Polymorphic Theorem Proving for DHOL

Daniel Ranalter
University of Innsbruck, Computational Logic, Austria

Florian Rabe
University of Erlangen-Nuremberg, Computer Science, Germany

Cezary Kaliszyk
University of Melbourne, School of Computing and Information Systems, Australia
University of Innsbruck, Computational Logic, Austria

Abstract

DHOL is an extensional, classical logic that equips the well-known higher-order logic (HOL) with dependent types. This allows for concise encodings of important domains like size-bounded data structures, category theory, or proof theory. Automation support is obtained by translating DHOL to HOL, for which powerful modern automated theorem provers are available. However, a critically missing feature of DHOL is polymorphism. Indeed, in many practical applications, it is important that types may take both term arguments (DHOL) and type arguments (polymorphism) e.g., in vectors of fixed length and type. We present a sweet spot in the design space by defining polymorphic DHOL. We extend the syntax and semantics as well as the translation and the judgment preservation/truth reflection proofs to the polymorphic case and consider two generalizations of PDHOL. The expressivity of polymorphic DHOL is demonstrated on a range of TPTP formalizations and we evaluate practical theorem proving support on off-the-shelf HOL theorem provers.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Higher order logic; Theory of computation → Type theory

Keywords and phrases Polymorphism, Dependent Types, Higher-order Logic, Automated Reasoning

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Supplementary Material The TPTP problems, the implementation of the translator and type checker: *Dataset, Software*: https://drive.google.com/drive/folders/14H9nH-voaF35X69y-IR0LiCMtJrakUvd

1 Introduction

Setting Monomorphic dependently typed higher-order logic (DHOL) was introduced in [25, 26]. It combines the simplicity of higher-order logic (HOL) [5, 10], particularly the use of extensionality and classical Booleans, with the often-wished-for feature of dependent types. Contrary to proof assistants based on dependent type theory [6, 18, 8], it uses dependent types in the simplest possible setting and, in particular, does not introduce universes or inductive types. Instead, it only changes HOL's simple function type $A \to B$ into a dependent one $\Pi_{x:A}B$ and allows for base types a to depend on typed arguments.

This makes typing undecidable [13] but has the benefit of being near to languages in the HOL ecosystem for which strong automated theorem proving (ATP) support exists. This is leveraged by giving a linear, compositional, and judgment preserving/truth reflecting translation from monomorphic DHOL to monomorphic HOL to obtain implementations of type-checking and theorem proving for DHOL [26], based on partial equivalence relations (PERs) [1]. Practical experiments show that this combination of expressivity and ATP support makes undecidable typing a price worth paying [16].

Contribution We greatly increase the expressivity of DHOL by extending it with shallow (ML style, rank 1) polymorphism and subtype definitions. The resulting PDHOL is deceptively simple, constituting a good trade-off between being expressive enough to meet practical needs and simple enough to retain strong automation support. We amend the translation to target polymorphic HOL (PHOL) and show it remains well-behaved.

To demonstrate the practical value, we give several elegant and concise formalizations of practically important problems. These include common polymorphic data structures where dependent types enable tracking key invariants, e.g., the dimensions of vectors and matrices or the black height of red-black trees, as well as algebraic data structures, where polymorphism is needed to abstract over the carrier, and dependent types occur naturally, e.g., in the type of homomorphisms between two groups. We prove multiple theorems automatically using PHOL ATPs and a polymorphic extension of the TPTP syntax for DHOL [29, 22].

Shallow polymorphism allows all global declarations (type symbols, term symbols, and axioms) to depend on type variables $\alpha : \mathtt{type}$. Consequently, all references to such a declaration must provide (or allow inferring) an appropriate substitution for the type variables. In particular, it allows declaring type operators $a : \Pi_\alpha : \mathtt{type}$ and polymorphic operations and axioms about them. In combination with dependent types, it allows, e.g., type operators like $\mathtt{vec} : \Pi_\alpha \Pi_{\mathtt{nat}} : \mathtt{type}$ for fixed-dimension vectors over type α. Shallow polymorphism does *not* allow deep quantification over types (e.g. $\neg\forall\alpha : \mathtt{type}\ldots$) or higher-order type variables (e.g. $\lambda_{\alpha:\mathtt{type}\to\mathtt{type}}\ldots$). However, it allows type variables to depend on terms (e.g. $\Pi_{\alpha:\mathtt{nat}\to\mathtt{type}}\ldots$), which we use to sketch a variant of PDHOL with dependent type variables in Sect. 7, using heterogeneous lists as an example.

2 Syntax

The grammar below shows both the DHOL language and our extension to PDHOL. We also mark the parts that must be removed or, in the case of the function types, adjusted to recover HOL as a fragment. $., \circ, \bullet$ denote the empty theory, context, and substitution respectively, and Φ represents a meta variable for terms of type o. The remaining notational peculiarities are explained in this section.

$$
\begin{array}{lcll}
T & ::= & . \mid T, a : \Pi_{\vec{\alpha}}\Pi_{\vec{x}:\vec{A}} : \mathtt{type} \mid T, c : \Pi_{\vec{\alpha}} A \mid T, \triangleright\, \Pi_{\vec{\alpha}}\Phi & \text{Theories} \\
\Gamma & ::= & \circ \mid \Gamma, \alpha : \mathtt{type} \mid \Gamma, x : A \mid \Gamma, \triangleright\, \Phi & \text{Context} \\
\gamma & ::= & \bullet \mid \gamma, A \mid \gamma, t \mid \gamma, \checkmark & \text{Substitutions} \\
A, B & ::= & a\, \vec{A}\, \vec{t} \mid \alpha \mid \Pi_{x:A} B \mid o & \text{Types} \\
t, u, \Phi & ::= & c\, \vec{A} \mid x \mid \lambda_{x:A} t \mid t\, u \mid t \Rightarrow u \mid t =_A u & \text{Terms (including formulas)}
\end{array}
$$

This suffices to define all the usual logical connectives and binders [2], and we write $A \to B$ as usual. We also use the following abbreviations for sequences of expressions:

abbr.	expansion	remark
$\vec{A}$	$A_1 \ \ldots\ A_n$	types in a substitution, application or binding
$\vec{t}$	$t_1 \ \ldots\ t_n$	terms in a substitution or application
$\vec{\alpha} : \mathtt{type}$	$\alpha_1 : \mathtt{type} \ldots \alpha_n : \mathtt{type}$	type variables in a context or binding
$\vec{x} : \vec{A}$	$x_1 : A_1 \ldots x_n : A_n$	term variables in a context or binding

To avoid case distinctions, we will occasionally merge lists $\Pi_{\vec{\alpha}}\Pi_{\vec{x}:\vec{A}}$ of types and term argument bindings into a single list Π_Δ for a context Δ. Similarly, we may merge list $\vec{A}\,\vec{t}$ of type and term arguments into a single substitution δ.

▶ **Example 1.** We present fixed-length lists, sometimes called vectors, as an intuitive running example. For that, we start with natural numbers:

$$\mathtt{nat} : \mathtt{type} \qquad 0 : \mathtt{nat} \qquad \mathtt{suc} : \mathtt{nat} \to \mathtt{nat} \qquad + : \mathtt{nat} \to \mathtt{nat} \to \mathtt{nat}$$

$$\triangleright\ \forall n : \mathtt{nat}. + 0\ n =_{\mathtt{nat}} n \qquad \triangleright\ \forall n, m : \mathtt{nat}. + (\mathtt{suc}\ n)\ m =_{nat} \mathtt{suc}\ (+\ n\ m)$$

Here, `nat` is a non-dependent, or simple, base type for which a constant, a constructor, and a function are declared. We will abbreviate $\texttt{suc}\ 0, \texttt{suc}\ (\texttt{suc}\ 0), \ldots$ with $1, 2, \ldots$.

Everything stated so far is expressible in regular HOL — neither dependent types nor polymorphism play a role. We now extend this theory to vectors, keeping the highlighting conventions used in the grammar.

$$\texttt{vec} : \Pi_\alpha \Pi_{n:\texttt{nat}} : \texttt{type} \quad \texttt{nil} : \Pi_\alpha \texttt{vec}\ \alpha\ 0 \quad \texttt{cons} : \Pi_\alpha \Pi_{n:\texttt{nat}} \alpha \to \texttt{vec}\ \alpha\ n \to \texttt{vec}\ \alpha\ (\texttt{suc}\ n)$$

$$++ : \Pi_\alpha \Pi_{n,m:\texttt{nat}} \texttt{vec}\ \alpha\ n \to \texttt{vec}\ \alpha\ m \to \texttt{vec}\ \alpha\ (+\ n\ m)$$

Removing the highlighted dependent part yields polymorphic, dynamic lists in PHOL while instantiating the highlighted polymorphic part results in fixed-type vectors in DHOL. Doing both of these gives fixed-type, dynamic lists in HOL.

Note that, if one would now want to prove the associativity of $++$, type checking the statement would require a proof of the associativity of $+$ — turning type checking into an undecidable problem in general.

Contexts and Substitutions Contexts Γ are lists of local declarations, subject to α-renaming and substitution as usual: (i) type variables $\alpha : \texttt{type}$ (ii) typed variables $x : A$ (iii) local assumptions $\rhd \Phi$.

Substitutions $\gamma : \Gamma \to \Gamma'$ provide type/term expressions for all type/term variables declared in Γ in such a way that the assumptions made in Γ are satisfied. To track when Γ contained an assumption that must be proved, we write ✓ in the corresponding place of a substitution. If we extended the formulation with a language of proofs, the ✓ would be replaced with an appropriate proof expression. We write $E[\gamma]$ for the result of substituting in expression E according to γ, and we abbreviate as $E[t]$ the common case where the substitution is the identity for all variables but the last one.

▶ **Example 2.** Assume a typing expression E for a 2-element vector `[n, m]`, represented formally as $\texttt{cons nat}\ 1\ n\ (\texttt{cons nat}\ 0\ m\ (\texttt{nil nat})) : \texttt{vec nat}\ 2$. For this to be properly typed, the context must include typing statements for n and m. For the sake of this example, we also add two assumptions resulting in the context: $n : \texttt{nat}, m : \texttt{nat}, \rhd n =_{\texttt{nat}} 2, \rhd m =_{\texttt{nat}} 3$.

A well-formed substitution γ for this context is $2, \mathit{suc}\ 2, ✓, ✓$. $E[\gamma]$ would then be $\texttt{cons nat}\ 1\ 2\ (\texttt{cons nat}\ 0\ (\texttt{suc}\ 2)\ (\texttt{nil nat})) : \texttt{vec nat}\ 2$. The ✓s in the substitution represent the proof obligations $2 =_{\texttt{nat}} 2$and $\mathit{suc}\ 2 =_{\texttt{nat}} 3$ which are trivial after unfolding the abbreviations used for numbers.

Theories Theories T are lists of global declarations: (i) base types a depending on typed arguments $x : A$ (ii) typed constants c (iii) global assumptions (i.e. axioms) $\rhd \Phi$.

