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 presentations of important domains like size-bounded data structures, category theory, or proof theory while retaining strong automated theorem support. The latter 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 offixed 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 soundness/completeness proofs to the polymorphic case. We demonstrate both the expressivity and evaluate the practical theorem proving support for polymorphic DHOL by presenting a range of formalizations as TPTP problems that we tackle with off-the-shelf theorem provers for HOL.

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-IROLiCMtJrakUvd

1 Introduction

Setting Monomorphic dependently-typed higher-order logic (DHOL) was introduced by Rothgang et al. [20]. It combines the simplicity of higher-order logic (HOL) [4, 8], 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 [5, 15, 6], 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 comes at the price of making typing undecidable, but it also has the benefit of being very similar in style to languages in the HOL ecosystem for which strong automated theorem proving (ATP) support is available. Rothgang et al. [20] leverage this similarity by giving a linear, compositional, and sound/complete translation that translates monomorphic DHOL to monomorphic HOL. This translation makes it possible to obtain implementations of both type-checking and theorem proving for DHOL. Practical experiments show that the obtained combination of expressivity and ATP support makes undecidable typing a price worth paying.

The translation is based on partial equivalence relations (PERs). Maybe surprisingly, even though the PER semantics of dependent types are well-known, the soundness/completeness proofs turned out to be very difficult in the presence of classical Booleans. The proof used in [20] uses a complex back-translation of HOL-proofs to DHOL, but it omitted important HOL features such as choice and polymorphism even though they are supported by most HOL ATPs.

Contribution The present paper introduces a polymorphic version of DHOL. We extend the syntax of DHOL with type variables and amend the semantics, meta-theory, and translation accordingly.

We leverage our polymorphic DHOL in a number of practical theorem-proving problems, enabling elegant and concise formalization.

The list of problems is formalized in a slightly modified variant of TPTP [23]. Problems include an encoding of Red-Black Trees and a conjecture establishing that the reversal function on them is an involution, as well as sub-problems of this conjecture. Furthermore, we extend previously presented problems for lists in monomorphic to polymorphic DHOL.

Our definitions are deceptively simple because the main difficulty of our work was choosing the right trade-off for the language design that keeps the language expressive enough to meet practical demand and simple enough to retain strong automation support. We opted for shallow (ML style) polymorphism, i.e., we allow all global declarations (type symbols, term symbols, and axioms) to be parametric in type variables $\alpha : \mathtt{type}$ that can be understood as being universally quantified. Consequently, all references to a declaration must provide or allow inferring an appropriate substitution for the type parameters.

Shallow polymorphism has the advantage of covering a large portion of practical needs while adding only minor complications to the meta-theory. In particular, it allows the declaration of type operators $a : \Pi : \mathtt{type}$, as well 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 (like in $\neg\forall\alpha : \mathtt{type}\ldots$) or higher-order type variables (like in $\lambda_{\alpha:\mathtt{type}\to\mathtt{type}}\ldots$). However, it allows type variables to depend on term arguments as in $(\Pi_{\alpha:\mathtt{nat}\to\mathtt{type}}\ldots)$, and we also introduce a variant of polymorphic DHOL with dependent type variables in Sect. 7.2 and use it to formalize heterogeneous lists.

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 [12], 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 both interactive HOL theorem provers [8, 14, 9] and automated ones [22, 18, 24]. 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 very similar to our design. It combines native decision procedures with interaction and automation but to is too expressive for easy translation into standard ATP tools. Recently the Vampire automated theorem prover has been extended with shallow polymorphism [3]. The extension is very elegant, but it focuses on non-dependent types so far.

The general idea of translating dependent type theories is not new: Felty and Miller [7] translated LF into hereditary Harrop formulas, a simply-typed meta-logic, over 30 years ago. The work of Jacobs and Melham [10], which presented the idea of translating dependent types into predicates which serve as type guards, is closer to our translation.

2 Syntax

The grammar below shows both the DHOL language and our extension for polymorphism. We also mark the parts that must be removed additionally to recover HOL as a fragment. (In that case, types can never contain terms so that $\Pi_{x:A}B$ can always be written as $A \to B$, a convention that we will use for non-dependent functions in DHOL too.) Note, that $., \circ, \bullet$ denote the empty theory, context, and substitution respectively, and F represents a meta variable for terms of type o.

