Encoding monomorphic and polymorphic types

Downloaded from: https://research.chalmers.se, 2026-11-10 06:28 UTC

Citation for the original published paper (version of record):
Blanchette, J., Böhme, S., Popescu, A. et al (2016). Encoding monomorphic and polymorphic types. Logical Methods in Computer Science, 12(4). http://dx.doi.org/10.2168/LMCS-12(4:13)2016

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

Logical Methods in Computer Science
Vol. 12(4:13)2016, pp. 1–52
www.lmcs-online.org

Submitted May 14, 2014
Published Dec. 31, 2016

ENCODING MONOMORPHIC AND POLYMORPHIC TYPES

JASMIN CHRISTIAN BLANCHETTE [a], SASCHA BÖHME [b], ANDREI POPESCU [c], AND NICHOLAS SMALLBONE [d]

[a] Inria & LORIA, Nancy, France; Max-Planck-Institut für Informatik, Saarbrücken, Germany
e-mail address: jasmin.blanchette@{inria.fr,mpi-inf.mpg.de}

[b] Fakultät für Informatik, Technische Universität München, Germany
e-mail address: boehmes@in.tum.de

[c] Department of Computer Science, School of Science and Technology, Middlesex University, UK
e-mail address: a.popescu@mdx.ac.uk

[d] Dept. of CSE, Chalmers University of Technology, Gothenburg, Sweden
e-mail address: nicsma@chalmers.se

Abstract. Many 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 when translating monomorphic to untyped first-order logic. 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 in Isabelle/HOL as part of the Sledgehammer tool. We include informal proofs of soundness and correctness, and have formalised the monomorphic part of this work in Isabelle/HOL. Our evaluation finds the new encodings vastly superior to previous schemes.

1. Introduction

Specification languages, proof assistants, and other theorem proving applications are typically based on polymorphism formalisms, but state-of-the-art automatic provers support only untyped or monomorphic logics. The existing sound (proof-reflecting) and complete (proof-preserving) translation schemes for polymorphic types, whether they revolve around functions (tags) or predicates (guards), produce clutter that severely hampers the proof search [22], and lighter approaches based on type arguments are unsound [22, 29]. As a result, application authors face a difficult choice between soundness and efficiency when interfacing with automatic provers.

The fourth author, together with Claessen and Lilliestrom [16], designed a pair of sound, complete, and efficient translations from monomorphic many-typed to untyped first-order logic with equality. The key insight is that *monotonic* types—types whose domain can be

2012 ACM CCS: [**Theory of computation**]: Logic—Automated reasoning.

LOGICAL METHODS IN COMPUTER SCIENCE
DOI:10.2168/LMCS-12(4:13)2016
© J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone
Creative Commons

extended with new elements while preserving satisfiability—can be merged. The remaining types can be made monotonic by introducing suitable protectors.

Example 1.1 (Monkey Village)**.** Imagine a village of monkeys [16] where each monkey owns at least two bananas. The predicate $\mathsf{owns} : \mathit{monkey} \times \mathit{banana} \to o$ (where o denotes truth values) associates monkeys with bananas, and the functions $\mathsf{b}_1, \mathsf{b}_2 : \mathit{monkey} \to \mathit{banana}$ witness the existence of each monkey's minimum supply of bananas:

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

The axioms are clearly 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 $\mathsf{g}_{\mathit{monkey}}$ to guard against ill-typed instantiations of M, M_1, and M_2:

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

The first axiom states the existence of a monkey; this is necessary for completeness, since model carriers are required to be nonempty. Thanks to monotonicity, it is sound to omit all type information regarding bananas. The intuition behind this is that the $\mathsf{g}_{\mathit{monkey}}$ predicate makes the problem fully monotonic, and for such problems it it possible to synchronise the cardinalities of the different types and to merge the types, yielding an equisatisfiable untyped (or singly typed) problem. For example, a model $\mathcal{M}$ of the typed problem with m monkeys and b bananas will give rise to a model $\mathcal{M}'$ of the untyped problem with b "bananamonkeys," among which m are monkeys according to the interpretation of $\mathsf{g}_{\mathit{monkey}}$; conversely, from a model of the untyped problem with b "bananamonkeys" including m values for which $\mathsf{g}_{\mathit{monkey}}$ is true (with $m > 0$ thanks to the first axiom), it is easy to construct a model of the typed problem with b bananas and m monkeys.

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

In addition, we generalise the monotonicity approach to a rank-1 polymorphic logic, as embodied by the typed first-order form TFF1 of the TPTP (Thousand of Problems for Theorem Provers) [7], a de facto standard implemented by a number of reasoning tools. 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

	Traditional (Polymorphic)	Cover-based (Polymorphic)	Monotonicity-based Monomorphic	Polymorphic
Full type erasure	e (§3.1)			
Type arguments	a (§3.2)			
Type tags	t (§3.3)	t@ (§4.2)	$\widetilde{\mathsf{t}}$?, $\widetilde{\mathsf{t}}$?? (§5.4)	t?, t?? (§6.5)
Type guards	g (§3.4)	g@ (§4.1)	$\widetilde{\mathsf{g}}$?, $\widetilde{\mathsf{g}}$?? (§5.5)	g?, g?? (§6.6)

Figure 1: The main encodings

observation that protectors are required only when translating the particular formulae 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 appealing to a more general notion of monotonicity, it is possible to eliminate all type information in the monkey village problem in a sound fashion.)

Encoding types in an untyped logic is an old problem, and several solutions have been proposed in the literature. We start by reviewing four main traditional approaches (Section 3), which prepare the ground for the more advanced encodings presented in this article. Next, we present improvements of the traditional encodings that aim at reducing the clutter associated with polymorphic symbols, based on the novel notion of "cover" (Section 4). Then we move our attention to monotonicity-based encodings, which try to minimise the number of added tags or guards. We first present known and novel monotonicity-based schemes that handle only ground types (Section 5); these are interesting in their own right and serve as stepping stones for the full-blown polymorphic encodings (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 for full type erasure, a for type arguments, t for type tags, g for type guards). The nontraditional encodings append a suffix to the letter: @ (= cover-based), ? (= monotonicity-based, lightweight), or ?? (= monotonicity-based, featherweight). 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. [16]; the other encodings are novel.

A formalisation [8] of the monomorphic part of our results has been developed in the proof assistant Isabelle/HOL [24, 25]. The encodings have been implemented in Sledgehammer [3, 22], which provides a bridge between Isabelle/HOL and automatic theorem provers (Section 7). They were evaluated with Alt-Ergo, E, SPASS, Vampire, and Z3 on a 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).

We presented an earlier version of this work at the TACAS 2013 conference [5]. The current text extends the conference paper with detailed proofs and a discussion of implementational issues (Section 7). It also corrects the side conditions of two encoding definitions, which resulted in an unexpected incompleteness.

Convention 1.2. Given a name, such as t, that ranges over a certain domain, such as the set of terms, its overlined version $\bar{t}$ ranges over lists (tuples) of items in this domain, and $|\bar{t}|$ denotes the length of $\bar{t}$. We also write $|A|$ for the cardinal of a set A. A set is *countable* if

it is finite or countably infinite. Given an element u, $(u)^n$ or u^n denotes the list consisting of n occurrences of u. Given a nonempty set A, we write $\varepsilon(A)$ for some arbitrary but fixed element of A. We use $\varepsilon(A)$ in definitions where the choice of the element does not matter, as long as it belongs to A.

2. Background: Logics

This article involves three versions of classical first-order logic with equality: polymorphic, monomorphic, and untyped. They correspond to the TPTP syntaxes TFF1 [7], TFF0 [32], and FOF [30], respectively, excluding interpreted arithmetic.

2.1. Polymorphic First-Order Logic.

The source logic is a rank-1 polymorphic logic as specified by TFF1 [7].

We fix $\mathcal{A}$, a countably infinite set of *type variables* with typical element α, and $\mathcal{V}$, a countably infinite set of *term variables* with typical element X.

Definition 2.1 (Syntax)**.** A *polymorphic signature* is a triple $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$, where $\mathcal{K}$ is a countable set of type constructors k with arities, $\mathcal{F}$ is a countable set of function symbols f with arities, and $\mathcal{P}$ is a countable set of predicate symbols p with arities.

For type constructors $k \in \mathcal{K}$, the arity is a natural number n. This association is written $k :: n$. Types, forming the set $\mathit{Type}_{\mathcal{K}}$, are then defined inductively starting with type variables and applying type constructors according to their arities:

Types:

σ	$::=$ $k(\bar{\sigma})$	where $k :: \lvert\bar{\sigma}\rvert$	constructor type
	$\mid$ α		type variable

For function symbols $f \in \mathcal{F}$, the arity is a triple $(\bar{\alpha}, \bar{\sigma}, \sigma)$, where $\bar{\alpha}$ is a list of distinct type variables, $\bar{\sigma}$ is a list of types, and σ is a type such that all the variables appearing in $\bar{\sigma}$ and σ are among the ones in $\bar{\alpha}$. We write this association as $f : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma$ or $f : \forall\bar{\alpha}.\ \sigma_1 \times \cdots \times \sigma_n \to \sigma$. Finally, for predicate symbols $p \in \mathcal{P}$, the arity is a pair $(\bar{\alpha}, \bar{\sigma})$, where $\bar{\alpha}$ is a list of distinct type variables and $\bar{\sigma}$ is a list of types. We write this association as $p : \forall\bar{\alpha}.\ \bar{\sigma} \to o$ or $p : \forall\bar{\alpha}.\ \sigma_1 \times \cdots \times \sigma_n \to o$.

Overloading of different arities for the same symbol is excluded by definition. The symbols $\forall$, $\times$, $\to$, and o (omicron) are not type constructors but syntax. The arity declarations for function and predicate symbols can be seen as instances of the general syntax $s : \forall\bar{\alpha}.\ \bar{\sigma} \to \varsigma$, where $s \in \mathcal{F} \uplus \mathcal{P}$ and ς is either a type or o. An application of s will require $\lvert\bar{\alpha}\rvert$ type arguments (written in angle brackets) and $\lvert\bar{\sigma}\rvert$ term arguments (written in parentheses). In examples, we may omit type arguments from $\bar{\alpha}$ that are irrelevant or clear from the context.

A *typed* (*term*) *variable* is a pair of a variable and a type, written X^σ. We fix $\mathcal{V}_{\text{typed}}$, the set of type variables. The terms and formulae are defined below.

Terms:

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

Formulae:

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

All type quantification is universal. We also impose the following syntactic restriction, not captured in the formation rules: type quantifiers may only occur at the top of the formula, i.e. not underneath a term quantifier or a connective, and formulae are in negation normal form (NNF), which is most suitable for defining and reasoning about translations. We sometimes use implication $\varphi_1 \wedge \cdots \wedge \varphi_m \to \psi_1 \vee \cdots \vee \psi_n$ as an abbreviation for $\neg\varphi_1 \vee \cdots \vee \neg\varphi_m \vee \psi_1 \vee \cdots \vee \psi_n$ to enhance readability. In examples, we freely nest quantifiers and connectives.

Definition 2.2 (Free and Fresh Variables, Groundness, and Sentences)**.** $\mathrm{TVars}(\sigma)$, $\mathrm{TVars}(t)$, and $\mathrm{FTVars}(\varphi)$ denote the sets of type variables occurring in the type σ, occurring in the term t, and occurring freely in the formula φ, respectively. Similarly, $\mathrm{Vars}(t)$ and $\mathrm{FVars}(\varphi)$ denote the sets of typed term variables occurring in the term t and occurring freely in the formula φ, respectively. For example, $\mathrm{TVars}(X^\sigma) = \mathrm{TVars}(\sigma)$, $\mathrm{Vars}(X^\sigma) = \{X^\sigma\}$, $\mathrm{FVars}(\forall X{:}\sigma.\ \varphi) = \mathrm{FVars}(\exists X{:}\sigma.\ \varphi) = \mathrm{FVars}(\varphi) - \{X^\sigma\}$. A type σ is *ground* if $\mathrm{TVars}(\sigma) = \emptyset$. We let GType_Σ denote the set of ground types. A term t is *ground* if $\mathrm{TVars}(t) = \emptyset$ and $\mathrm{Vars}(t) = \emptyset$. A formula φ is a *sentence* if $\mathrm{TVars}(\varphi) = \emptyset$ and $\mathrm{FVars}(\varphi) = \emptyset$.

Convention 2.3. We assume that the set of variables $\mathcal{V}$ is partitioned in two infinite sets: $\mathcal{V}^\forall$, of *universal* variables, and $\mathcal{V}^\exists$, of *existential* variables. The former are the only ones allowed to be universally quantified, and the latter are the only ones allowed to be existentially quantified. We let $\mathcal{V}^\forall_{\text{typed}}$ for the set of typed variable X^σ with $X \in \mathcal{V}^\forall$; and similarly for $\mathcal{V}^\exists_{\text{typed}}$. We also assume that each type or term variable is bound only once in a formula, and that if the typed variable X^σ appears in the scope of a quantification $\forall X : \sigma'$ or $\exists X : \sigma'$, then $\sigma = \sigma'$. The last assumption allows us to omit the superscript σ from X^σ in examples.

The typing rules and semantics of the logic are modelled after those of TFF1. 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 domains corresponding to ground types. All types are inhabited (nonempty).

Definition 2.4 (Type Substitution)**.** A *type substitution* ρ is a function that maps every type variable α to a type, written $\alpha\rho$. It is extended from type variables to types and type tuples in the standard way: $k(\bar{t})\rho = k(\bar{t}\rho)$ and $(t_1, \ldots, t_n)\rho = (t_1\rho, \ldots, t_n\rho)$.

Definition 2.5 (Typing Rules)**.** A judgement $t : \sigma$ expresses that the term t is *well typed* and has type σ. A judgement $\varphi : o$ expresses that the formula φ is *well typed*. The *typing rules* of polymorphic first-order logic are given below:

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

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

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

$$\frac{\varphi_1 : o \quad \varphi_2 : o}{\varphi_1 \vee \varphi_2 : o} \qquad \frac{\varphi : o}{\forall X : \sigma.\, \varphi \;:\; o} \qquad \frac{\varphi : o}{\exists X : \sigma.\, \varphi \;:\; o} \qquad \frac{\varphi : o}{\forall \alpha.\, \varphi \;:\; o}$$

Definition 2.6 (Problem)**.** A *problem* is a set of (well-typed) sentences.

Lemma 2.7. *For all terms t, there exists at most one type σ such that $t : \sigma$.*

Proof. Immediate by induction on t. □

Convention 2.8. We write t^σ to indicate that the term t is well typed and its (unique) type is σ. This convention is consistent with the notation X^σ for term variables of type σ.

Definition 2.9 (Semantics)**.** Let $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ be a polymorphic signature. A *structure* $\mathcal{M}$ for Σ is a tuple of families $\big(\mathbb{D}, (k^{\mathcal{M}})_{k\in\mathcal{K}}, (f^{\mathcal{M}})_{f\in\mathcal{F}}, (p^{\mathcal{M}})_{p\in\mathcal{P}}\big)$, where

- $\mathbb{D}$ is a nonempty collection of nonempty sets called the *domains*;
- if $k :: n$, then $k^{\mathcal{M}} : \mathbb{D}^n \to \mathbb{D}$. Given a *type variable valuation* $\theta : \mathcal{A} \to \mathbb{D}$, this induces an *interpretation* of types $[\![\ ]\!]^{\mathcal{M}}_\theta$ defined by the equations $[\![k(\bar{\sigma})]\!]^{\mathcal{M}}_\theta = k^{\mathcal{M}}([\![\bar{\sigma}]\!]^{\mathcal{M}}_\theta)$ and $[\![\alpha]\!]^{\mathcal{M}}_\theta = \theta(\alpha)$;
- if $f : \forall \alpha_1, \ldots, \alpha_m.\, \sigma_1 \times \cdots \times \sigma_n \to \sigma$, then $f^{\mathcal{M}} : \prod_{\bar{D}\in\mathbb{D}^m} [\![\sigma_1]\!]^{\mathcal{M}}_{\theta_{\bar{D}}} \times \cdots \times [\![\sigma_n]\!]^{\mathcal{M}}_{\theta_{\bar{D}}} \to [\![\sigma]\!]^{\mathcal{M}}_{\theta_{\bar{D}}}$, where $\theta_{\bar{D}}$ is a type variable valuation mapping each α_i to D_i;
- if $p : \forall \alpha_1, \ldots, \alpha_m.\, \sigma_1 \times \cdots \times \sigma_n \to o$, then $p^{\mathcal{M}} \subseteq \prod_{\bar{D}\in\mathbb{D}^m} [\![\sigma_1]\!]^{\mathcal{M}}_{\theta_{\bar{D}}} \times \cdots \times [\![\sigma_n]\!]^{\mathcal{M}}_{\theta_{\bar{D}}}$, where $\theta_{\bar{D}}$ is as above.

As expected from the arity of f, the interpretation $f^{\mathcal{M}}$ is a function first taking m type arguments and then n data (element) arguments. Similarly, the interpretation $p^{\mathcal{M}}$ is a predicate respecting the arity of p.

Given a type variable valuation θ and a compatible *term variable valuation* $\xi : \mathcal{V} \to \prod_{\sigma\in Type} [\![\sigma]\!]^{\mathcal{M}}_\theta$, the *interpretation* of terms and formulae by the structure $\mathcal{M}$ is as follows:

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

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

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

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

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 consisting of the three axioms and the negated conjecture is unsatisfiable. The conjecture is a consequence of the axioms.

We are interested in encoding polymorphic problems Φ in a manner that preserves and reflects their satisfiability. It will be technically convenient to assume that their signatures have at least one nullary type constructor, so that the set of ground types is nonempty. It is obvious that this assumption is harmless: if it is not satisfied, we simply extend the signature with a distinguished nullary type constructor $\iota :: 0$. Since ι does not appear in the formulae of Φ and since in models the set of domains is assumed to be nonempty, this signature extension does not affect its satisfiability: given a model of Φ in the original signature, we obtain one in the extended signature by interpreting ι as an arbitrary domain; given a model in the extended signature, we obtain one in the original signature by omitting the interpretation of ι.

Convention 2.11. For all polymorphic signatures $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ that we consider, we assume that $\mathcal{K}$ contains at least one nullary type constructor.

The following lemma shows that, in structures, the collection of domains can be regarded as a copy of the ground types.

Lemma 2.12. *If a polymorphic Σ-problem Φ has a model, it also has a model $\mathcal{M} = \big(\mathbb{D}, (k^{\mathcal{M}})_{k\in\mathcal{K}}, _, _\big)$ such that the following conditions are met*:

(1) *each $k^{\mathcal{M}}$ is injective, and $k^{\mathcal{M}}(\bar{D}) \neq k'^{\mathcal{M}}(\bar{E})$ whenever $k \neq k'$;*
(2) $\mathbb{D} = \{[\![\tau]\!]^{\mathcal{M}} \mid \tau \in \mathit{GType}_{\Sigma}\}$;
(3) *the type interpretation function $[\![\ ]\!]^{\mathcal{M}}$ is a bijection between GType and $\mathbb{D}$;*
(4) *$\mathbb{D}$ is countable*;
(5) *$\mathbb{D}$ is disjoint from each $D \in \mathbb{D}$, and any distinct $D_1, D_2 \in \mathbb{D}$ are disjoint.*

Proof. Assume Φ has a model $\mathcal{M}$. To prove (1), we construct a model $\mathcal{M}'$ from $\mathcal{M}$ by tagging the domains with types, i.e. by defining $\mathbb{D}' = \{D \times \{\sigma\} \mid D \in \mathbb{D}' \ \wedge\ \sigma \in \mathit{GType}_{\Sigma}\}$ and maintaining types across the application of type constructors: $k^{\mathcal{M}'}(D_1 \times \{\sigma_1\}, \ldots, D_n \times \{\sigma_n\}) = k^{\mathcal{M}}(D_1, \ldots, D_n) \times \{k(\sigma_1, \ldots, \sigma_n)\}$. The interpretations of the function and predicate symbols are adjusted accordingly. It is easy to prove that $\mathcal{M}$ and $\mathcal{M}'$ satisfy the same polymorphic formulae; in particular, $\mathcal{M}'$ is a model of Φ. For (2), we define a model $\mathcal{M}'' = (\mathbb{D}'', (k^{\mathcal{M}''})_{k\in\mathcal{K}}, _, _)$ from $\mathcal{M}'$ by taking $\mathbb{D}'' \subseteq \mathbb{D}'$ to be the image of $[\![\ ]\!]^{\mathcal{M}'} : \mathit{GType}_{\Sigma} \to \mathbb{D}'$ and by taking $k^{\mathcal{M}''}$ and $s^{\mathcal{M}''}$ to be the restrictions of $k^{\mathcal{M}'}$ and $s^{\mathcal{M}'}$. Thanks to Convention 2.11, $\mathbb{D}''$ is nonempty, and moreover $k^{\mathcal{M}''}$ is well defined on $\mathbb{D}$. Again, it is easy to prove that $\mathcal{M}''$ satisfies all the polymorphic formulae that $\mathcal{M}'$ satisfies. In particular, $\mathcal{M}''$ is a model of Φ. Points (3) and (4) follow from (1) and (2). Finally, we prove (5) by replacing the domains with disjoint copies of them. □

2.2. **Monomorphic First-Order Logic.** Monomorphic first-order logic, more commonly known as many-sorted first-order logic and corresponding to TPTP TFF0 [32], has signatures $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$, where Type is a countable set of types (or sorts) ranged over by σ, $\mathcal{F}$ is a countable set of function symbols $f : \bar{\sigma} \to \sigma$ with arities, and $\mathcal{P}$ is a countable set of predicate symbols $p : \bar{\sigma} \to o$ with arities. Σ-structures $\mathcal{M} = \big((D_\sigma)_{\sigma \in \mathit{Type}}, (f^{\mathcal{M}})_{f \in \mathcal{F}}, (p^{\mathcal{M}})_{p \in \mathcal{P}}\big)$ interpret the types as sets and the function and predicate symbols as functions and predicates of the suitable arities. Given a model $\mathcal{M}$ and a valuation $\xi : \mathcal{V} \to \prod_{\sigma \in \mathit{Type}} D_\sigma$, the interpretations of terms and formulae, $[\![t]\!]^{\mathcal{M}}_{\xi}$ and $[\![\varphi]\!]^{\mathcal{M}}_{\xi}$, are defined as expected.

Monomorphic first-order logic can be viewed as a special case of polymorphic first-order logic, with a polymorphic signature considered monomorphic when all its type constructors are nullary and the arities of its function and predicate symbols contain no type variables.

Example 2.13. A monomorphised version of the algebraic list problem of Example 2.10, with α instantiated by b, follows:

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

Like the original polymorphic problem, it is unsatisfiable.

2.3. **Untyped First-Order Logic.** The final target logic for all our encodings, untyped first-order logic, coincides with the TPTP first-order form FOF [30]. This is the logic traditionally implemented in automatic theorem provers and finite model finders. An *untyped signature* is a pair $\Sigma = (\mathcal{F}, \mathcal{P})$, where $\mathcal{F}$ and $\mathcal{P}$ are countable sets of function and predicate symbols with arities, where the arities are natural numbers. The notation s^n indicates that the symbol s has arity n. The untyped syntax is identical to that of the monomorphic logic, except that variable terms do not contain types and quantification is written $\forall X.\ \varphi$ and $\exists X.\ \varphi$. The structures for $\Sigma = (\mathcal{F}, \mathcal{P})$ are triples $\mathcal{M} = \big(D, (f^{\mathcal{M}})_{f \in \mathcal{F}}, (p^{\mathcal{M}})_{p \in \mathcal{P}}\big)$, where D is the domain and $f^{\mathcal{M}}$ and $p^{\mathcal{M}}$ are n-ary functions and predicates on $\mathbb{D}$, with n being the symbol's arity.