Contrary to contexts, declarations in theories may depend on arguments, and this is the mechanism how both monomorphic DHOL [26] and our extension thereof are defined as generalizations of HOL. The following table gives an overview of the possible combinations:

	declaration of		
depending on	type symbol	term symbol	global assumption
type variable	allowed in PDHOL and PHOL		
term variable	allowed in DHOL	definable via Π	definable via $\forall$
local assumption	not allowed		definable via $\Rightarrow$

DHOL arises from HOL by allowing type symbols to depend on term arguments. PDHOL arises by allowing all declarations to depend on type arguments. Three combinations are always definable and therefore redundant. The remaining two cases (local assumptions as parameters of type/term symbols) would allow declaring partial functions and partial type symbols. We could easily extend PDHOL to allow this, but as it would make the translation to PHOL significantly more complicated, we do not consider that here.

Types and Terms Types $A, B, \dots$ and terms $t, u, \dots$ are formed from

- references to symbols declared in the theory, which must always be fully instantiated with type and term arguments where applicable
- references to variables declared in the context
- the usual production rules of the grammar for dependent function types $\Pi_{x:A}B$, function formation $\lambda_{x:A}t$ and application $t\ u$,
- production rules of the grammar for Booleans o formed from typed equality $t =_A u$ and dependent implication $\Phi \Rightarrow \Psi$.

We discuss a few non-obvious design choices in the sequel.

▶ Remark 3 (Parametric Declarations). DHOL avoids kinds and `type`-level λ-abstraction: the production for dependent base types uses the normal form $(\Pi_{x:A})^* : \texttt{type}$ rather than $a : K$ for an arbitrary kind. Consequently, type symbols can only be used in fully applied form $a\ \delta$. PDHOL follows that decision for polymorphic symbols. λ-abstraction over type variables in terms/types or over term variables in types can be added easily.

▶ Remark 4 (Dependent Connectives). PDHOL follows DHOL in using *dependent* implication: in $F \Rightarrow G$, the well-formedness of G may already assume F is true. This is critical for type-checking, e.g., in $a =_A b \Rightarrow f\ a =_{B[a]} f\ b$ for a dependent function f. Unable to define dependent implication from equality, DHOL makes implication primitive as well, and uses that to define corresponding dependent variants of conjunction and disjunction. While the loss of commutativity of conjunction/disjunction may seem odd at first, the behavior is well-known from short circuit evaluation in some programming languages.

▶ Remark 5 (Reasoning about Types). Like PHOL and DHOL, PDHOL does not allow direct reasoning about types: Equality $A \equiv B$ of types is only a meta-level judgment and not a Boolean term. And there is no quantification over type variables except for the implicit universal quantification over the type variables of a polymorphic axiom.

3 Inference System

Name	Judgment	Intuition
theories	$\vdash T$ `Thy`	T is a well-formed theory
contexts	$\vdash_T \Gamma$ `Ctx`	Γ is a well-formed context
substitutions	$\Gamma \vdash_T \Delta \ \leftarrow\ \delta$	δ is a well-formed substitution for Δ
types	$\Gamma \vdash_T A\ :\ \texttt{type}$	A is a well-formed type
typing	$\Gamma \vdash_T t\ :\ A$	t is a well-formed term of well-formed type A
validity	$\Gamma \vdash_T \Phi$	Boolean Φ is derivable
equality of types	$\Gamma \vdash_T A \equiv B$	well-formed types A and B are equal

Figure 1 DHOL Judgments

In Fig. 1, we give the judgments of PDHOL. These look the same as for DHOL—but as contexts Γ can now contain type variables, they correspond to polymorphic statements.

We use $\leftarrow$ in the judgment for well-formed substitutions to avoid confusion with the simple function type constructor $\to$. Fig. 2 and 3 present the rules of PDHOL. They arise as an extension of the rules of DHOL. We use $\in, \notin$ on lists in the obvious manner.

Well-formed theories T

$$\frac{}{\vdash .\ \mathtt{Thy}}\,\mathit{ThyBase} \qquad \frac{\vdash T\ \mathtt{Thy} \quad a \notin T \quad \vdash_T \Delta\ \mathtt{Ctx} \quad \Delta = \vec{\alpha} : \mathtt{type}, \vec{x} : \vec{A}}{\vdash T, a : \Pi_\Delta : \mathtt{type}\ \mathtt{Thy}}\,\mathit{ThyType}$$

$$\frac{\vdash T\ \mathtt{Thy} \quad c \notin T \quad \Delta \vdash_T A : \mathtt{type} \quad \Delta = \vec{\alpha} : \mathtt{type}}{\vdash T, c : \Pi_\Delta A\ \mathtt{Thy}}\,\mathit{ThyCon}$$

$$\frac{\vdash T\ \mathtt{Thy} \quad \Delta \vdash_T \Phi : o \quad \Delta = \vec{\alpha} : \mathtt{type}}{\vdash T, \rhd \Pi_\Delta \Phi\ \mathtt{Thy}}\,\mathit{ThyAss}$$

Contexts Δ (left) and substitutions δ for them (right)

$$\frac{\vdash T\ \mathtt{Thy}}{\vdash_T \circ\ \mathtt{Ctx}}\,\mathit{CtxBase} \qquad \frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad \alpha \notin \Gamma}{\vdash_T \Gamma, \alpha : \mathtt{type}\ \mathtt{Ctx}}\,\mathit{CtxTp} \qquad \frac{\vdash_T \Gamma\ \mathtt{Ctx}}{\Gamma \vdash_T \bullet \leftarrow \circ}\,\mathit{SubBase}$$

$$\frac{\Gamma \vdash_T \Delta \leftarrow \delta \quad \alpha \notin \Delta \quad \Gamma \vdash_T A : \mathtt{type}}{\Gamma \vdash_T \Delta, \alpha : \mathtt{type} \leftarrow \delta, A}\,\mathit{SubTp} \qquad \frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad x \notin \Gamma \quad \Gamma \vdash_T A : \mathtt{type}}{\vdash_T \Gamma, x : A\ \mathtt{Ctx}}\,\mathit{CtxVar}$$

$$\frac{\Gamma \vdash_T \Delta \leftarrow \delta \quad x \notin \Delta \quad \Gamma \vdash_T A : \mathtt{type} \quad \Gamma \vdash_T t : A[\delta]}{\Gamma \vdash_T \Delta, x : A \leftarrow \delta, t}\,\mathit{SubVar}$$

$$\frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad \Gamma \vdash_T \Phi : o}{\vdash_T \Gamma, \rhd \Phi\ \mathtt{Ctx}}\,\mathit{CtxAss} \qquad \frac{\Gamma \vdash_T \Delta \leftarrow \delta \quad \Gamma \vdash_T \Phi : o \quad \Gamma \vdash_T \Phi[\delta]}{\Gamma \vdash_T \Delta, \rhd \Phi \leftarrow \delta, \checkmark}\,\mathit{SubAss}$$

Lookup of type/term/assumption in a theory (left) or context (right)

$$\frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad a : \Pi_\Delta : \mathtt{type} \in T \quad \Gamma \vdash_T \Delta \leftarrow \delta}{\Gamma \vdash_T a\ \delta : \mathtt{type}}\,\mathit{TpSym} \qquad \frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad \alpha : \mathtt{type} \in \Gamma}{\Gamma \vdash_T \alpha : \mathtt{type}}\,\mathit{TpVar}$$

$$\frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad c : \Pi_\Delta A \in T \quad \Gamma \vdash_T \Delta \leftarrow \delta}{\Gamma \vdash_T c\ \delta : A[\delta]}\,\mathit{TermSym} \qquad \frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad x : A \in \Gamma}{\Gamma \vdash_T x : A}\,\mathit{TermVar}$$

$$\frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad \rhd \Pi_\Delta \Phi \in T \quad \Gamma \vdash_T \Delta \leftarrow \delta}{\Gamma \vdash_T \Phi[\delta]}\,\mathit{ValidSym} \qquad \frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad \rhd \Phi \in \Gamma}{\Gamma \vdash_T \Phi}\,\mathit{ValidVar}$$

Figure 2 DHOL Rules for theories and contexts.

The rules in Fig. 2 cover the structural parts, i.e., the declaration of and references to declarations in the theory and context. Note how the rules for theory and context formation are parallel. For example, Rule ThyType and Rule CtxTp handle the declaration of types $a : \Pi_{\vec{\alpha}}, \Pi_{\vec{x}:\vec{A}} : \mathtt{type}$ in a theory resp. $\alpha : \mathtt{type}$ in a context. The main difference is that the former allows type symbols a, term symbols c, and global assumptions to depend on a list Δ of parameters. To emphasize the common structure of the handling of parameters, we unify all parameter lists notationally into a single context Δ.

Each of these six rules for making the declaration is paralleled by one of six rules for referencing (looking up). For example, Rule TpSym and Rule TpVar reference type symbols a resp. type variables α. Because the former is parametric in some context Δ, their references must be instantiated with an appropriate substitution δ for the parameters.

Finally, the four rules for forming contexts are parallel to the four rules for forming substitutions. For example, Rule SubTp shows how to extend a substitution δ for Δ to a substitution for $\Delta, \alpha : \mathtt{type}$ by adding a type substitute A for α.

The rules in Fig. 3 cover the rules for expressions. We only point out some specialties:

Booleans: type formation, terms for equality and implication

$$\frac{\vdash_T \Gamma\ \texttt{Ctx}}{\Gamma \vdash_T o\ :\ \texttt{type}}\,\mathit{TpBool} \qquad \frac{\Gamma \vdash_T t\ :\ A \quad \Gamma \vdash_T u\ :\ A}{\Gamma \vdash_T t =_A u\ :\ o}\,\mathit{TermEq}$$

$$\frac{\Gamma \vdash_T \Phi\ :\ o \quad \Gamma, \triangleright\, \Phi \vdash_T G\ :\ o}{\Gamma \vdash_T \Phi \Rightarrow \Psi\ :\ o}\,\mathit{TermImpl}$$

Functions: type formation, λ-abstraction, application

$$\frac{\Gamma \vdash_T A\ :\ \texttt{type} \quad \Gamma, x : A \vdash_T B\ :\ \texttt{type}}{\Gamma \vdash_T \Pi_{x:A}B\ :\ \texttt{type}}\,\mathit{TpPi}$$

$$\frac{\Gamma \vdash_T A\ :\ \texttt{type} \quad \Gamma, x : A \vdash_T t\ :\ B}{\Gamma \vdash_T \lambda_{x:A}t\ :\ \Pi_{x:A}B}\,\mathit{TermLambda} \qquad \frac{\Gamma \vdash_T t\ :\ \Pi_{x:A}B \quad \Gamma \vdash_T u\ :\ A}{\Gamma \vdash_T t\ u\ :\ B[u]}\,\mathit{TermApply}$$

Type equality: congruence for all constructors and conversion of types

$$\frac{\vdash_T \Gamma\ \texttt{Ctx} \quad \alpha\ :\ \texttt{type} \in \Gamma}{\Gamma \vdash_T \alpha \equiv \alpha}\,\mathit{TpEqVar} \qquad \frac{\vdash_T \Gamma\ \texttt{Ctx} \quad a : \Pi_\Delta : \texttt{type} \in T \quad \Gamma \vdash_T \delta \equiv_\Delta \delta'}{\Gamma \vdash_T a\ \delta \equiv a\ \delta'}\,\mathit{TpEqSym}$$

$$\frac{\vdash_T \Gamma\ \texttt{Ctx}}{\Gamma \vdash_T o \equiv o}\,\mathit{TpEqBool} \qquad \frac{\Gamma \vdash_T A \equiv A' \quad \Gamma, x : A \vdash_T B \equiv B'}{\Gamma \vdash_T \Pi_{x:A}B \equiv \Pi_{x:A'}B'}\,\mathit{TpEqPi}$$

$$\frac{\Gamma \vdash_T t\ :\ A \quad \Gamma \vdash_T A \equiv A'}{\Gamma \vdash_T t\ :\ A'}\,\mathit{TermConv} \qquad \frac{\Gamma \vdash_T t\ \bot \quad \Gamma \vdash_T t\ \top \quad \top, \bot \text{ defined as usual}}{\Gamma, x : o \vdash_T t\ x}\,\mathit{BoolExt}$$

where $\Gamma \vdash_T \delta \equiv_\Delta \delta'$ abbreviates the expression-wise provable equality of two substitutions for Δ

We omit the following routine validity rules: congruence rules for application; β, η for functions; introduction and elimination for $\Rightarrow$
Note that propositional and functional extensionality is implied by the rules for equality and the omitted eta rule [26].

