way as the representative element. Finally, if some of the tagged

variables are instantiated by "ill-typed" values, the tags return the representatives, and non-tagged instances of the same variables are treated as the representatives, so again this case coincides with an already considered case.

COMPLETE: The $\mathsf{a}^{\mathsf{phan}}$ encoding is complete by Theorem 3.9. It remains to show that $[\![\,]\!]_{\mathtt{t@}}$ is complete. Given a model of $[\![\Phi]\!]_{\mathtt{t@}}$, we construct a canonical model of $[\![\Phi]\!]_{\mathtt{t@}}$, from which we extract a model of Φ, as in the proof of Theorem 5.13. The typing axioms ensure that $\mathsf{t}\langle\sigma\rangle$ is the identity for at least one element and for all "well-typed" terms; furthermore, the equations $\mathsf{t}\langle\sigma\rangle(X) \approx X$ generated for existential variables ensures that $\mathsf{t}\langle\sigma\rangle$ is the identity for witnesses, as required to apply Lemma 4.13. □

6.2 Cover-Based Type Guards

The cover-based $\mathtt{g@}$ encoding is defined analogously.

Definition 6.6 (Cover Guards g@). The *polymorphic cover-based type guards* encoding $\mathtt{g@}$ is identical to the traditional g encoding except for the $\forall$ case:

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

The encoding is complemented by typing axioms:

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

Example 6.7. The $\mathtt{g@}$ encoding of the algebraic list problem is identical to the g encoding (Example 3.15), except that the guard $\mathsf{g}(\mathsf{list}(A), Xs)$ 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 6.8 (Correctness of g@). *The cover-based type guards encoding* $\mathtt{g@}$ *is correct.*

Proof. SOUND: Given a model of Φ, we extend it to a model $\mathcal{M}$ of $\Phi' = [\![\Phi]\!]_{\mathtt{g@}}$ by interpreting $\mathsf{g}\langle\sigma\rangle$ as the true predicate. We assume the domains of $\mathcal{M}$ are disjoint and choose from each domain a representative element. We then construct a model $\mathcal{M}'$ exactly as in the proof of Theorem 6.5, except that $\mathsf{g}(\langle\!\langle\sigma\rangle\!\rangle, a)$ is set to be false for "ill-typed" values a (i.e. $a \notin [\![\sigma]\!]^{\mathcal{M}}$). We must show that $\mathcal{M}'$ is a model of $[\![\Phi']\!]_{\mathsf{a}^{\mathsf{phan}};\mathsf{e}}$. Thanks to step 1 of the construction of $\mathcal{M}'$, the semantics coincide if universal variables are instantiated by "well-typed" values. If all guarded variables are instantiated by "well-typed" values but some of the unguarded variables are not, the semantics of the formula coincides with, or is weaker than, that of an already considered "well-typed" instance thanks to step 2. Finally, if some of the guarded variables are instantiated by "ill-typed" values, the guard returns false, making the subformula true.

COMPLETE: The proof is analogous to the corresponding case of Theorems 5.13. □

7 Implementation

Our research on polymorphic type encodings was driven by Sledgehammer, a component of Isabelle/HOL that harnesses first-order automatic theorem provers to discharge interactive proof obligations. The tool heuristically selects hundreds of background facts, translates them to untyped or monomorphic first-order logic, invokes the external provers in parallel, and reconstructs machine-generated proofs in Isabelle.

All the encodings presented in this report, including the traditional encodings, are implemented in Sledgehammer and can be used to target external first-order provers. The rest of this section considers implementation issues in more detail.

7.1 Finite Monomorphisation Algorithm

The monomorphisation algorithm implemented in Sledgehammer translates a polymorphic problem into a monomorphic problem by heuristically instantiating type variables. It involves three stages:

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

To ensure termination, the iterations performed in stage 2 are limited to a configurable number K. To curb the exponential growth, the algorithm also enforce an upper bound Δ on the number of new formulas. Sledgehammer operates with $K = 3$ and $\Delta = 200$ by default, so that a problem with 500 formulas comprises at most 700 formulas after monomorphisation. Experiments found these values suitable. Given formulas about b and $\mathit{list}(\alpha)$, the third iteration already generates $\mathit{list}(\mathit{list}(\mathit{list}(b)))$ instances; adding yet another layer of *list* is unlikely to help. Increasing Δ sometimes helps solve more problems, but its potential for clutter is real.