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

This suffices to define all the usual constants, connectives and binders [1] including $\Rightarrow$. To accommodate dependent types, however, we need at least one of the connectives as a primitive, see also Remark 4.

Above and in the sequel, we use the following abbreviations to handle sequences of expressions concisely:

abbr.	expansion	remark
$\vec{A}$	$A_1 \ \ldots \ A_n$	types in a substitution or application
$\vec{t}$	$t_1 \ \ldots \ t_n$	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:

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

Here, $\mathtt{nat}$ is a non-dependent, or simple, base type for which a constant, a constructor, and a function are declared. In the remainder, we will abbreviate $\mathtt{suc}\ 0, \mathtt{suc}\ (\mathtt{suc}\ 0), \ldots$ with $1, 2, \ldots$. To define $+$ we add axioms.

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

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.

$$\mathtt{vec} : \Pi_\alpha \Pi_{n:\mathtt{nat}} : \mathtt{type} \quad \mathtt{nil} : \Pi_\alpha \mathtt{vec}\ \alpha\ 0 \quad \mathtt{cons} : \Pi_\alpha \Pi_{n:\mathtt{nat}} \alpha \to \mathtt{vec}\ \alpha\ n \to \mathtt{vec}\ \alpha\ (\mathtt{suc}\ n)$$
$$++ : \Pi_\alpha \Pi_{n,m:\mathtt{nat}} \mathtt{vec}\ \alpha\ n \to \mathtt{vec}\ \alpha\ m \to \mathtt{vec}\ \alpha\ (+\ n\ m)$$

Removing the highlighted dependent part and/or instantiating the highlighted polymorphic part yields fixed-type, dynamic lists in HOL or fixed-type vectors in (monomorphic) DHOL.

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

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 $\mathtt{cons\ nat}\ 1\ n\ (\mathtt{cons\ nat}\ 0\ m\ (\mathtt{nil\ nat})) : \mathtt{vec\ nat}\ 2$. 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: $n : \mathtt{nat}, m : \mathtt{nat}, n =_{\mathtt{nat}} 2, m =_{\mathtt{nat}} 3$.

A well-formed substitution γ for this context is $n, suc\ n, ✓, ✓$. $E[\gamma]$ would then be $\mathtt{cons\ nat}\ 1\ n\ (\mathtt{cons\ nat}\ 0\ (\mathtt{suc}\ n)\ (\mathtt{nil\ nat})) : \mathtt{vec\ nat}\ 2$. The ✓s in the substitution represent the proof obligations $n =_{\mathtt{nat}} 2$, which follows from the context, and $suc\ n =_{\mathtt{nat}} 3$ for which the validity is easy to see.

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) $\triangleright F$.

Contrary to contexts, declarations in theories may depend on arguments, and this is the mechanism how both monomorphic DHOL [20] and our extension thereof are defined as generalizations of HOL. The following table gives an overview of the possible combinations:

depending on	declaration of type symbol	term symbol	global assumption
type variable	allowed in polymorphic (D)HOL		
term variable	allowed in DHOL	definable via Π	definable via $\forall$
local assumption	not allowed		definable via $\Rightarrow$

Monomorphic DHOL arises by allowing type symbols to depend on term arguments. Polymorphic HOL and DHOL arise by allowing all declarations to depend on type arguments. The three other combinations are definable in all languages and, therefore, are not included in our grammar.

The remaining two cases are local assumptions as parameters of type/term symbols. This would allow declaring partial functions and (less commonly used) partial type symbols axioms. Our definition of DHOL can easily be amended to allow this, but it would make the translation to HOL significantly more complicated. Therefore, we do not consider those extensions here.

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

- references to symbols declared in the theory, which in the polymorphic case must be instantiated with type arguments for the type variables and (in the case of type symbols) term arguments,
- 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 $F \Rightarrow G$.

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

▶ Remark 3 (Parametric Declarations). The grammar of monomorphic DHOL grammar avoids kinds and `type`-level λ-abstraction: the production rule for declaring dependent base types spells out the normal form $(\Pi_{x:A})^*\mathtt{type}$ of kinds rather than using $a : K$ for an arbitrary kind. Consequently, type symbols can only be used in fully applied form $a\,\delta$. Our polymorphic version follows that design principle for declaring and referencing polymorphic symbols. λ-abstraction over type variables in terms/types and over term variables in types can be added easily, and that is reasonable in practical implementations.