Figure 3 DHOL Rules for Types and Terms

The rule Rule TermImpl makes implication $\Phi \Rightarrow \Psi$ dependent by assuming Φ while in the well-formedness of Ψ. The rule Rule TpEqSym, while looking like a routine congruence rule, is the rule that makes type-checking undecidable by making two types equal if their arguments are equal component-wise. Here the equality $\Gamma \vdash_T \delta \equiv_\Delta \delta'$ of substitutions holds if the term/type equality judgments hold for all corresponding pairs of terms/types in δ and δ'. And because term equality may depend on arbitrary assumptions in theory and contexts, so do all judgments. Finally, Rule TermConv is only needed for constants and variables; for any other term, it can be derived using congruence.

4 Translating PDHOL to PHOL

Overview Like for DHOL the translation applies dependency erasure, written with an overline $\overline{X}$, to turn PDHOL syntax X into PHOL syntax. Note that because the latter is a fragment of the former, we can reuse all PDHOL notations to write PHOL syntax. Intuitively, term-dependent types are translated into types without their term arguments. The lost information is captured in a partial equivalence relation (PER) in the style of [1]. Readers may consult the examples below or the invariants stated in Thm. 9 to help their intuitions.

All term arguments of type symbols are erased; type arguments are kept, i.e., polymorphic symbols remain polymorphic. In particular, we translate the type $a\ \vec{A}\ \vec{t}$ to $a\ \overline{\vec{A}}$ and the type $\Pi_{x:A}B$ to $\overline{A} \to \overline{B}$. To recover the erased typing information, we define for every PDHOL-type A a PER A^* on $\overline{A}$ in PHOL such that the PHOL-formula $A^*\ \overline{t}\ \overline{t}$ captures the PDHOL-typing judgment $t : A$. Critically, term equality $s =_A t$ is translated to $A^*\ \overline{s}\ \overline{t}$.

PERs are symmetric and transitive relations, and an element in a PER is related to itself iff it is related to any element. So A^* is an equivalence relation on a subtype of $\overline{A}$. The corresponding quotient of that subtype of $\overline{A}$ is the semantics of A under our translation. We write $\mathrm{PER}(r)$ to abbreviate that r is a PER (i.e., symmetric and transitive).

Expanding the usual definitions of the quantifiers reveals that (up to provable equivalence) $A^*\ x\ x$ acts as a guard when quantifying over x. One might think that a unary predicate (indicating a subtype) would suffice as a type guard instead of a binary predicate. But using quotients of subtypes (and thus PERs) becomes necessary at higher function types where two functions are equal if they map type-guarded inputs to equal outputs.

Auxiliary Notations While conceptually straightforward, binding the parameters in theories in all generated declarations is notationally complex. Therefore, we abbreviate as follows:

▶ **Definition 6** (Abbreviations for PHOL-Contexts). *Given a PHOL-context Γ and a PHOL-substitution γ, we abbreviate*

- *Γ^{y} is like Γ but contains only the type variables.*
- *Γ^{ye} is like Γ but contains only the type and term variables. (In HOL, as opposed to DHOL, such taking of subcontexts is always legal as long as the order is preserved.)*
- *γ^{y} is like γ but contains only the type substitutes.*
- *γ^{ye} is like γ but contains only the type and term substitutes.*
- *If Γ consists of only type variables, we write $\Gamma \to \mathtt{type}$ for the kind $\mathtt{type} \to \ldots \to \mathtt{type} \to \mathtt{type}$ (taking one type argument for every type variable in Γ).*
- *If Γ consists of type variables $\vec{\alpha}$ and term variables $\vec{x} : \vec{A}$, we write $\Gamma \to B$ for $\Pi_{\vec{\alpha}} A_1 \to \ldots \to A_n \to B$.*
- *If Γ consists of type variables $\vec{\alpha}$, term variables $\vec{x} : \vec{A}$, and assumptions $\triangleright\ \Phi_i$, we write $\Gamma \Rightarrow \Psi$ for the PHOL-formula $\Pi_{\vec{\alpha}} \forall x_1 : A_1. \ldots \forall x_m : A_m. \Phi_1 \Rightarrow \ldots \Phi_n \Rightarrow \Psi$; if Γ contains alternating term variables and assumptions, we alternate the $\forall$ and $\Rightarrow$ bindings accordingly.*

The abbreviations Γ^{y} and Γ^{ye} remove parameters that a PHOL-declaration cannot bind: term symbol declarations cannot bind assumptions, and type symbol declarations cannot bind assumptions or term variables. These occur because every type A yields a translated type $\overline{A}$, a binary relation A^*, and a statement $\mathrm{PER}(A^*)$. Consequently, a type variable α is translated to three declarations: a type variable α, a binary relation α^* on it, and a local assumption $\mathrm{PER}(\alpha^*)$. Similarly, every term $t : A$ yields a translated term $\overline{t}$ and a statement $A^*\ \overline{t}\ \overline{t}$. Consequently, every term variable x is translated to two declarations: a term variable and a local assumption. Thus, $\overline{\Gamma}$ contains a mixture of type variables, term variables, and assumptions even if Γ contains only type variables. The latter have to be removed if we want to bind $\overline{\Gamma}$ in a PHOL-type or term symbol declaration.

The abbreviations $\Gamma \to \mathtt{type}$, $\Gamma \to A$, and $\Gamma \Rightarrow \Phi$ efficiently perform these bindings. For example, if a PDHOL axiom $\triangleright\ \Pi_{\vec{\alpha}}\Phi$ is parametric in type variables $\alpha_1, \ldots, \alpha_n$, then $\overline{\Gamma}$ contains $3n$ declarations. $\overline{\Gamma} \Rightarrow \overline{\Phi}$ binds all of them to construct a PHOL-axiom: the type variables in $\overline{\Gamma}$ remain type variables, the term variables are $\forall$-bound, and the unnamed assumptions are bound by $\Rightarrow$. Any PDHOL usage of this axiom instantiates each type variable α with a type A. In PHOL, this corresponds to instantiating α with $\overline{A}$, universal elimination with A^*, and implication elimination with $\mathrm{PER}(A^*)$.

Formal Definition We translate types and terms as follows:

PDHOL type A	Translation $\overline{A}$ in PHOL	PDHOL term t	Translation $\overline{t}$ in PHOL
$a\ \delta$	$a\ \overline{\delta}^{y}$	$c\ \delta$	$c\ \overline{\delta}^{ye}$
α	α	x	x
o	o	$t =_A u$	$A^*\ t\ u$
		$\Phi \Rightarrow \Psi$	$\overline{\Phi} \Rightarrow \overline{\Psi}$
$\Pi_{x:A}B$	$\overline{A} \to \overline{B}$	$\lambda_{x:A}t$	$\lambda_{x:\overline{A}}\overline{t}$
		$t\ u$	$\overline{t}\ \overline{u}$

For the translation of type and term symbol *references*, compare the matching translation of the corresponding *declarations* in theories below. Contexts (left) and substitutions (right) are translated by concatenating the translations of their components:

PDHOL	Translation	PDHOL	Translation
$\alpha : \texttt{type}$	$\alpha : \texttt{type},\ \alpha^* : \alpha \to \alpha \to o,\ \triangleright\, \mathrm{PER}(\alpha^*)$	A	$\overline{A},\ A^*,\ \checkmark$
$x : A$	$x : \overline{A},\ \triangleright\, A^*\ x\ x$	t	$\overline{t},\ \checkmark$
$\triangleright\, \Phi$	$\triangleright\, \overline{\Phi}$	$\checkmark$	$\checkmark$

Note how substitutions for Δ are translated to substitutions for $\overline{\Delta}$: for example, just like every type variable produces 3 declarations, every type substitute produces 3 corresponding substitutes. In particular, each $\checkmark$ in the translation of a substitution represents the invariant that the respective formula is in fact provable in the translated context: for example, for every PDHOL-type A, PHOL can prove $\mathrm{PER}(A^*)$, and for every PDHOL-term $t : A$, PHOL can prove $A^*\ t\ t$. These properties are shown in Thm. 9.

The definition of the PER follows the principles of logical relations:

PDHOL type A	$A^*\ t\ u$ in PHOL
$a\ \delta$	$a^*\ \overline{\delta}^{ye}\ \ t\ u$
α	$\alpha^*\ t\ \ u$
o	$t =_o u$
$\Pi_{x:A}B$	$\forall x, y : \overline{A}. A^*\ x\ y \Rightarrow B^*\ (t\ x)\ (u\ y)$

Here a^* and α^* are names introduced during the translation of theories and contexts. These declare the respective PER axiomatically for type symbols and type variables.

▶ **Example 7.** We use the expression E from Ex. 2 to create E_2, a list `[E, E]` of lists by $E_2 := \texttt{cons}\ (\texttt{vec nat}\ 2)\ 1\ E\ (\texttt{cons}\ (\texttt{vec nat}\ 2)\ 0\ E\ (\texttt{nil}\ (\texttt{vec nat}\ 2))) : \texttt{vec}\ (\texttt{vec nat}\ 2)\ 2$. Its translation $\overline{E_2}$ yields $\texttt{cons}\ (\texttt{vec nat})\ 1\ \overline{E}\ (\texttt{cons}\ (\texttt{vec nat})\ 0\ \overline{E}\ (\texttt{nil}\ (\texttt{vec nat})))$.

Here δ is $(\texttt{vec nat}\ 2)\ 1\ E\ (\ldots)$. Observe that the erasure recurses through the argument list, i.e., $\overline{(\texttt{vec nat}\ 2)}\ \overline{1}\ \overline{E}\ \overline{(\ldots)}$. The type argument `vec nat 2` is translated into `vec nat` removing the term argument to the type as well as the generated PER $(\texttt{vec nat}\ 2)^*$ in accordance with our definition of δ^{y}. The remaining arguments are terms that do not take term arguments, meaning that the translation does not change them.