2.4. **Type Encodings.** The type encodings discussed in this article are given by functions that take problems Φ in a logic $\mathcal{L}$ to problems Φ' in a logic $\mathcal{L}'$, where $\mathcal{L}$ and $\mathcal{L}'$ are among the three logics introduced above.

Convention 2.14. Each of the considered encodings will be specified by the following data:

- a function that maps $\mathcal{L}$-signatures Σ to $\mathcal{L}'$-signatures Σ';
- for all $\mathcal{L}$-signatures Σ and problems Φ over Σ:
  - a (possibly empty) set Ax_Φ of sentences over Σ, the *axioms*, added by the translation;
  - a function $\langle\!\langle\ \rangle\!\rangle$ between formulae over Σ and formulae over Σ', the *formula translation*.

The formula translation is typically based on a *term translation* $\langle\!\langle\ \rangle\!\rangle$. The encoding $\langle\!\langle \Phi \rangle\!\rangle$ of a problem Φ is given by the union between the axioms and the componentwise translations: $\langle\!\langle \Phi \rangle\!\rangle = Ax_\Phi \cup \{\langle\!\langle \varphi \rangle\!\rangle \mid \varphi \in \Phi\}$.

Central to this article are the notions of soundness and completeness of an encoding:

Definition 2.15 (Correctness)**.** An encoding as above is *sound* for a class of problems $\mathcal{C}$ if satisfiability of $\Phi \in \mathcal{C}$ implies satisfiability of $\langle\!\langle \Phi \rangle\!\rangle$; it is *complete* for $\mathcal{C}$ if, given $\Phi \in \mathcal{C}$, satisfiability of $\langle\!\langle \Phi \rangle\!\rangle$ implies satisfiability of Φ; it is *correct* for $\mathcal{C}$ if it is both sound and complete (i.e. Φ and $\langle\!\langle \Phi \rangle\!\rangle$ are equisatisfiable). In case $\mathcal{C}$ is the class of all problems, we omit it and simply call the encoding sound, complete, or correct.

3. Traditional Type Encodings

There are four main traditional approaches to encoding polymorphic types: full type erasure, type arguments, type tags, and type guards [18,22,29,36]. Before introducing them, we first establish some conventions that will be useful throughout the article.

Convention 3.1. We will often need to extend signatures Σ with one or more of the following distinguished symbols. Whenever we employ them, we assume they are fresh with respect to Σ and have the indicated arities:

- a nullary type constructor ϑ;
- a function symbol $\mathsf{t} : \forall\alpha.\, \alpha \to \alpha$;
- a predicate symbol $\mathsf{g} : \forall\alpha.\, \alpha \to o$.

Terms of type ϑ will be used to represent the types of Σ, t will be used to tag terms with type information, and g will be used to guard formulae with type information.

Convention 3.2. Since the sets of type and term variables, $\mathcal{A}$ and $\mathcal{V}$, are countably infinite, we can fix a function from $\mathcal{A}$ to $\mathcal{V}$, $\alpha \mapsto \mathcal{V}(\alpha)$, such that

(1) it is injective, i.e. $\mathcal{V}(\alpha_1) = \mathcal{V}(\alpha_2)$ implies $\alpha_1 = \alpha_2$;
(2) it allows for an infinite supply of term variables that do not correspond to type variables, i.e. the set of term variables not having the form $\mathcal{V}(\alpha)$ is infinite;
(3) each $\mathcal{V}(\alpha)$ is a universal variable.

This function can be used to encode types as terms. Thanks to (2), we can safely assume that the source problems Φ do not contain variables of the form $\mathcal{V}(\alpha)$.

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 3.3 (Full Erasure e)**.** The *full type erasure* encoding e translates a polymorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ into an untyped problem over $\Sigma' = (\mathcal{F}', \mathcal{P}')$, where the symbols in Σ' have the same term arities as in Σ (but without type arguments). Thus, if $f : \forall\bar{\alpha}.\, \sigma_1 \times \ldots \times \sigma_n \to \sigma$ and $p : \forall\bar{\alpha}.\, \sigma_1 \times \ldots \times \sigma_n \to o$ are in $\mathcal{F}$ and $\mathcal{P}$, respectively, then f and p have arities n in $\mathcal{F}'$ and $\mathcal{P}'$, respectively. The encoding adds no axioms, and the term and formula translations $\langle\!\langle\ \rangle\!\rangle_{\mathsf{e}}$ are defined as follows:

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

Here and elsewhere, we omit the trivial cases where the function is simply applied to its subterms or subformulae, as in $\langle\!\langle \varphi_1 \wedge \varphi_2 \rangle\!\rangle_{\mathsf{e}} = \langle\!\langle \varphi_1 \rangle\!\rangle_{\mathsf{e}} \wedge \langle\!\langle \varphi_2 \rangle\!\rangle_{\mathsf{e}}$. Recall that, according to Section 2.4, the e translation of a problem Φ is simply the componentwise translation of its formulae: $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{e}} = \{\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{e}} \mid \varphi \in \Phi\}$.

Example 3.4. 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).

However, full type erasure is generally unsound in the presence of equality because equality can be used to encode cardinality constraints on domains. For example, the axiom $\forall U : \mathit{unit}.\ U \approx \mathsf{unity}$ 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; a contradiction is derivable from any disequality $t \not\approx u$ or any pair of clauses $p(\bar{t})$ and $\neg\, p(\bar{u})$. An expedient proposed by Meng and Paulson [22, §2.8], which they 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 more intuitive example might be $N \not\approx 0 \rightarrow N > 0$, 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 3.5 (Completeness of e)**.** *Full type erasure is complete.*

Proof. From a model $\mathcal{M}' = \big(D, (f^{\mathcal{M}'})_{f \in \mathcal{F}'}, (p^{\mathcal{M}'})_{p \in \mathcal{P}'}\big)$ of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{e}}$, we construct a structure $\mathcal{M} = \big(\mathbb{D}, (k^{\mathcal{M}})_{k \in \mathcal{K}}, (f^{\mathcal{M}})_{f \in \mathcal{F}}, (p^{\mathcal{M}})_{p \in \mathcal{P}}\big)$ for the signature of Φ by taking the same domain for all types and interpreting all instances of each polymorphic symbol in the same way as $\mathcal{M}$:

- $\mathbb{D} = \{D\}$, and $k^{\mathcal{M}}$ maps everything to D;
- if $s \in \mathcal{F} \uplus \mathcal{P}$, then $s^{\mathcal{M}}(\bar{D})(\bar{d}) = s^{\mathcal{M}'}(\bar{d})$.

Given $\theta : \mathcal{A} \rightarrow \mathbb{D}$ and $\xi : \mathcal{V} \rightarrow \prod_{\sigma \in \mathit{Type}_\Sigma} [\![\sigma]\!]^{\mathcal{M}}_{\theta}$, we define $\xi' : \mathcal{V} \rightarrow D$ by $\xi'(X) = \xi(X)(\alpha)$, where α is any type variable. (The choice of α is irrelevant because θ maps all type variables to D, the only element of $\mathbb{D}$.) The next facts follow by structural induction on t and φ (for arbitrary θ and ξ):

- $[\![t]\!]^{\mathcal{M}}_{\theta,\xi} = [\![\langle\!\langle t \rangle\!\rangle_{\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'}$;
- $[\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi} = [\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'}$.

In particular, for sentences, we have $[\![\varphi]\!]^{\mathcal{M}} = [\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{e}}]\!]^{\mathcal{M}'}$; and since $\mathcal{M}'$ is a model of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{e}}$, it follows that $\mathcal{M}$ is a model of Φ. □

By way of composition, the e encoding lies at the heart of all the encodings presented in this article. Given n encodings $x_1, \ldots, x_n$, we write $\langle\!\langle\ \rangle\!\rangle_{x_1;\ldots;x_n}$ for the composition $\langle\!\langle\ \rangle\!\rangle_{x_n} \circ \cdots \circ \langle\!\langle\ \rangle\!\rangle_{x_1}$. Typically, n will be 2 or 3 and x_n will be e. Moreover, x_i will be correct and will

transform the problem so that it belongs to a fragment for which x_{i+1} is also correct. This will ensure that the whole composition is correct. Finally, because $x_2, \ldots, x_n$ will always be fixed for a given x_1, we will call the entire composition $\langle\!\langle\ \rangle\!\rangle_{x_1;\ldots;x_n}$ the "x_1 encoding".

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

We now proceed with first encoding the types in isolation and then the typed terms.

Definition 3.6 (Term Encoding of Types)**.** Let $\mathcal{K}$ be a finite set of n-ary type constructors. The *term encoding* of a polymorphic type over $\mathcal{K}$ is a term over the signature $(\{\vartheta\}, \mathcal{K}', \emptyset)$, where $\mathcal{K}'$ contains a function symbol $\mathsf{k} : \vartheta^n \to \vartheta$ for each $k \in \mathcal{K}$ with $k :: n$. 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 3.7 (Traditional Arguments a)**.** We first define the encoding function $\langle\!\langle\ \rangle\!\rangle_{\mathsf{a}}$, which translates polymorphic problems over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ to polymorphic problems over $(\mathcal{K} \uplus \{\vartheta\}, \mathcal{F} \uplus \mathcal{K}', \mathcal{P})$, where ϑ and $\mathcal{K}'$ are as in Definition 3.6. It adds no axioms, and its term and formula translations are defined as follows:

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

(Again, we omit the trivial cases, e.g. $\langle\!\langle\forall X : \sigma.\,\varphi\rangle\!\rangle_{\mathsf{a}} = \forall X : \sigma.\,\langle\!\langle\varphi\rangle\!\rangle_{\mathsf{a}}$.) The *traditional type arguments* encoding a is defined as the composition $\langle\!\langle\ \rangle\!\rangle_{\mathsf{a};\mathsf{e}}$. It follows from the definition that 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 $\mathcal{F}, \mathcal{P}$; and for each symbol $s : \forall\bar\alpha.\,\bar\sigma \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$, the arity of s in Σ' is $|\bar\alpha| + |\bar\sigma|$.

Example 3.8. The a encoding translates the algebraic list problem of Example 2.10 into the following untyped problem:

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

The a encoding coincides with e for monomorphic problems and suffers from the same unsoundness with respect to equality and cardinality constraints. Nonetheless, a will form the basis of all the sound polymorphic encodings in a slightly generalised version, called a^x, for suitable instances of x.

Definition 3.9 (Type Argument Filter)**.** Given a signature $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$, a *type argument filter* x maps any $s : \forall\alpha_1, \ldots, \alpha_m.\ \bar{\sigma} \to \varsigma$ to a subset $\mathsf{x}_s = \{i_1, \ldots, i_{m'}\} \subseteq \{1, \ldots, m\}$ of its type argument indices. Given a list $\bar{z}$ of length m, $\mathsf{x}_s(\bar{z})$ 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 $\mathsf{t}, \mathsf{g} \notin \mathcal{F} \uplus \mathcal{P}$ introduced in Convention 3.1.

Definition 3.10 (Generic Arguments a^{x})**.** Given a type argument filter x, we first define the encoding $\langle\!\langle\ \rangle\!\rangle_{\mathsf{a}^{\mathsf{x}}}$ that translates polymorphic problems over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ to polymorphic problems over $(\mathcal{K} \uplus \{\vartheta\}, \mathcal{F} \uplus \mathcal{K}', \mathcal{P})$, where ϑ and $\mathcal{K}'$ are as in Definition 3.6. It adds no axioms, and its term and formula translations are defined as follows:

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

The *generic type arguments* encoding a^{x} is the composition $\langle\!\langle\ \rangle\!\rangle_{\mathsf{a}^{\mathsf{x}};\mathsf{e}}$. It 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 $\mathcal{F}, \mathcal{P}$; and for each symbol $s : \forall\bar{\alpha}.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$, the arity of s in Σ' is $|\mathsf{x}_s| + |\bar{\sigma}|$.

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

Theorem 3.11 (Completeness of a^{x})**.** *The type arguments encoding* a^{x} *is complete.*

Proof. Recall that a^{x} is defined as the composition of $\langle\!\langle\ \rangle\!\rangle_{\mathsf{a}^{\mathsf{x}}}$ and $\langle\!\langle\ \rangle\!\rangle_{\mathsf{e}}$. Since $\langle\!\langle\ \rangle\!\rangle_{\mathsf{e}}$ is complete by Theorem 3.5, it suffices to show that $\langle\!\langle\ \rangle\!\rangle_{\mathsf{a}^{\mathsf{x}}}$ is complete. Let $\mathcal{M}' = \big(\mathbb{D}', (k^{\mathcal{M}'})_{k \in \mathcal{K}' \uplus \{\vartheta\}}, (f^{\mathcal{M}'})_{f \in \mathcal{F} \uplus \mathcal{K}'}, (p^{\mathcal{M}'})_{p \in \mathcal{P}}\big)$ be a model of $\langle\!\langle\Phi\rangle\!\rangle_{\mathsf{a}^{\mathsf{x}}}$. We will construct a structure $\mathcal{M} = \big(\mathbb{D}, (k^{\mathcal{M}})_{k \in \mathcal{K}}, (f^{\mathcal{M}})_{f \in \mathcal{F}}, (p^{\mathcal{M}})_{p \in \mathcal{P}}\big)$ for the signature of Φ by taking the same domains as $\mathcal{M}'$, interpreting the type constructors other than ϑ in the same way, and interpreting the function and predicate symbols s as in $\mathcal{M}'$, but supplying, for the extra arguments, a suitable tuple from $[\![\vartheta]\!]^{\mathcal{M}'}$ that reflects the type arguments.

To this end, we first apply Lemma 2.12 to obtain that every element of $\mathbb{D}'$ is uniquely represented as $[\![\tau]\!]^{\mathcal{M}'}$ with $\tau \in GType_{\Sigma'}$. We define

- $\mathbb{D} = \mathbb{D}'$ and $k^{\mathcal{M}} = k^{\mathcal{M}'}$;
- if $s \in \mathcal{F} \uplus \mathcal{P}$, then

$$s^{\mathcal{M}}([\![\bar{\tau}]\!]^{\mathcal{M}'})(\bar{d}) = \begin{cases} s^{\mathcal{M}'}([\![\bar{\tau}]\!]^{\mathcal{M}'})([\![\langle\!\langle \mathsf{x}_s(\bar{\tau})\rangle\!\rangle]\!]^{\mathcal{M}'}, \bar{d}) & \text{if } \mathsf{x}_s(\bar{\tau}) \in GType^n_\Sigma \\ \text{anything} & \text{otherwise} \end{cases}$$

  where n is the length of $\mathsf{x}_s(\bar{\tau})$.

Note how the tuple $[\![\langle\!\langle \mathsf{x}_s(\bar{\tau})\rangle\!\rangle]\!]^{\mathcal{M}'}$ reflects the x_s-selection from the type arguments $[\![\bar{\tau}]\!]^{\mathcal{M}'}$.

Given $\theta : \mathcal{A} \to \mathbb{D}$ and $\xi : \mathcal{V} \to \prod_{\sigma \in Type_\Sigma} [\![\sigma]\!]^{\mathcal{M}}_\theta$, we define $\xi' : \mathcal{V} \to \prod_{\sigma' \in Type_{\Sigma'}} [\![\sigma']\!]^{\mathcal{M}'}_\theta$ by

$$\xi'(X)(\sigma') = \begin{cases} \xi'(X)(\sigma') & \text{if } \sigma' \in Type_\Sigma \\ [\![\langle\!\langle\tau\rangle\!\rangle]\!]^{\mathcal{M}'} & \text{if } \sigma' = \vartheta,\ X = \mathcal{V}(\alpha), \text{ and } \theta(\alpha) = [\![\tau]\!]^{\mathcal{M}'} \text{ for } \tau \in GType_\Sigma \\ \text{anything} & \text{otherwise} \end{cases}$$

The next facts follow by structural induction on t and φ (for arbitrary θ and ξ):

- $[\![t]\!]^{\mathcal{M}}_{\theta,\xi} = [\![\langle\!\langle t \rangle\!\rangle_{\mathsf{a}^x}]\!]^{\mathcal{M}'}_{\theta,\xi'}$;
- $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{a}^x}]\!]^{\mathcal{M}'}_{\theta,\xi'}$ implies $[\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi}$.

(The reason why for formula interpretation we have only implication and not equality is the $\forall\alpha$ case: when encoding a universally quantified formula $\forall\alpha.\,\varphi$, the result $\forall\alpha.\forall\langle\!\langle\alpha\rangle\!\rangle : \vartheta.\,\langle\!\langle\varphi\rangle\!\rangle_{\mathsf{a}^x}$ introduces quantification over two variables, α and $\langle\!\langle\alpha\rangle\!\rangle$, whose interpretations need not be synchronised. As a result, $\forall\alpha.\forall\langle\!\langle\alpha\rangle\!\rangle : \vartheta.\,\langle\!\langle\varphi\rangle\!\rangle_{\mathsf{a}^x}$ could be stronger than $\forall\alpha.\varphi$.) In particular, for a sentence φ, $[\![\langle\!\langle\varphi\rangle\!\rangle_{\mathsf{a}^x}]\!]^{\mathcal{M}'}$ implies $[\![\varphi]\!]^{\mathcal{M}}$, and hence $\mathcal{M}$ is a model of Φ because $\mathcal{M}'$ is a model of $\langle\!\langle\Phi\rangle\!\rangle_{\mathsf{a}^x}$. □

3.3. **Type Tags.** An intuitive approach to encode type information soundly (and 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 encoded as a term $\langle\!\langle\sigma\rangle\!\rangle$. 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 3.12 (Phantom Type Argument)**.** Let $s : \forall\alpha_1,\ldots,\alpha_m.\,\bar\sigma \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. The ith type argument is a *phantom* if α_i does not occur in $\bar\sigma$ or ς. Given a list $\bar z \equiv z_1,\ldots,z_m$, $\mathsf{phn}_s(\bar z)$ denotes the sublist $z_{i_1},\ldots,z_{i_{m'}}$ corresponding to the positions in $\bar\alpha$ of the phantom type arguments.

Definition 3.13 (Traditional Tags t)**.** We first define the encoding $\langle\!\langle\ \rangle\!\rangle_{\mathsf{t}}$ that translates polymorphic problems over $\Sigma = (\mathcal{K},\mathcal{F},\mathcal{P})$ to polymorphic problems over $(\mathcal{K},\mathcal{F} \uplus \{\mathsf{t} : \forall\alpha.\ \alpha \to \alpha\},\mathcal{P})$. It adds no axioms, and its term and formula translations are defined as follows:

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

The *traditional type tags* encoding t is the composition $\langle\!\langle\ \rangle\!\rangle_{\mathsf{t};\mathsf{a}^{\mathsf{phn}};\mathsf{e}}$. It translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}' \uplus \{\mathsf{t}^2\},\mathcal{P}')$, where $\mathcal{K}',\mathcal{F}',\mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{phn}}$ (i.e. a^x with $x = \mathsf{phn}$).

Example 3.14. The t encoding translates the algebraic list problem of Example 2.10 into the following equisatisfiable untyped problem:

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

Since there are no phantoms in this example, $\mathsf{a}^{\mathsf{phn}}$ adds no extra arguments. All type information is carried by the t function's first argument.

Example 3.15. Consider the following formula, with $\mathsf{linorder} : \forall\alpha.\,o$ and $\mathsf{less_eq} : \forall\alpha.\ \alpha\times\alpha \to o$:

$$\forall\alpha.\, \forall X : \alpha,\, Y : \alpha.\ \mathsf{linorder}\langle\alpha\rangle \to \mathsf{less_eq}(X, Y) \vee \mathsf{less_eq}(Y, X)$$

The α variable in linorder's arity declaration is a phantom. The t encoding preserves it as an explicit term argument:

$$\forall A, X, Y.\ \mathsf{linorder}(A) \to \mathsf{less_eq}(\mathsf{t}(A, X), \mathsf{t}(A, Y)) \vee \mathsf{less_eq}(\mathsf{t}(A, Y), \mathsf{t}(A, X))$$

As the formula suggests, phantom type arguments can be used to encode predicates on types, mimicking type classes [35].

We postpone the proof of the following theorem to Section 4.2, in the context of our improved encodings:

Theorem 3.16 (Correctness of t)**.** *The traditional type tags encoding* t *is correct.*

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. The terms are smaller than with tags, but the formulae contain more disjuncts.

With the type tags encoding, only phantom type arguments need 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 3.17 (Inferable Type Argument)**.** Let $s : \forall\alpha_1, \ldots, \alpha_m.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. A type argument α_j is *inferable* if it occurs in some of the term arguments' types, i.e. if there exists an index i such that α_j occurs in σ_i. Given a list $\bar{z} \equiv z_1, \ldots, z_m$, let $\mathsf{inf}_s(\bar{z})$ denote the sublist $z_{i_1}, \ldots, z_{i_{m'}}$ corresponding to the positions in $\bar{\alpha}$ of the inferable type arguments, and let $\mathsf{ninf}_s(\bar{z})$ denote the sublist for noninferable type arguments.

Observe that a phantom type argument is in particular noninferable, and a noninferable nonphantom type argument is one that appears in ς but not in $\bar{\sigma}$.

Definition 3.18 (Traditional Guards g)**.** We first define the encoding $\langle\!\langle\ \rangle\!\rangle_{\mathsf{g}}$, which translates a polymorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ into an untyped problem over $(\mathcal{K}, \mathcal{F}, \mathcal{P} \uplus \{\mathsf{g} : \forall\alpha.\, \alpha \to o\})$. Its term and formula translations are defined as follows:

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

The encoding also adds the following *typing axioms*:

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

(Following Convention 2.14, the translation of a problem is given by $\langle\!\langle\Phi\rangle\!\rangle_{\mathsf{g}} = \mathrm{Ax} \cup \{\langle\!\langle\varphi\rangle\!\rangle_{\mathsf{g}} \mid \varphi \in \Phi\}$, where Ax are the typing axioms.) The *traditional type guards* encoding g is defined as the composition $\langle\!\langle\ \rangle\!\rangle_{\mathsf{g};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$. It translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}', \mathcal{P}' \uplus \{\mathsf{g}^2\})$, where $\mathcal{K}', \mathcal{F}', \mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ninf}}$.

The last typing axiom in the above definition 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 3.19. The g encoding translates the algebraic list problem of Example 2.10 into the following:

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

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 guards, any cons, hd, or tl term could be typed with any type, defeating the purpose of the guard predicates.

Example 3.20. Consider the following formula, where $\mathsf{inl} : \forall \alpha, \beta.\ \alpha \rightarrow \mathit{sum}(\alpha,\beta)$ and $\mathsf{inr} : \forall \alpha, \beta.\ \beta \rightarrow \mathit{sum}(\alpha,\beta)$:

$$\forall \alpha,\beta.\ \forall X : \alpha,\ Y : \beta.\ \mathsf{inl}\langle\alpha,\beta\rangle(X) \not\approx \mathsf{inr}\langle\alpha,\beta\rangle(Y)$$

The β variable in inl's arity declaration and the α in inr's are noninferable. The g encoding preserves them as explicit term arguments:

$$\forall A, B, X, Y.\ \mathsf{g}(A, X) \wedge \mathsf{g}(B, Y) \rightarrow \mathsf{inl}(B, X) \not\approx \mathsf{inr}(A, Y)$$

Theorem 3.21 (Correctness of g)**.** *The traditional type guards encoding* g *is correct.*

Proof. This will be a consequence of Theorem 4.7 for our parameterised cover-based encoding g@, since g is a particular case of g@. □

Remark 3.22. The above encodings, as well as those discussed in the next sections, all lead to an untyped problem. An increasing number of automatic provers support monomorphic types, and it may seem desirable to exploit such support when it is available. With such provers, we can replace the e encoding with a variant that enforces a basic type discipline by distinguishing two types, ϑ (for encoded types) and ι (for encoded terms). An incomplete (non-proof-preserving) alternative is to perform heuristic monomorphisation (Section 5.6). Hybrid schemes that exploit monomorphic types, including interpreted types, are also possible and have been studied by other researchers (Section 9).