7.2 Extension to Higher-Order Logic

Sledgehammer's source logic, polymorphic higher-order logic (HOL) with axiomatic type classes [31], is not the same as the polymorphic first-order logic considered in this report. The translation is a three-step process, where the first and last step may be omitted, depending on whether monomorphisation is desired and whether the target prover supports monomorphic types:

1. *Optionally monomorphise the problem.*

2. *Eliminate the higher-order constructs* [21, § 2.1]. λ-abstractions are rewritten to SK combinators or to supercombinators (λ-lifting). Functions are passed varying numbers of arguments via an apply operator $\mathsf{hAPP} : \forall\alpha\beta.\, \mathit{fun}(\alpha,\beta) \times \alpha \to \beta$ (where *fun* is uninterpreted). Boolean terms are converted to formulas using a unary predicate $\mathsf{hBOOL} : \mathit{bool} \to \mathrm{o}$ (where *bool* is uninterpreted).
3. *Encode the type information.* Polymorphic types are encoded using the techniques described in this report. Type classes are essentially sets of types; they are encoded as polymorphic predicates $\forall\alpha.\,\mathrm{o}$, where α is a phantom type variable. For example, a predicate $\mathsf{linorder} : \forall\alpha.\,\mathrm{o}$ could be used to restrict the axioms specifying that $\mathsf{less} : \forall\alpha.\,\alpha \times \alpha \to \mathrm{o}$ is a linear order to those types that satisfy the linorder predicate. The type class hierarchy is expressible as Horn clauses [21, § 2.3].

The symbol hAPP would hugely burden problems if it were introduced systematically for all arguments to functions. To reduce clutter, Sledgehammer computes the minimum arity n needed for each symbol and pass the first n arguments directly, falling back on hAPP for additional arguments. In general, more arguments can be passed directly if monomorphisation is performed before hAPP is introduced, because each monomorphic instance of a polymorphic symbol is considered individually. Similar observations can be made for hBOOL.

7.3 Infinite Types and Constructors

The monotonicity calculus $\rhd$ is parameterised by a set $\mathrm{Inf}(\Phi)$ of infinite types. One could employ an approach similar to that implemented in Infinox [13] to automatically infer finite unsatisfiability of types. This tool relies on various proof principles to show that a set of untyped first-order formulas only has models with infinite domains. For example, it can infer that *list_b* is infinite in Example 2.6 because cons_b is injective in its second argument but not surjective. However, in a proof assistant such as Isabelle, it is simpler to exploit metainformation available through introspection. Isabelle's datatypes are registered with their constructors; if some of them are recursive, or take an argument of an infinite type, the datatype must be infinite and hence monotonic.

More specifically, the monotonicity inference is run on the entire problem and maintains two finite sets of polymorphic types: the surely infinite types J and the possibly nonmonotonic types N. Every type of a naked variable in the problem is tested for infinity. If the test succeeds, the type is inserted into J; otherwise, it is inserted into N. Simplifications are performed: there no need to insert σ to J or N if it is an instance of a type already in the set; when inserting σ to a set, it is safe to remove any type in the set that is an instance of σ. The monotonicity check then becomes

$$\sigma \rhd \Phi \iff (\exists\tau \in J.\, \exists\rho.\, \sigma = \tau\rho) \vee (\forall\tau \in N.\, \nexists\rho.\, \sigma\rho = \tau\rho)$$

Introspection also plays a role in the $\mathsf{a}^{\mathsf{ctor}}$ encoding and its derivatives (t?, t??, g?, and g??). These are parameterised by a set of constructors $\mathrm{Ctor}(\Phi)$. Querying the datatype package is again simpler than attempting to detect distinctness and injectivity axioms in individual problems.

7.4 Proof Reconstruction

To guard against bugs in the external provers, Sledgehammer reconstructs machine-generated proofs in Isabelle. This is usually accomplished by the *metis* proof method [23], supplying it with the short list of facts referenced in the proof found by the prover. The proof method is based on the Metis prover [18], a complete resolution prover for untyped first-order logic. The *metis* call is all that remains from the Sledgehammer invocation in the Isabelle theory, which can then be replayed without external provers. Given only a handful of facts, *metis* usually succeeds within milliseconds. Prior to our work, a large share of the reconstruction failures were caused by type-unsound proofs found by the external provers, due to the use of the unsound encoding a [11, § 4.1]. We now replaced the internals of Sledgehammer and *metis* so that they use a translation module supporting all the type encodings described in this report. However, despite the typing information, individual inferences in Metis can be ill-typed when types are reintroduced, causing the *metis* proof method to fail. There are three main failure scenarios.

First, the prover may in principle instantiate variables with "ill-typed" terms at any point in the proof. Fortunately, this hardly ever arises in practice, because like other resolution provers Metis heavily restricts paramodulation from and into variables [1].