The type of the expression $\texttt{vec}\ (\texttt{vec nat}\ 2)\ 2$ is correspondingly erased to $\texttt{vec}\ (\texttt{vec nat})$.

Finally, the cases for declarations in theories are more complex because they contain contexts. This is where the abbreviations from Def. 6 come in:

PDHOL		Translation in PHOL
$a : \Pi_\Delta : \texttt{type}$	where $\Delta = \vec{\alpha} : \texttt{type},\ \vec{x} : \vec{A}$	$a : \overline{\Delta}^{y} \to \texttt{type}$ $a^* : \overline{\Delta}^{ye} \to a\ \vec{\alpha} \to a\ \vec{\alpha} \to o$ $\rhd\ \overline{\Delta} \Rightarrow \mathrm{PER}(a^*\ \vec{\alpha}\ \vec{\alpha^*}\ \vec{x})$
$c : \Pi_\Delta A$	where $\Delta = \vec{\alpha} : \texttt{type}$	$c : \overline{\Delta}^{ye} \to \overline{A}$ $\rhd\ \overline{\Delta} \Rightarrow A^*\ (c\ \vec{\alpha})\ (c\ \vec{\alpha})$
$\rhd\ \Pi_\Delta \Phi$	where $\Delta = \vec{\alpha} : \texttt{type}$	$\rhd\ \overline{\Delta} \Rightarrow \overline{\Phi}$

▶ **Example 8.** Translating our running example's theory yields $\texttt{vec} : \texttt{type} \to \texttt{type}$ for the vector declaration. Note that while Δ includes type and term variables, the first part of the erasure only considers Δ^y, doing away with the term variables, therefore $\overline{\Delta}^{y} \to \texttt{type}$ is the kind that takes an equal number of type variables as arguments and returns a type.

The second part of the erasure of vector yields $\texttt{vec}^* : \Pi_\alpha(\alpha \to \alpha \to o) \to \texttt{nat} \to (\texttt{vec}\ \alpha) \to (\texttt{vec}\ \alpha) \to o$. Δ^{ye} includes, additionally to the type variables from the previous point, the term variables. This includes the PER generated by the erasure of the type variables as well as the term argument $\texttt{nat}$.

Finally, we get the axiom $\rhd\ \Pi_\alpha \forall \alpha^* : \alpha \to \alpha \to o. \forall n : \texttt{nat}. PER(\alpha^*) \Rightarrow PER(\texttt{vec}^*\ \alpha\ \alpha^*\ n)$. Compared to the last two results of the erasure, we now have additionally the assertion that the type argument comes equipped with a PER. As this is now a validity statement (as opposed to a typing statement) term arguments are now bound by $\forall$ and $\Rightarrow$.

Readers might note the absence of the according statements for the type $\texttt{nat}$. In the case of non-dependent base types, the PER collapses to standard equality, resulting in trivial axioms, which we omit.

Our translation subsumes that of [26]: If we specialize to DHOL, contexts must not contain type variables and the corresponding cases in the translation of contexts and substitutions and the definition of the PER can be dropped. The arguments Δ of a type symbol declaration a in a theory contain only type variables, and references $a\,\delta$ take only type arguments, so that (i) $\overline{\Delta}^{y}$ and $\overline{\delta}^{y}$ are empty (ii) $\overline{\Delta}^{ye}$ contains only the declarations $x : \overline{A}$ for every type A in Δ (iii) $\overline{\delta}^{ye}$ contains only the terms $\overline{t}$ for every term t in δ Finally, the argument list δ of a term symbol c is empty and thus so is $\overline{\delta}^{ye}$.

5 Properties of the Translation

▶ **Theorem 9** (Preservation of Judgments and Substitution)**.** *Every PDHOL judgment in the table below implies the corresponding PHOL judgment about the translated syntax:*

PDHOL	*PHOL*
$\vdash T\ \texttt{Thy}$	$\vdash \overline{T}\ \texttt{Thy}$
$\vdash_T \Gamma\ \texttt{Ctx}$	$\vdash_{\overline{T}} \overline{\Gamma}\ \texttt{Ctx}$
$\Gamma \vdash_T \Delta\ \leftarrow\ \delta$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{\Delta}\ \leftarrow\ \overline{\delta}$
$\Gamma \vdash_T A\ :\ \texttt{type}$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{A}\ :\ \texttt{type}$ *and* $\overline{\Gamma} \vdash_{\overline{T}} A^*\ :\ \overline{A} \to \overline{A} \to o$ *and* $\overline{\Gamma} \vdash_{\overline{T}} \mathrm{PER}(A^*)$
$\Gamma \vdash_T t\ :\ A$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{t}\ :\ \overline{A}$ *and* $\overline{\Gamma} \vdash_{\overline{T}} A^*\ \overline{t}\ \overline{t}$
$\Gamma \vdash_T F$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{F}$
$\Gamma \vdash_T A \equiv B$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{A} \equiv \overline{B}$ *and* $\overline{\Gamma} \vdash_{\overline{T}} A^* =_{\overline{A} \to \overline{A} \to o} B^*$

Moreover, whenever $\Gamma \vdash_T \Delta\ \leftarrow\ \delta$*, we have*

PDHOL	*PHOL*
$\Gamma, \Delta \vdash_T A\ :\ \texttt{type}$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{A[\delta]} \equiv \overline{A}[\overline{\delta}]$
$\Gamma, \Delta \vdash_T t\ :\ A$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{t[\delta]} =_{\overline{A[\delta]}} \overline{t}[\overline{\delta}]$

The proof of Theorem 9 is straightforward and performed by induction over derivations. An illustrative example is given in Appendix A.

▶ **Theorem 10** (Reflection of Truth). *Assume a well-formed PDHOL theory* $\vdash T$ Thy.

$$\textit{If } \Gamma \vdash_T F : o \textit{ and } \overline{\Gamma} \vdash_{\overline{T}} \overline{F} \textit{ then } \Gamma \vdash_T F.$$

In particular, if $\Gamma \vdash_T s : A$ *and* $\Gamma \vdash_T t : A$ *and* $\overline{\Gamma} \vdash_{\overline{T}} A^* \ \overline{s}\ \overline{t}$, *then* $\Gamma \vdash s =_A t$.

The assumption about the well-typedness of the statement is necessary: Consider two PDHOL-terms $u := \lambda x : A\ s.x$ and $v := \lambda x : A\ t.x$ of some dependent type a and different terms s, t. In (P)DHOL an equality $u =_{\Pi x:A\ s.A\ s} t$ between them would be ill-typed. The erased equality however is not only well-typed but provable.

The original proof for DHOL [26] is rather involved. Luckily it is easy to extend it to PDHOL. Intuitively, a proof of a PHOL statement $\overline{F}$ is translated into a proof that exists in the image of the translation. This allows us to subsequently read off a PDHOL proof of the untranslated conjecture F. An overview of the original proof together with the necessary adaptations is given in Appendix B. In the remainder, we will call the combination of these properties *well-behavedness* of the translation.

6 Case Studies

To evaluate and leverage PDHOL, we implemented the translation as a part of the logic-embedding tool by Steen [27]. Our implementation builds on the one for monomorphic DHOL [26], and more details regarding the TPTP dialect and type checking can be found in [22]. As an optimization, we replace PERs in our translation with equality whenever there is no dependence on the term arguments. From an informal comparison with previous data, this seems to yield a speedup of about 5% on the presented problems.

We created a set of 52 problems to evaluate and test our implementation on. 36 problems are polymorphic extensions of a subset of the problem files created by Niederhauser et al. [16] to test their implementation of DHOL in the tableaux prover Lash. Of these, 17 are variations where type variables have been instantiated in the conjecture. 11 further problems describe that the reversal function on red-black trees is an involution. These will be the focus of this section, while we give a summary of the other results. The remaining problems focus on the formalization capabilities of PDHOL. The translated problems were evaluated using Vampire 4.7 using its portfolio mode, with a timeout of 120 seconds. All experiments ran on a Intel Core i5-6200U CPU at 2.30GHz and 8 GB of RAM. All files and binaries are online.[1]

Fixed-Length Lists We did a deep formalization of lists following the examples in Section 2 and 4. We also formalized finite types `fin` : `nat` → `type`, and used an accessor `elemAt` : $\Pi_\alpha.\Pi_{n:\texttt{nat}}.\texttt{vec}\,\alpha\,n \rightarrow \texttt{fin}\,n \rightarrow \alpha$ for safe access to list elements. The conjectures we investigated expressed properties of the append function — associativity and the identity of nil — as well as the involution property of the reverse function. These problems generated more complex type checking obligations than the red-black trees, due to the arithmetic needed to show that lengths of lists line up after appending. Matrices can then be represented by nesting vectors. Then we formalize, e.g., matrix transposition as `transpose` : $\Pi_\alpha.\Pi_{m,n:\texttt{nat}}.\texttt{vec}\,(\texttt{vec}\,\alpha\,m)\,n \rightarrow \texttt{vec}\,(\texttt{vec}\,\alpha\,n)\,m$.

[1] https://drive.google.com/drive/folders/14H9nH-voaF35X69y-IROLiCMtJrakUvd

This leverages both dependent types and polymorphism to concisely represent complex data structures in a way that guarantees their dimensional invariants. E.g., Haskell's default List package[2] includes distinct functions `zipN` for $N \in \{3, \dots, 7\}$ arguments due to the lack of dependent types. Note that an arbitrary dimensional zip function acts like matrix transposition on lists of lists. No extra cost is incurred: For all of the above, the induced type checking obligations (TCOs) are discharged easily by PHOL ATPs.

However, for many advanced conjectures, e.g., showing that transposing matrices is an involution, the proofs timed out.

We believe this is because the necessary induction arguments are too difficult for current ATPs (independent of how the data structures are formalized). This is in line with the results reported by Niederhauser et al. [16], indicating that performance improvements can be obtained by building DHOL-specific provers, or splitting problems in a way that helps the ATP system with the induction arguments.

Red-Black Trees Red-black trees are binary search trees for which self-balancing is achieved by coloring nodes red or black. Leaves are always black. In addition, two constraints are maintained: *The child of a red not must not be red* and *Every path from a node to its leaves has the same number of black nodes.* Such invariants are difficult to capture by non-dependent inductive types. However, in PDHOL, we can use a declaration rbtree : $\Pi_\alpha \Pi_{b:o} \Pi_{n:\texttt{nat}}$: `type` where rbtree $A\ b\ n$ is the type of red-black trees holding values of type A containing n black nodes. This allows capturing the invariant directly in the type. The formalized conjecture states that reversing is an involution and is presented in Appendix C.

The standard proof of this fact involves inductive definitions that are difficult for ATP systems to deal with. In line with previous work in DHOL [16, 21] we split the problem into smaller sub-problems, ensuring well-typedness.

The problem is split into a base case for red (`Base-Red`) and black (`Base-Black`) leaves. Additionally, there is a problem file asserting that, if a property holds for red leaves and black leaves, it holds for all red-black trees of black-height 1 (`Base-Lemma`). This lemma is used to prove the base case for any tree of black-height 1 (`Base`).

Next are the step cases: Again there is a step case for red (`Step-Red`) and for black (`Step-Black`) leaves. While the red step case is easy, the black is challenging due to the raised complexity of black nodes. While possible to prove directly, we also added three sub-steps (`Step-subBlack1-3`) to help the ATP system along.