4. Cover-Based Encodings of Polymorphism

Type tags and guards considerably increase the size of the problems passed to the automatic provers, with a dramatic impact on their performance. A lot of the type information generated by the traditional encodings t and g is redundant. For example, t translates $\mathsf{cons}\langle\alpha\rangle(X, Xs)$ to $\mathsf{t}(\mathsf{list}(A), \mathsf{cons}(\mathsf{t}(A, X), \mathsf{t}(\mathsf{list}(A), Xs)))$, but intuitively only one of the three tags is necessary to specify the right type for the expression if we know the arity of cons. The cover-based encodings capitalise on this, by supplying only a minimum of protectors and

adding typing axioms that effectively compute the type of function symbols from a selection of their term arguments' types—the "cover".

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

Definition 4.1 (Cover)**.** Let $s : \forall \bar{\alpha}.\ \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$. A (*type argument*) *cover* $C \subseteq \{1, \ldots, |\bar{\sigma}|\}$ for s is a set of term argument indices such that any inferable type argument can be inferred from a term argument whose index belongs to C, i.e. for all j, if α_j appears in $\bar{\sigma}$, it also appears in some σ_i such that $i \in C$. A cover C of s is *minimal* if no proper subset of C is a cover for s; it is *maximal* if $C = \{1, \ldots, |\bar{\sigma}|\}$. We let Cover_s denote an arbitrary but fixed cover for s.

In practice, we would normally take a minimal cover for Cover_s to reduce clutter. Accordingly, $\{1\}$ and $\{2\}$ are minimal covers for $\mathsf{cons} : \forall \alpha.\ \alpha \times \mathit{list}(\alpha) \to \mathit{list}(\alpha)$, whereas $\{1, 2\}$ is a maximal cover.

Convention 4.2. As canonical cover, we arbitrarily choose $\mathrm{Cover}_{\mathsf{cons}} = \{1\}$.

The cover-based encoding g@ introduced below is a generalisation of the traditional encoding g. The two encodings coincide if Cover_s is chosen to be maximal for all symbols s. In contrast, the cover-based encoding t@ is not exactly a generalisation of t, although they share many ideas. For this reason, we momentarily depart from our general policy of considering tags before guards so that we can present the easier case first.

Intuitively, g@ and t@ ensure that each term argument position that is part of its enclosing function or predicate symbol's cover has a unique type associated with it, from which the omitted type arguments can be inferred. Thus, t@ translates $\mathsf{cons}\langle\alpha\rangle(X, Xs)$ to $\mathsf{cons}(\mathsf{t}(A, X), Xs)$ with a type tag around X, effectively protecting the term from an ill-typed instantiation of X that would result in the wrong type argument being inferred for cons.

There is no need to protect the second argument, Xs, since it is not part of the cover. We call variables that occur in their enclosing symbol's cover (and hence that "carry" some type arguments) "undercover variables". It may seem dangerous to allow ill-typed terms to instantiate Xs, but this is not an issue because such terms cannot contribute meaningfully to a proof. At most, they can act as witnesses for the existence of terms of given types, but even in that capacity they are not necessary.

Definition 4.3 (Undercover Variable)**.** The set of *undercover variables* $\mathrm{UV}(\varphi)$ of a formula φ is defined by the equations

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

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

4.1. **Cover-Based Type Guards.** The cover-based encoding g@ is similar to the traditional encoding g, except that it guards only undercover occurrences of variables.

Definition 4.4 (Cover Guards g@)**.** The encoding $\langle\!\langle\ \rangle\!\rangle_{\mathsf{g@}}$ is defined similarly to the encoding $\langle\!\langle\ \rangle\!\rangle_{\mathsf{g}}$ (used for the traditional g encoding) except for the $\forall$ case in its formula translation and the typing axioms. Namely, the $\forall$ case adds guards only for universally quantified variables that are undercover:

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

Moreover, the typing axioms take the cover into consideration:

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

The *cover-based type guards* encoding g@ is defined as the composition $\langle\!\langle\ \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$. It translates a polymorphic problem Φ over Σ into an untyped problem $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$ over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}', \mathcal{P}' \uplus \{\mathsf{g}^2\})$, where $\mathcal{K}', \mathcal{F}', \mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ninf}}$.

Example 4.5. If we choose the cover for cons as in Convention 4.2, the g@ encoding of the algebraic list problem is identical to the g encoding (Example 3.19), except that the guard $\mathsf{g}(\mathsf{list}(A), Xs)$ is omitted in typing axiom and one of the problem axioms:

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

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

Lemma 4.6. *Let $\mathcal{M} = \left(\mathbb{D}, (k^{\mathcal{M}})_{k \in \mathcal{K}}, (f^{\mathcal{M}})_{f \in \mathcal{F}}, (p^{\mathcal{M}})_{p \in \mathcal{P}}\right)$ be such that the domains in $\mathbb{D}$ are mutually disjoint and the type constructors $k^{\mathcal{M}}$ are injective. Assume $\bar{\alpha} = (\alpha_1, \ldots, \alpha_m)$, $\bar{\beta} = (\beta_1, \ldots, \beta_n)$, $\bar{\sigma} = (\sigma_1, \ldots, \sigma_u)$, $\bar{\tau} = (\tau_1, \ldots, \tau_v)$, and $s : \forall(\bar{\alpha}, \bar{\beta}).\ (\bar{\sigma}, \bar{\tau}) \to \varsigma$ in $\mathcal{F} \uplus \mathcal{P}$, such that the last n type arguments (corresponding to $\bar{\beta}$) are inferable and the first u term arguments (corresponding to $\bar{\sigma}$) constitute a cover for s. Let $(d_1, \ldots, d_u) \in (\bigcup_{D \in \mathbb{D}} D)^u$. Then there exists at most one tuple $\bar{E} = (E_1, \ldots, E_n) \in \mathbb{D}^n$ such that each d_i is in $[\![\sigma_i]\!]^{\mathcal{M}}_{\theta}$ for some θ that maps $\bar{\beta}$ to $\bar{E}$.*

Proof. From the inferability and cover assumptions, we have the condition $\mathrm{TVars}(\bar{\beta}) \subseteq \mathrm{TVars}(\bar{\sigma}, \bar{\tau}) = \mathrm{TVars}(\bar{\sigma})$ on type variables. Assume another such tuple $\bar{E}'$ exists. Let θ' be its corresponding substitution, and let $j \in \{1, \ldots, n\}$. By the type variable condition, there exists $i \in \{1, \ldots, u\}$ such that $\beta_j \in \mathrm{TVars}(\sigma_i)$; since $d_i \in [\![\sigma_i]\!]^{\mathcal{M}}_{\theta} \cap [\![\sigma_i]\!]^{\mathcal{M}}_{\theta'}$ and distinct domains are disjoint, we have $[\![\sigma_i]\!]^{\mathcal{M}}_{\theta} = [\![\sigma_i]\!]^{\mathcal{M}}_{\theta'}$, which, together with $\beta_j \in \mathrm{TVars}(\sigma_i)$ and the injectivity of the type constructors, implies $\theta(\beta_j) = \theta'(\beta_j)$, hence $E_j = E'_j$; and since j was arbitrary, we obtain $\bar{E} = \bar{E}'$, as desired. □

Theorem 4.7 (Correctness of g@)**.** *The cover-based type guards encoding* g@ *is correct.*

Proof. Let $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ be the signature of a polymorphic problem Φ.

Sound: Let $\mathcal{M} = \big(\mathbb{D}, (k^{\mathcal{M}})_{k\in\mathcal{K}}, (f^{\mathcal{M}})_{f\in\mathcal{F}}, (p^{\mathcal{M}})_{p\in\mathcal{P}}\big)$ be a model of Φ. By Lemma 2.12, we may assume that $\mathbb{D}$ is disjoint from each of its elements, its elements are mutually disjoint, and the type constructors $k^{\mathcal{M}}$ satisfy distinctness and injectivity.

We define a structure $\mathcal{M}' = \big(D', (f^{\mathcal{M}'})_{f\in\mathcal{F}'\uplus\mathcal{K}'}, (p^{\mathcal{M}'})_{p\in\mathcal{P}'\uplus\{\mathsf{g}^2\}}\big)$ for the untyped signature $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}', \mathcal{P}' \uplus \{\mathsf{g}^2\})$ as follows. D' is the union of the domains of $\mathcal{M}$ and their elements, $\mathsf{g}^{\mathcal{M}'}$ is the set membership relation, and the symbols of $\mathcal{M}'$ common to those of $\mathcal{M}$ try to emulate the $\mathcal{M}$ interpretation as closely as possible: $k^{\mathcal{M}'}$ acts like $k^{\mathcal{M}}$ on the $\mathbb{D}$ subset of D', and similarly for function and predicate symbols s whose noninferable type arguments in $\mathcal{M}$ become regular (term) arguments in $\mathcal{M}'$. The reason why we can omit the inferable type arguments of s is their recoverability from the type of s and the domains of the genuine term arguments. Additionally, when applying $s^{\mathcal{M}'}$ to inputs outside s's cover that do not belong to their proper domains in $\mathcal{M}$, we correct these inputs by replacing them with arbitrary inputs from the proper domains. We do this because, in formulae, these inputs will not be guarded by the encoding, but we will nevertheless want to infer the encoded formula holding in $\mathcal{M}'$ from the original formula holding in $\mathcal{M}$.

Formally, we define the components of $\mathcal{M}'$ as follows. First, $D' = \mathbb{D} \uplus (\bigcup_{D\in\mathbb{D}} D)$ and $\mathsf{g}^{\mathcal{M}'}(a, b) = (a \in \mathbb{D} \wedge b \in a)$. Assume $k :: n$ is in $\mathcal{K}$, meaning that k is an n-ary function symbol in $\mathcal{K}'$. Then

$$k^{\mathcal{M}'}(\bar{D}) = \begin{cases} k^{\mathcal{M}}(\bar{D}) & \text{if } \bar{D} \in \mathbb{D}^n \\ \varepsilon(\mathbb{D}) & \text{otherwise} \end{cases}$$

Assume $\bar{\alpha} = (\alpha_1, \ldots, \alpha_m)$, $\bar{\beta} = (\beta_1, \ldots, \beta_n)$, $\bar{\sigma} = (\sigma_1, \ldots, \sigma_u)$, $\bar{\tau} = (\tau_1, \ldots, \tau_v)$, and $s : \forall\bar{\alpha}, \bar{\beta}.\ \bar{\sigma} \times \bar{\tau} \to \varsigma$ is in $\mathcal{F} \uplus \mathcal{P}$, such that the first m type arguments are noninferable, the last n type arguments are inferable, and the first u term arguments constitute s's cover. (The general case, with arbitrary permutations of noninferable and cover arguments, can be handled similarly, albeit with heavier notation.) Then s is an $(m + u + v)$-ary symbol in $\mathcal{F}' \uplus \mathcal{P}'$. Let $\bar{D} = (D_1, \ldots, D_m) \in D'^m$, $\bar{d} = (d_1, \ldots, d_u) \in D'^u$ and $\bar{e} = (e_1, \ldots, e_v) \in D'^v$, and consider the following condition on domains:

$$(D_1, \ldots, D_m) \in \mathbb{D}^m \text{ and there exists } \bar{E} = (E_1, \ldots, E_n) \in \mathbb{D}^n \text{ such that each } d_i \text{ is in } [\![\sigma_i]\!]^{\mathcal{M}}_{\theta}, \\ \text{where } \theta \text{ maps } (\bar{\alpha}, \bar{\beta}) \text{ to } (\bar{D}, \bar{E})$$

Assuming the condition holds, by Lemma 4.6 there exists precisely one tuple $\bar{E}$ satisfying it. We let $\bar{e}' = (e'_1, \ldots, e'_v)$, where

$$e'_i = \begin{cases} e_i & \text{if } e_i \in [\![\tau_i]\!]^{\mathcal{M}}_{\theta} \\ \varepsilon([\![\tau_i]\!]^{\mathcal{M}}_{\theta}) & \text{otherwise} \end{cases}$$

Finally, we define

$$s^{\mathcal{M}'}(\bar{D}, \bar{d}, \bar{e}) = \begin{cases} s^{\mathcal{M}}(\bar{D}, \bar{E})(\bar{d}, \bar{e}') & \text{if the domain condition holds} \\ \varepsilon(\mathbb{D}) & \text{if the domain condition fails and } s \in \mathcal{F} \\ \varepsilon(o) & \text{if the domain condition fails and } s \in \mathcal{P} \end{cases}$$

We will show that $\mathcal{M}'$ is a model of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$. Let $\xi' : \mathcal{V} \to D'$ be a valuation that respects types, in the sense that $\xi'(\mathcal{V}(\alpha)) \in \mathbb{D}$ for all $\alpha \in \mathcal{A}$. We define $\theta : \mathcal{A} \to \mathbb{D}$ by $\theta(\alpha) = \xi'(\mathcal{V}(\alpha))$ and $\xi : \mathcal{V} \to \prod_{\sigma\in \mathit{Type}_{\Sigma}} [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ by $\xi(X)(\sigma) = \xi'(X)$ if $\xi'(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ and

$= \varepsilon([\![\sigma]\!]^{\mathcal{M}}_{\theta})$ otherwise. Facts (1)–(3) below follow by induction on σ, t or φ (for arbitrary ξ'), and (1′) is a consequence of (1) and the definition of $\mathsf{g}^{\mathcal{M}'}$:

(1) $[\![\langle\!\langle \sigma \rangle\!\rangle]\!]^{\mathcal{M}'}_{\xi'} = [\![\sigma]\!]^{\mathcal{M}}_{\theta}$;

(1′) $[\![\mathsf{g}(\langle\!\langle \sigma \rangle\!\rangle, X)]\!]^{\mathcal{M}'}_{\xi'} = (\xi'(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta})$;

(2) if $t \notin \mathcal{V}_{\text{typed}}$ and $\xi(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ for all $X^{\sigma} \in \mathrm{UV}(t)$, then $[\![\langle\!\langle t \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = [\![t]\!]^{\mathcal{M}}_{\theta,\xi}$;

(3) if $\xi'(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ for all $X^{\sigma} \in \mathrm{UV}(\varphi)$, then $[\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi}$ implies $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'}$.

Let us detail a few interesting cases in these proofs:

Inductive case for (2): Assume $t = f\langle\bar{\tau}\rangle(t_1, \ldots, t_n)$, and let $i \in \{1, \ldots, n\}$. If $t_i \notin \mathcal{V}_{\text{typed}}$, then the induction hypothesis applies to it, yielding

$$[\![\langle\!\langle t_i \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = [\![t_i]\!]^{\mathcal{M}}_{\theta,\xi}$$

If $t_i = X^{\sigma}$, then $[\![\langle\!\langle t_i \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = \xi'(X)$, $[\![t_i]\!]^{\mathcal{M}}_{\theta,\xi} = \xi(X)(\sigma)$, and we have two cases:

- If $\xi'(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$, then $\xi'(X)(\sigma) = \xi(X)$.
- Otherwise, by the assumptions $i \notin \mathrm{Cover}_f$, and hence the definition of $f^{\mathcal{M}'}$ effectively replaces the argument $\xi'(X)$ by $\varepsilon([\![\sigma]\!]^{\mathcal{M}}_{\theta})$, which also equals $\xi(X)(\sigma)$.

The above shows $[\![\langle\!\langle t \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = [\![t]\!]^{\mathcal{M}}_{\theta,\xi}$.

Disequality case for (3): The subcase when one of the terms is a variable and the other is not. Assume φ has the form $X^{\sigma} \not\approx t$. If $\xi'(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$, then $\xi'(X)(\sigma) = \xi(X)$, hence $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = [\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi}$. Otherwise, $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'}$ since $[\![\langle\!\langle t \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ (and hence is different from $\xi'(X)$).

Universal quantifier case for (3): Assume (A) $[\![\forall X : \sigma.\ \varphi]\!]^{\mathcal{M}}_{\theta,\xi}$. We must show $[\![\langle\!\langle \forall X : \sigma.\ \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'}$. Fix $d \in D$ and let $\xi'_1 = \xi'[X \to d]$.

- Assume $X \in \mathrm{UV}(\varphi)$. Then we assume $[\![\mathsf{g}(\langle\!\langle \sigma \rangle\!\rangle, X)]\!]^{\mathcal{M}'}_{\xi'}$, i.e. (B) $d \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$, and need to show $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'_1}$. From (B), we have $\xi(X)(\sigma) = \xi'(X)$, hence $\xi_1 = \xi[X \mapsto d]$; with (A), this implies $[\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi_1}$, and hence the desired fact follows from the induction hypothesis.
- Assume $X \notin \mathrm{UV}(\varphi)$. We must show $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'_1}$. If $d \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$, then the argument goes the same as above. So assume $d \notin [\![\sigma]\!]^{\mathcal{M}}_{\theta}$. Then $\xi(X)(\sigma) = \varepsilon([\![\sigma]\!]^{\mathcal{M}}_{\theta})$, hence $\xi_1 = \xi[X \mapsto \varepsilon([\![\sigma]\!]^{\mathcal{M}}_{\theta})]$; with (A), this implies $[\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi_1}$, and hence the desired fact follows from the induction hypothesis.

From (1′) and the definition of $f^{\mathcal{M}'}$, it follows that $\mathcal{M}'$ satisfies the necessary axioms, namely, $\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$ for each typing axiom φ. It remains to show that $\mathcal{M}'$ satisfies $\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$ for each $\varphi \in \Phi$. So let $\varphi \in \Phi$. Then $[\![\varphi]\!]^{\mathcal{M}}$. We pick any ξ' that respects types and has the property that $\xi'(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ for all $X^{\sigma} \in \mathrm{UV}(\varphi)$; by (3) and the fact that φ and $\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$ are sentences, it follows that $[\![\varphi]\!]^{\mathcal{M}}_{\xi,\theta}$, hence $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'}$, hence $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}$.

Complete: This part is easy. By Theorem 3.11, it suffices to show that $\langle\!\langle\ \rangle\!\rangle_{\mathsf{g@}}$ is complete. Let $\mathcal{M}' = \big(\mathbb{D}', (k^{\mathcal{M}'})_{k \in \mathcal{K}}, (f^{\mathcal{M}'})_{f \in \mathcal{F}}, (p^{\mathcal{M}'})_{p \in \mathcal{P} \uplus \{\mathsf{g}\}}\big)$ be a model of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{g@}}$, for which again we

may assume that the domains are mutually disjoint. We define $\mathcal{M} = \big(\mathbb{D}, (k^{\mathcal{M}})_{k\in\mathcal{K}}, (f^{\mathcal{M}})_{f\in\mathcal{F}}, (p^{\mathcal{M}})_{p\in\mathcal{P}}\big)$ by restricting the domains according to the guards and restricting the operations correspondingly. Namely, for each $D \in \mathbb{D}'$, let $D' = \{d \in D \mid \mathsf{g}^{\mathcal{M}'}(D)(d)\}$. We take

- $\mathbb{D} = \{D' \mid D \in \mathbb{D}'\}$;
- $k^{\mathcal{M}}(\bar{D}') = k^{\mathcal{M}'}(\bar{D})$;
- $s^{\mathcal{M}}(\bar{D}')(\bar{d}) = s^{\mathcal{M}'}(\bar{D})(\bar{d})$.

Thanks to the typing axioms, each D' is nonempty. Since the domains are disjoint, each D' determines its D uniquely; together with the typing axioms, this also ensures that each $k^{\mathcal{M}}$ and $s^{\mathcal{M}}$ are well defined. Now, $[\![U]\!]^{\mathcal{M}}_{\theta,\xi} = [\![\langle\!| U |\!\rangle_{\mathsf{g@}}]\!]^{\mathcal{M}'}_{\theta,\xi}$, where U is first a term and then a formula, follows by induction on U (for arbitrary θ and ξ). From this, we obtain that $\mathcal{M}$ is a model of $\langle\!| \Phi |\!\rangle_{\mathsf{g@}}$ by the usual route. □

4.2. **Cover-Based Type Tags.** The cover-based encoding t@ is similar to the traditional encoding t, except that it tags only undercover occurrences of variables and requires typing axioms to add or remove tags around function terms.

Definition 4.8 (Cover Tags t@)**.** The encoding $\langle\!| \, |\!\rangle_{\mathsf{t@}}$ translates polymorphic problems over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ to polymorphic problems over $(\mathcal{K}, \mathcal{F} \uplus \{\mathsf{t} : \forall\alpha.\ \alpha \to \alpha\}, \mathcal{P})$. Its term and formula translations are the following:

$$\begin{aligned}
\langle\!| f\langle\bar{\sigma}\rangle(\bar{t}) |\!\rangle_{\mathsf{t@}} &= f\langle\bar{\sigma}\rangle(\lfloor\langle\!| \bar{t} |\!\rangle_{\mathsf{t@}}\rfloor_f) \\
\langle\!| p\langle\bar{\sigma}\rangle(\bar{t}) |\!\rangle_{\mathsf{t@}} &= p\langle\bar{\sigma}\rangle(\lfloor\langle\!| \bar{t} |\!\rangle_{\mathsf{t@}}\rfloor_p) \qquad \langle\!| t_1 \approx t_2 |\!\rangle_{\mathsf{t@}} = \lfloor\langle\!| t_1 |\!\rangle_{\mathsf{t@}}\rfloor_{\approx} \approx \lfloor\langle\!| t_2 |\!\rangle_{\mathsf{t@}}\rfloor_{\approx} \\
\langle\!| \neg\, p\langle\bar{\sigma}\rangle(\bar{t}) |\!\rangle_{\mathsf{t@}} &= \neg\, p\langle\bar{\sigma}\rangle(\lfloor\langle\!| \bar{t} |\!\rangle_{\mathsf{t@}}\rfloor_p) \quad \langle\!| \exists X : \sigma.\ \varphi |\!\rangle_{\mathsf{t@}} = \exists X : \sigma.\ \mathsf{t}\langle\sigma\rangle(X) \approx X \wedge \langle\!| \varphi |\!\rangle_{\mathsf{t@}}
\end{aligned}$$

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

$$u_j = \begin{cases} \mathsf{t}\langle\sigma_j\rangle(t_j) & \text{if } j \in \mathrm{Cover}_s \text{ and } t_j \in \mathcal{V}^{\forall}_{\mathrm{typed}} \\ t_j & \text{otherwise} \end{cases}$$

taking $\mathrm{Cover}_{\approx} = \{1, 2\}$. The encoding adds the following typing axioms:

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

The *cover-based type tags* encoding t is the composition $\langle\!| \, |\!\rangle_{\mathsf{t@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$. It translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}' \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{K}', \mathcal{F}', \mathcal{P}'$ are as for $\mathsf{a}^{\mathsf{ninf}}$.