An issue that is more likely to plague users concerns the infinite types $\mathrm{Inf}(\Phi)$. Assuming *nat* is known to be infinite, the monotonicity-based encodings will not introduce any protectors around the naked variables M and N when translating the problem

$$\mathsf{on} \not\approx \mathsf{off}^{state} \wedge (\forall X, Y : nat.\ X \approx Y)$$

(presumably a negated conjecture). That problem is satisfiable on its own but unsatisfiable with respect to the background theory. Untyped provers will happily instantiate X and Y with on and off to derive a contradiction; and no "type-sound" proof is possible unless we also provide characteristic theorems for *nat*. In general, we would need to provide infinity axioms for all types in $\mathrm{Inf}(\Phi)$ to make the encoding sound irrespective of the background theory; for example:

$$\forall N : nat.\ \mathsf{zero} \not\approx \mathsf{suc}(N) \qquad \forall M, N : nat.\ \mathsf{suc}(M) \approx \mathsf{suc}(N) \rightarrow M \approx N$$

Although this now makes a sound proof possible (by instantiating X and Y with zero and $\mathsf{suc}(\mathsf{zero})$), it does not prevent the prover from discovering the spurious proof with on and off, which cannot be reconstructed by *metis*.

A similar issue affects the constructors $\mathrm{Ctor}(\Phi)$. Let $\mathsf{c}, \mathsf{d} : \forall \alpha.\ k(\alpha)$ be among the constructors for k. The encodings based on $\mathsf{a}^{\mathsf{ctor}}$ will translate the satisfiable problem

$$\mathsf{c}\langle a \rangle \not\approx \mathsf{d}\langle a \rangle \wedge \mathsf{c}\langle b \rangle \approx \mathsf{d}\langle b \rangle$$

into an unsatisfiable one: $\mathsf{c} \not\approx \mathsf{d} \wedge \mathsf{c} \approx \mathsf{d}$. In general, we would need to provide distinctness and injectivity axioms for all constructors in $\mathrm{Ctor}(\Phi)$ to make the encoding sound irrespective of the background theory; for example: $\forall \alpha.\ \mathsf{c}\langle \alpha \rangle \not\approx \mathsf{d}\langle \alpha \rangle$.

We stress that the issues above do not indicate unsoundness in our encodings. Rather, we use metainformation from Isabelle to guide the translation; the translation is then sound assuming that the metainformation holds. If we do not encode that metainformation in the typed problem, it might have a model where the metainformation is

false, while the translation might exploit the metainformation and be unsatisfiable, as in the examples above. Proof reconstruction will then fail.

Although the above scenarios rarely occur in practice, it would be more satisfactory if proof reconstruction were always possible. A solution would be to connect our formalised soundness proofs with a verified checker for untyped first-order proofs. This remains for future work.

8 Evaluation

To evaluate the type encodings described in this report, we put together two sets of 1000 polymorphic first-order problems originating from 10 existing Isabelle theories, translated with Sledgehammer's help (100 problems per theory).[3] Nine of the theories are the same as in a previous evaluation [3]; the tenth one is an optimality proof for Huffman's algorithm. Our test data are publicly available [4].

The problems in the first benchmark set include about 50 heuristically selected facts (before monomorphisation); that number is increased to 500 for the second set, to reveal how well the encodings scale with the problem size.

We evaluated each type encoding with five modern automatic theorem provers: the resolution provers E 1.6 [25], SPASS 3.8ds [30], and Vampire 2.6 [24], the instantiation prover iProver 0.99 [19], and the SMT solver Z3 4.0 [16]. 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 $\widetilde{\mathtt{n}}$. Each prover was invoked with the set of options we had previously determined worked best for Sledgehammer.[4] The provers were granted 20 seconds of CPU time per problem on one core of a 3.06 GHz Dual-Core Intel Xeon processor. Most proofs were found within a few seconds; a higher time limit would have had little impact on the success rate [11, § 4]. To avoid giving the unsound encodings (e, a, and $\mathtt{a}^{\mathsf{ctor}}$) 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.