▶ Remark 4 (Dependent Implication). DHOL requires using *dependent* implication: Consider the implication $a =_A b \Rightarrow f\ a =_{B[a]} f\ b$. Well-formedness of the right-hand side is only given if the left-hand side is assumed to be true. In HOL, equality suffices to define all the usual connectives. But dependent implication cannot be defined from equality alone. Therefore, DHOL makes implication primitive as well. In examples, we will use all the usual connectives. It is important to note that due to this change connectives like con- and disjunction lose their commutativity properties.

▶ Remark 5 (Reasoning about Types). Our formulation keeps the style of monomorphic HOL and DHOL not to allow reasoning about types. In particular, equality $A \equiv B$ of types is not a term and is instead only introduced as a meta-level judgment.

Correspondingly we do not allow any quantification over type variables. We only allow the implicit universal quantification that is provided by Π-binding type variables at the top level of a declaration. That is sufficient to state axioms that universally quantify over types.

3 Inference System

Name	Judgment	Intuition
theories	$\vdash T\ \mathtt{Thy}$	T is well-formed theory
contexts	$\vdash_T \Gamma\ \mathtt{Ctx}$	Γ is well-formed context
substitutions	$\Gamma \vdash_T \Delta \leftarrow \delta$	δ is well-formed substitution for Δ
types	$\Gamma \vdash_T A : \mathtt{type}$	A is well-formed type
typing	$\Gamma \vdash_T t : A$	t is well-formed term of well-formed type A
validity	$\Gamma \vdash_T F$	Boolean F 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 kinds of judgments possible in DHOL. The kinds are the same as in monomorphic DHOL, however, since contexts Γ can now contain type variables, these judgments now correspond to polymorphic statements. In Fig. 2 and 3 we present the rules of polymorphic DHOL. They arise as a straightforward extension of the rules of monomorphic DHOL. They are the standard proof rules and do not support any meta-reasoning about types.

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 CtxType handle the declaration of types

Well-formed theories T

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

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

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

$$\frac{\vdash T\ \mathtt{Thy}}{\vdash_T \circ\ \mathtt{Ctx}}\,\mathit{CtxEmpty} \qquad \frac{\vdash_T \Gamma\ \mathtt{Ctx}}{\Gamma \vdash_T \bullet \leftarrow \circ}\,\mathit{SubEmpty}$$

$$\frac{\vdash_T \Delta\ \mathtt{Ctx}}{\vdash_T \Delta, \alpha : \mathtt{type}\ \mathtt{Ctx}}\,\mathit{CtxType} \qquad \frac{\Gamma \vdash_T \Delta \leftarrow \delta \quad \Gamma \vdash_T A : \mathtt{type}}{\Gamma \vdash_T \Delta, \alpha : \mathtt{type} \leftarrow \delta, A}\,\mathit{SubType}$$

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

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

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

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

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

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

Figure 2 DHOL Rules for theories and contexts

$a : \Pi_{\vec{\alpha}}, \Pi_{\vec{x}:\vec{A}} : \mathtt{type}$ 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 TypeSym and Rule TypeVar 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 that instantiates the parameters.

Finally, the four rules for forming contexts are parallel to the four rules for forming substitutions. For example, Rule SubType 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. The rules for Booleans and functions introduce the type and its terms. The rules for type equality are routine syntax-driven congruence rules, and we omit the remaining rules for validity, which are straightforward. We only point out two specialties of DHOL: The rule Rule TermImpl makes implication $F \Rightarrow G$ dependent by requiring the well-formedness of G only under the assumption that F holds. The rule Rule TypeEqSym, 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

Booleans: type formation, terms for equality and implication

$$\frac{\vdash_T \Gamma\ \mathtt{Ctx}}{\Gamma \vdash_T o\ :\ \mathtt{type}}\mathit{TypeBool} \qquad \frac{\Gamma \vdash_T t\ :\ A \quad \Gamma \vdash_T u\ :\ A}{\Gamma \vdash_T t =_A u\ :\ o}\mathit{TermEq} \qquad \frac{\Gamma \vdash_T F\ :\ o \quad \Gamma, \triangleright F \vdash_T G\ :\ o}{\Gamma \vdash_T F \Rightarrow G\ :\ o}\mathit{TermImpl}$$