Example 4.9. The t@ encoding of Example 2.10 is as follows (again, choosing the cover for cons as in Convention 4.2):

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

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

Theorem 4.10 (Correctness of t@)**.** *The cover-based type tags encoding* t@ *is correct.*

Proof. Let $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ be the signature of a polymorphic problem Φ.

Sound: Let $\mathcal{M} = \big(\mathbb{D}, (k^{\mathcal{M}})_{k\in\mathcal{K}}, (f^{\mathcal{M}})_{f\in\mathcal{F}}, (p^{\mathcal{M}})_{p\in\mathcal{P}}\big)$ be a model of Φ. By Lemma 2.12, we may assume that $\mathbb{D}$ is disjoint from each of its elements, its elements are mutually disjoint, and the type constructors $k^{\mathcal{M}}$ satisfy distinctness and injectivity.

We define a structure $\mathcal{M}' = \big(D', (f^{\mathcal{M}'})_{f\in\mathcal{F}'\uplus\mathcal{K}'\uplus\{\mathsf{t}^2\}}, (p^{\mathcal{M}'})_{p\in\mathcal{P}'}\big)$ for $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}', \mathcal{P}' \uplus \{\mathsf{g}^2\})$ intended to be a model of $\langle\!\langle\Phi\rangle\!\rangle_{\mathsf{t@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$. The construction of $\mathcal{M}'$ proceeds very similarly to the case of guards from the proof of Theorem 4.7: D', $k^{\mathcal{M}'}$ for $k \in \mathcal{K}'$, and $s^{\mathcal{M}'}$ for $s \in \mathcal{F}' \uplus \mathcal{P}'$ are all defined in the same way as in that proof. It remains to define $\mathsf{t}^{\mathcal{M}'} : D' \times D' \to D'$:

$$\mathsf{t}^{\mathcal{M}'}(a, b) = \begin{cases} b & \text{if } a \in \mathbb{D} \text{ and } b \in a \\ \varepsilon(a) & \text{if } a \in \mathbb{D} \text{ and } b \notin a \\ \varepsilon(\mathbb{D}) & \text{if } a \notin \mathbb{D} \end{cases}$$

In the proof of Theorem 4.7, $\mathsf{g}^{\mathcal{M}'}(a, b)$ captured the notion that a is a domain of $\mathcal{M}$ and b an element in it. The same can now be expressed by $\mathsf{t}^{\mathcal{M}'}(a, b) = b$. Additionally, here $\mathsf{t}^{\mathcal{M}'}$ is useful for redirecting any ill-typed element b (one not in the first argument a) to a well-typed element $\varepsilon(a) \in a$.

To show that $\mathcal{M}'$ is a model of $\langle\!\langle\Phi\rangle\!\rangle_{\mathsf{t@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$, we proceed almost identically to the proof of Theorem 4.7. Starting with a valuation $\xi' : \mathcal{V} \to D'$ that respects types, we define ξ and θ in the same way and state facts (1)–(3) as for guards, but adding a condition about the "well-typedness" of the existential variables interpretation, since only universal variables (i.e. variables in $\mathcal{V}^{\forall}_{\text{typed}}$) are tagged by t@:

(1) $[\![\langle\!\langle\sigma\rangle\!\rangle]\!]^{\mathcal{M}'}_{\xi'} = [\![\sigma]\!]^{\mathcal{M}}_{\theta}$;

(2) if $t \notin \mathcal{V}_{\text{typed}}$ and $\xi(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ for all $X^{\sigma} \in \mathrm{UV}(t) \cup \mathcal{V}^{\exists}_{\text{typed}}$, then $[\![\langle\!\langle t\rangle\!\rangle_{\mathsf{t@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = [\![t]\!]^{\mathcal{M}}_{\theta,\xi}$;

(3) if $\xi'(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ for all $X^{\sigma} \in \mathrm{UV}(\varphi) \cup \mathcal{V}^{\exists}_{\text{typed}}$, then $[\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi}$ implies $[\![\langle\!\langle\varphi\rangle\!\rangle_{\mathsf{g@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'}$.

Moreover, (1′) from the proof of Theorem 4.7 is replaced with the following two facts:

(1′) $([\![\mathsf{t}(\langle\!\langle\sigma\rangle\!\rangle, X)]\!]^{\mathcal{M}'}_{\xi'} = \xi'(X)) = (\xi'(X) \in [\![\sigma]\!]^{\mathcal{M}}_{\theta})$;

(1″) $[\![\mathsf{t}(\langle\!\langle\sigma\rangle\!\rangle, X)]\!]^{\mathcal{M}'}_{\xi'} \in [\![\sigma]\!]^{\mathcal{M}}_{\theta}$.

Again, (1)–(3) follow by induction on the involved type, term, or formula; (1′) and (1″) follow from (1) and the definition of $\mathsf{t}^{\mathcal{M}'}$ (and are in turn used to establish (2) and (3)). The fact that $\mathcal{M}'$ satisfies $\langle\!\langle\varphi\rangle\!\rangle_{\mathsf{t@};\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$ for each $\varphi \in \Phi$ follows in the same way as in Theorem 4.7. It remains to show that $\mathcal{M}'$ satisfies the necessary axioms, namely, the $\langle\!\langle\ \rangle\!\rangle_{\mathsf{a}^{\mathsf{ninf}};\mathsf{e}}$-translations of the $\langle\!\langle\ \rangle\!\rangle_{\mathsf{t@}}$ typing axioms. This follows from (1′) and the definition of $f^{\mathcal{M}'}$.

Complete: This part is again similar to the corresponding one for guards. By Theorem 3.11, it suffices to show $\langle\!\langle\ \rangle\!\rangle_{\mathsf{t@}}$ complete. Let $\mathcal{M}' = \big(\mathbb{D}', (k^{\mathcal{M}'})_{k\in\mathcal{K}}, (f^{\mathcal{M}'})_{f\in\mathcal{F}\uplus\{\mathsf{t}\}}, (p^{\mathcal{M}'})_{p\in\mathcal{P}}\big)$ be a model of $\langle\!\langle\Phi\rangle\!\rangle_{\mathsf{t@}}$, for which we may assume that the domains are mutually disjoint. We define $\mathcal{M} = \big(\mathbb{D}, (k^{\mathcal{M}})_{k\in\mathcal{K}}, (f^{\mathcal{M}})_{f\in\mathcal{F}}, (p^{\mathcal{M}})_{p\in\mathcal{P}}\big)$ by restricting the domains not according to the guards as in the proof of Theorem 4.7, but according to the property that tags be identity: for each $D \in \mathbb{D}'$, let $D' = \{d \in D \mid \mathsf{t}^{\mathcal{M}'}(D)(d) = d\}$; we define $\mathbb{D} = \{D' \mid D \in \mathbb{D}'\}$. The

other components of $\mathcal{M}$ are defined as in Theorem 3.11, and the proof is analogous to that for guards. □

Unlike in the case of guards, the traditional type tag encoding t is not a particular case of the cover-based encoding t@, since only t@ introduces typing axioms, and also t further restricts the type arguments to phantom arguments. Nevertheless, we can provide a somewhat similar argument for t's correctness. Although the encoding is well known, we are not aware of any soundness proof in the literature.

Proof of Theorem 3.16 (Correctness of t).

SOUND: Let $\mathcal{M}$ be as in the proof of Theorem 4.10. We define $\mathcal{M}'$ similar to there, but with the following difference: D' contains not only the domains in $\mathbb{D}$ and their elements, but also the set P of "polymorphic values", i.e. functions that take a domain $D \in \mathbb{D}$ and return an element of D. The reason for the polymorphic values is that, due to the switch from noninferable arguments to phantom arguments, the domain of the result for $f^{\mathcal{M}'}$ with $f : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma$ in $\mathcal{F}$ will no longer be completely inferable from the arguments, but will miss precisely the nonphantom noninfrebale arguments, which correspond to type variables belonging to the result type σ but not to the argument types $\bar{\sigma}$. Consequently, the result of applying $f^{\mathcal{M}'}$ to "well-typed" arguments will be polymorphic values that wait for the result domain D and, if it has the form $[\![\sigma]\!]_\theta$ for an appropriate θ, return the result from $[\![\sigma]\!]_\theta$ according to the interpretation of $f^{\mathcal{M}'}$. Then the interpretation of the tag applied to polymorphic values will select the desired element by providing the domain. The reason why this approach works with t but not with t@ is that in t tags are applied everywhere, thus resolving immediately any polymorphic value emerging from an application of $f^{\mathcal{M}'}$.

Formally, we define the components of $\mathcal{M}'$ as follows. First, $k^{\mathcal{M}'}$ and $p^{\mathcal{M}'}$ are as in the proofs of Theorems 4.7 and 4.10. (For predicate symbols, the notions of phantom and noninferable type arguments coincide.) $D' = \mathbb{D} \cup E \cup P$, where $E = \bigcup_{D\in\mathbb{D}} D$ (the elements) and $P = \prod_{D\in\mathbb{D}} D$ (the polymorphic values). Moreover, $\mathrm{t}^{\mathcal{M}'} : D' \times D' \to D'$ is defined as follows:

$$\mathrm{t}^{\mathcal{M}'}(a,b) = \begin{cases} b & \text{if } a \in \mathbb{D} \text{ and } b \in a \\ \varepsilon(a) & \text{if } a \in \mathbb{D},\ b \in E \text{ and } b \notin a \\ b(a) & \text{if } a \in \mathbb{D} \text{ and } b \in P \\ \varepsilon(\mathbb{D}) & \text{otherwise} \end{cases}$$

Assume $\bar{\alpha} = (\alpha_1, \ldots, \alpha_m)$, $\bar{\gamma} = (\gamma_1, \ldots, \gamma_r)$, $\bar{\beta} = (\beta_1, \ldots, \beta_n)$, $\bar{\sigma} = (\sigma_1, \ldots, \sigma_u)$, $\bar{\tau} = (\tau_1, \ldots, \tau_v)$, and $s : \forall(\bar{\alpha}, \bar{\gamma}, \bar{\beta}).\ (\bar{\sigma}, \bar{\tau}) \to \varsigma$ is in $\mathcal{F} \uplus \mathcal{P}$, such that the first m type arguments are phantom, the middle r arguments are noninferable nonphantoms, the last n type arguments are inferable, and the first u term arguments constitute s's cover. (Again, the general case with arbitrary permutations of the arguments can be handled similarly.) Then s is an $(m+u+v)$-ary symbol in $\mathcal{F}' \uplus \mathcal{P}'$. Let $\bar{D} = (D_1, \ldots, D_m) \in D'^m$, $\bar{d} = (d_1, \ldots, d_u) \in D'^u$ and $\bar{e} = (e_1, \ldots, e_v) \in D'^v$, and consider the following condition on domains:

$$(D_1, \ldots, D_m) \in \mathbb{D}^m \text{ and there exists } \bar{E} = (E_1, \ldots, E_n) \in \mathbb{D}^n \text{ such that each } d_i \text{ is in } [\![\sigma_i]\!]^{\mathcal{M}}_\theta, \text{ where } \theta \text{ maps } (\bar{\alpha}, \bar{\beta}) \text{ to } (\bar{D}, \bar{E})$$

Assuming the condition holds, again by Lemma 4.6 (taking $\bar{\alpha}$ from there to be $(\bar{\alpha}, \bar{\gamma})$) there exists precisely one tuple $\bar{E}$ satisfying it, and we define the "correction" vector $\bar{e}'$ from $\bar{e}$ as

in the proof of Theorems 4.7 and 4.10. We now define

$$f^{\mathcal{M}'}(\bar{D},\bar{d},\bar{e}) = \begin{cases} \pi_{\bar{D},\bar{d},\bar{e}} & \text{if the domain condition holds} \\ \varepsilon(\mathbb{D}) & \text{otherwise} \end{cases}$$

where the polymorphic value $\pi_{\bar{D},\bar{d},\bar{e}} \in P$ is defined as follows:

$$\pi_{\bar{D},\bar{d},\bar{e}}(D) = \begin{cases} s^{\mathcal{M}}(\bar{D},\bar{G},\bar{E})(\bar{d},\bar{e}') & \text{if } D \text{ has the form } [\![\sigma]\!]^{\mathcal{M}}_{\theta} \text{ and } \theta \text{ assigns } (\bar{\alpha},\bar{\gamma},\bar{\beta}) \text{ to } (\bar{D},\bar{G},\bar{E}) \\ \varepsilon(\mathbb{D}) & \text{otherwise} \end{cases}$$

To show that $\mathcal{M}'$ is a model of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{t};\mathsf{a}^{\mathsf{phn}};\mathsf{e}}$, we proceed almost identically to the proof of Theorem 4.10. Starting with a valuation $\xi' : \mathcal{V} \to D'$ that respects types (as in the proofs of Theorems 4.7 and 4.10). We define $\theta : \mathcal{A} \to \mathbb{D}$ by $\theta(\alpha) = \xi'(\mathcal{V}(\alpha))$ and $\xi : \mathcal{V} \to \prod_{\sigma \in Type_\Sigma} [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ by $\xi(X)(\sigma) = \mathsf{t}^{\mathcal{M}'}([\![\sigma]\!]^{\mathcal{M}}_{\theta}, \xi'(X))$. Facts (1)–(3) below follow by induction on σ, t, or φ (for arbitrary ξ'):

(1) $[\![\langle\!\langle \sigma \rangle\!\rangle]\!]^{\mathcal{M}'}_{\xi'} = [\![\sigma]\!]^{\mathcal{M}}_{\theta}$;

(2) $[\![\langle\!\langle t \rangle\!\rangle_{\mathsf{t};\mathsf{a}^{\mathsf{phn}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = [\![t]\!]^{\mathcal{M}}_{\theta,\xi}$;

(3) $[\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi} = [\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{t};\mathsf{a}^{\mathsf{phn}};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'}$.

It follows by the usual route that $\mathcal{M}'$ is a model of each $\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{t};\mathsf{a}^{\mathsf{phn}};\mathsf{e}}$ with $\varphi \in \Phi$, hence of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{t};\mathsf{a}^{\mathsf{phn}};\mathsf{e}}$.

Complete: By Theorem 3.11, it suffices to show $\langle\!\langle\ \rangle\!\rangle_{\mathsf{t}}$ complete. Let $\mathcal{M}' = \big(\mathbb{D}', (k^{\mathcal{M}'})_{k\in\mathcal{K}}, (f^{\mathcal{M}'})_{f\in\mathcal{F}\uplus\{\mathsf{t}\}}, (p^{\mathcal{M}'})_{p\in\mathcal{P}}\big)$ be a model of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{t}}$, for which we may assume that the domains are mutually disjoint. We must construct a model $\mathcal{M} = \big(\mathbb{D}, (k^{\mathcal{M}})_{k\in\mathcal{K}}, (f^{\mathcal{M}})_{f\in\mathcal{F}}, (p^{\mathcal{M}})_{p\in\mathcal{P}}\big)$ of Φ by somehow removing the tags from $\mathcal{M}'$. Unlike for $\mathsf{t@}$, here $\langle\!\langle\ \rangle\!\rangle_{\mathsf{t}}$ does not contain typing axioms ensuring that a the restriction of $\mathcal{M}'$ to elements for which the tag interpretation is the identity forms a valid submodel; thus $\mathcal{M}'$ cannot be defined that way. On the other hand, we know that the formulae in $\langle\!\langle\ \rangle\!\rangle_{\mathsf{t}}$ are fully tagged; in particular, all variables are accessed through tags. This means we can take the domains of $\mathcal{M}$ by restricting those of $\mathcal{M}'$ to the images of the tag interpretations. The result will not be a submodel, so cannot take the functions and predicates $s^{\mathcal{M}}$ of $\mathcal{M}$ to be restrictions of those of $\mathcal{M}'$. Instead, $s^{\mathcal{M}}$ will be defined by applying $s^{\mathcal{M}'}$ through the tag "interface".

Formally, let, for each $D \in \mathbb{D}'$, let D' be the image of $\mathsf{t}^{\mathcal{M}'}(D)$ (which is a subset of D). Each D' uniquely determines its D. We define the auxiliary function $T : \prod_{D\in\mathbb{D}'} D \to D$, which only applies $\mathsf{t}^{\mathcal{M}'}$ if the element is not already in the image:

$$T_D(d) = \begin{cases} d & \text{if } d \in D' \\ \mathsf{t}^{\mathcal{M}'}(D)(d) & \text{otherwise} \end{cases}$$

Notice that T_D adds no tags around an existing tag: $T_D(\mathsf{t}^{\mathcal{M}'}(D)(d)) = \mathsf{t}^{\mathcal{M}'}(D)(d)$. We define the components of $\mathcal{M}$ as follows: $\mathbb{D} = \{D' \mid D \in \mathbb{D}'\}$; $k^{\mathcal{M}}(\bar{D}') = k^{\mathcal{M}'}(\bar{D})$; if $f : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma$ is in $\mathcal{F}$, then $f^{\mathcal{M}}(\bar{D}')(d_1,\ldots,d_n) = \mathsf{t}^{\mathcal{M}'}([\![\sigma]\!]^{\mathcal{M}'}_{\theta})(f^{\mathcal{M}'}(\bar{D})(T_{[\![\sigma_1]\!]^{\mathcal{M}'}_{\theta}}(d_1),\ldots, T_{[\![\sigma_n]\!]^{\mathcal{M}'}_{\theta}}(d_n)))$, where θ maps each α_i to D_i; if $p : \forall\bar{\alpha}.\ \bar{\sigma} \to o$ is in $\mathcal{P}$, then $p^{\mathcal{M}}(\bar{D}')(d_1,\ldots,d_n) = p^{\mathcal{M}'}(\bar{D})(T_{[\![\sigma_1]\!]^{\mathcal{M}'}_{\theta}}(d_1),\ldots,T_{[\![\sigma_n]\!]^{\mathcal{M}'}_{\theta}}(d_n))$, where θ maps each α_i to D_i. The interpretation $f^{\mathcal{M}}$ applies tags $\mathsf{t}^{\mathcal{M}'}$ at the top, whereas the interpretations of $f^{\mathcal{M}}$ and $p^{\mathcal{M}}$ apply the modified tag T at the bottom. This is to ensure that the interpretations of terms and atomic formulae in $\mathcal{M}$ do not add several consecutive layers of tags. For example, when interpreted in $\mathcal{M}$

within $f(X^\sigma)$, f should add a tag around X^σ, as well as one around $f(X^\sigma)$; however, when interpreted in $\mathcal{M}$ within $f(g(X^\sigma))$, f should not add a tag around $g(X^\sigma)$, since g already adds one.

Given $\theta : \mathcal{A} \to \mathbb{D}$ and $\xi' : \mathcal{V} \to \prod_{\sigma \in Type_{\Sigma'}} [\![\sigma]\!]^{\mathcal{M}'}_{\theta'}$, we define $\theta' : \mathcal{A} \to \mathbb{D}'$ by letting $\theta'(\alpha)$ be the unique $D \in \mathbb{D}'$ such that $D' = \theta(\alpha)$, and $\xi : \mathcal{V} \to \prod_{\sigma \in Type_{\Sigma}} [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ by $\xi(X)(\sigma) = \mathsf{t}^{\mathcal{M}'}([\![\sigma]\!]^{\mathcal{M}'}_{\theta})(\xi'(X)(\sigma))$. The next facts follow by induction on σ, t, or φ (for arbitrary θ and ξ'):

(1) $[\![\sigma]\!]^{\mathcal{M}}_{\theta}$ is in the image of $\mathsf{t}^{\mathcal{M}'}([\![\sigma]\!]^{\mathcal{M}'}_{\theta'})$;
(2) $[\![t]\!]^{\mathcal{M}}_{\theta,\xi} = [\![\langle\!\langle t \rangle\!\rangle_{\mathsf{t}}]\!]^{\mathcal{M}'}_{\theta',\xi}$;
(3) $[\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi} = [\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{t}}]\!]^{\mathcal{M}'}_{\theta',\xi}$.

Let $\varphi \in \Phi$. To show that $\mathcal{M}$ is a model of φ, let $\theta : \mathcal{A} \to \mathbb{D}$ and $\xi : \mathcal{V} \to \prod_{\sigma \in Type_{\Sigma}} [\![\sigma]\!]^{\mathcal{M}}_{\theta}$. By (1), there exists $\xi' : \mathcal{V} \to \prod_{\sigma \in Type_{\Sigma'}} [\![\sigma]\!]^{\mathcal{M}'}_{\theta'}$ such that $\xi(X)(\sigma) = \mathsf{t}^{\mathcal{M}'}([\![\sigma]\!]^{\mathcal{M}'}_{\theta})(\xi'(X)(\sigma))$ for all X and σ—simply take $\xi'(X)(\sigma)$ to be any element d' in $[\![\sigma]\!]^{\mathcal{M}'}_{\theta'}$ such that $\mathsf{t}^{\mathcal{M}'}([\![\sigma]\!]^{\mathcal{M}'}_{\theta'})(d') = \xi(X)(\sigma)$. Since $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{t}}]\!]^{\mathcal{M}'}_{\theta',\xi}$ holds, it follows from (3) that $[\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi}$, i.e. $[\![\varphi]\!]^{\mathcal{M}}$, also holds, as desired. □

4.3. **Polymorphic Löwenheim–Skolem.** The previous completeness results can be used to prove Löwenheim–Skolem-like properties for polymorphic first-order logic by appealing to the more manageable (and better known) monomorphic first-order logic.

Lemma 4.11. *If a polymorphic Σ-problem Φ has a model, it also has a model $\mathcal{M} = \big(\mathbb{D}, (k^{\mathcal{M}})_{k \in \mathcal{K}}, _, _\big)$ such that the following conditions are met:*[1]
(1) *each $D \in \mathbb{D}$ is countable;*
(2) *each $k^{\mathcal{M}}$ is injective, and $k^{\mathcal{M}}(\bar{D}) \neq k'^{\mathcal{M}}(\bar{E})$ whenever $k \neq k'$;*
(3) $\mathbb{D} = \{[\![\tau]\!]^{\mathcal{M}} \mid \tau \in GType_{\Sigma}\}$;
(4) *the type interpretation function $[\![\ ]\!]^{\mathcal{M}}$ is a bijection between $GType$ and $\mathbb{D}$;*
(5) *$\mathbb{D}$ is countable;*
(6) *$\mathbb{D}$ is disjoint from each $D \in \mathbb{D}$, and any distinct $D_1, D_2 \in \mathbb{D}$ are disjoint.*

Proof. Assume Φ has a model, and recall that by definition the polymorphic signatures that we consider are countable. By the soundness of g@ (Theorem 4.7), $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{g@}}$ also has a model and its signature is countable; hence by classical Löwenheim–Skolem [19], $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{g@}}$ has a countable model $\mathcal{M}'$. It follows from the proof of completeness of g@ that Φ has a model $\mathcal{M} = (\mathbb{D}, _, _, _)$ constructed from $\mathcal{M}'$ in such a way that each $D \in \mathbb{D}$ is countable—hence $\mathcal{M}'$ satisfies (1). Now, (2)–(6) follow by applying the same reasoning as in Lemma 2.12 and noticing that all the structure modifications from there do not alter the countability of the domains $D \in \mathbb{D}$. □

[1]These correspond to the conditions from Lemma 2.12 plus countability of each $D \in \mathbb{D}$.