Figures 2 and 3 give, for each combination of prover and encoding, the number of solved problems from each problem set. Rows marked with ~ concern the monomorphic encodings. The encodings $\widetilde{\mathtt{a}}$, $\widetilde{\mathtt{a}}^{\mathsf{ctor}}$, $\widetilde{\mathtt{t@}}$, and $\widetilde{\mathtt{g@}}$ are omitted; the first two coincide with $\widetilde{\mathtt{e}}$, whereas $\widetilde{\mathtt{t@}}$ and $\widetilde{\mathtt{g@}}$ are identical to versions of $\widetilde{\mathtt{t??}}$ and $\widetilde{\mathtt{g??}}$ that treat all types as possibly nonmonotonic. We observe the following:

- Among the encodings to untyped first-order logic, the monomorphic monotonicity-based encodings (especially $\widetilde{\mathtt{g??}}$ but also $\widetilde{\mathtt{t??}}$, $\widetilde{\mathtt{g?}}$, and $\widetilde{\mathtt{t?}}$) performed best overall. In many cases, these even outperformed the provers' native support for simple types ($\widetilde{\mathtt{n}}$). Polymorphic encodings lag behind, especially for resolution provers;

[3] The TPTP benchmark suite [27], which is customarily used for evaluating theorem provers, has just begun collecting polymorphic (TFF1) problems [6, § 6].

[4] 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	a$^{\mathsf{ctor}}$	t	t?	t??	t@	g	g?	g??	g@	n
E		319	**328**	**328**	304	298	319	258	267	316	323	285	–
	~	333	–	–	325	337	333	–	320	336	**340**	–	–
iProver		243	220	225	**249**	190	236	157	173	235	243	219	–
	~	233	–	–	263	261	**264**	–	222	259	261	–	–
SPASS		276	286	286	267	272	**296**	196	237	283	294	262	–
	~	298	–	–	289	304	**310**	–	291	308	305	–	309
Vampire		**325**	312	312	319	307	314	260	255	289	300	270	–
	~	330	–	–	327	336	338	–	299	337	**340**	–	332
Z3		299	**321**	**321**	288	236	304	290	263	281	301	275	–
	~	319	–	–	300	313	319	–	322	318	321	–	**325**

Figure 2. Number of solved problems with 50 facts

		e	a	a$^{\mathsf{ctor}}$	t	t?	t??	t@	g	g?	g??	g@	n
E		116	**361**	360	263	275	347	228	216	344	349	262	–
	~	393	–	–	328	390	397	–	337	393	**401**	–	–
iProver		243	212	207	231	202	**262**	135	140	242	257	169	–
	~	210	–	–	243	246	245	–	180	**247**	241	–	–
SPASS		131	292	292	262	245	**299**	164	164	283	296	208	–
	~	331	–	–	293	326	330	–	237	320	334	–	**356**
Vampire		120	**341**	**341**	277	281	314	212	171	271	299	241	–
	~	393	–	–	309	379	382	–	265	390	**403**	–	372
Z3		281	355	**357**	250	238	350	279	213	291	351	268	–
	~	354	–	–	268	343	346	–	328	**355**	349	–	350

Figure 3. Number of solved problems with 500 facts

this is partly due to the synergy between the monomorphiser and the translation of higher-order constructs (cf. Section 7.2).

- 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).
- As suggested in the literature, there is no clear winner between tags and guards. We expected monomorphic guards to be especially effective with SPASS, since they are internally mapped to soft sorts, but this is not corroborated by the data.
- Despite the proof reconstruction phase, the unsound encodings a and a$^{\mathsf{ctor}}$ achieved similar results to the sound polymorphic encodings. In contrast, their monomorphic cousin $\widetilde{\mathsf{e}}$ is no match for the sound monomorphic schemes.

For the second benchmark set, Figure 4 presents the average number of clauses, literals per clause, symbols per atom, and symbols for clausified problems (using E's clausifier), to give an idea of each encoding's overhead. The main surprise here is the

		e	a	a$^{\mathsf{ctor}}$	t	t?	t??	t@	g	g?	g??	g@
Clauses		832	901	901	918	947	1129	1307	1304	1127	1126	1304
	~	1188	–	–	1207	1207	1273	–	1890	1273	1273	–
Literals per clause		2.55	2.77	2.77	2.77	2.71	2.97	3.33	5.88	4.63	3.20	5.01
	~	2.58	–	–	2.58	2.58	2.57	–	5.41	2.87	2.66	–
Symbols per atom		6.5	8.5	8.5	21.0	18.4	11.8	9.9	5.8	8.6	10.7	5.9
	~	6.9	–	–	11.7	7.7	6.9	–	4.3	6.2	6.6	–
Symbols ('000)		13.8	21.2	21.1	53.4	47.4	39.7	43.1	44.8	45.0	38.5	38.5
	~	21.0	–	–	36.4	24.0	22.5	–	44.5	22.8	22.3	–

Figure 4. Average size of clausified problems with 500 facts

lightness of the monomorphic encodings. Because they give rise to duplicate formulas, we could have expected them to result in larger problems; but this underestimates the cost of encoding types as terms in the polymorphic encodings. The strong correlation between the success rates in Figure 3 and the average number of symbols in Figure 4 confirms the expectation that clutter (whether type arguments, guards, or tags) slows down automatic provers.