Problem	Time to prove (incl. type check)
Base-Black	<1s
Base-Red	9s
Base-Lemma	18s
Base	82s
Step-subBlack1	<1s
Step-subBlack2	<1s
Step-subBlack3	<1s
Step-Black	11s
Step-Black (direct)	66s
Step-Red	14s
Instanced Induction	timeout
Combined	14s
Bad Tree	timeout

Figure 4 Experimental results.

While the instanced induction scheme (`Instanced Induction`) timed out, type checking was successful. Assuming the instanced induction allows to proof the main conjecture (`Combined`) in just 14s.

Finally, we specified one problem not related to the reversal function (`Bad Tree`). It serves as a sanity check for the formulation of red-black trees: a conjecture trying to create

[2] hackage.haskell.org/package/base-4.21.0.0/docs/Data-List.html

an ill-formed red-black tree. It postulates that for every red tree, there are two children of arbitrary color. This violates the invariant that a red node must not have red children. Indeed, neither the conjecture itself nor the TCOs trying to establish that black is any color can be proven.

Notably, the red-black tree examples generate very few type checking obligations (3 in total) compared to the list examples, where the majority of examples generated 1+ type checking obligations. This is due to the fact that most constraints, thanks to the efficient formulation of the problems provided by the dependent types, reduce to reflexivity. See Figure 4 for a summary.

7 Generalizations

Subtype Definitions Large HOL theories are usually built by chaining conservative extension principles. Besides direct definitions, the most important such principle used in HOL-based ITPs is definitional subtypes [11]. For example, the theory-level declaration $a := A|p$ for some predicate $p : A \to o$ conservatively extends the containing theory with declarations for a fresh (undefined) type a and fresh functions $Abs : A \to a$ and $Rep : a \to A$ that are axiomatized to be bijections between a and the set of elements of A that satisfy p. This extension principle is expressive enough to define, e.g., record and inductive types. But it becomes particularly powerful in the presence of polymorphism, where it can introduce new type *operators* a. For example, simple product types can be defined using $a\,\alpha\,\beta := (\alpha \to \beta \to o)|p$ where p is the property that f is true for exactly one argument pair.

Generalizing this extension principle to dependent types further increases its expressivity and enables PDHOL to make complex type definitions. We extend our language as follows:

$$T \quad ::= \quad T,\, a := \lambda_\Delta.A|p \quad \text{for some} \quad \Delta = \vec{\alpha} \;:\; \mathtt{type},\, \vec{x} : \vec{A}$$

with a typing rule requiring $\Delta \vdash_T p \;:\; A \to o$ and a proof obligation that we discuss below. Abbreviating $\delta := \vec{\alpha}\,\vec{x}$, its semantics is defined by elaboration into

$$a : \Pi_\Delta.\mathtt{type} \qquad Abs : \Pi_\Delta.A \to a\,\delta \qquad Rep : \Pi_\Delta.a\,\delta \to A$$

$$\begin{aligned}\rhd\, \Pi_{\vec{\alpha}}\forall\vec{x} : \vec{A}.\big(&\forall u : a\,\delta.\ p\,(Rep\,\delta\,u) \wedge Abs\,\delta\,(Rep\,\delta\,u) =_{a\,\delta} u\big)\\ &\wedge \big(\forall v : A.\ p\,v \Rightarrow Rep\,\delta\,(Abs\,\delta\,v) =_A v\big)\end{aligned}$$

In the same way as for HOL subtype definitions, in the second part of the axiom, the predicate p occurs crucially to require $Rep\,\delta\,(Abs\,\delta\,v) =_A v$ only for those v that are in the intended subtype, capturing that we think of Abs as a partial function. Because all functions are total, $Abs\,\delta\,v$ is also well-typed if $\neg(p\,v)$ but remains unspecified. That is conservative if the new type a is non-empty, which is why HOL additionally requires the proof obligation $\exists v : A.p\,v$. While this is natural for HOL (where all types are assumed to be non-empty anyway), this is not ideal for DHOL, where empty types are useful and allowed. Nonetheless, we adopt the same proof obligation here for simplicity, i.e., a subtype definition is well-typed only if $\Delta \vdash_T \exists v : A.p\,v$. Our translation is in fact judgment preserving and truth reflecting for weaker conditions, but we leave the details to future work.

We extend our translation as follows: every subtype definition $a := \lambda_\Delta.A|p$ is translated to the corresponding PHOL subtype definition $a := \lambda_{\vec{\alpha}}\overline{A}|(\lambda_{u:\overline{A}}.A^*\ u\ u \wedge p\,u)$. This translation commutes with elaboration: we obtain isomorphic PHOL theories if we (i) elaborate a PDHOL subtype definition and then translate it to PHOL, or (ii) translate it to a PHOL definition and then apply the usual PHOL elaboration. Consequently, the translation remains well-behaved.

▶ **Example 11** (Algebraic Structures)**.** Let us assume we have already formalized types for algebraic structures, such as a type `group` : $\Pi_\alpha.\mathtt{type}$ with (among others) a selector `op` : $\Pi_\alpha.\alpha \to \alpha \to \alpha$. (In principle, the type $\mathtt{group}\,\alpha$ could itself be defined as a subtype of $\alpha \to \alpha \to \alpha$. But that would require a more complex analysis of conservativity because there is no group on the empty type.)

We can now use polymorphism to abstract over the carrier set and then use dependent types to formalize various constructions from universal algebra. For example, we can define the predicate `ishom` : $\Pi_{\alpha,\beta}.\,\mathtt{group}\,\alpha \to \mathtt{group}\,\beta \to (\alpha \to \beta) \to o$ by

$$\mathtt{ishom}\,\alpha\,\beta\,G\,H\,m =_o \forall x, y : \alpha.m\,(\mathtt{op}\,G\,x\,y) =_\beta \mathtt{op}\,H\,(m\,x)\,(m\,y)$$

and use that to define the polymorphic and dependent type of group homomorphisms by

$$\mathtt{hom} := \lambda_{\alpha\,:\,\mathtt{type},\,\beta\,:\,\mathtt{type},\,G:\mathtt{group}\,\alpha,\,H:\mathtt{group}\,\beta}(\alpha \to \beta)|\mathtt{ishom}\,\alpha\,\beta\,G\,H$$

Similar examples are the types of subgroups of a group, conjugacy classes of a group, or actions of a group on a set, and accordingly for all other algebraic theories.

As an example theorem, we state that homomorphisms preserve the neutral element:

$$\Pi_{\alpha,\beta}.\,\forall G : \mathtt{group}\,\alpha,\ H : \mathtt{group}\,\beta,\ m : \mathtt{hom}\,\alpha\,\beta\,G\,H.\,\forall e : \alpha.\mathtt{neutral}\,\alpha\,G\,e \Rightarrow \mathtt{neutral}\,\beta\,H\,(\mathrm{Rep}_{\mathtt{hom}}\,m\,e)$$

for an appropriately defined predicate `neutral`. Of course, in practical type checkers, we should infer omitted arguments and insert the $\mathrm{Rep}_{\mathtt{hom}}$ function so that this becomes:

$$\Pi_{\alpha,\beta}.\,\forall G : \mathtt{group}\,\alpha,\ H : \mathtt{group}\,\beta,\ m : \mathtt{hom}\,G\,H.\,\forall e.\,\mathtt{neutral}\,G\,e \Rightarrow \mathtt{neutral}\,H\,(m\,e)$$

A formalization of this example is available with the problem files discussed in Section 6.

Dependent Type Variables In Sect. 2 we discussed the various design options of how a declaration in a *theory* may depend on variables/assumptions. We can similarly analyze declarations in *contexts*.

	declaration of		
depending on	type variable	term variable	local assumption
type variable	not allowed due to size issues		
term variable	this section	definable via Π	definable via $\forall$
local assumption	not allowed		definable via $\Rightarrow$

Just like for theories, several combinations are problematic or definable. In particular, contrary to theories, contexts must be small i.e. declarations must not depend on type variables. This ensures that we can formulate the semantics without requiring a hierarchy of universes or similar. We have so far omitted the desirable feature of type variables depending on term variables as in $\alpha : \mathtt{type},\ \beta : \alpha \to \mathtt{type}$.

▶ **Example 12.** We give heterogeneous lists as a variant of vectors where each component may have a different type. Consequently, all operations must take a dependent type variable $\alpha : \mathtt{fin}\,n \to \mathtt{type}$ that provides the types of the n components. This uses finite types $\mathtt{fin}\,n = \{0, \dots, n-1\}$, which we can declare by

$$\mathtt{fin} : \mathtt{nat} \to \mathtt{type} \qquad \mathtt{fnext} : \Pi_{n:\mathtt{nat}}\mathtt{fin}\,n \to \mathtt{fin}(\mathtt{suc}\ n) \qquad \mathtt{ftop} : \Pi_{n:\mathtt{nat}}\mathtt{fin}(\mathtt{suc}\ n)$$

Then heterogeneous lists can be declared as

$$\begin{array}{l}
\texttt{Hlist} : \Pi_{n:\texttt{nat}}(\texttt{fin}n \to \texttt{type}) \to \texttt{type} \\
\texttt{hlist} : \Pi_{n:\texttt{nat}}\Pi_{L:\texttt{fin}n\to\texttt{type}}(\texttt{fin}n \to L\ n) \to \texttt{Hlist}\ n\ L \\
\texttt{hget} : \Pi_{n:\texttt{nat}}\Pi_{L:\texttt{fin}n\to\texttt{type}}\texttt{Hlist}\ n\ L \to \Pi_{i:\texttt{fin}n}L\ i
\end{array}$$

Note how this requires allowing type and term variables to alternate as parameters of type variables may depend on previous term parameters.

This feature is theoretically non-problematic but requires a more complex presentation. We change the grammar as follows:

$$\begin{array}{|lcl|lcl|}
\hline
T & ::= & .\ \mid T, a : \Pi_{\Gamma} : \texttt{type} \mid T, c : \Pi_{\Gamma} A \mid T, \rhd \Pi_{\Gamma} F & & & \\
K & ::= & \texttt{type} \mid \Pi_{x:A} K & k & ::= & A \mid \lambda_{x:A} k \\
\Gamma & ::= & \circ \mid \Gamma, \alpha : K \mid \Gamma, x : A \mid \Gamma, \rhd F & \gamma & ::= & \bullet \mid \gamma, k \mid \gamma, t \mid \gamma, \checkmark \\
A, B & ::= & a\,\gamma \mid \alpha\, t\ \ldots\ t \mid \Pi_{x:A} B \mid o & t, u & ::= & c\,\gamma \mid x \mid \lambda_{x:A} t \mid t\, u \mid t \Rightarrow u \mid t =_A u \\
\hline
\end{array}$$

Here k produces types with uninstantiated term dependencies, and K produces their kinds. The grammar used so far arises as the special case $K = \texttt{type}$ and $k = A$. Their typing is given by the rules

$$\frac{\Gamma, \Delta \vdash_T B\ :\ \texttt{type}}{\Gamma \vdash_T \lambda_{\Delta} B\ :\ \Pi_{\Delta} : \texttt{type}} \qquad \frac{\alpha : \Pi_{\Delta} : \texttt{type} \in \Gamma \quad \Gamma \vdash_T \Delta\ \leftarrow\ \delta}{\Gamma \vdash_T \alpha\ \delta\ :\ \texttt{type}} \quad \text{where } \Delta = \vec{x} : \vec{A}$$

Adjusting Rule CtxTp and Rule SubTp accordingly is straightforward.