Lemma 4.12. *If Φ has a model $\mathcal{M} = (\mathbb{D}, _, _, _)$ where all $[\![\tau]\!]^{\mathcal{M}}$ are infinite for all $\tau \in GType$, it also has a model $\mathcal{M}' = (\mathbb{D}', _, _, _)$ where $\mathbb{D}'$ and all $D' \in \mathbb{D}'$ are countably infinite.*

Proof. Let Φ' be Φ extended with axioms $Ax = \{\exists X_1 : \sigma, \ldots, X_n : \sigma.\ \bigwedge_{i<j} X_i \not\approx X_j \mid \sigma \in GType_\Sigma, n \in \mathbb{N}\}$ that state that all ground types are infinite. Then $\mathcal{M} = (\mathbb{D}, _, _, _)$ is a model of Φ'. By Lemma 4.11, Φ' also has a model $\mathcal{M}' = (\mathbb{D}', _, _, _)$ with $\mathbb{D}' = \{[\![\tau]\!]^{\mathcal{M}'} \mid \tau \in GType_\Sigma\}$ (which is, in particular, countably infinite) and each $D' \in \mathbb{D}'$ at most countably infinite, i.e. due to Ax and the fact that each $D' \in \mathbb{D}$ has the form $[\![\tau]\!]^{\mathcal{M}'}$, actually countably infinite. $\mathcal{M}'$ is the desired model of Φ. □

5. Monotonicity-Based Type Encodings — The Monomorphic Case

The cover-based encodings of Section 4 removed some of the clutter associated with type arguments, which are in general necessary to encode polymorphism soundly. Another family of encodings focuses on the quantifiers and exploit monotonicity. For types that are inferred monotonic, the translation can omit the type information on term variables of these types. Informally, a type is monotonic in a problem when, for any model of that problem, we can increase the size of the domain that interprets the type while preserving satisfiability.

This section focuses on the monomorphic case, where the input problem contains no type variables or polymorphic symbols. This case is interesting in its own right and serves as a stepping stone towards polymorphic monotonicity-based encodings (Section 6).

Before we start, we need to define variants of the traditional e, t, and g encodings that operate on monomorphic problems. Since the monomorphic e is essentially identical to the polymorphic e restricted to monomorphic signatures, we use the same notation for both. 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 [36, § 4].

5.1. Monotonicity.

The concept of monotonicity used by Claessen et al. [16, § 2.2] declares a type τ monotonic for a finite problem Φ if for any model $\mathcal{M} = \big((D_\sigma)_{\sigma \in Type}, _, _\big)$ of Φ such that D_τ is finite, there exists another model $\mathcal{M}' = \big((D'_\sigma)_{\sigma \in Type}, _, _\big)$ of Φ such that $|D'_\tau| = |D_\tau| + 1$ and $|D'_\sigma| = |D_\sigma|$ for all $\sigma \neq \tau$. Their notion, which we call *finite monotonicity*, is designed to ensure that it is possible to produce a model having all types interpreted by countably infinite sets, and finally a model having all types interpreted as the same set, so that type information can be soundly erased. In this article, we take directly the infinite-interpretation property as definition of monotonicity and extend the notion to sets of types:

Definition 5.1 (Monotonicity)**.** Let S be a set of types and Φ be a problem. The set S is *monotonic* in Φ if for all models $\mathcal{M} = \big((D_\sigma)_{\sigma \in Type}, _, _\big)$ of Φ, there exists a model $\mathcal{M}' = \big((D'_\sigma)_{\sigma \in Type}, _, _\big)$ of Φ such that for all types σ, D'_σ is infinite if $\sigma \in S$ and $|D'_\sigma| = |D_\sigma|$ otherwise. A type σ is *monotonic* if $\{\sigma\}$ is monotonic. The problem Φ is *monotonic* if the set *Type* of all types is monotonic.

Full type erasure is sound for monotonic monomorphic 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 5.2. The monkey village of Example 1.1 is 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 3.4). However, the type of monkeys is not monotonic on its own, as we argued intuitively in Example 1.1, nor is it finitely monotonic.

Theorem 5.3 (Monotonic Erasure)**.** *Full type erasure is sound for monotonic monomorphic problems.*

Proof. Let Φ be such a problem, let $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$ be its signature, and let $\Sigma' = (\mathcal{F}', \mathcal{P}')$ be the target signature of e. If Φ is satisfiable, it has a model where all domains are infinite by definition of monotonicity. Since Σ is countable, by the downward Löwenheim–Skolem theorem [19], Φ also has a model where all domains are countably infinite, hence also a model $\mathcal{M} = \big((D_\sigma)_{\sigma \in \mathit{Type}}, (f^{\mathcal{M}})_{f \in \mathcal{F}}, (p^{\mathcal{M}})_{p \in \mathcal{P}}\big)$ where all domains are interpreted as the same set D, i.e. each D_σ is D. We define the Σ'-structure $\mathcal{M}' = \big(D, (f^{\mathcal{M}'})_{f \in \mathcal{F}'}, (p^{\mathcal{M}'})_{p \in \mathcal{P}'}\big)$ by taking D to be D and each $s^{\mathcal{M}'}$ to be $s^{\mathcal{M}}$. For each $\xi' : \mathcal{V} \to D$, we define $\xi : \mathcal{V} \to \prod_{\sigma \in \mathit{Type}} D_\sigma$, i.e. $\xi : \mathcal{V} \to \mathit{Type} \to D$, by $\xi(X)(\sigma) = \xi'(X)$. The next facts follow by induction on t and φ (for arbitrary ξ):

(1) $[\![\langle\!\langle t \rangle\!\rangle_{\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = [\![t]\!]^{\mathcal{M}}_{\xi}$;

(2) $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = [\![\varphi]\!]^{\mathcal{M}}_{\xi}$.

It follows by the usual route that $\mathcal{M}'$ is a model of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{e}}$. □

Remark 5.4. An alternative to invoking the Löwenheim–Skolem theorem would have been to require countable infinity in the definition of monotonicity. Although it makes no difference in this article, the more general definition would also apply in the presence of uncountable interpreted types (e.g. for the real numbers). The proof of Theorem 5.3 can be adapted to go beyond countable infinity if desired.

It is often convenient to determine the monotonicity of single types separately, viewed as singleton sets. This is enough to make the set of all types, i.e. the problem, monotonic:

Lemma 5.5 (Global Monotonicity from Separate Monotonicity)**.** *If σ is monotonic in Φ for each type σ, then Φ is monotonic.*

Proof. Let $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$ be the signature of Φ, and assume Φ is satisfiable. Let $\sigma_1, \sigma_2, \ldots$ be an enumeration of *Type*, and let φ_k be a formula stating that for all $i \in \{1, \ldots, k\}$, σ_i's domain has at least k elements. Finally, let $\Phi' = \{\varphi_i \mid i \in \mathbb{N}\}$. By the monotonicity of the individual types, it follows by induction on k that each $\Phi \cup \{\varphi_i \mid i \in \{1, \ldots, k\}\}$ is satisfiable, and hence that each finite subset of $\Phi \cup \Phi'$ is satisfiable. By the compactness theorem [19], it follows that $\Phi \cup \Phi'$ is itself satisfiable, and hence Φ has a model with only infinite domains. □

5.2. **Monotonicity Inference.** Claessen et al. introduced a simple calculus to infer finite monotonicity for monomorphic first-order logic [16, §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 unavoidable to express upper bounds on the cardinality of types in first-order logic.

Definition 5.6 (Naked Variable)**.** The set of *naked variables* $\mathrm{NV}(\varphi)$ of a formula φ is defined as follows:

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

For a problem Φ, we define $\mathrm{NV}(\Phi) = \bigcup_{\varphi \in \Phi} \mathrm{NV}(\varphi)$.

We see from the $\exists$ case that existential variables never occur naked in sentences.

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 2.13, will cause us to classify σ as possibly nonmonotonic. We regain some precision by extending the calculus with an infinity analysis, as suggested by Claessen et al.: trivially, all types with no finite models are monotonic.

Convention 5.7. Abstracting over the specific analysis used to detect infinite types (e.g. Infinox [15]), we fix a set $\mathrm{Inf}(\Phi)$ of types whose interpretations are guaranteed to be infinite in all models of Φ, More precisely, the following property is assumed to hold: if $\tau \in \mathrm{Inf}(\Phi)$ and $\mathcal{M} = \big((D_\sigma)_{\sigma \in \mathit{Type}}, _, _\big)$ is a model of Φ, then D_τ is infinite.

Our monotonicity calculus takes $\mathrm{Inf}(\Phi)$ into account:

Definition 5.8 (Monotonicity Calculus $\rhd$)**.** Let Φ be a monomorphic problem over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$. A judgement $\sigma \rhd \Phi$ indicates that the ground type σ is inferred monotonic in Φ. The *monotonicity calculus* consists of the following rules:

$$\frac{\sigma \in \mathrm{Inf}(\Phi)}{\sigma \rhd \Phi} \qquad \frac{\mathrm{NV}(\Phi) \cap \{X^\sigma \mid X \in \mathcal{V}\} = \emptyset}{\sigma \rhd \Phi}$$

We write $\sigma \rhd \Phi$ to indicate that the judgement is derivable and $\sigma \not\rhd \Phi$ otherwise.

Claessen et al. designed a second, more powerful calculus that extends their first calculus to detect predicates that act as guards for naked variables. Whilst the calculus proved successful on a subset of the TPTP benchmarks [30], we assessed its suitability on about 1000 problems generated by Sledgehammer and found no improvement on the simple calculus. For this reason, we restrict our attention to the first calculus.

Theorem 5.9 (Soundness of $\rhd$)**.** *Let Φ be a monomorphic problem. If $\tau \rhd \Phi$, then τ is monotonic in Φ.*

Proof. Let $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$ be the signature of Φ, and let $\tau \in \mathit{Type}$ such that $\tau \rhd \Phi$. Assume Φ has a model $\mathcal{M} = \big((D_\sigma)_{\sigma \in \mathit{Type}}, (f^{\mathcal{M}})_{f \in \mathcal{F}}, (p^{\mathcal{M}})_{p \in \mathcal{P}}\big)$. We will construct a model $\mathcal{M}' = \big((D'_\sigma)_{\sigma \in \mathit{Type}}, (f^{\mathcal{M}'})_{f \in \mathcal{F}}, (p^{\mathcal{M}'})_{p \in \mathcal{P}}\big)$ with the same domains as $\mathcal{M}$ for $\sigma \neq \tau$ and with D'_τ infinite. If $\tau \in \mathrm{Inf}(\Phi)$, then D_τ is already infinite, so can we take $\mathcal{M}' = \mathcal{M}$. Otherwise,

$\tau \notin \mathrm{Inf}(\Phi)$, which means the second rule of the calculus must have been applied and no variables of type τ occur naked, i.e. $\mathrm{NV}(\Phi) \cap \{X^\tau \mid X \in \mathcal{V}\} = \emptyset$. We define D'_τ by extending D_τ with an infinite number of fresh elements $e_1, e_2, \ldots$ and define the functions and predicates of $\mathcal{M}'$ to treat these in the same way as $\varepsilon(D_\tau)$. Intuitively, $\mathcal{M}'$ also satisfies Φ because, since no variables of type τ occur naked, the formulae of Φ cannot tell the difference between a "clone" e_i and the original $\varepsilon(D_\tau)$ except for the case of disequality—and there the formula instantiated with clones is more likely to be true than the one instantiated with $\varepsilon(D_\tau)$ (since clones are mutually disequal and disequal to $\varepsilon(D_\tau)$).

Formally, we let $E = \{e_1, e_2, \ldots\}$ be an infinite set disjoint from D_τ, and define

$$D'_\sigma = \begin{cases} D_\sigma \cup E & \text{if } \sigma = \tau \\ D_\sigma & \text{otherwise} \end{cases}$$

Given $s : \bar{\sigma} \to \varsigma$, we let $s^{\mathcal{M}'}(\bar{d}) = s^{\mathcal{M}}(\bar{d}')$, where

$$d'_i = \begin{cases} \varepsilon(D_\tau) & \text{if } \sigma_i = \tau \text{ and } d_i \in E \\ d_i & \text{otherwise} \end{cases}$$

Given $\xi' : \mathcal{V} \to \prod_{\sigma \in Type} D'_\sigma$, we define $\xi : \mathcal{V} \to \prod_{\sigma \in Type} D_\sigma$ by

$$\xi(X)(\sigma)(d) = \begin{cases} \varepsilon(D_\tau) & \text{if } \sigma = \tau \text{ and } d \in E \\ \xi'(X)(\sigma)(d) & \text{otherwise} \end{cases}$$

The next facts follow by induction on t or φ (for arbitrary ξ'):

(1) $t \notin \{X^\tau \mid X \in \mathcal{V}\}$ implies $[\![t]\!]^{\mathcal{M}'}_{\xi'} = [\![t]\!]^{\mathcal{M}}_{\xi}$;

(2) $\mathrm{NV}(\varphi) \cap \{X^\sigma \mid X \in \mathcal{V}\} = \emptyset$ and $[\![\varphi]\!]^{\mathcal{M}}_{\xi}$ imply $[\![\varphi]\!]^{\mathcal{M}'}_{\xi'}$.

(Notice that (2) is an implication, not an equivalence. An inductive proof of the converse would fail for the case of disequalities.) In particular, thanks to the absence of naked variables of type τ in all $\varphi \in \Phi$, $\mathcal{M}'$ is the desired model of Φ. □

In the light of the above soundness result, we will allow ourselves to write that σ is monotonic if $\sigma \rhd \Phi$ and possibly nonmonotonic if $\sigma \not\rhd \Phi$.

5.3. **Encoding Nonmonotonic Types.** Monotonic types can be soundly erased when translating to untyped first-order logic, by Theorem 5.3. Nonmonotonic types in general cannot. Claessen et al. [16, §3.2] point out that adding sufficiently many protectors to a nonmonotonic problem will make it monotonic, at which point its types can be erased. Thus the following general two-stage procedure translates monomorphic problems to untyped first-order logic:

1. Selectively introduce protectors (tags or guards) without erasing any types:
   - 1.1. Infer monotonicity to identify the possibly nonmonotonic types in the problem.
   - 1.2. Introduce protectors for the universal variables of possibly nonmonotonic types.
   - 1.3. If necessary (depending on the encoding), generate typing axioms for any function symbol whose result type is possibly nonmonotonic, to make it possible to remove protectors for terms with the right type.
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 5.10 (Correctness Conditions)**.** *Let Φ be a monomorphic problem, and let x be a monomorphic encoding. The problems Φ and $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{x};\mathsf{e}}$ are equisatisfiable provided that the following conditions hold*:

MONO: *$\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{x}}$ is monotonic.*

SOUND: *If Φ is satisfiable, so is $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{x}}$.*

COMPLETE: *If $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{x}}$ is satisfiable, so is Φ.*

Proof. Immediate from Theorems 3.5 and 5.3. □

5.4. **Monotonicity-Based Type Tags.** The monotonicity-based encoding $\widetilde{\mathsf{t}?}$ specialises the above procedure for tags. It is similar to the traditional encoding $\widetilde{\mathsf{t}}$ (the monomorphic version of 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 5.11 (Lightweight Tags $\widetilde{\mathsf{t}?}$)**.** The encoding $\langle\!\langle\ \rangle\!\rangle_{\widetilde{\mathsf{t}?}}$ translates monomorphic problems over $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$ to monomorphic problems over $(\mathit{Type}, \mathcal{F} \uplus \{\mathsf{t}_\sigma : \sigma \to \sigma \mid \sigma \in \mathit{Type}\}, \mathcal{P})$. It adds no axioms, and its term and formula translations are defined as follows:

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

The *monomorphic lightweight type tags* encoding $\widetilde{\mathsf{t}?}$ is the composition $\langle\!\langle\ \rangle\!\rangle_{\widetilde{\mathsf{t}?};\mathsf{e}}$. It translates a monomorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \{\mathsf{t}^1_\sigma \mid \sigma \in \mathit{Type}\}, \mathcal{P}')$, where $\mathcal{F}', \mathcal{P}'$ are as for e.

Example 5.12. For the algebraic list problem of Example 2.13, the type *list_b* is monotonic by virtue of being infinite, whereas b cannot be inferred monotonic. The $\widetilde{\mathsf{t}?}$ encoding of the problem follows:

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

The $\widetilde{\mathsf{t}?}$ encoding treats all variables of the same type uniformly. Hundreds of axioms can suffer because of one unhappy formula that uses a type nonmonotonically (or in a way that cannot be inferred monotonic). To address this, we introduce a lighter encoding: if a universal variable does not occur naked in a formula, its tag can safely be omitted.[2]

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

[2]This is related to the observation that only paramodulation from or into a variable can cause ill-typed instantiations in a resolution prover [36, §4].

Definition 5.13 (Featherweight Tags $\widetilde{\mathsf{t}}??$)**.** The encoding $\langle\!\langle\ \rangle\!\rangle_{\widetilde{\mathsf{t}}??}$ translates monomorphic problems over $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$ to monomorphic problems over $(\mathit{Type}, \mathcal{F} \uplus \{\mathsf{t}_\sigma : \sigma \to \sigma \mid \sigma \in \mathit{Type}\}, \mathcal{P})$. Its term and formula translations are defined as follows:

$$\langle\!\langle t_1 \approx t_2 \rangle\!\rangle_{\widetilde{\mathsf{t}}??} \;=\; \lfloor \langle\!\langle t_1 \rangle\!\rangle_{\widetilde{\mathsf{t}}??} \rfloor \approx \lfloor \langle\!\langle t_2 \rangle\!\rangle_{\widetilde{\mathsf{t}}??} \rfloor$$

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

with

$$\lfloor t^\sigma \rfloor \;=\; \begin{cases} t & \text{if } \sigma \rhd \Phi \text{ or } t \notin \mathcal{V}^{\forall}_{\text{typed}} \\ \mathsf{t}_\sigma(t) & \text{otherwise} \end{cases}$$

The encoding adds the following typing axioms:

$\forall \bar{X} : \bar{\sigma}.\ \mathsf{t}_\sigma(f(\bar{X})) \approx f(\bar{X})$ for $f : \bar{\sigma} \to \sigma \in \mathcal{F}$ such that $\sigma \not\rhd \Phi$

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

The *monomorphic featherweight type tags* encoding $\widetilde{\mathsf{t}}??$ is the composition $\langle\!\langle\ \rangle\!\rangle_{\widetilde{\mathsf{t}}??;\mathsf{e}}$. The target signature for $\widetilde{\mathsf{t}}??$ is the same as for $\widetilde{\mathsf{t}}?$.

The axioms are necessary for the completeness of $\widetilde{\mathsf{t}}??$. They would have been harmless for $\widetilde{\mathsf{t}}?$: for soundness, we can think of the t_σ functions as identities. The side condition of 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 5.14. The $\widetilde{\mathsf{t}}??$ encoding of Example 2.13 requires fewer tags than $\widetilde{\mathsf{t}}?$, at the cost of a typing axiom for hd and typing equations for the existential variables of type b:

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

$\forall X, Xs.\ \mathsf{nil}_b \not\approx \mathsf{cons}_b(X, Xs)$

$\forall Xs.\ Xs \approx \mathsf{nil}_b \vee (\exists Y, Ys.\ \mathsf{t}_b(Y) \approx Y \wedge Xs \approx \mathsf{cons}_b(Y, Ys))$

$\forall X, Xs.\ \mathsf{hd}_b(\mathsf{cons}_b(X, Xs)) \approx \mathsf{t}_b(X) \wedge \mathsf{tl}_b(\mathsf{cons}_b(X, Xs)) \approx Xs$

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

Theorem 5.15 (Correctness of $\widetilde{\mathsf{t}}?$, $\widetilde{\mathsf{t}}??$)**.** *The monomorphic type tags encodings* $\widetilde{\mathsf{t}}?$ *and* $\widetilde{\mathsf{t}}??$ *are correct.*

Proof. It suffices to show the three conditions of Lemma 5.10.

MONO: By induction on φ, it follows that

(1) $X^\sigma \in \mathrm{NV}(\langle\!\langle \varphi \rangle\!\rangle_{\widetilde{\mathsf{t}}?})$ implies $\sigma \in \mathrm{Inf}(\Phi)$;
(2) $X^\sigma \in \mathrm{NV}(\langle\!\langle \varphi \rangle\!\rangle_{\widetilde{\mathsf{t}}??})$ implies $\sigma \in \mathrm{Inf}(\Phi)$.

(Indeed, while transforming φ into $\langle\!\langle \varphi \rangle\!\rangle_{\widetilde{\mathsf{t}}??}$, $\widetilde{\mathsf{t}}??$ tags precisely the variables that would cause the condition (1) to fail; and $\widetilde{\mathsf{t}}?$ tags even more.) Moreover, the typing axioms of $\widetilde{\mathsf{t}}??$ have no naked variables, and hence $\sigma \rhd \langle\!\langle \varphi \rangle\!\rangle_{\widetilde{\mathsf{t}}?}$ and $\sigma \rhd \langle\!\langle \varphi \rangle\!\rangle_{\widetilde{\mathsf{t}}??}$ hold for all types σ. Therefore, by Theorem 5.9 and Lemma 5.5, $\sigma \rhd \langle\!\langle \varphi \rangle\!\rangle_{\widetilde{\mathsf{t}}?}$ and $\sigma \rhd \langle\!\langle \varphi \rangle\!\rangle_{\widetilde{\mathsf{t}}??}$ are monotonic.

SOUND: This is immediate for both $\langle\!\langle\ \rangle\!\rangle_{\widetilde{\mathsf{t}}?}$ and $\langle\!\langle\ \rangle\!\rangle_{\widetilde{\mathsf{t}}??}$: given a model of Φ, we extend it to a model of the encoded Φ by interpreting all type tags as the identity.

COMPLETE FOR $\langle\!\langle\ \rangle\!\rangle_{\widetilde{\mathsf{t}}?}$: The proof is analogous to the corresponding case for t (Theorem 3.16), the differences being that here we do not face the complication of interpreting polymorphic

types, and only terms of certain types are tagged. Let $\mathcal{M}' = \big((D'_\sigma)_{\sigma\in Type}, (f^{\mathcal{M}'})_{f\in\mathcal{F}\uplus\{\mathsf{t}_\sigma \mid \sigma\not\rhd\Phi\}}, (p^{\mathcal{M}'})_{p\in\mathcal{P}}\big)$ be a model of $\langle\!\langle\Phi\rangle\!\rangle_{\widetilde{\mathsf{t}?}}$. For each $\sigma \in Type$, we let

$$D_\sigma = \begin{cases} D'_\sigma & \text{if } \sigma \rhd \Phi \\ \{d \in D'_\sigma \mid d \text{ is in the image of } \mathsf{t}_\sigma^{\mathcal{M}'}\} & \text{otherwise} \end{cases}$$

We define $T_\sigma : D'_\sigma \to D'_\sigma$ so as to apply $\mathsf{t}_\sigma^{\mathcal{M}'}$ only if the element is not already in its image:

$$T_\sigma(d) = \begin{cases} d & \text{if } d \in D_\sigma \\ \mathsf{t}_\sigma^{\mathcal{M}'}(d) & \text{otherwise} \end{cases}$$

Let $\mathcal{M} = \big((D_\sigma)_{\sigma\in Type}, (f^{\mathcal{M}})_{f\in\mathcal{F}}, (p^{\mathcal{M}})_{p\in\mathcal{P}}\big)$, where

- $f^{\mathcal{M}}(d_1,\ldots,d_n) = \mathsf{t}_\sigma^{\mathcal{M}'}(f^{\mathcal{M}'}(T_{\sigma_1}(d_1),\ldots,T_{\sigma_n}(d_n)))$ for $f : \bar\sigma \to \sigma$ is in $\mathcal{F}$;
- $p^{\mathcal{M}}(d_1,\ldots,d_n) = p^{\mathcal{M}'}(T_{\sigma_1}(d_1),\ldots,T_{\sigma_n}(d_n))$ if $p : \bar\sigma \to o$ is in $\mathcal{F}$.