Independently of these empirical results, the new type encodings made an impact at the 2012 edition of CASC, the annual automatic prover competition [28]. Isabelle's automatic proof tools, including Sledgehammer, compete against the automatic provers 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.

9 Related Work

The earliest descriptions of type tags and type guards we are aware of are due to Enderton [17, § 4.3] and Stickel [26, p. 99]. Wick and McCune [32, § 4] compare type arguments, tags, and guards in a monomorphic setting. Type arguments are reminiscent of System F; they are described by Meng and Paulson [21], who also consider full type erasure and polymorphic type tags and present a translation of axiomatic type classes. As part of the MPTP project, Urban [29] extended the untyped TPTP FOF syntax with dependent types to accommodate Mizar and designed translations to plain FOF.

The intermediate verification language and tool Boogie 2 [20] supports a restricted form of higher-rank polymorphism (with polymorphic maps), and its cousin Why3 [9] provides rank-1 polymorphism. Both define translations to a monomorphic logic and rely on proxies to handle interpreted types [10, 15, 20]. One of the Boogie translations [20, § 3.1] uses SMT triggers to prevent ill-typed instantiations in conjunction with type arguments; however, this approach is risky in the absence of a semantics for triggers. Bouillaguet et al. [12, § 4] 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 or monomorphising them away is to support them natively in the prover. This is ubiquitous in interactive theorem provers, but perhaps the only automatic prover that supports polymorphism is Alt-Ergo [8].[5]

Blanchette and Krauss [5] studied monotonicity inferences for higher-order logic without polymorphism. Claessen et al. [14] were first to apply them to type erasure.

10 Conclusion

This report 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 the Monotonox translator [14]. Applications such as Boogie [20], LEO-II [2], and Why3 [9] also stand to gain from lighter encodings.

The TPTP family recently welcomed the addition of TFF1 [6], 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. It should also be possible to extend our approach to interpreted arithmetic.

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

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 authors are listed in alphabetical order regardless of individual contributions or seniority.

References

[1] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Inf. Comput.*, 121(2):172–192, 1995.

[5] Regrettably, limitations in its type system make it unsuitable as a Sledgehammer backend.

[2] C. Benzmüller, L. C. Paulson, F. Theiss, and A. Fietzke. LEO-II—A cooperative automatic theorem prover for higher-order logic. In A. Armando, P. Baumgartner, and G. Dowek, editors, *IJCAR 2008*, volume 5195 of *LNAI*, pages 162–170. Springer, 2008.

[3] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. *J. Autom. Reasoning*, 51(1):109–128, 2013.

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] S. Böhme and T. Nipkow. Sledgehammer: Judgement Day. In J. Giesl and R. Hähnle, editors, *IJCAR 2010*, volume 6173 of *LNAI*, pages 107–121. Springer, 2010.

[12] 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.

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

[14] 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.

[15] J.-F. Couchot and S. Lescuyer. Handling polymorphism in automated deduction. In F. Pfenning, editor, *CADE-21*, volume 4603 of *LNAI*, pages 263–278. Springer, 2007.

[16] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

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

[18] J. Hurd. First-order proof tactics in higher-order logic theorem provers. In M. Archer, B. Di Vito, and C. Muñoz, editors, *Design and Application of Strategies/Tactics in Higher Order Logics*, number CP-2003-212448 in NASA Tech. Reports, pages 56–68, 2003.

[19] K. Korovin. Instantiation-based automated reasoning: From theory to practice. In R. A. Schmidt, editor, *CADE-22*, volume 5663 of *LNAI*, pages 163–166. Springer, 2009.

[20] 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.

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

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

[23] L. C. Paulson and K. W. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *TPHOLs 2007*, volume 4732 of *LNCS*, pages 232–245. Springer, 2007.

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

[25] S. Schulz. System description: E 0.81. In D. Basin and M. Rusinowitch, editors, *IJCAR 2004*, volume 3097 of *LNAI*, pages 223–228. Springer, 2004.

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

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

[28] 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.

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

[30] C. Weidenbach. Combining superposition, sorts and splitting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1965–2013. Elsevier, 2001.

[31] M. Wenzel. Type classes and overloading in higher-order logic. In E. L. Gunter and A. Felty, editors, *TPHOLs 1997*, volume 1275 of *LNCS*, pages 307–322. Springer, 1997.

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