Function: type formation, λ-abstraction, application

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

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

Type equality: congruence for all constructors and conversion of types

$$\frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad \alpha\ :\ \mathtt{type}\ \text{in}\ \Gamma}{\Gamma \vdash_T \alpha \equiv \alpha}\mathit{TypeEqVar} \qquad \frac{\vdash_T \Gamma\ \mathtt{Ctx} \quad a : \Pi_\Delta \mathtt{type}\ \text{in}\ T \quad \Gamma \vdash_T \delta \equiv_\Delta \delta'}{\Gamma \vdash_T a\ \delta \equiv a\ \delta'}\mathit{TypeEqSym}$$

$$\frac{\vdash_T \Gamma\ \mathtt{Ctx}}{\Gamma \vdash_T o \equiv o}\mathit{TypeEqBool} \qquad \frac{\Gamma \vdash_T A \equiv A' \quad \Gamma, x : A \vdash_T B \equiv B'}{\Gamma \vdash_T \Pi_{x:A}B \equiv \Pi_{x:A'}B'}\mathit{TypeEqPi}$$

$$\frac{\Gamma \vdash_T t\ :\ A \quad \Gamma \vdash_T A \equiv A'}{\Gamma \vdash_T t\ :\ A'}\mathit{TermConvert} \qquad \frac{\Gamma \vdash_T t\bot \quad \Gamma \vdash_T t(\neg\bot)}{\Gamma, x : o \vdash_T sx}\mathit{BoolExt}$$

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 $A \equiv st$; β, η for functions; introduction and elimination for $\Rightarrow$
Note that propositional, functional extensionality is implied by the rules for equality and the omitted eta rule. [19]

Figure 3 DHOL Rules for Types and Terms

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. Rule TermConvert is only needed for constants and variables; for any other term, it can be derived using congruence.

4 Translating DHOL to HOL

Overview Like in the original formulation of Rothgang et al. [20], the general idea of the translation is to apply dependency erasure, written with an overline $\overline{\square}$, for all DHOL syntax, resulting in the syntax of polymorphic HOL. Because the latter is a fragment of the former, we can reuse all DHOL notations to build HOL syntax. Intuitively, term-dependent types are translated into types without their term arguments. The hereby lost information is captured in a partial equivalence relation (PER) serving as a predicate to ensure well-typedness. Examples 7 and 8 will illustrate this after the following definition.

Formally, all term arguments of type symbols are erased. Our dependency erasure does not strip away type arguments: references to 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 DHOL-type A a PER in HOL A^* on $\overline{A}$ such that the provability of $A^*\ \overline{t}\ \overline{t}$ captures the typing judgment $t : A$. Critically, term equality

$s =_A t$ is translated to $A^* \; \overline{s} \; \overline{t}$.

PERs are symmetric and transitive relations. It is easy to see that an element in a PER is related to itself iff it is related to any element. So a PER A^* on a type $\overline{A}$ 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 will write PER(r) to abbreviate that r is a PER (i.e., symmetric and transitive).

As will become apparent shortly, expanding the usual definitions of the quantifiers indeed 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 introduces a lot of notational complexity. To keep things as intuitive as possible, we introduce the following abbreviations:

▶ **Definition 6** (Abbreviations for HOL Contexts). *Given a HOL-context Γ, we abbreviate*

- Γ^y*: like* Γ *but containing only the type variables*
- Γ^{ye}*: like* Γ *but containing only the type and term variables*

Given a HOL-substitution γ*, we abbreviate*

- γ^y*: like* γ *but containing only the type substitutes*
- γ^{ye}*: like* γ *but containing only the type and term substitutes*

We also abbreviate as follows:

- *if* Γ *contains only type variables, we write* $\Gamma \to \texttt{type}$ *for the kind* $\texttt{type} \to \ldots \to \texttt{type} \to \texttt{type}$ *(taking one type argument for every type variable in* Γ*)*
- *if* Γ *contains only type variables* $\vec{\alpha}$ *and term variables* $x_i : A_i$*, we write* $\Gamma \to B$ *for the polymorphic simple function type* $\Pi_{\vec{\alpha}} A_1 \to \ldots \to A_n \to B$
- *if* Γ *contains type variables* $\vec{\alpha}$*, term variables* $x_i : A_i$*, and assumptions* $\rhd F_i$*, we write* $\Gamma \Rightarrow G$ *for the polymorphic HOL formula* $\Pi_{\vec{\alpha}} \forall x_1 : A_1 \ldots . \forall x_m : A_m . F_1 \Rightarrow \ldots F_n \Rightarrow G$