Given $\xi' : \mathcal{V} \to \prod_{\sigma\in Type} D'_\sigma$, we define $\xi : \mathcal{V} \to \prod_{\sigma\in Type} D_\sigma$ as follows:

$$\xi(X)(\sigma) = \begin{cases} \xi'(X)(\sigma) & \text{if } \sigma \rhd \Phi \\ \mathsf{t}_\sigma(\xi'(X)(\sigma)) & \text{otherwise} \end{cases}$$

The next facts follow by induction on t or φ (for arbitrary ξ'):

(1) $[\![t]\!]^{\mathcal{M}}_\xi = [\![\langle\!\langle t\rangle\!\rangle_{\mathsf{t}}]\!]^{\mathcal{M}'}_{\xi'}$;
(2) $[\![\varphi]\!]^{\mathcal{M}}_\xi = [\![\langle\!\langle \varphi\rangle\!\rangle_{\mathsf{t}}]\!]^{\mathcal{M}'}_{\xi'}$.

Let $\varphi \in \Phi$. To show that $\mathcal{M}$ is a model of φ, let $\xi : \mathcal{V} \to \prod_{\sigma\in Type} D_\sigma$. By the definition of D_σ, there exists $\xi' : \mathcal{V} \to \prod_{\sigma\in Type} D'_\sigma$ such that the defining property of ξ holds. Now, since $[\![\langle\!\langle\varphi\rangle\!\rangle_{\widetilde{\mathsf{t}?}}]\!]^{\mathcal{M}'}_{\xi'}$ holds, it follows from (2) that $[\![\varphi]\!]^{\mathcal{M}}_\xi$, i.e. $[\![\varphi]\!]^{\mathcal{M}}$ also holds, as desired.

Complete for $\langle\!\langle\ \rangle\!\rangle_{\widetilde{\mathsf{t}??}}$: The proof is analogous to the corresponding case for $\mathsf{t@}$ (Theorem 4.10). Starting with a model $\mathcal{M}' = \big((D'_\sigma)_{\sigma\in Type}, (f^{\mathcal{M}'})_{f\in\mathcal{F}\uplus\{\mathsf{t}_\sigma \mid \sigma\not\rhd\Phi\}}, (p^{\mathcal{M}'})_{p\in\mathcal{P}}\big)$ of $\langle\!\langle\Phi\rangle\!\rangle_{\widetilde{\mathsf{t}??}}$, we define a model $\mathcal{M}$ of Φ as follows, where the typing axioms ensure that each D_σ is nonempty and each $s^{\mathcal{M}}$ is well defined:

- $D_\sigma = \begin{cases} D'_\sigma & \text{if } \sigma \rhd \Phi \\ \{d \in D'_\sigma \mid \mathsf{t}_\sigma^{\mathcal{M}'}(d) = d\} & \text{otherwise} \end{cases}$
- $s^{\mathcal{M}}$ is the restriction of $s^{\mathcal{M}'}$.

The next facts follow by induction on t or φ (for arbitrary $\xi : \mathcal{V} \to \prod_{\sigma\in Type} D_\sigma$):

(1) $[\![t]\!]^{\mathcal{M}}_\xi = [\![\langle\!\langle t\rangle\!\rangle_{\mathsf{t}}]\!]^{\mathcal{M}'}_{\xi'}$;
(2) $[\![\varphi]\!]^{\mathcal{M}}_\xi = [\![\langle\!\langle \varphi\rangle\!\rangle_{\mathsf{t}}]\!]^{\mathcal{M}'}_{\xi'}$.

Then, by the usual route, it follows that $\mathcal{M}$ is a model of Φ. □

5.5. **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 5.16 (Lightweight Guards $\widetilde{\mathsf{g}}?$)**.** The encoding $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{g}}?}$ translates monomorphic problems over $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$ to monomorphic problems over $(\mathit{Type}, \mathcal{F}, \mathcal{P} \uplus \{\mathsf{g}_\sigma : \sigma \to o \mid \sigma \in \mathit{Type}\})$. Its term and formula translations are defined as follows:

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

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

The encoding adds the following typing axioms:

$$\forall \bar{X} : \bar{\sigma}.\ \mathsf{g}_\sigma(f(\bar{X}^{\bar{\sigma}})) \qquad \text{for } f : \bar{\sigma} \to \sigma \in \mathcal{F} \text{ such that } \sigma \not\rhd \Phi$$

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

The *monomorphic lightweight type guards* encoding $\widetilde{\mathsf{g}}?$ is the composition $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{g}}?;\mathsf{e}}$. It translates a monomorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}', \mathcal{P}' \uplus \{\mathsf{g}^1_\sigma \mid \sigma \in \mathit{Type}\})$, where $\mathcal{F}', \mathcal{P}'$ are as for e.

Example 5.17. The $\widetilde{\mathsf{g}}?$ encoding of Example 2.13 is as follows:

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

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

Our novel encoding $\widetilde{\mathsf{g}}??$ omits the guards for variables that do not occur naked, regardless of whether they are of a monotonic type.

Definition 5.18 (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 5.19. 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 5.20 (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 5.10.

Mono: By Lemma 5.5, it suffices to show that each type τ such that $\sigma \not\rhd \Phi$ is monotonic in the encoded problem. By Theorem 5.9, 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 5.9 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 the encoded problem $\mathcal{M} = \big((D_\sigma)_{\sigma \in \mathit{Type}}, (f^{\mathcal{M}})_{f \in \mathcal{F}}, (p^{\mathcal{M}})_{p \in \mathcal{P} \uplus \{\mathsf{g}_\sigma \mid \sigma \not\rhd \Phi\}}\big)$, we construct a model $\mathcal{M}' = \big((D'_\sigma)_{\sigma \in \mathit{Type}},$

$\left(f^{\mathcal{M}'}\right)_{f\in\mathcal{F}}, \left(p^{\mathcal{M}'}\right)_{p\in\mathcal{P}}\big)$ with all types infinite as in the proof of Theorem 5.9, but defining the τ-guard interpretation to be false on newly added elements (recall that $D'_\tau = D_\tau \uplus E$ for some countably infinite set E):

$$\mathsf{g}_\sigma^{\mathcal{M}'}(d) = \begin{cases} \mathsf{g}_\sigma^{\mathcal{M}}(d) & \text{if } \sigma \neq \tau \text{ or } \sigma = \tau \text{ and } d \in D_\sigma \\ \text{false} & \text{if } \sigma = \tau \text{ and } d \in E \end{cases}$$

Given $\xi' : \mathcal{V} \to \prod_{\sigma\in Type} D'_\sigma$, we define $\xi : \mathcal{V} \to \prod_{\sigma\in Type} D_\sigma$ by

$$\xi(\sigma)(d) = \begin{cases} \varepsilon(D_\tau) & \text{if } \sigma = \tau \text{ and } d \in E \\ \xi'(\sigma)(d) & \text{otherwise} \end{cases}$$

The encoded problem has the form $\Phi_1 \cup \Phi_2$, where Φ_1 are the added axioms and Φ_2 are the formula translations from Φ. It is easy to check that $\mathcal{M}'$ satisfies Φ_1. We say that a formula is *τ-guarded* if all its subformulae of the form $\forall X : \tau.\, \varphi$ have φ of the form $\neg\, \mathsf{g}_\tau(X^\tau) \vee \chi$ and all its subformulae of the form $\exists X : \tau.\, \varphi$ have φ of the form $\mathsf{g}_\tau(X^\tau) \wedge \chi$. All the formulae in Φ_2 are clearly guarded. The next facts follow by induction on t or φ (for arbitrary ξ'):

(1) $[\![t]\!]^{\mathcal{M}'}_{\xi'} = [\![t]\!]^{\mathcal{M}}_{\xi}$;

(2) If φ is guarded and $\mathsf{g}_\tau\left(\xi\left(\tau\right)\left(X\right)\right)$ is false whenever $X^\tau \in \mathrm{NV}(\varphi) \cap \mathrm{FVars}(\varphi)$, then $[\![\varphi]\!]^{\mathcal{M}}_{\xi}$ implies $[\![\varphi]\!]^{\mathcal{M}'}_{\xi'}$.

It follows that $\mathcal{M}'$ satisfies Φ_2 as well. Hence it is the desired model of the encoded problem.

Sound: Just like for tags, this is immediate for both $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{g}}?}$ and $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{g}}??}$: given a model of Φ, we extend it to a model of the encoded Φ by interpreting all type guards as everywhere-true predicates.

Complete: The proofs for both $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{g}}?}$ and $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{g}}??}$ are very similar to that for $\mathsf{t}??$ (from Theorem 5.15)—the only change is the replacement of the condition $\mathsf{t}_\sigma^{\mathcal{M}'}(d) = d$ by $\mathsf{g}_\sigma^{\mathcal{M}'}(d)$ in the definition of the model $\mathcal{M}$. (Just like for $\mathsf{t}??$, the typing axioms ensure that the structure is well defined.) □

A simpler but less instructive way to prove Mono is to observe that the second monotonicity calculus by Claessen et al. [16, §2.4] can infer monotonicity of all problems generated by $\widetilde{\mathsf{g}}$, $\widetilde{\mathsf{g}}?$, and $\widetilde{\mathsf{g}}??$.

Remark 5.21. The proofs of Theorems 5.15 and 5.20 remain valid as they are even if the generated problems contain more tags or guards than inferred as unnecessary by the monotonicity calculus, i.e. if in the definitions of $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{t}}?}$, $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{t}}??}$, $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{g}}?}$, and $\langle\!\langle\, \rangle\!\rangle_{\widetilde{\mathsf{g}}??}$ we replace the condition "$\sigma \rhd \Phi$" by "$\sigma \in J$", where $J \subseteq \{\sigma \in Type_\Sigma \mid \sigma \rhd \Phi\}$. (The replacement should be done everywhere, including in the axioms.)

5.6. **Heuristic Monomorphisation.** Section 6 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 *heuristic monomorphisation* on the polymorphic problem:

1. Heuristically instantiate all type variables with suitable ground types, taking finitely many copies of each formula if desired.

2. Map each ground occurrence $s\langle\bar{\alpha}\rho\rangle$ of a polymorphic symbol $s : \forall\bar{\alpha}.\ \bar{\sigma} \to \sigma$ to a fresh monomorphic symbol $s_{\bar{\alpha}\rho} : \lfloor\bar{\sigma}\rho\rfloor \to \lfloor\sigma\rho\rfloor$, 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. $\{b \mapsto b,\ \mathit{list}(b) \mapsto \mathit{list_}b\}$).

Heuristic monomorphisation is generally incomplete [12, §2] and often overlooked in the literature, but by eliminating type variables it considerably simplifies the generated formulae, 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.

6. Complete Monotonicity-Based Encodings of Polymorphism

Heuristic 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 type encodings to a polymorphic setting.

We start by defining a correct but infinitary translation of a polymorphic into a monomorphic problem, called complete monomorphisation. Then we address the genuinely polymorphic issue of encoding type arguments, proving conditions under which composition of an encoding x with the type arguments encoding a is correct: x must be complete and produce monotonic problems. Finally, we define polymorphic counterparts of the guard and tag encodings and show that they satisfy the required conditions by reducing (most of) the problem to the monomorphic case via complete monomorphisation.

6.1. Complete Monomorphisation.

The main insight behind complete monomorphisation is that a polymorphic formula having all its type quantification at the top is equisatisfiable to the (generally infinite) set of its monomorphic instances. Complete monomorphisation does not obey our convention about encodings, since it translates each polymorphic formula not into a single monomorphic formula but into a set of formulae.

Definition 6.1 (Instance Type and Most General Instance)**.** A type τ is an *instance* of a type σ if there exists a type substitution ρ such that $\tau = \sigma\rho$. If this is the case, we also say that τ is *less general* than σ and that σ is *more general* than τ, and we write $\tau \leq \sigma$. Given two types σ and τ, if they have a common instance (i.e. a unifier), then they also have a most general common instance, which we denote by $\mathrm{mgi}(\sigma, \tau)$.

Definition 6.2 (Complete Monomorphisation ∞)**.** We define the encoding $\langle\!\langle\ \rangle\!\rangle_\infty$ that translates polymorphic problems over $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ to monomorphic problems over $\Sigma' = (\mathit{Type}', \mathcal{F}', \mathcal{P}')$, where

- $\mathit{Type}' = \mathit{GType}_\Sigma$;
- for each $s : \forall\bar{\alpha}.\ \bar{\sigma} \to \varsigma$ in $\mathcal{F} \uplus \mathcal{P}$ and each ρ such that each α_i is mapped to a ground type $\tau_i \in \mathit{GType}_\Sigma$, $\mathcal{F}' \uplus \mathcal{P}'$ contains a symbol $s_{\bar{\tau}} : \bar{\sigma}\rho \to \sigma\rho$.

We first define the translation of terms and formulae that contain no type variables, i.e. that have no type quantifiers and such that all the occurring types in applications of the function

or predicate symbols are ground:

$$\begin{aligned} \langle\!\langle f\langle\bar{\sigma}\rangle(\bar{t})\rangle\!\rangle_\infty &= f_{\bar{\sigma}}(\langle\!\langle \bar{t}\rangle\!\rangle_\infty) & \langle\!\langle X^\sigma\rangle\!\rangle_\infty &= X^\sigma \\ \langle\!\langle p\langle\bar{\sigma}\rangle(\bar{t})\rangle\!\rangle_\infty &= p_{\bar{\sigma}}(\langle\!\langle \bar{t}\rangle\!\rangle_\infty) & \langle\!\langle \forall X:\sigma.\ \varphi\rangle\!\rangle_\infty &= \forall X:\sigma.\ \langle\!\langle\varphi\rangle\!\rangle_\infty \\ \langle\!\langle \neg\, p\langle\bar{\sigma}\rangle(\bar{t})\rangle\!\rangle_\infty &= \neg\, p_{\bar{\sigma}}(\langle\!\langle \bar{t}\rangle\!\rangle_\infty) & \langle\!\langle \exists X:\sigma.\ \varphi\rangle\!\rangle_\infty &= \exists X:\sigma.\ \langle\!\langle\varphi\rangle\!\rangle_\infty \end{aligned}$$

Now, given a sentence $\forall\bar{\alpha}.\,\varphi$ where $\bar{\alpha}$ indicates all its universally quantified types, we encode it as the set of encodings of its monomorphic instances (via ground type substitutions ρ):

$$\langle\!\langle \forall\bar{\alpha}.\,\varphi\rangle\!\rangle_\infty = \{\langle\!\langle\varphi\rho\rangle\!\rangle_\infty \mid \rho : \mathcal{A} \to \mathit{GType}\}$$

Finally, the encoding of a problem is the union of the encoding of its formulae:

$$\langle\!\langle\Phi\rangle\!\rangle_\infty = \bigcup_{\varphi\in\Phi}\langle\!\langle\varphi\rangle\!\rangle_\infty$$

Convention 6.3. Whenever we write a polymorphic formula as $\forall\bar{\alpha}.\,\varphi$, we implicitly assume that $\bar{\alpha}$ indicates all its universally quantified types, so that φ has no type quantifiers.

Lemma 6.4 (Correctness of ∞)**.** *The complete monomorphisation encoding ∞ is correct.*

Proof. First, observe that in any model $\mathcal{M}$ of a polymorphic signature, the interpretation $[\![\ ]\!]^{\mathcal{M}}_{\theta,\xi}$ of a term or formula that does not contain type variables does not depend on θ and, from ξ, it only depends on the restriction of ξ to ground types, $\xi' : \mathcal{V} \to \prod_{\sigma\in \mathit{GType}_\Sigma} [\![\sigma]\!]^{\mathcal{M}}$. We can therefore write $[\![\ ]\!]^{\mathcal{M}}_{\xi'}$ instead of $[\![\ ]\!]^{\mathcal{M}}_{\theta,\xi'}$.

Sound: Assume Φ is satisfiable and let $\Sigma = (\mathcal{K},\mathcal{F},\mathcal{P})$ be its polymorphic signature. By Lemma 4.11, Φ also has a model $\mathcal{M} = \big(\mathbb{D}, (f^{\mathcal{M}})_{f\in\mathcal{F}}, (p^{\mathcal{M}})_{p\in\mathcal{P}}\big)$ for which $[\![\ ]\!]^{\mathcal{M}} : \mathit{GType}_\Sigma \to \mathbb{D}$ is a bijection—let $\nu : \mathbb{D} \to \mathit{GType}_\Sigma$ be its inverse. We define a structure $\mathcal{M}' = \big((D'_\tau)_{\tau\in \mathit{Type}'}, (f^{\mathcal{M}'})_{f\in\mathcal{F}}, (p^{\mathcal{M}'})_{p\in\mathcal{P}'}\big)$ for Σ' as follows:

- $D'_\tau = [\![\tau]\!]^{\mathcal{M}}$;
- if $s : \forall\bar{\alpha}.\,\bar{\sigma} \to \varsigma$ is in $\mathcal{F}\uplus\mathcal{P}$ and $\bar{\tau} = (\tau_1,\dots,\tau_m)$ are ground instances of $\bar{\alpha} = (\alpha_1,\dots,\alpha_m)$ via ρ, we define by $s^{\mathcal{M}'}_{\bar{\tau}}(\bar{d}) = s^{\mathcal{M}}(\nu(\tau_1),\dots,\nu(\tau_m))(\bar{d})$.

The next fact follows by induction on the term or formula δ (for arbitrary $\xi : \mathcal{V} \to \prod_{\tau\in \mathit{Type}_{\Sigma'}} D'_\tau$):

(1) If δ contains no type variables, then $[\![\langle\!\langle\delta\rangle\!\rangle_\infty]\!]^{\mathcal{M}'}_{\xi'} = [\![\delta]\!]^{\mathcal{M}}_{\xi}$.

In particular, for a sentence φ, we have $[\![\langle\!\langle\varphi\rangle\!\rangle_\infty]\!]^{\mathcal{M}'} = [\![\varphi]\!]^{\mathcal{M}}$.

Now let $\forall\bar{\alpha}.\,\varphi \in \Phi$. Since $\mathcal{M}$ is a model of $\forall\bar{\alpha}.\,\varphi$, $\mathcal{M}$ is also a model of all its ground-type instances $\varphi\rho$, and hence, since $[\![\langle\!\langle\varphi\rangle\!\rangle_\infty]\!]^{\mathcal{M}'} = [\![\varphi]\!]^{\mathcal{M}}$, $\mathcal{M}'$ is a model of each formula in $\langle\!\langle\forall\bar{\alpha}.\,\varphi\rangle\!\rangle_\infty$. Thus, $\mathcal{M}'$ is a model of $\langle\!\langle\Phi\rangle\!\rangle_\infty$, as desired.

Complete: Let $\mathcal{M}' = \big((D'_\tau)_{\tau\in \mathit{Type}'}, (f^{\mathcal{M}'})_{f\in\mathcal{F}}, (p^{\mathcal{M}'})_{p\in\mathcal{P}'}\big)$ be a model of $\langle\!\langle\Phi\rangle\!\rangle_\infty$, for which we can assume without loss of generality that $D'_\tau \cap D'_{\tau'} = \emptyset$ if $\tau \neq \tau'$. We define a structure $\mathcal{M} = \big(\mathbb{D}, (f^{\mathcal{M}})_{f\in\mathcal{F}}, (p^{\mathcal{M}})_{p\in\mathcal{P}}\big)$ for Σ as follows:

- $\mathbb{D} = \{D'_\tau \mid \tau \in \mathit{Type}'\}$—for each $D\in\mathbb{D}$, we let $\nu(D)$ be the unique $\tau \in \mathit{Type}' = \mathit{GType}_\Sigma$ such that $D = D_\sigma$;
- if $\bar{\alpha} = (\alpha_1,\dots,\alpha_m)$, $s : \forall\bar{\alpha}.\,\bar{\sigma} \to \varsigma \in \mathcal{F}\uplus\mathcal{P}$, and $\bar{D}\in\mathbb{D}^m$, we define $s^{\mathcal{M}}(\bar{D}) = s^{\mathcal{M}'}_{(\nu(D_1),\dots,\nu(D_m))}$.

The next fact follows by induction on τ:

(1) $\tau \in \mathit{GType}$ implies $[\![\tau]\!]^{\mathcal{M}} = D'_\tau$.

The next fact follows by induction on the term or formula δ (for arbitrary $\xi : \mathcal{V} \to \prod_{\tau \in GType_\Sigma} [\![\tau]\!]^{\mathcal{M}}$):

(2) If δ contains no type variables, then $[\![\delta]\!]^{\mathcal{M}}_{\xi} = [\![\langle\!|\delta|\!\rangle_\infty]\!]^{\mathcal{M}'}_{\xi'}$.

In particular, for a sentence φ, we have $[\![\varphi]\!]^{\mathcal{M}} = [\![\langle\!|\varphi|\!\rangle_\infty]\!]^{\mathcal{M}'}$. Now let $\forall \bar{\alpha}.\, \varphi \in \Phi$, and assume by absurdity that $\mathcal{M}$ is not a model of $\forall \bar{\alpha}.\, \varphi$. Then, since by (1) and the definition of $\mathbb{D}$ we have that $[\![\ ]\!]^{\mathcal{M}} : GType \to \mathbb{D}$ is surjective, we obtain a ground-type instance $\varphi\rho$ of φ such that $\mathcal{M}$ is not a model of $\varphi\rho$. By $[\![\varphi]\!]^{\mathcal{M}} = [\![\langle\!|\varphi|\!\rangle_\infty]\!]^{\mathcal{M}'}$, $\mathcal{M}'$ is not a model of $[\![\langle\!|\varphi\rho|\!\rangle_\infty]\!]^{\mathcal{M}'}$, hence not a model of $\langle\!|\forall \bar{\alpha}.\, \varphi|\!\rangle_\infty$, which is a contradiction. Therefore $\mathcal{M}$ is a model of $\forall \bar{\alpha}.\, \varphi$. We obtain that $\mathcal{M}$ is a model Φ, as desired. □

6.2. **Monotonicity.** The definition of monotonicity from Section 5.1 (Definition 5.1) must be adapted to the polymorphic case.

Definition 6.5 (Monotonicity)**.** Let S be a set of types and Φ be a polymorphic problem. The set S is *monotonic* in Φ if for all models $\mathcal{M}$ of Φ, there exists a model $\mathcal{M}'$ of Φ such that for all ground types σ, $[\![\sigma]\!]^{\mathcal{M}'}$ is infinite if σ is an instance of a type in S and $|[\![\sigma]\!]^{\mathcal{M}'}| = |[\![\sigma]\!]^{\mathcal{M}}|$ otherwise. A type σ is *monotonic* if $\{\sigma\}$ is monotonic. The problem Φ is *monotonic* if the set *Type* of all types is monotonic.

Example 6.6. For the algebraic list problem of Example 2.10, $S = \{list(\alpha)\}$ is monotonic because all models $\mathcal{M}$ of the problem necessarily interpret all of the ground instances of $list(\alpha)$ (e.g. $list(b)$, $list(list(b))$) by infinite domains. Monotonicity is trivially witnessed by taking $\mathcal{M}' = \mathcal{M}$.