Additionally, the grammar above deviates from the one given before by allowing all theory declarations to depend on arbitrary contexts rather than regulating which kind of declarations (type variable, term variable, or local assumption) are allowed as parameters of which declaration. This is necessitated by the new dependent type variables: if type variables can depend on terms, term variable bindings $x : A$ in constant declarations or axioms are no longer definable because they might have to be bound *before* the type variables whose argument types depend on them. Thus, term variable parameters must be allowed explicitly and must be allowed to alternate with type variable parameters.

Our grammar also allows local assumptions $\rhd F$ as parameters of theory declarations. This is not necessary for dependent type variables and is an optional feature in the language design. We allow it as it simplifies the notations. If desired, it is easy to focus on the language fragment where the contexts in theory declarations do not introduce assumptions. The typing rules for theories remain unchanged except that we need to remove the syntactic restrictions on the context Δ in Rule ThyType, Rule ThyCon, and Rule ThyAss.

We adjust three cases in the translation:

DHOL		Translation
type variable declaration	$\alpha : \Pi_{\Delta} : \texttt{type}$ where $\Delta = \vec{x} : \vec{A}$	$\alpha\ :\ \texttt{type}$ $\alpha^* : \overline{\Delta}^{ye} \to \alpha \to \alpha \to o,$ $\rhd \overline{\Delta} \Rightarrow \text{PER}(\alpha^*\ \vec{x})$
type variable substitute	$\lambda_{\Delta} A$ where $\Delta = \vec{x} : \vec{A}$	$\overline{A},\ \lambda_{\overline{\Delta}^{ye}} A^*,\ \checkmark$
type variable reference	$\alpha\ \delta$	α

The translation rules for declarations in a theory remain unchanged except for generalizing the syntactic restrictions on the contexts and substitutions. The invariants are the same.

8 Conclusion and Future Work

Summary We extended Dependently-typed Higher-order Logic with polymorphism and subtype definitions. A translation to polymorphic HOL yields an efficient automated reasoning procedure for the calculus. We demonstrated the practical usability of the language and its automation by encoding and automatically proving several practical problems that combine polymorphism with dependent types and cannot be represented as concisely in either polymorphic HOL or monomorphic DHOL.

Related Work While there are some practical examples that make deep polymorphism desirable, they tend to interact poorly with ATP systems. Indeed, most logics that support deep polymorphism introduce a hierarchy of universes as in Martin-Löf type theory [15], which goes far beyond the expressivity of current ATP tools. On the other hand, shallow polymorphism still allows standard set-theoretic semantics without any size issues. That makes it the variant of choice in HOL theorem provers — both interactive [10, 17, 12] and automated [28, 23, 30]. Indeed, our definition has the key advantage that DHOL polymorphism can be directly translated to HOL polymorphism, ensuring that proof obligations remain efficiently solvable for the ATP system.

PVS provides a combination of dependent types and shallow polymorphism similar to our design. It combines native decision procedures with interaction and automation but it is too expressive for easy translation into standard ATP tools. Recently the Vampire automated theorem prover has been extended with shallow polymorphism [4]. The extension is very elegant, but it focuses on non-dependent types so far.

CoqHammer [7] tackles a similar problem but sits on a different end of the expressivity and automation spectrum: It translates Rocq into first-order logic. It is, however, incomplete.

The general idea of translating dependent type theories is not new: Felty and Miller [9] translated LF into hereditary Harrop formulas, a simply-typed meta-logic. The work of Jacobs and Melham [14], which introduced the idea of translating dependent types into predicates which serve as type guards, is an ancestor of our translation.

Our use of PERs follows established practice. The ultimate reason for the prevalence of PERs in type theory is that refinement types are not closed under function type formation, i.e., the function type on refinement types cannot be represented as a refinement of a function type. A similar argument applies to quotients. But PERs, which subsume refinements and quotients, are closed under function type formation. Thus, many constructions that involve refinements or quotients of base types are eventually generalized to PERs in order to complete inductive definitions or proofs. Because dependent base types can be understood as refinements of a bigger type, the semantics of dependent types is one such construction. PERs have been used to formulate the semantics of dependent types, in essentially the same way as we do, going back to at least [1] and were used for the semantics of NuPRL [20]. Another example are parametricity arguments in polymorphic type theories such as [3]. For example, recently, [19] presented a parametricity translation from HOL to itself that interprets every type as a PER. That translation arises as the special case of ours where the input is restricted to the HOL fragment of DHOL. Because, their source and target language are the same, they additionally study which HOL-style subtype definitions can be translated to themselves, leading them to introduce the notion of wideness.

Future Work Even though our automation is well-behaved, it is still not as strong as desirable. To improve this, we plan to define dedicated automated reasoning rules for PDHOL similar to Niederhauser et al. [16] for DHOL. Furthermore, a larger case study involving dependently typed examples from ITPs would allow checking if PDHOL constitutes a useful intermediate language for proof automation.

Finally, we want to combine our work with the extension of DHOL with subtyping from [24]. While we expect simply merging the language extensions to be straightforward, their union allows adding *bounded* polymorphism where type variables $\alpha <: A$ can only be instantiated with subtypes of A. Intuitively, the type system and translation can be extended easily to allow for such upper bounds on type variables, but it is unclear how difficult the judgment preservation and truth reflection proofs and the design of a (sub)type-checking algorithm will be in that case.

References

1 S. Allen. *A Non-Type-Theoretic Semantics for Type-Theoretic Language*. PhD thesis, Cornell University, 1987.

2 P. Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.

3 Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Parametricity and dependent types. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 345–356. ACM, 2010. doi:10.1145/1863543.1863592.

4 A. Bhayat and G. Reger. A polymorphic vampire - (short paper). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 361–368. Springer, 2020. doi:10.1007/978-3-030-51054-1_21.

5 A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5(1):56–68, 1940. doi:10.2307/2266170.

6 Coq Development Team. The Coq Proof Assistant: Reference Manual. Technical report, INRIA, 2015.

7 Lukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *J. Autom. Reason.*, 61(1-4):423–453, 2018. URL: https://doi.org/10.1007/s10817-018-9458-4, doi:10.1007/S10817-018-9458-4.

8 L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean Theorem Prover (System Description). In A. Felty and A. Middeldorp, editors, *Automated Deduction*, pages 378–388. Springer, 2015.

9 Amy P. Felty and Dale Miller. Encoding a dependent-type lambda-calculus in a logic programming language. In Mark E. Stickel, editor, *10th International Conference on Automated Deduction, Kaiserslautern, FRG, July 24-27, 1990, Proceedings*, volume 449 of *Lecture Notes in Computer Science*, pages 221–235. Springer, 1990. doi:10.1007/3-540-52885-7_90.

10 M. Gordon. HOL: A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, pages 73–128. Kluwer-Academic Publishers, 1988.

11 M. Gordon and A. Pitts. The HOL Logic. In M. Gordon and T. Melham, editors, *Introduction to HOL, Part III*, pages 191–232. Cambridge University Press, 1993.

12 J. Harrison. HOL Light: A Tutorial Introduction. In *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, pages 265–269. Springer, 1996.

13 M. Hofmann. *Extensional constructs in intensional type theory*. CPHC/BCS distinguished dissertations. Springer, 1997.

14 B. Jacobs and T. Melham. Translating dependent type theory into higher order logic. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, pages 209–29, 1993.

15 P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Proceedings of the '73 Logic Colloquium*, pages 73–118. North-Holland, 1974.

16 J. Niederhauser, C. E. Brown, and C. Kaliszyk. Tableaux for automated reasoning in dependently-typed higher-order logic. In Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I*, volume 14739 of *Lecture Notes in Computer Science*, pages 86–104. Springer, 2024. doi:10.1007/978-3-031-63498-7_6.

17 T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

18 U. Norell. The Agda WiKi, 2005. http://wiki.portal.chalmers.se/agda.

19 Andrei Popescu and Dmitriy Traytel. Admissible types-to-pers relativization in higher-order logic. In A. Ahmed, R. Findler, D. Garg, and F. Pottier, editors, *Principles of Programming Languages*, pages 1214–1245. ACM, 2023.

20 Vincent Rahli, Mark Bickford, and Abhishek Anand. Formal program optimization in nuprl using computational equivalence and partial types. In Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie, editors, *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, volume 7998 of *Lecture Notes in Computer Science*, pages 261–278. Springer, 2013. doi:10.1007/978-3-642-39634-2_20.

21 D. Ranalter, C. E. Brown, and C. Kaliszyk. Experiments with choice in dependently-typed higher-order logic. In Nikolaj S. Bjørner, Marijn Heule, and Andrei Voronkov, editors, *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024*, volume 100 of *EPiC Series in Computing*, pages 311–320. EasyChair, 2024. URL: https://doi.org/10.29007/2v8h, doi:10.29007/2V8H.

22 Daniel Ranalter, Cezary Kaliszyk, Florian Rabe, and Geoff Sutcliffe. The dependently typed higher-order form for the tptp world. 2025. preprint; to appear in FROCOS 25. URL: https://arxiv.org/abs/2507.03208, arXiv:2507.03208.

23 A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15:91–110, 2002.

24 C. Rothgang and F. Rabe. Subtyping in DHOL – extended preprint, 2025. preprint; to appear in FROCOS 25. URL: https://arxiv.org/abs/2507.02855, arXiv:2507.02855.

25 C. Rothgang, F. Rabe, and C. Benzmüller. Theorem Proving in Dependently Typed Higher-Order Logic. In B. Pientka and C. Tinelli, editors, *Automated Dedution*, pages 438–455. Springer, 2023.

26 C. Rothgang, F. Rabe, and C. Benzmüller. Theorem proving in dependently-typed higher-order logic, 2024. journal version, under review. doi:10.48550/arXiv.2305.15382.

27 A. Steen. An extensible logic embedding tool for lightweight non-classical reasoning (short paper). In B. Konev, C. Schon, and A. Steen, editors, *Practical Aspects of Automated Reasoning*, volume 3201 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

28 A. Steen, M. Wisniewski, and C. Benzmüller. Going polymorphic - th1 reasoning for leo-iii. In Thomas Eiter, David Sands, Geoff Sutcliffe, and Andrei Voronkov, editors, *IWIL Workshop and LPAR Short Presentations*, volume 1 of *Kalpa Publications in Computing*, pages 100–112. EasyChair, 2017. doi:10.29007/jgkw.

29 G. Sutcliffe. Stepping stones in the tptp world. In C. Benzmüller, M. Heule, and R. Schmidt, editors, *Proceedings of the 12th International Joint Conference on Automated Reasoning*, number 14739 in Lecture Notes in Artificial Intelligence, pages 30–50, 2024.