The purpose of the abbreviations Γ^y and Γ^{ye} is to remove declarations from contexts that a declaration cannot bind: term symbol declarations cannot bind assumptions, and type symbol declarations cannot bind assumptions or term variables. These become necessary because every type A is translated to a triple of translated type $\overline{A}$, binary relation A^*, and statement that PER(A^*). Consequently, every type variable α is translated to three declarations: a type variable α, a binary relation α^* on it, and an assumption that α^* is a PER (i.e., the length of all type variable contexts triples). Similarly, every term $t : A$ is translated to a pair of a translated term $\overline{t}$ and the statement of $A^* \; \overline{t} \; \overline{t}$. Consequently, every term variable x is translated to two declarations: a term and an assumption. Thus, the translation of a context Γ 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 type or term symbol declaration.

Secondly, if a context has been stripped of the unwanted declarations, the purpose of the abbreviations $\Gamma \to \texttt{type}$, $\Gamma \to A$, and $\Gamma \Rightarrow F$ is to efficiently perform these bindings. For example, if a DHOL axiom $\rhd \Pi_{\vec{\alpha}} F$ is parametric in n type variables, $\overline{\Gamma}$ contains $3n$

declarations, all of which must be bound in a polymorphic HOL axiom. We use the notation $\overline{\Gamma} \Rightarrow \overline{F}$ to capture this binding, where the type variables in $\overline{\Gamma}$ remain type variables, the term variables are $\forall$-bound, and the unnamed assumptions are bound by $\Rightarrow$.

Formal Definition We define the translation in chunks for the various kinds of syntax. For types and terms, the translation is defined as follows:

DHOL type A	Translation $\overline{A}$ in HOL	DHOL term t	Translation $\overline{t}$ in HOL
$a\ \delta$	$a\ \overline{\delta}^y$	$c\ \delta$	$c\ \overline{\delta}^{ye}$
α	α	x	x
o	o	$t =_A u$	$A^*\ t\ u$
		$F \Rightarrow G$	$\overline{F} \Rightarrow \overline{G}$
$\Pi_{x:A}B$	$\overline{A} \to \overline{B}$	$\lambda_{x:A}t$	$\lambda_{x:\overline{A}}\overline{t}$
		$t\ u$	$\overline{t}\ \overline{u}$

For the translation of type and term symbol references, compare the translation of the corresponding declarations below, which matches the translation of the symbol references.

Contexts and substitutions are translated by concatenating the translations of their components. The cases are:

DHOL	Translation	DHOL	Translation
Contexts		Substitutions	
$\alpha\ :\ \mathtt{type}$	$\alpha\ :\ \mathtt{type},\ \alpha^* : \alpha \to \alpha \to o,\ \triangleright\ \mathrm{PER}(\alpha^*)$	A	$\overline{A},\ A^*,\ \checkmark$
$x : A$	$x : \overline{A},\ \triangleright\ A^*\ x\ x$	t	$\overline{t},\ \checkmark$
$\triangleright\ F$	$\triangleright\ \overline{F}$	$\checkmark$	$\checkmark$

Note how the translation of substitutions matches that of contexts: 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 DHOL-type A, HOL can prove $\mathrm{PER}(A^*)$, and for every DHOL-term $t : A$, HOL can prove $A^*\ t\ t$. These properties are made explicit in Thm. 9 below, which states that substitutions for Δ do indeed translate to substitutions for $\overline{\Delta}$.

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

DHOL type A	$A^*\ t\ u$ in HOL
$a\ \delta$	$a^*\ \overline{\delta}^{ye}\ t\ u$
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 extend the expression E from Ex. 2 to a list `[E, E]` of lists by $E_2 :=$ `cons` (`vec` `nat` 2) 1 E (`cons` (`vec` `nat` 2) 0 E (`nil` (`vec` `nat` 2))) : `vec` (`vec` `nat` 2) 2. Its translation $\overline{E_2}$ yields `cons` (`vec` `nat`) 1 $\overline{E}$ (`cons` (`vec` `nat`) 0 $\overline{E}$ (`nil` (`vec` `nat`))).