Theorem 6.7 (Monotonic Erasure)**.** *The traditional type arguments encoding* a *is sound for monotonic polymorphic problems.*

Proof. Let Φ be such a problem and $\Sigma = (\mathcal{K}, \mathcal{F}, \mathcal{P})$ be its signature, and assume Φ is satisfiable. By monotonicity and Lemma 4.12, it also has a model $\mathcal{M}$ where all $D \in \mathbb{D}$ are countably infinite. From this model, we construct a model $\mathcal{M}'$ of $\langle\!|\Phi|\!\rangle_{\mathsf{a};\mathsf{e}}$ with a countably infinite domain E that interprets the encoded types as distinct elements of E. The function and predicate tables for $\mathcal{M}'$ are based on those from $\mathcal{M}$, with encoded type arguments corresponding to actual type arguments.

More precisely, let E be an countably infinite set and consider the following functions:

- the mutually inverse bijections $u : \mathbb{D} \to E$ and $v : E \to \mathbb{D}$;
- for each $D \in \mathbb{D}$, the mutually inverse bijections $u_D : D \to E$ and $v_D : E \to D$.

Let $\Sigma' = (\mathcal{F}' \uplus \mathcal{K}', \mathcal{P}')$ be the target untyped signature of a. We define the structure $\mathcal{M}'$ for Σ' as follows:

- $D = E$.
- Assume k is an n-ary function in $\mathcal{K}'$, meaning $k :: n$ is in $\mathcal{K}$. Then $k^{\mathcal{M}'}(\bar{e}) = u(k^{\mathcal{M}}(v(\bar{e})))$.
- Assume $\bar{\alpha} = \alpha_1, \ldots, \alpha_m$, $\bar{\sigma} = \sigma_1, \ldots, \sigma_n$, and $s : \forall \bar{\alpha}.\, \bar{\sigma} \to \varsigma \in \mathcal{F} \uplus \mathcal{P}$, meaning that s is an $(m+n)$-ary symbol in $\mathcal{F}' \uplus \mathcal{P}'$. Given $\bar{c} = (c_1, \ldots, c_m) \in E^m$ and $\bar{e} = (e_1, \ldots, e_n) \in E^n$, we define $s^{\mathcal{M}'}(\bar{c}, \bar{e}) = u_{[\![\sigma]\!]^{\mathcal{M}}_\theta}(s^{\mathcal{M}}(v(\bar{c}))(v_{[\![\sigma_1]\!]^{\mathcal{M}}_\theta}(e_1), \ldots, v_{[\![\sigma_n]\!]^{\mathcal{M}}_\theta}(e_n)))$, where θ maps each α_i to $v(c_i)$.

Given $\xi' : \mathcal{V} \to E$, we define $\theta : \mathcal{A} \to \mathbb{D}$ by $\theta(\alpha) = \nu(\xi'(\mathcal{V}(\alpha)))$ and $\xi : \mathcal{V} \to \prod_{\sigma \in Type_\Sigma} [\![\sigma]\!]^{\mathcal{M}}_{\theta}$ by $\xi(X)(\sigma) = \nu_{[\![\langle\!\langle \sigma \rangle\!\rangle]\!]^{\mathcal{M}'}_{\xi'}}(\xi'(X))$, where $\mathcal{V}(\alpha)$ and $\langle\!\langle \sigma \rangle\!\rangle$ are as in Definition 3.6.

The next facts follow by structural induction on σ, t, and φ (for arbitrary ξ'):

- $[\![\langle\!\langle \sigma \rangle\!\rangle]\!]^{\mathcal{M}'}_{\xi'} = u([\![\sigma]\!]^{\mathcal{M}}_{\theta})$;
- $t : \sigma$ implies $[\![\langle\!\langle t \rangle\!\rangle_{\mathsf{a};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = u_{[\![\sigma]\!]^{\mathcal{M}}_{\theta}}([\![t]\!]^{\mathcal{M}}_{\theta,\xi})$;
- $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{a};\mathsf{e}}]\!]^{\mathcal{M}'}_{\xi'} = [\![\varphi]\!]^{\mathcal{M}}_{\theta,\xi}$.

In particular, for sentences, we have $[\![\langle\!\langle \varphi \rangle\!\rangle_{\mathsf{a};\mathsf{e}}]\!]^{\mathcal{M}'} = [\![\varphi]\!]^{\mathcal{M}}$; and since $\mathcal{M}$ is a model of Φ, it follows that $\mathcal{M}'$ is a model of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{a};\mathsf{e}}$. □

Lemma 6.8 (Correctness Conditions)**.** *Let Φ be a polymorphic problem, and let x be a polymorphic encoding. The problems Φ and $\langle\!\langle \Phi \rangle\!\rangle_{x;\mathsf{a};\mathsf{e}}$ are equisatisfiable provided that the following conditions hold*:

MONO: *$\langle\!\langle \Phi \rangle\!\rangle_x$ is monotonic.*

SOUND: *If Φ is satisfiable, so is $\langle\!\langle \Phi \rangle\!\rangle_x$.*

COMPLETE: *If $\langle\!\langle \Phi \rangle\!\rangle_x$ is satisfiable, so is Φ.*

Proof. Immediate from Theorems 3.11 and 6.7. □

Lemma 6.9 (Monotonicity Preservation and Reflection by ∞)**.** *A polymorphic problem Φ is monotonic iff its monomorphic encoding $\langle\!\langle \Phi \rangle\!\rangle_\infty$ is monotonic.*

Proof. Let φ_k be the polymorphic formula stating that all type domains have at least k elements, and let $\Phi' = \{\varphi_k \mid k \in \mathbb{N}\}$. We note that the following four statements are equivalent:

(1) Φ has a model with all domains infinite;
(2) $\Phi \cup \Phi'$ has a model;
(3) $\langle\!\langle \Phi \rangle\!\rangle_\infty \cup \langle\!\langle \Phi' \rangle\!\rangle_\infty$ has a model;
(4) $\langle\!\langle \Phi \rangle\!\rangle_\infty$ has a model with all domains infinite.

The equivalences "(1) iff (2)" and "(3) iff (4)" are obvious, and "(2) iff (3)" follows from the correctness of ∞ (Lemma 6.4). □

6.3. **Monotonicity Inference.** The monotonicity inference of Section 5.2 must be adapted to the polymorphic setting. We start by generalising the notion of naked variable.

Definition 6.10 (Polymorphic Naked Variable)**.** The set of *naked variables* $\mathrm{NV}(\varphi)$ of a polymorphic formula φ is defined similarly to the monomorphic case (Definition 5.6). The following equations are new or slightly different from there:

$$\mathrm{NV}(p\langle\bar{\sigma}\rangle(\bar{t})) = \emptyset \qquad \mathrm{NV}(\neg\, p\langle\bar{\sigma}\rangle(\bar{t})) = \emptyset \qquad \mathrm{NV}(\forall\alpha.\, \varphi) = \mathrm{NV}(\varphi)$$

Again, we take $\mathrm{NV}(\Phi) = \bigcup_{\varphi\in\Phi} \mathrm{NV}(\varphi)$.

Note from Definitions 5.6 and 6.10 that the naked variables of a polymorphic formula occur at the same positions as in all its monomorphic instances. Moreover, the types of the naked variables in the completely monomorphised problem are simply the ground instances of the types of the naked variables in the original problem:

Lemma 6.11. $\mathrm{NV}(\langle\!\langle \Phi \rangle\!\rangle_\infty) = \{X^\tau \mid \tau \in GType \text{ and there exists } \sigma \geq \tau \text{ such that } X^\sigma \in \mathrm{NV}(\Phi)\}$.

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. Similarly to its monomorphic counterpart, it is parameterised by a fixed set $\mathrm{Inf}(\Phi)$ of (not necessarily ground) types for which any interpretation $[\![\sigma]\!]^{\mathcal{M}}_{\theta}$ (for any type valuation θ) in any model $\mathcal{M}$ of Φ is known to be infinite.

Convention 6.12. It is easy to see that if $\sigma \in \mathrm{Inf}(\Phi)$ and $\rho : \mathcal{A} \to \mathit{GType}_\Sigma$, then $\sigma\rho$ is infinite in all models of $\langle\!\langle \Phi \rangle\!\rangle_\infty$. We fix the "known as infinite" types of $\langle\!\langle \Phi \rangle\!\rangle_\infty$, $\mathrm{Inf}(\langle\!\langle \Phi \rangle\!\rangle_\infty)$, to be $\{\sigma\rho \mid \sigma \in \mathrm{Inf}(\Phi) \text{ and } \rho : \mathcal{A} \to \mathit{GType}_\Sigma\}$.

Definition 6.13 (Polymorphic Monotonicity Calculus $\rhd$)**.** A judgement $\sigma \rhd \Phi$ indicates that the type σ is inferred monotonic in Φ. The *monotonicity calculus* consists of the single rule

$$\frac{\text{for all } X^{\sigma'} \in \mathrm{NV}(\Phi), \text{ if } \sigma \text{ and } \sigma' \text{ have a common instance, then } \mathsf{mgi}(\sigma, \sigma') \in \mathrm{Inf}^*(\Phi)}{\sigma \rhd \Phi}$$

where $\mathrm{Inf}^*(\Phi) = \{\sigma \in \mathit{Type}_\Sigma \mid \text{there exists } \sigma' \in \mathrm{Inf}(\Phi) \text{ such that } \sigma \leq \sigma'\}$ consists of all instances of all types in $\mathrm{Inf}(\Phi)$.

Example 6.14. For the algebraic list problem of Example 2.10, the only naked variables are X^α and $Xs^{\mathit{list}(\alpha)}$, i.e. $\mathrm{NV}(\Phi) = \{X^\alpha, Xs^{\mathit{list}(\alpha)}\}$. Assume $\mathrm{Inf}(\Phi) = \{\mathit{list}(\alpha)\}$. Then $\mathrm{Inf}^*(\Phi) = \{\mathit{list}(\sigma) \mid \sigma \in \mathit{Type}_\Sigma\}$. The type $\mathit{list}(\alpha)$ is inferred monotonic by $\rhd$ because its most general common instance with α (the type of X) and $\mathit{list}(\alpha)$ (the type of Xs) is in both cases $\mathit{list}(\alpha)$, which is known to be infinite. In contrast, α and b cannot be inferred monotonic: each of them is its most general common instance with α, and neither of them is among the types that are known to be infinite.

Lemma 6.15. *The following properties hold for any polymorphic signature Σ and any Σ-problem Φ:*

(1) *If $\sigma \rhd \Phi$ and $\tau \in \mathit{GType}_\Sigma$ such that $\tau \leq \sigma$, then $\tau \rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$.*
(2) *If $\tau \in \mathit{GType}_\Sigma$, then $\tau \rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$ iff $\tau \rhd \Phi$.*
(3) *If $\sigma \rhd \Phi$ and $\sigma' \leq \sigma$, then $\sigma' \rhd \Phi$.*

Proof. (1): Assume $\sigma \rhd \Phi$ and $\sigma \geq \tau \in \mathit{GType}_\Sigma$. To show $\tau \rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$, it suffices to let $X^\tau \in \mathrm{NV}(\langle\!\langle \Phi \rangle\!\rangle_\infty)$ and prove $\tau \in \mathrm{Inf}(\langle\!\langle \Phi \rangle\!\rangle_\infty)$. By Lemma 6.12, we obtain $\sigma' \geq \tau$ such that $X^{\sigma'} \in \mathrm{NV}(\Phi)$. Since σ and σ' have a common instance, by the first assumption we have $\mathrm{mgi}(\sigma, \sigma') \in \mathrm{Inf}^*(\Phi)$, and hence, since $\tau \leq \mathrm{mgi}(\sigma, \sigma')$, we have $\tau \in \mathrm{Inf}^*(\Phi)$—and since τ is ground, we obtain $\tau \in \mathrm{Inf}(\langle\!\langle \Phi \rangle\!\rangle_\infty)$, as desired.

(2): Let (A) $\tau \in \mathit{GType}_\Sigma$. One implication follows immediately from (1). For the second, assume (B) $\tau \rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$. To prove $\tau \rhd \Phi$, we fix (C) $X^\sigma \in \mathrm{NV}(\Phi)$ such that σ and τ have a common instance (i.e. by (A), $\tau \leq \sigma$) and prove $\mathrm{mgi}(\sigma, \tau) \in \mathrm{Inf}^*(\Phi)$, i.e. by (A), $\tau \in \mathrm{Inf}^*(\Phi)$, i.e. again by (A), $\tau \in \mathrm{Inf}(\langle\!\langle \Phi \rangle\!\rangle_\infty)$. From (C), $\tau \leq \sigma$ and Lemma 6.12, we obtain $X^\tau \in \mathrm{NV}(\langle\!\langle \Phi \rangle\!\rangle_\infty)$; hence, by (A), we have $\tau \in \mathrm{Inf}(\langle\!\langle \Phi \rangle\!\rangle_\infty)$, as desired.

(3): Immediate from the definition. □

Note that property (1) of the above lemma states that if a type is inferred monotonic in the polymorphic calculus, all its ground instances can be inferred monotonic in the monomorphic calculus associated to the completely monomorphised problem. This allows us to prove soundness of the former from soundness of the latter:

Theorem 6.16 (Soundness of $\rhd$)**.** *Let Φ be a polymorphic problem. If $\sigma \rhd \Phi$ for all $\sigma \in \mathit{Type}_\Sigma$, then Φ is monotonic.*

Proof. Assume $\sigma \rhd \Phi$ for all $\sigma \in \mathit{Type}_\Sigma$. By Lemma 6.15, $\tau \rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$ for all ground instances τ of types in Type_Σ, i.e. for all τ in the signature of $\langle\!\langle \Phi \rangle\!\rangle_\infty$. By Theorem 5.9 and Lemma 5.5, this makes $\langle\!\langle \Phi \rangle\!\rangle_\infty$ monotonic. Then, by Lemma 6.4, Φ is also monotonic. □

6.4. **General Strategy.** We will define polymorphic versions of the featherweight and lightweight tags and guard encodings, altogether four encodings: $\mathsf{t?}$, $\mathsf{t??}$, $\mathsf{g?}$, and $\mathsf{g??}$. If x ranges over the polymorphic encodings, $\widetilde{x} \in \{\widetilde{\mathsf{t?}}, \widetilde{\mathsf{t??}}, \widetilde{\mathsf{g?}}, \widetilde{\mathsf{g??}}\}$ denotes its monomorphic counterpart. We base the correctness of each x on the correctness of $\widetilde{x}$, by proving the problems $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{x}}$ and $\langle\!\langle \Phi \rangle\!\rangle_{x;\infty}$ equisatisfiable in a monotonicity-preserving way. The following lemma implicitly assumes that the signatures of $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{x}}$ and $\langle\!\langle \Phi \rangle\!\rangle_{x;\infty}$ coincide, which is easy to check for each encoding x.

Lemma 6.17 (Correctness Conditions via Complete Monomorphisation)**.** *Let Φ be a polymorphic problem and let $x \in \{\mathsf{t?}, \mathsf{t??}, \mathsf{g?}, \mathsf{g??}\}$. The problems Φ and $\langle\!\langle \Phi \rangle\!\rangle_{x;\mathsf{a};\mathsf{e}}$ are equisatisfiable provided that the following conditions hold*:

SOUND: *If $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{x}}$ has a model, $\langle\!\langle \Phi \rangle\!\rangle_{x;\infty}$ has a model with the same interpretation of the type constructors.*

COMPLETE: *If $\langle\!\langle \Phi \rangle\!\rangle_{x;\infty}$ has a model, $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{x}}$ has a model with the same interpretation of the type constructors.*

Proof. By the correctness of ∞ (Lemma 6.4) and the corresponding correctness theorems for $\widetilde{x}$ (more precisely, from the SOUND and COMPLETE statements from the proofs of Theorems 5.15 and 5.20), we have that Φ and $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{x}}$ are equisatisfiable. Together with the assumptions, this implies that (A) Φ and $\langle\!\langle \Phi \rangle\!\rangle_{x;\infty}$ are equisatisfiable. Moreover, from Lemma 6.9 and monotonicity of $\widetilde{x}$ (more precisely, from the MONO statements from the proofs of Theorems 5.15 and 5.20), we know that $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{x}}$ is monotonic. Hence, by the assumptions, (B) $\langle\!\langle \Phi \rangle\!\rangle_{x;\infty}$ is also monotonic. The desired fact follows from (A), (B), and Lemma 6.8 (with x instantiated with $x;\infty$). □

To apply the above lemma, we need to provide an x such that the equation $\langle\!\langle \Phi \rangle\!\rangle_{x;\infty} = \langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{x}}$ almost holds, in the sense that the two problems are equisatisfiable without changing the type constructor interpretation, but possibly changing some of the function or predicate symbol interpretation.

Given $\widetilde{x}$, we will come up with an encoding x in a systematic way. But first we need to define some relevant sets of types for a polymorphic Σ-problem Φ:

Definition 6.18. Let

- $\mathit{Type}_\Phi = \{\sigma \in \mathit{Type}_\Sigma \mid \sigma \text{ is the type of a subterm occurring in } \Phi\}$;
- $S_\Phi = \{\tau \in \mathit{GType}_\Sigma \mid \tau \not\rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$ and there exists $\sigma \in \mathit{Type}_\Phi$ such that $\tau \leq \sigma\}$;
- $T_\Phi = \{\tau \in \mathit{GType}_\Sigma \mid$ there exists $\sigma \in \mathit{Type}_\Phi$ such that $\sigma \not\rhd \Phi$ and $\tau \leq \sigma\}$.

Here is how we proceed with the definition of x:

(1) We emulate the definition of $\widetilde{x}$, using the polymorphic monotonicity calculus instead of the monomorphic one.

(2) Step (1) will cause $\langle\!\langle \Phi \rangle\!\rangle_{x;\infty}$ to introduce more protectors than $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\tilde{x}}$. This is because the former protects all ground types τ that are instances of types $\sigma \in \mathit{Type}_\Phi$ such that $\sigma \not\rhd \Phi$ (namely, T_Φ), whereas the latter protects all ground types τ that are instances of types in Type_Φ and satisfy $\tau \not\rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$ (namely, S_Φ). We have $S_\Phi \subseteq T_\Phi$ but generally not vice versa. To repair this mismatch, we add axioms that semantically eliminate the protectors for the types in T_Φ but not in S_Φ. To achieve this, we must characterise $T_\Phi - S_\Phi$ (which is an infinite set even for finite problems Φ) as the set of ground instances of a set U_Φ of types so that U_Φ is finite whenever Φ is.

We define U_Φ next:

Definition 6.19 (Cap)**.** Given a set of types S, a *cap* for it is a set $S' \subseteq S$ such that all types in S are instances of types in S'. A cap is *minimal* if it contains no two distinct types σ and σ' such that $\sigma \leq \sigma'$. For each σ, let U_σ be a minimal cap of the set

$$U'_\sigma = \{\sigma' \leq \sigma \mid \sigma' \in \mathrm{Inf}^*(\Phi) \text{ or there exists no } X^{\sigma''} \in \mathrm{NV}(\Phi) \text{ such that } \sigma' \text{ and } \sigma'' \text{ have a common instance}\}$$

We let U_Φ be an arbitrary cap of $\bigcup\{U_\sigma \mid \sigma \in \mathit{Type}_\Phi \text{ and } \sigma \not\rhd \Phi\}$.

The set U_Φ is both a subset of the monotonic types and a precise characterisation of the sets of types whose ground instances give our difference of interest, $T_\Phi - S_\Phi$.

Lemma 6.20. *The following properties hold*:

(1) $S_\Phi \subseteq T_\Phi$;
(2) *If* $\sigma' \in U'_\sigma$, *then* $\sigma' \rhd \Phi$;
(3) *If* $\sigma \in U_\Phi$, *then* $\sigma \rhd \Phi$;
(4) *If* $\tau \in T_\Phi - S_\Phi$, *then* $\tau \rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$;
(5) $T_\Phi - S_\Phi = \{\tau \in \mathit{GType}_\Sigma \mid \textit{there exists } \sigma \in U_\Phi \textit{ such that } \tau \leq \sigma\}$.

Proof. (1): Assume $\tau \in S_\Phi$ and let $\sigma \geq \tau$ as in the definition of S_Φ. By Lemma 6.15(2), we have $\tau \not\rhd \Phi$, and hence by Lemma 6.15(3) we have $\sigma \not\rhd \Phi$, ensuring that $\tau \in T_\Phi$.

(2): It is clear that the condition defining U'_σ is a strengthening of that defining $\sigma \rhd \Phi$.

(3): Immediate from (2).

(4): Immediate from the definitions of S_Φ and T_Φ.

(5): Let $\tau \in T_\Phi$ such that (A) $\tau \notin S_\Phi$. There exists σ such that $\tau \leq \sigma \in \mathit{Type}_\Phi$ and $\sigma \not\rhd \Phi$. From (4), we have $\tau \rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$; hence, by Lemma 6.15(2), $\tau \rhd \Phi$. Since τ is ground, either $\tau \in \mathrm{Inf}^*(\Phi)$ or there exists no $X^{\sigma''} \in \mathrm{NV}(\Phi)$ such that $\tau \leq \sigma''$, implying $\tau \in U'_\sigma$. It follows that τ is an instance of an element of U_σ, hence of an element of U_Φ, as desired. Now, assume τ is ground such that $\tau \leq \sigma' \in U_\Phi$. We obtain $\sigma \in \mathit{Type}_\Phi$ such that $\sigma \not\rhd \Phi$ and $\sigma' \in U_\sigma$. In particular, we have $\sigma' \leq \sigma$, hence $\tau \in T_\Phi$. Moreover, by $\sigma' \in U_\sigma$ and (2), we have $\sigma' \rhd \Phi$; hence by Lemma 6.15(1), we have $\tau \rhd \langle\!\langle \Phi \rangle\!\rangle_\infty$, implying $\tau \notin S_\Phi$, as desired. □

The following results will hold for any choice of set V_Φ between U_Φ and $\{\sigma \in \mathit{Type}_\Sigma \mid \sigma \rhd \Phi\}$. The smaller the chosen set, the lighter the encoding.

Convention 6.21. We fix V_Φ such that $U_\Phi \subseteq V_\varphi \subseteq \{\sigma \in \mathit{Type}_\Sigma \mid \sigma \rhd \Phi\}$.

6.5. **Monotonicity-Based Type Tags.** We are now equipped to present the definitions of the polymorphic monotonicity-based encodings and prove their correctness. The polymorphic t? encoding can be seen as a hybrid between traditional tags (t) and monomorphic lightweight tags ($\widetilde{\mathsf{t}}$?): as in t, tags take the form of a function $\mathsf{t}\langle\sigma\rangle(t)$; as in $\widetilde{\mathsf{t}}$?, tags are omitted for types that are inferred monotonic.

The main novelty concerns the typing axioms. The $\widetilde{\mathsf{t}}$? encoding omits all typing axioms for monotonic types. In the polymorphic case, the monotonic type σ might be an instance of a more general, potentially nonmonotonic 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 and hence monotonic), there will be mismatches between tagged and untagged terms. Our solution is to add the typing axiom $\mathsf{t}\langle\mathit{list}(\alpha)\rangle(Xs) \approx Xs$, 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 6.22 (Lightweight Tags t?)**.** The encoding t? translates polymorphic problems over $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$ to polymorphic problems over $(\mathit{Type}, \mathcal{F} \uplus \{\mathsf{t} : \forall\alpha.\ \alpha \to \alpha\}, \mathcal{P})$. Its term and formula translations are defined as follows:

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

The encoding adds the following typing axioms:

$$\forall \mathrm{TVars}(\sigma).\ \forall X : \sigma.\ \mathsf{t}\langle\sigma\rangle(X^{\sigma}) \approx X^{\sigma} \quad \text{for } \sigma \in V_{\Phi}$$

The *polymorphic lightweight type tags* encoding t? is the composition $\langle\!\langle\ \rangle\!\rangle_{\mathsf{t?};\mathsf{a};\mathsf{e}}$. It translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}' \uplus \{\mathsf{t}^2\}, \mathcal{P}')$, where $\mathcal{F}', \mathcal{P}'$ are as for a.