30 P. Vukmirovic, A. Bentkamp, J. Blanchette, S. Cruanes, V. Nummelin, and S. Tourret. Making higher-order superposition work. *J. Autom. Reason.*, 66(4):541–564, 2022. URL: https://doi.org/10.1007/s10817-021-09613-z, doi:10.1007/S10817-021-09613-Z.

A Preservation

Proof. Thoerem 9 is proved by straightforward induction on derivations. The sub-proofs for all proof steps follow the same structure. We therefore present here only one example (the Rule TpSym rule) applying the induction hypothesis followed by definitions of the erasure in full detail:

$$
\begin{array}{lll}
\vdash_{\overline{T}} \overline{\Gamma}\ \texttt{Ctx} & \textit{assumption} & (1)\\
\overline{a : \Pi_\Delta : \texttt{type}} \ \in\ \overline{T} & \textit{assumption} & (2)\\
a : \overline{\Delta}^{y} \to \texttt{type} \ \in\ \overline{T} & \overline{\ }\text{-def of DHOL Theories on } 2 & (3)\\
a^* : \overline{\Delta}^{ye} \to a\ \vec{\alpha} \to a\ \vec{\alpha} \to o \in \overline{T} & \overline{\ }\text{-def of DHOL Theories on } 2 & (4)\\
\triangleright\ \overline{\Delta} \Rightarrow PER(a^*\ \vec{\alpha}\ \vec{\alpha}^*\ \vec{x}) \in \overline{T} & \overline{\ }\text{-def of DHOL Theories on } 2 & (5)\\
\overline{\Gamma} \vdash_{\overline{T}} \overline{\Delta} \leftarrow \overline{\delta} & \textit{assumption} + \textit{IH} & (6)\\
\overline{\Gamma} \vdash_{\overline{T}} \overline{\Delta}^{y} \leftarrow \overline{\delta}^{y} & \text{see Note in Def. } 6 & (7)\\
\overline{\Gamma} \vdash_{\overline{T}} \overline{\Delta}^{ye} \leftarrow \overline{\delta}^{ye} & \text{see Note in Def. } 6 & (8)\\
\overline{\Gamma} \vdash_{\overline{T}} a\ \overline{\delta}^{y} : \texttt{type} & \text{Rule TpSym},\ 1,\ 3,\ 8 & (9)\\
\overline{\Gamma} \vdash_{\overline{T}} \overline{a\ \delta} : \texttt{type} & \overline{\ }\text{-def of DHOL Types on } 9 & (10)\\
\overline{\Gamma} \vdash_{\overline{T}} a^*\ \overline{\delta}^{ye} : a\ \overline{\delta}^{y} \to a\ \overline{\delta}^{y} \to o & \text{Rule TermSym},\ 1,\ 4,\ 7 & (11)\\
\overline{\Gamma} \vdash_{\overline{T}} \overline{(a\ \delta)^*} : \overline{a\ \delta} \to \overline{a\ \delta} \to o & {}^{*}\text{-def and } \overline{\ }\text{-def of DHOL types} & (12)\\
\overline{\Gamma} \vdash_{\overline{T}} PER(a^*\ \overline{\delta}^{ye}) & \text{Rule ValidSym},\ 1,\ 5,\ 7 & (13)\\
\overline{\Gamma} \vdash_{\overline{T}} PER(\overline{(a\ \delta)^*}) & {}^{*}\text{-def},\ 13 & (14)
\end{array}
$$

A remark about steps 7 and 8: Inspection of the inference rules for substitutions shows, that the remark in Def. 6 extends to substitutions by just traversing the list and throwing out the corresponding elements of δ.

The same proof structure is directly applicable to the other rules, only note that Rule TpSym, Rule TermSym and Rule ValidSym all proceed by substituting the Δ in the premises by the δ of the substitution. ◀

B Reflection

The proof is taken from Rothgang et al. [26] and extended to allow for the changes we performed on the grammar and inference system. Large parts of the proof remain identical and we will point out where changes are made to accommodate our formulation. This similarity stems from the fact that most changes to the system happen to accommodate the declarations in theory and context, while validity rules stay mostly the same.

Despite the extension to the proof being trivial, we will give an overview of the original argument for the benefit of the interested reader, making heavy references to it throughout the proof. Intuitively, the extension is trivial because Theorem 10 assumes a well-typed and valid derivation of the translated conjecture. Our extension does not change any validity rules and merely affects which types are possible and provides the option for polymorphic conjectures. As reasoning can only happen on fully applied base types, there are only minor changes to the formulation of some of the intermediate results.

The main challenge lies in the fact that there are situations in which the erasure of ill-typed DHOL terms results in terms that are well-typed in HOL. This is mainly due to the non-injectivity of the translation: two fixed length lists $a : lst\ 2$ and $b : lst\ 3$ of length 2 and 3 respectively are incomparable in DHOL, but erasing them results in $a : lst$ and $b : lst$ for which equality would be well-typed.

Note that the opposite problem, namely that a well-typed DHOL term t translates to an ill-typed HOL term $\overline{t}$, cannot happen. This is clear by the definition of the translation.

As a result of this non-injectivity, a valid HOL-derivation cannot be translated into a valid DHOL-derivation without further processing. The proof idea, then, is to show that it is possible to transform a HOL proof of a translated, well-typed DHOL statement, into a proof with qualities that allow for such a direct translation back into a DHOL proof.

We proceed to show this in the following steps: First, we will show that the translation, while not injective in general, is type-wise injective — meaning that if $t : A$ and $s : A$ are distinct, then so are $\overline{t} : \overline{A}$ and $\overline{s} : \overline{A}$. This will allow us to associate a unique DHOL term with HOL terms that come from the erasure, assuming the type is known. Using this, we show that HOL proofs can be transformed into HOL proofs that guarantee certain properties which will finally allow us to map said proofs to DHOL proofs of the untranslated conjecture.

We front-load this section with the necessary definitions to highlight the relationship between them.

▶ **Definition 13.** *Ill-typed DHOL terms t with a well-typed counterpart $\overline{t}$ will be called* spurious *while terms in which both — erased and original — terms are well-typed will be called* proper*. A* improper *term $\overline{t}$ is not in the (translation-)image of any DHOL term t.*

Normalizing *an improper term results in a proper or spurious one. We introduce the normalizing function used in the proof in Figure 5 and expand on it in the sequel. Normalizing an already proper or spurious term returns the same term.*

We extend the notion of "proper" from terms to contexts Δ, whenever $\overline{\Gamma}$ can be obtained from Δ by adding typing assumptions. Then Γ is called the quasi-preimage *of the* proper *context Δ.*

Furthermore, given a proper HOL context Δ, a statement ϕ over this context is called quasi-proper*, iff the normalization of ϕ is $\overline{F}$ for $\Gamma \vdash F : o$ and Γ quasi-preimage of Δ. In this case, F is called a* quasi-preimage *of ϕ.*

As a last extension to this terminology, a validity judgment $\Delta \vdash \phi$ is also called proper *iff Δ is proper and ϕ is quasi-proper in this context. Then $\overline{\Gamma} \vdash_{\overline{T}} \overline{F}$ is called a* relativization *of $\Delta \vdash_T \phi$ and $\Gamma \vdash_T F$ is called a* quasi-preimage *of $\Delta \vdash_T \phi$.*

We call an improper term almost proper *iff its normalization is not spurious. This is equivalent to saying an improper term is almost proper iff it is quasi-proper (has a well-typed quasi-preimage). Otherwise, it is called* unnormalizably spurious*.*

Finally, we give a definition of the property which allows us to translate HOL proofs into DHOL proofs. A valid HOL derivation is called admissible iff all terms occurring in it are almost proper.

B.1 Type-wise injectivity of the translation

Compared to the original formulation of Rothgang et al. [26] there are some changes to the translation. It is now possible to apply type arguments to base types and constants. These, however, are preserved in the erasure: base types $\overline{a\ \delta}$ and constants $\overline{c\ \delta}$ result in $a\ \overline{\delta}^{y}$ and $c\ \overline{\delta}^{ye}$ respectively.

The other relevant change to the system is the addition of type variables α : `type` as an option to the type system. These are straightforwardly translated into α : `type`, $\alpha^* : \alpha \to \alpha \to o$, $\triangleright \mathrm{PER}(\alpha^*)$.

▶ **Lemma 14.** *Let s, t be DHOL terms of type A. Assuming s and t are different, then $\overline{s} : \overline{A}$ and $\overline{t} : \overline{A}$ are different.*

Proof. The proof proceeds by induction over the term structure. Different top-level productions result in different terms after the translation, so we can limit ourselves to

$$\begin{aligned}
norm[\overline{t}] &:= t\\
norm[norm[s]] &:= norm[s]\\
norm[A^*\ s] &:= \lambda y : \overline{A}.A^*\ s\ y\\
norm[A^*] &:= \lambda x : \overline{A}.\lambda y : \overline{A}.A^*\ x\ y\\
norm[c\ \delta^{ye}] &:= c\ \delta^{ye}\\
norm[x] &:= x\\
norm[f\ t] &:= norm[f]\ norm[t]\\
norm[\lambda x : C.t] &:= \lambda x : C.norm[t]\\
norm[s =_{\overline{A}} t] &:= A^*\ s\ t\\
norm[s \Rightarrow t] &:= norm[s] \Rightarrow norm[t]\\
norm[\forall x : \overline{A}.A^*\ x\ x\ \Rightarrow G] &:= \forall x, y : \overline{A}.A^*\ x\ y \Rightarrow G
\end{aligned}$$

If F not of shape $A^*\ \_\ \_ \Rightarrow \_$ or $\forall x' : \overline{A}.A^*\ x\ x' \Rightarrow \_$:

$$norm[\forall x : \overline{A}.F] := norm[\forall x : \overline{A}.A^*\ x\ x \Rightarrow F]$$

Figure 5 Definition of $norm[t]$ with changes highlighted.

the cases where both terms have the same root symbol. For non-equality terms, the only cases that need to be adjusted from the original proof, are where we performed changes to the translation.

For that, notice that erasing applied base types and constants results in recursive calls to the translations of the applied types. By the induction hypothesis, these are already different, concluding the proof.

Another interesting case occurs when the terms t, s are equalities over two types erased to the same type. This is not possible for type variables α as their translation yields themselves. All remaining cases are identical to the original presentation. ◀

B.2 Transforming HOL proofs into admissible HOL proofs

In order to transform HOL proofs into admissible HOL proofs, the original proof defines two functions. First, they define a normalization function, given in Figure 5, with changes due to the incorporation of polymorphism highlighted. This normalization turns improper HOL-statements into proper or spurious ones, i.e. the term $norm[t]$ is in the image of the translation after normalization.

Second, a *normalizing statement transformation* $sRed(t)$ is applied to the derivation. A normalizing statement transformation is a function that replaces terms and their context in statements in such a way that unnormalizably spurious terms end up as almost proper ones. In contrast to $norm[t]$ the changes to accommodate polymorphism do not require any changes in the definition of $sRed(t)$, so the function (given in Figure 6) is identical to the one in [26]. $sRed(t)$ proceeds by beta-eta normalizing terms and, in case this does not make them almost proper, replaces unnormalizably spurious function applications of type B by a "default term" $\omega_B : B$ which is proper and exists due to the non-emptiness assumption in HOL.