Here δ is (`vec` `nat` 2) 1 E (...). 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 (`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 `vec` (`vec` `nat` 2) 2 is correspondingly erased to `vec` (`vec` `nat`).

Finally, the cases for declarations in theories are more complex because they involve contexts that must be translated and bound recursively. This is where the abbreviations from Def. 6 are most useful:

DHOL		Translation
$a : \Pi_\Delta \texttt{type}$	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 $\Delta = \vec{\alpha} : \texttt{type}$	$c : \overline{\Delta}^{ye} \to \overline{A}$ $\rhd \overline{\Delta} \Rightarrow A^*\ (c\ \vec{\alpha})\ (c\ \vec{\alpha})$
$\rhd \Pi_\Delta F$	where $\Delta = \vec{\alpha} : \texttt{type}$	$\rhd \overline{\Delta} \Rightarrow \overline{F}$

▶ **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 $\Rightarrow$.

The astute reader might note the absence of the according statements for the type $\texttt{nat}$. In the case of non-dependent base types, the partial equivalence relation just collapses to standard equality, resulting in axioms stating trivial reflexivity of equality. As such we chose to omit them.

Note that our translation subsumes that of Rothgang et al. [20]. If we specialize to monomorphic 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,
- declarations of type symbols a in a theory declare only type arguments Δ and references take only type arguments δ and thus
  - $\overline{\Delta}^y$ and $\overline{\delta}^y$ are empty
  - $\overline{\Delta}^{ye}$ contains only the declarations $x : \overline{A}$ for every type A in Δ
  - $\overline{\delta}^{ye}$ contains only the terms $\overline{t}$ for every term t in δ
- the argument list δ of a term symbol c is empty and thus so is $\overline{\delta}^{ye}$.

5 Invariants of the Translation

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

DHOL	*HOL*
$\vdash T$ Thy	$\vdash \overline{T}$ Thy
$\vdash_T \Gamma$ Ctx	$\vdash_{\overline{T}} \overline{\Gamma}$ Ctx
$\Gamma \vdash_T \Delta \leftarrow \delta$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{\Delta} \leftarrow \overline{\delta}$
$\Gamma \vdash_T A : \mathtt{type}$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{A} : \mathtt{type}$ *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*

DHOL	*HOL*
$\Gamma, \Delta \vdash_T A : \mathtt{type}$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{A[\delta]} \equiv \overline{A}[\overline{\delta}]$
$\Gamma, \Delta \vdash_T t : A$	$\overline{\Gamma} \vdash_{\overline{T}} \overline{t[\delta]} =_{\overline{A[\delta]}} \overline{t}[\overline{\delta}]$

Proof. This is proved by straightforward induction on derivations. To illustrate we present the corresponding proof for the rule Rule TypeSym:

$$\vdash_{\overline{T}} \overline{\Gamma}\ \mathtt{Ctx} \qquad assumption + IH \qquad (1)$$

$$\overline{a : \Pi_\Delta} \in T \qquad assumption + IH \qquad (2)$$

$$\overline{\Gamma \vdash_T \Delta \leftarrow \delta} \qquad assumption + IH \qquad (3)$$

$$\overline{\Gamma} \vdash_{\overline{T}} \overline{\Delta} \leftarrow \overline{\delta} \qquad \overline{\ }\text{-def} \qquad (4)$$

$$\overline{\Gamma} \vdash_{\overline{T}} a\ \overline{\delta} : \mathtt{type} \qquad \text{Rule TypeSym} \qquad (5)$$

$$a^* : \overline{\Delta}^{ye} \to a\ \vec{\alpha} \to a\ \vec{\alpha} \to o \in T \qquad \text{Eq } 2 \qquad (6)$$

$$\overline{\Gamma} \vdash_{\overline{T}} a^*\ \overline{\delta}^{ye} : a\ \overline{\delta}^{y} \to a\ \overline{\delta}^{y} \to o \qquad \text{Rule TermSym} \qquad (7)$$

$$\overline{\Gamma} \vdash_{\overline{T}} (a\ \delta)^* : \overline{a\ \delta} \to \overline{a\ \delta} \to o \qquad \overline{\ }\text{-def} \qquad (8)$$

$$\triangleright\ \overline{\Delta} \Rightarrow PER(a^*\ \vec{\alpha}\ \vec{\alpha}^*\ \vec{x}) \in T \qquad \text{Eq } 2 \qquad (9)$$

$$\overline{\Gamma} \vdash_{\overline{T}} PER(a^*\ \overline{\delta}^{ye}) \qquad \text{Rule ValidSym} \qquad (10)$$

$$\overline{\Gamma} \vdash_{\overline{T}} PER((a\ \delta)^*) \qquad \overline{\ }\text{-def} \qquad (11)$$