Example 6.23. The t? encoding of of Example 2.10 follows:

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

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

Definition 6.24 (Featherweight Tags t??)**.** The encoding t?? translates polymorphic problems over $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$ to polymorphic problems over $(\mathit{Type}, \mathcal{F} \uplus \{\mathsf{t} : \forall\alpha.\ \alpha \to \alpha\}, \mathcal{P})$.

Its term and formula translations are defined as follows:

$$\langle\!\langle t_1 \approx t_2 \rangle\!\rangle_{\mathsf{t??}} = \lfloor \langle\!\langle t_1 \rangle\!\rangle_{\mathsf{t??}} \rfloor \approx \lfloor \langle\!\langle t_2 \rangle\!\rangle_{\mathsf{t??}} \rfloor$$

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

with

$$\lfloor t^\sigma \rfloor = \begin{cases} t & \text{if } \sigma \rhd \Phi \text{ or } t \notin \mathcal{V}^{\forall}_{\text{typed}} \\ \mathsf{t}\langle\sigma\rangle(t) & \text{otherwise} \end{cases}$$

The encoding adds the typing axioms of t? (from Definition 6.22) and the following:

$\forall \bar\alpha.\ \forall \bar X : \bar\sigma.\ \mathsf{t}\langle\sigma\rangle(f\langle\bar\alpha\rangle(\bar X^{\bar\sigma})) \approx f\langle\bar\alpha\rangle(\bar X^{\bar\sigma})$	for $f : \forall\bar\alpha.\ \bar\sigma \to \sigma \in \mathcal{F}$ such that $\sigma \not\rhd \Phi$
$\forall \mathrm{TVars}(\sigma).\ \exists X : \sigma.\ \mathsf{t}\langle\sigma\rangle(X^\sigma) \approx X^\sigma$	for $\sigma \in \mathit{Type}_\Phi$ such that $\sigma \not\rhd \Phi$ and σ is not an instance of the result type of some $f \in \mathcal{F}$

The *polymorphic featherweight type tags* encoding t?? is the composition $\langle\!\langle\ \rangle\!\rangle_{\mathsf{t??};\mathsf{a};\mathsf{e}}$. The target signature of t?? is the same as that of t?.

Example 6.25. The t?? encoding of Example 2.10 requires fewer tags than $\widetilde{\mathsf{t}?}$, at the cost of two additional typing axioms and two typing equations for the existential variables of type b:

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

Theorem 6.26 (Correctness of t?, t??)**.** *The polymorphic type tags encodings* t? *and* t?? *are correct.*

Proof. First we discuss the case of t?. The following property can be routinely checked:
(A) $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{t?};\infty} = \{\varphi^\# \mid \varphi \in \langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{\mathsf{t}?}}\} \cup \{\forall X^\tau.\ \mathsf{t}_\tau(X^\tau) \approx X^\tau \mid \tau \in \mathrm{GInst}(V_\Phi)\}$
where $\mathrm{GInst}(V_\Phi)$ denotes the set of ground instances of types in V_Φ and $\varphi^\#$ denotes the modification of φ obtained by adding, for each $\tau \in T_\Phi - S_\Phi$, a tag t_τ around every term of type τ. Recall that $U_\Phi \subseteq V_\varphi \subseteq \{\sigma \in \mathit{Type}_\Sigma \mid \sigma \rhd \Phi\}$. We have the following:
(1) $T_\Phi - S_\Phi \subseteq \mathrm{GInst}(V_\Phi)$;
(2) $\mathrm{GInst}(V_\Phi) \subseteq \{\sigma \in \mathit{GType}_\Sigma \mid \sigma \rhd \Phi\}$;
(3) if $\tau \in V_\Phi$, the symbol t_τ does not occur in $\langle\!\langle\ \rangle\!\rangle_{\infty;\widetilde{\mathsf{t}?}}$.
Property (1) follows from Lemma 6.20(5), (2) from Lemma 6.15(3), and (3) from (2) and the definition of $\langle\!\langle\ \rangle\!\rangle_{\widetilde{\mathsf{t}?}}$. Now we are ready to check the conditions of Lemma 6.17.

Sound: Given a model $\mathcal{M}$ of $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{\mathsf{t}?}}$, we modify it into a structure $\mathcal{M}'$ by reinterpreting, for all $\tau \in \mathrm{GInst}(V_\Phi)$, the symbols t_τ as identity. Thanks to (3), $\mathcal{M}'$ is still a model of $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{\mathsf{t}?}}$ and, thanks to (A) and (1), it is also a model of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{t?};\infty}$.

COMPLETE: Any model of $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{t?};\infty}$ is also a model of $\langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{\mathsf{t?}}}$, since, thanks to (1), in the presence of $\{\forall X^\tau.\, \mathsf{t}_\tau(X^\tau) \approx X^\tau \mid \tau \in \mathrm{GInst}(V_\Phi)\}$ each $\varphi^\#$ is equivalent to φ.

The case of t?? is similar, with the following modifications:

(A) $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{t?};\infty} = \{\varphi^\# \mid \varphi \in \langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{\mathsf{t?}}}\} \cup \{\forall X^\tau.\, \mathsf{t}_\tau(X^\tau) \approx X^\tau \mid \tau \in \mathrm{GInst}(V_\Phi)\} \cup Ax$

where Ax consists of all ground instances $\varphi\rho$ of the additional axioms $\forall \bar\alpha.\, \varphi$ from Definition 6.24 where the occurring t_τ is such that $\tau \in T_\Phi - S_\Phi$. What we retain about Ax is that Ax is satisfied by any structure that interprets as identity each t_τ with $\tau \in T_\Phi - S_\Phi$, and hence, by (1), we have

(4) Ax is satisfied by any structure that interprets as identity each t_τ with $\tau \in \mathrm{GInst}(V_\Phi)$.

Moreover, $\varphi^\#$ is modified to add tags to φ in fewer places—not to arbitrary terms, but only to naked universal variables. Then the proof for t? works here too, additionally invoking (4) in the proof of soundness. □

6.6. **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 monotonic types. In the polymorphic case, the monotonic type σ might be an instance of a more general, potentially nonmonotonic 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 monotonic type σ.

Definition 6.27 (Lightweight Guards g?)**.** The encoding g? translates polymorphic problems over $\Sigma = (\mathit{Type}, \mathcal{F}, \mathcal{P})$ to polymorphic problems over $(\mathit{Type}, \mathcal{F}, \mathcal{P} \uplus \{\mathsf{g} : \forall\alpha.\, \alpha \to o\})$. Its term and formula translations $\langle\!\langle\ \rangle\!\rangle_{\mathsf{g?};\mathsf{a};\mathsf{e}}$ are defined as follows:

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

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

The encoding adds the following typing axioms:

$\forall \mathrm{TVars}(\sigma).\, \forall X : \bar\sigma.\ \mathsf{g}\langle\sigma\rangle(X^\sigma)$	for $\sigma \in V_\Phi$
$\forall \bar\alpha.\, \forall \bar X : \bar\sigma.\ \mathsf{g}\langle\sigma\rangle(f\langle\bar\alpha\rangle(\bar X^{\bar\sigma}))$	for $f : \forall\bar\alpha.\, \bar\sigma \to \sigma \in \mathcal{F}$ such that $\sigma \not\rhd \Phi$
$\forall \mathrm{TVars}(\sigma).\, \exists X : \sigma.\ \mathsf{g}\langle\sigma\rangle(X^\sigma)$	for $\sigma \in \mathit{Type}_\Phi$ such that $\sigma \not\rhd \Phi$ and σ is not an instance of the result type of some $f \in \mathcal{F}$

The *polymorphic lightweight type guards* encoding g? is the composition $\langle\!\langle\ \rangle\!\rangle_{\mathsf{g?};\mathsf{a};\mathsf{e}}$. It translates a polymorphic problem over Σ into an untyped problem over $\Sigma' = (\mathcal{F}', \mathcal{P}' \uplus \{\mathsf{g}^2\})$, where $\mathcal{F}', \mathcal{P}'$ are as for a.

The featherweight cousin is a straightforward generalisation of g? along the lines of the generalisation of $\widetilde{\mathsf{g}}$? into $\widetilde{\mathsf{g}}$??.

Definition 6.28 (Featherweight Guards g??)**.** The *polymorphic featherweight type guards* encoding g?? is identical to the lightweight encoding g? except that the condition "if $\sigma \rhd \Phi$" in the $\forall$ case is weakened to "if $\sigma \rhd \Phi$ or $X^\sigma \notin \mathrm{NV}(\varphi)$".

Example 6.29. The g?? encoding of Example 2.10 follows:

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

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

Theorem 6.30 (Correctness of g?, g??)**.** *The polymorphic type guards encodings* g? *and* g?? *are correct.*

Proof. The argument is very similar to that for tags (Theorem 6.26), with the following modifications:
(A) $\langle\!\langle \Phi \rangle\!\rangle_{\mathsf{t?};\infty} = \{\varphi^\# \mid \varphi \in \langle\!\langle \Phi \rangle\!\rangle_{\infty;\widetilde{\mathsf{t?}}}\} \cup \{\forall X^\tau.\ \mathsf{g}_\tau(X^\tau) \mid \tau \in \mathrm{GInst}(V_\Phi)\} \cup Ax$
where Ax consists of a set of formulae which is satisfied by each structure that interprets as everywhere-true each g_τ with $\tau \in \mathrm{GInst}(V_\Phi)$ and $\varphi^\#$ denotes the modification of φ obtained by adding, for each $\tau \in T_\Phi - S_\Phi$, a guard g_τ on several negative positions in φ.

Apart from this, the proofs for g? and g?? are identical to that for t??, except that instead of interpreting tags as identity we interpret guards as everywhere-true. □

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 article except for the complete monomorphisation translation from Section 6.1 (which plays only an auxiliary theoretical role in our development) enjoy the desirable property that if the source signature and problem are finite, then the target signature and problem are also finite. All of these encodings, including the traditional ones, 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. Heuristic 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 formulae, and collect all symbols occurring in the monomorphic formulae (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 formulae. Only keep fully monomorphic formulae.

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 formulae. Sledgehammer operates with $K = 3$ and $\Delta = 200$ by default, so that a problem with 500 formulae comprises at most 700 formulae after monomorphisation. Experiments found these values suitable. Given formulae 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.** Isabelle/HOL's logic, polymorphic higher-order logic with axiomatic type classes [35], is not the same as the polymorphic first-order logic considered in this article. Sledgehammer's 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* [22, §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 formulae using a unary predicate $\mathsf{hBOOL} : \mathit{bool} \to o$ (where *bool* is uninterpreted).
3. *Encode the type information.* Polymorphic types are encoded using the techniques described in this article. Type classes are essentially sets of types; they are encoded as polymorphic predicates $\forall\alpha.\ o$ (where α is a phantom type variable, Definition 3.12). For example, a predicate $\mathsf{linorder} : \forall\alpha.\ o$ could be used to restrict the axioms specifying that $\mathsf{less_eq} : \forall\alpha.\ \alpha \times \alpha \to o$ is a linear order to those types that satisfy the $\mathsf{linorder}$ predicate (cf. Example 3.15). The type class hierarchy is expressible as Horn clauses [22, §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 passes 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 [15] to automatically infer finite unsatisfiability of types. This tool relies on various proof principles to show that a set of untyped first-order formulae only has models with infinite domains. For example, it can infer that *list_b* is infinite in Example 2.13 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

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

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 [26], 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 [20], 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 [13, §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 article.

Nonetheless, 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 two 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 $\mathrm{Inf}(\Phi)$. In the theoretical part of the paper, we required infinity to be a consequence of the problem Φ. The implementation is less rigorous; it will happily treat types that are known to be infinite in the Isabelle background theories even if Φ itself does not imply infinity of the types. For example, assuming *nat* is known to be infinite, the implementation of the monotonicity-based encodings will not introduce any protectors around the naked variables M and N when translating the problem

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

(where the second conjunct is presumably the negation of a conjecture stating that there exist two distinct natural numbers). That problem is satisfiable on its own but unsatisfiable with respect to the background theory. Untyped provers will 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:

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

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

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 article, 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 1.8 [28], SPASS 3.8ds [9], and Vampire 3.0 (revision 1803) [27] and the SMT solvers Alt-Ergo 0.95.2 [10] and Z3 4.3.2 (revision 944dfee008) [23]. To make the evaluation more informative, we also tested the provers' native support for types where it is available; it is referred to as $\widetilde{\mathsf{n}}$ (monomorphic) and n (polymorphic). Only Alt-Ergo supports polymorphic types natively.

Each prover was invoked with the set of options we had previously determined worked best for Sledgehammer.[4] The provers were granted 15 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 a restricted impact on the success rate [13, §4]. To avoid giving the unsound encodings (e and a) 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 in practice.

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 $\widetilde{\mathsf{a}}$, $\widetilde{\mathsf{t}}$@, and $\widetilde{\mathsf{g}}$@ are omitted; the first two coincide with $\widetilde{\mathsf{e}}$, whereas $\widetilde{\mathsf{t}}$@ and $\widetilde{\mathsf{g}}$@ are identical to degenerate versions of $\widetilde{\mathsf{t}}$?? and $\widetilde{\mathsf{g}}$?? 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 $\widetilde{\mathsf{g}}$?? but also $\widetilde{\mathsf{t}}$??, $\widetilde{\mathsf{g}}$?, and $\widetilde{\mathsf{t}}$?) performed best overall. Their performance is close to that of the provers' native support for simple types ($\widetilde{\mathsf{n}}$). Polymorphic encodings lag behind; this is likely due in part to the synergy between the monomorphiser and the translation of higher-order constructs (cf. Section 7.2).

[3] The TPTP benchmark suite [30], which is customarily used for evaluating theorem provers, has just begun collecting polymorphic (TFF1) problems.

[4] The setup for E was suggested by Stephan Schulz and includes the little known "symbol offset" weight function. We ran Alt-Ergo with the default setup, SPASS in Isabelle mode, Vampire in CASC mode, and Z3 through the `z3_tptp` wrapper but otherwise with the default setup.

		e	a	t	t?	t??	t@	g	g?	g??	g@	n
Alt-Ergo		268	275	243	218	296	272	233	291	295	259	301
	~	293	–	245	293	301	–	296	303	**304**	–	302
E		319	330	322	311	325	269	268	325	336	288	–
	~	338	–	337	**353**	343	–	335	351	352	–	–
SPASS		289	316	290	275	319	196	223	302	314	244	–
	~	322	–	322	328	332	–	317	330	337	–	**349**
Vampire		328	335	291	304	320	243	281	320	328	301	–
	~	340	–	331	355	351	–	329	356	354	–	**370**
Z3		295	323	299	245	317	311	289	301	317	295	–
	~	326	–	307	320	341	–	342	**344**	**344**	–	343

Figure 2: Number of solved problems with 50 facts

		e	a	t	t?	t??	t@	g	g?	g??	g@	n
Alt-Ergo		268	218	183	223	273	216	157	240	273	190	260
	~	285	–	209	302	311	–	251	311	313	–	**318**
E		170	355	261	271	353	193	205	345	355	237	–
	~	376	–	331	388	**393**	–	316	388	390	–	–
SPASS		136	321	289	259	299	154	163	278	296	194	–
	~	333	–	287	323	333	–	229	327	334	–	**343**
Vampire		318	396	164	286	365	230	219	312	349	259	–
	~	406	–	231	384	406	–	314	406	405	–	**440**
Z3		247	358	253	241	362	273	213	305	363	270	–
	~	363	–	260	358	369	–	349	**370**	369	–	**370**

Figure 3: Number of solved problems with 500 facts

		e	a	t	t?	t??	t@	g	g?	g??	g@
Clauses		89	99	100	108	140	167	166	139	139	166
	~	125	–	127	127	141	–	242	141	141	–
Literals per clause		2.3	2.4	2.4	2.3	2.2	2.5	4.3	3.2	2.6	3.8
	~	2.3	–	2.3	2.3	2.2	–	4.4	2.5	2.4	–
Symbols per atom		6.3	8.0	18.3	16.0	10.3	9.9	5.7	8.0	8.6	5.7
	~	6.2	–	10.6	7.0	6.2	–	4.3	5.5	5.7	–
Symbols		1276	1870	4339	3924	3235	4070	4051	3609	3103	3610
	~	1757	–	3060	2040	1935	–	4548	1951	1904	–

Figure 4: Average size of clausified problems with 50 facts

- Among the polymorphic encodings, our novel cover-based and monotonicity-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 (an optimised representation of unary predicates [34]), but this is not corroborated by the data.
- Despite the proof reconstruction phase, the unsound encoding a achieved similar results to the sound polymorphic encodings. In contrast, its monomorphic cousin $\widetilde{\mathsf{e}}$ is generally no match for the sound monomorphic schemes.
- Oddly, the polymorphic prover Alt-Ergo performs significantly better on monomorphised problems than on the corresponding polymorphic ones. This raises serious doubts about the quality of the prover's heuristics for instantiating type variables.

For the first 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 striking point is the lightness of the monomorphic encodings, as witnessed by the number of symbols. Because monomorphisation generates several copies of the same formulae, we could have expected it to lead to larger problems, but this underestimates the cost of encoding types as terms in the polymorphic encodings.[5] 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, tags, or guards) 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 [31]. 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 [18, §4.3] and Stickel [29, p. 99]. Wick and McCune [36, §4] compare type arguments, tags, and guards in a monomorphic setting. Type arguments are described by Meng and Paulson [22], 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 [33] 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 [21] supports a restricted form of higher-rank polymorphism (with polymorphic maps), and Why3 [11] provides rank-1 polymorphism. Both define translations to a monomorphic logic and rely on proxies to handle interpreted types [12, 17, 21]. One of the Boogie translations [21, §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. [14, §4]

[5] The increase visible in the e column, from 89 to 125 clauses and from 1276 to 1757 symbols, is due to the multiple copies arising from monomorphisation. Even though these become identical after type erasure, they are counted as separate in the statistics, and it is up to the automatic provers to notice that they are the same.

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 production-quality automatic prover that supports polymorphism is Alt-Ergo [10].

Blanchette and Krauss [6] studied monotonicity inferences for higher-order logic without polymorphism. Claessen et al. [16] were first to apply them to type erasure.

10. Conclusion

This article 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, 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 [16]. Applications such as Boogie [21], LEO-II [2], and Why3 [11] also stand to gain from lighter encodings.

The TPTP family recently welcomed the addition of TFF1 [7], 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 strategy scheduling or parallelism, it is advantageous 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, Nir Piterman, Alexander Steen, Mark Summerfield, Tjark Weber, and several anonymous reviewers suggested dozens of textual improvements; in particular, the three reviewers of this extended, journal version accomplished a remarkable work, going well beyond the call of duty. Blanchette was partly supported by the Deutsche Forschungsgemeinschaft (DFG) projects Quis Custodiet and Hardening the Hammer (grants NI 491/11-2 and NI 491/14-1). Popescu was partly supported by the DFG project Security Type Systems and Deduction (grant NI 491/13-2) as part of the program Reliably Secure Software Systems (RS^3, priority program 1496). The authors are listed alphabetically.

References

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

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

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

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

[5] Blanchette, J.C., Böhme, S., Popescu, A., Smallbone, N.: Encoding monomorphic and polymorphic types. In: N. Piterman, S. Smolka (eds.) TACAS 2013, *LNCS*, vol. 7795, pp. 493–507. Springer (2013)

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

[7] Blanchette, J.C., Paskevich, A.: TFF1: The TPTP typed first-order form with rank-1 polymorphism. In: M.P. Bonacina (ed.) CADE-24, *LNAI*, vol. 7898, pp. 414–420. Springer (2013)

[8] Blanchette, J.C., Popescu, A.: Mechanizing the metatheory of Sledgehammer. In: P. Fontaine, C. Ringeissen, R.A. Schmidt (eds.) FroCoS 2013, *LNCS*, vol. 8152, pp. 245–260. Springer (2013)

[9] Blanchette, J.C., Popescu, A., Wand, D., Weidenbach, C.: More SPASS with Isabelle—Superposition with hard sorts and configurable simplification. In: L. Beringer, A. Felty (eds.) ITP 2012, *LNCS*, vol. 7406, pp. 345–360. Springer (2012)

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

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

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

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

[14] Bouillaguet, C., Kuncak, V., Wies, T., Zee, K., Rinard, M.: Using first-order theorem provers in the Jahob data structure verification system. In: B. Cook, A. Podelski (eds.) VMCAI 2007, *LNCS*, vol. 4349, pp. 74–88. Springer (2007)

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

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

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

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

[19] Hodges, W.: Model Theory. Cambridge University Press (1993)

[20] Hurd, J.: First-order proof tactics in higher-order logic theorem provers. In: M. Archer, B. Di Vito, C. Muñoz (eds.) Design and Application of Strategies/Tactics in Higher Order Logics, no. CP-2003-212448 in NASA Tech. Reports, pp. 56–68 (2003)

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

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

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

[24] Nipkow, T., Klein, G.: Concrete Semantics—with Isabelle/HOL. Springer (2014)

[25] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic, *LNCS*, vol. 2283. Springer (2002)

[26] Paulson, L.C., Susanto, K.W.: Source-level proof reconstruction for interactive theorem proving. In: K. Schneider, J. Brandt (eds.) TPHOLs 2007, *LNCS*, vol. 4732, pp. 232–245. Springer (2007)
[27] Riazanov, A., Voronkov, A.: The design and implementation of Vampire. AI Comm. **15**(2-3), 91–110 (2002)
[28] Schulz, S.: System description: E 0.81. In: D. Basin, M. Rusinowitch (eds.) IJCAR 2004, *LNAI*, vol. 3097, pp. 223–228. Springer (2004)
[29] Stickel, M.E.: Schubert's steamroller problem: Formulations and solutions. J. Autom. Reasoning **2**(1), 89–101 (1986)
[30] Sutcliffe, G.: The TPTP problem library and associated infrastructure—The FOF and CNF parts, v3.5.0. J. Autom. Reasoning **43**(4), 337–362 (2009)
[31] Sutcliffe, G.: Proceedings of the 6th IJCAR ATP system competition (CASC-J6). In: G. Sutcliffe (ed.) CASC-J6, *EPiC*, vol. 11, pp. 1–50. EasyChair (2012)
[32] Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: N. Bjørner, A. Voronkov (eds.) LPAR-18, *LNCS*, vol. 7180, pp. 406–419. Springer (2012)
[33] Urban, J.: MPTP 0.2: Design, implementation, and initial experiments. J. Autom. Reasoning **37**(1-2), 21–43 (2006)
[34] Weidenbach, C.: Combining superposition, sorts and splitting. In: A. Robinson, A. Voronkov (eds.) Handbook of Automated Reasoning, vol. 2, pp. 1965–2013. Elsevier (2001)
[35] Wenzel, M.: Type classes and overloading in higher-order logic. In: E.L. Gunter, A. Felty (eds.) TPHOLs 1997, *LNCS*, vol. 1275, pp. 307–322. Springer (1997)
[36] Wick, C.A., McCune, W.W.: Automated reasoning about elementary point-set topology. J. Autom. Reasoning **5**(2), 239–255 (1989)

This work is licensed under the Creative Commons Attribution-NoDerivs License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/2.0/ or send a letter to Creative Commons, 171 Second St, Suite 300, San Francisco, CA 94105, USA, or Eisenacher Strasse 2, 10777 Berlin, Germany