This is aided by passing, for each term, a DHOL type A to the function, effectively

$$
\begin{aligned}
sRed(t_A) &:= t_A \\
&\quad \text{if } t \text{ has quasi-preimage of type A} \\
sRed(f_{\Pi x:A.B}\ t_A) &:= sRed(sRed(f_{\Pi x:A.B})\ sRed(t_A)) \\
&\quad \text{if } f_{\Pi x:A.B}\ t_A \text{ not beta-eta reducible}
\end{aligned}
$$

In the following, the term t_A on the left-hand side is assumed to not be almost proper with a quasi-preimage of type A :

$$
\begin{aligned}
sRed(t_A) &:= sRed(t_A^{\beta\eta}) \\
&\quad \text{if } t \text{ eta-beta reducible} \\
sRed(s_A =_{\overline{A}} t_{A'}) &:= sRed(s_A) =_{\overline{A}} sRed(t_A) \\
sRed(F_o \Rightarrow G_o) &:= sRed(F_o) \Rightarrow sRed(G_o) \\
sRed(\lambda x : A.s_B) &:= \lambda x : A.sRed(s_B) \\
sRed((sRed(f_{\Pi x:A.B})_{\Pi x:A.B}\ sRed(t_{A'})_{A'})_{B'}) &:= \omega_{\overline{B}} \\
&\quad \text{if } A \neq A' \text{ or } B \neq B'
\end{aligned}
$$

Figure 6 Definition of $sRed(t)$.

associating them with a quasi-preimage. This is mainly necessary for λ-functions where there are potentially many quasi-preimages of differing types. To ensure correctness, it is, of course, required that an indexed term t_A is of type $\overline{A}$ and, if it is almost proper with a unique quasi-preimage, that the quasi-preimage has type A.

Using the definition of the normalizing statement transformation, we can go on and state

▶ **Lemma 15.** *Assume a well-typed DHOL theory T and a conjecture $\Gamma \vdash_T \phi$ with Γ well-formed and ϕ well-typed. Assume a valid HOL derivation of $\overline{\Gamma} \vdash_{\overline{T}} \overline{\phi}$. Then, we can index the terms in the derivations s.t. any steps S in the derivation can be replaced by a macro-step (i.e. a step with the same assumptions and conclusion as the original step, composed of multiple micro-steps) for the normalizing statement transformation, replacing step S s.t. after replacing all steps with their macro-steps:*

- *the resulting derivation is valid,*
- *all terms occurring in the derivation are almost proper*

▶ Remark 16 (A note about indices). It is reasonable to assume that the addition of type variables changes the indexing procedure, so we give the full (original) procedure here:

Indexing starts at the end of the derivation. We pick identical indices for identical terms. Whenever we need to index a constant or variable, and its preimage exists in the context of the DHOL conjecture, we pick the type of the preimage. Term equalities always have the same index on both sides. For non-atomic terms t, we pick indices for the atomic subterms and choose for t a type of the unique (by Lemma 14) quasi-preimage, such that the indices match up. If possible, we choose indices such that there exists a well-typed quasi-preimage of that type. Unless otherwise indexed, the index for $sRed(t_A)$ will also be A. For λ-functions that already have an index assigned, the variable and the body are assigned matching indices. If the λ function does not yet have an index, but is applied to an argument with an index, we assign that index to the variable of the λ-function.

If none of these rules apply, we pick an arbitrary index that does not violate any of the future applications of the rules.

Inspecting this algorithm, it becomes clear that it forbids at no point the choice of a type variable. Indeed, the labeling process is completely agnostic to the underlying set of types. Type variables syntactically act like non-dependent base types and do not interfere with this procedure at all.

Due to the fact that no changes to the definition of $sRed$ are necessary, and the conjecture only talks about validity statements, we refer to the original proof in [26] for details. We will nevertheless give one example derivation to illustrate how the function interacts with the labels:

Proof. The proof proceeds by induction on the inference rules, and we pick the beta rule as example: $\dfrac{\Gamma \vdash_T (\lambda x : A.s)\ t : B}{\Gamma \vdash_T (\lambda x : A.s)\ t =_B s[x/t]}beta$

For the sake of clarity, we will use the substitution notation used in the original paper.

By assumption we get $\Delta \vdash_{\overline{T}} sRed((\lambda x_A : \overline{A}.s_B)\ t_A)_{B'} : \overline{B}$ and applying the definition of $sRed$ gives $sRed(\lambda x_A : \overline{A}.s_B) = \lambda x_A : \overline{A}.sRed(s_B)$. We index this new term such that it is consistent with the image of the function and so we give it the index B.

We proceed by case distinction on whether $(\lambda x_A : \overline{A}.sRed(s_B)_B)\ t_A$ is almost proper with quasi-preimage of type $B \equiv B'$:

If it is, we get $\Delta \vdash_{\overline{T}} (\lambda x_A : \overline{A}.sRed(s_B))\ sRed(t_A) : \overline{B}$ by the second line in the definition of $sRed$. We apply the beta rule and the definition of $sRed$ twice to that result to get the goal $\Delta \vdash_{\overline{T}} sRed((\lambda x_A : \overline{A}.sRed(s_B))\ sRed(t_A) =_{\overline{B}} sRed(s_B)[x_A/sRed(t_A)])$.

If not, we observe that $sRed((\lambda x_A : \overline{A}.s_B)\ t_A) = sRed(sRed(s_B)[x_A/sRed(t_A)]) = sRed(s_B)[x_A/sRed(t_A)]$. From reflexivity we have $\Delta \vdash_{\overline{T}} sRed(s_B)[x_A/sRed(t_A)] =_{\overline{B}} sRed(s_B)[x_A/sRed(t_A)]$. Due to the induction hypothesis and the choice of indices, we can assume that the $sRed$ terms in the equality have a quasi-preimage of type B, and by the definition of $sRed$ we conclude $\Delta \vdash_{\overline{T}} sRed((\lambda x_A : \overline{A}.sRed(s_B))\ sRed(t_A) =_{\overline{B}} sRed(s_B)[x_A/sRed(t_A)])$.

Note how this transformed a single beta rule step into a macro-step. The derivation shows that even if during a regular proof step the statement would become unnormalizably spurious, we can transform the statements in a way that yields almost proper terms. ◀

B.3 Translation of HOL proofs into DHOL proofs

Finally, we show the reflection of truth. As previously, we give the general outline of the proof and refer to [26] for details.

Proof. According to Lemma 15 we can assume that the proof of $\overline{\Gamma} \vdash_{\overline{T}} \overline{F}$ is admissible, as we can always transform a valid HOL proof into an admissible and valid HOL proof. Because admissibility implies the existence of a well-typed quasi-preimage and the fact that the translation is type-wise injective, we therefore have that the translated conjecture is a proper validity statement with unique quasi-preimage in DHOL.

It remains to show that it is possible to lift the HOL derivation of the conjecture to a DHOL derivation of its quasi-preimage. For that, we can inspect the validity rules one for one and show that — assuming the conclusion is proper and has a quasi-preimage — all validity assumptions and their contexts are well-formed and proper respectively. From this, we continue to prove that in this case, the quasi-preimage of the conclusion of the rule is valid.

As stated previously, the validity rules for the polymorphic extension do not change compared to their monomorphic variants. However, we now have to consider applied types. Inspecting the normalization function shows that normalizing constants applied to type arguments is the identity function. Therefore, there is no change in the validity of the proofs as performed in [26]. ◀

C Red-Black Tree in PDHOL

Following is the theory of red-black trees from our case study in PDHOL:

$$\mathtt{color} : \mathtt{type} \quad \mathtt{color} : \mathtt{color} \quad \mathtt{red} : \mathtt{color}$$
$$\mathtt{nat} : \mathtt{type} \quad 0 : \mathtt{nat} \quad \mathtt{suc} : \mathtt{nat} \to \mathtt{nat}$$
$$\mathtt{tree} : \Pi_\alpha \Pi_{c:\mathtt{color}, n:\mathtt{nat}} : \mathtt{type} \quad \mathtt{leaf} : \Pi_\alpha . \mathtt{tree}\ \alpha\ \mathtt{color}\ 0$$
$$\mathtt{red\_tree} : \Pi_\alpha \Pi_{n:\mathtt{nat}} . \mathtt{tree}\ \alpha\ \mathtt{color}\ n \to \alpha \to \mathtt{tree}\ \alpha\ \mathtt{color}\ n \to \mathtt{tree}\ \alpha\ \mathtt{red}\ n$$
$$\mathtt{black\_tree} : \Pi_\alpha \Pi_{c_1:\mathtt{color}, c_2:\mathtt{color}, n:\mathtt{nat}} . \mathtt{tree}\ \alpha\ c_1\ n \to \alpha \to \mathtt{tree}\ \alpha\ c_2\ n \to \mathtt{tree}\ \alpha\ \mathtt{color}\ (\mathtt{suc}\ n)$$
$$\mathtt{rev} : \Pi_\alpha \Pi_{c:\mathtt{color}, n:\mathtt{nat}} . \mathtt{tree}\ \alpha\ c\ n \to \mathtt{tree}\ \alpha\ c\ n$$
$$\rhd\ \forall \alpha : \mathtt{type} . \mathtt{rev}\ \alpha\ \mathtt{color}\ 0\ (\mathtt{leaf}\ \alpha) = \mathtt{leaf}\ \alpha$$
$$\rhd\ \forall \alpha : \mathtt{type}, x : \alpha, n : \mathtt{nat}, c_1 : \mathtt{color}, c_2 : \mathtt{color}, t_1 : \mathtt{tree}\ \alpha\ c_1\ n, t_2 : \mathtt{tree}\ \alpha\ c_2\ n.$$
$$\quad \mathtt{rev}\ \alpha\ \mathtt{color}\ (\mathtt{suc}\ n)\ (\mathtt{black\_tree}\ \alpha\ n\ c_1\ c_2\ t_1\ x\ t_2) =$$
$$\quad \mathtt{black\_tree}\ \alpha\ n\ c_2\ c_1\ (\mathtt{rev}\ \alpha\ c_2\ n\ t_2)\ x\ (\mathtt{rev}\ \alpha\ c_1\ n\ t_1)$$
$$\rhd\ \forall \alpha : \mathtt{type}, x : \alpha, n : \mathtt{nat}, t_1 : \mathtt{tree}\ \alpha\ \mathtt{color}\ n, t_2 : \mathtt{tree}\ \alpha\ \mathtt{color}\ n.$$
$$\quad \mathtt{rev}\ \alpha\ \mathtt{red}\ n\ (\mathtt{red\_tree}\ \alpha\ n\ t_1\ x\ t_2) =$$
$$\quad \mathtt{red\_tree}\ \alpha\ n\ (\mathtt{rev}\ \alpha\ \mathtt{color}\ n\ t_2)\ x\ (\mathtt{rev}\ \alpha\ bl\ n\ t_1)$$

This theory was used to show, with various in-between steps, the following conjecture:

$$\rhd\ \forall \alpha : \mathtt{type}, n : \mathtt{nat}, c : \mathtt{color}, t : \mathtt{tree}\ \alpha\ c\ n.$$
$$\quad \mathtt{rev}\ \alpha\ c\ n\ (\mathtt{rev}\ \alpha\ c\ n\ t) = t$$