The other proofs follow the same structure of applying the induction hypothesis followed by definitions of the erasure. ◀

▶ **Theorem 10** (Soundness). *Assume a well-formed DHOL 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* $\Gamma \vdash s =_A t$*.*

Proof. The original soundness proof due to Rothgang et al. [20] is rather involved. Luckily it is easy to extend to the polymorphic case. Intuitively, a proof of a HOL statement $\overline{F}$ is translated into a proof that exists in the image of the translation. This allows us to subsequently read off a DHOL proof of the untranslated conjecture F.

An overview of the original proof together with the necessary adaptions is given in Appendix A. ◀

Depending on how one thinks about the translation — either as a way to provide semantics to DHOL or to enable HOL provers for DHOL — one can justify calling either of these properties Soundness and/or Completeness. In the remainder, we will informally refer to Theorem 9 as Completeness and continue to refer to Theorem 10 as Soundness.

6 Implementation and Case Studies

6.1 Translation

To evaluate and leverage polymorphic DHOL, we implemented the translation as a part of the logic-embedding tool by Steen [21]. Our implementation builds on the one for monomorphic DHOL [20]. 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.

Most automated theorem-proving systems support the format established by TPTP [23] as their input specification, and the logic-embedding tool operates as a translation tool between various forms thereof. TPTP supports polymorphic HOL through the TH1 format [11] which specifies the application of type arguments to, and the form of, polymorphic types. However, the associated syntax can be easily applied to *term* arguments and types depending on terms as well.

For example, we represent polymorphic dependent type symbol declarations $a : \Pi_{\vec{\alpha}}\Pi_{\vec{x}:\vec{A}} :$ type as a `:!>[`α_1`:$tType, ...,` α_m`:$tType,` x_1 `:` A_1`, ...,` x_m `:` A_n`]:$tType`, polymorphic term symbol declarations $c : \Pi_{\vec{\alpha}}A$ as c `: !>[`α_1`:$tType, ...,` α_n`:$tType,]:`A, and polymorphic axioms $\Pi_\alpha F$ as `![`α_1`:$tType, ...,` α_n`:$tType,]:`F. Their instantiations are written by applying every type/term argument individually via `@`.

6.2 Type Checking

In addition to the translation we implemented type checking for polymorphic DHOL. Theorem 10 establishes that provability in HOL carries over to polymorphic DHOL if the problem is well-typed. However, especially in the case of complex problems, it is often not easy to manually verify whether a problem is well-typed. Indeed, most of the issues we initially encountered in our examples were due to typing errors.

Our type checker is built on top of the TPTP parser utility used in the logic-embedding tool and reuses part of the provided infrastructure. The input is an arbitrary TPTP file. After assigning the type `$o` to the conjecture it proceeds to apply the typing rules of the inference system. As opposed to the logic-embedding tool, however, the output consists of several TPTP problems describing the different type checking obligations (TCOs) that need to be verified for the problem to be well-typed. Trivial obligations are dismissed.

This corresponds to how Rothgang et al. [20] implemented type-checking. However, whereas they implemented the checker inside the MMT framework [16], we used a standalone implementation (which thus provides a second implementation of type-checking for monomorphic DHOL). As TPTP gives all arguments — including type arguments — explicitly, this currently does not support inferring implicit arguments, a feature provided by MMT.

Our presentation assumes that all quantifiers and connectives are defined from dependent implication and equality. However, in our implementation, we make them primitive and translate them to their TPTP analogues. In particular, all binary connectives are dependent. This means that during type checking, conjunction (disjunction) makes its left argument available when type-checking its right argument as a (negated) assumption.

One challenge of using TPTP is that the equality of TPTP is binary and does not take the type as the third argument. Our type checker addresses this issue by additionally performing type inference whenever it encounters an equality. The types of both sides are inferred separately and, if not identical, generate a TCO to be proved. The algorithm is based on the Hindley-Milner type inference system. This is the major source of complexity in the type checking procedure, especially for problems with many equalities. However, the time required to type check a problem is, in general, still significantly shorter than the time required to prove it.

6.3 Evaluation

We created a set of 47 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. [13] 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. The remaining 11 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 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.

Polymorphic Vector Properties The formulation of the list problems closely follows that of the examples given in Section 2 and 4. The conjectures that were investigated expressed properties of the append function — namely the identity of nil and associativity of it — as well as the involution property of the reverse function. These problem files generated more complex type checking obligations than the red-black trees, due to the arithmetic needed to show that lengths of lists will line up after appending other lists. While we were able to discharge all of the 45 type checking obligations (TCOs) for our list examples, most of the actual conjecture proofs timed out. This is in line with the results reported by Niederhauser et al. [13]: DHOL's automation can perform well in the case of complex problems only with dedicated automated reasoning rules.

Reversal of Red-Black Trees We encoded red-black trees in TPTP, leveraging the expressive type system to ensure that well-formed red-black trees always satisfy the eponymous invariant. Crucially, red-black trees require all branches to have the same number of black nodes but do not restrict where on the path they occur. Such invariants are difficult to capture by simply-typed inductive types. However, in polymorphic DHOL, we can use a declaration `rbtree: !>[A:$tType]: bool > nat > $tTtype` 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 shows that the reversal of the reversal of a red-black tree is the identity.

The standard proof of this fact involves inductive definitions which are often difficult for ATP systems to deal with. In line with previous work in DHOL [13, 17] we split the problem up into smaller sub-problems, ensuring well-typedness.

The problem is split into a base case for red leaves and 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. This lemma is used to prove the base case for any tree of black-height 1.

Next are the step cases: Again there is a step case for red and for black leaves. While the red step case is easy, the black is challenging due to the complexity of black nodes compared to red nodes. While possible to prove directly, we also added three sub-steps, basically equational reasoning steps, to help the ATP system along.

The instantiated induction scheme timed out. This was expected, as for red-black trees, instantiating the scheme involves complex reasoning about the black-height of the tree for the step case. However, the type checking was successful.

Finally, we specified one problem not related to the reversal function, but which serves as a sanity check for the formulation of red-black trees. It takes the form of a conjecture trying to create an ill-formed red-black tree. It postulates that for every red tree, there are two children of arbitrary color. This would of course violate 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.

It is notable that the red-black tree examples generate very few type checking obligations (3 in total), compared to the previously discussed list examples. 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.

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.

7 Generalizations

7.1 Dependent Sums

DHOL can be extended with dependent sums $\Sigma x : A.B(x)$ for $x : A \vdash B(x) : \mathtt{type}$ in a straightforward way. The TPTP syntax again anticipates this extension by reserving the syntax $?*[X : A] : B(X)$ for it. When translating to HOL, we have two options: We can translate to HOL with or without simply-typed product types.

If our target language has simple products, we can use the usual dependency erasure and put $\overline{\Sigma x : A.B(x)} = \overline{A} \times \overline{B}$ as well as $(\Sigma x : A.B(x))^*\ t\ u = A^*\ t_1\ u_1 \wedge B^*\ t_2\ u_2$. We have not proved or implemented this, but we expect the soundness and completeness results and our theorem proving system to generalize seamlessly.

Another option is to prepare to use a HOL theorem prover that does not support simple products. In that case, we can use the fact that, in the presence of PER-based equality, product types are definable. The basic idea is to represent pairs (a, b) as binary predicates that are true in exactly one place. This translation depends on the presence of a choice operator in HOL (which is typically the case in provers for HOL) to translate the two projections. Concretely, we would put

$$\overline{\Sigma x : A.B(x)} = \overline{A} \to \overline{B} \to o \qquad \overline{(a, b)} = \lambda x : \overline{A}.\lambda y : \overline{B}.x =_{\overline{A}} a \wedge y =_{\overline{B}} b$$

$$\overline{t_1} = \epsilon x : \overline{A}.\ \exists y : \overline{B}.\ \overline{t}\ x\ y \qquad \overline{t_2} = \epsilon x : B.\ \exists y : A.\ \overline{t}\ x\ y$$

$$(\Sigma x : A.B(x))^*\ t\ u = t =_{\overline{A} \to \overline{B} \to o} u \wedge \exists a : \overline{A}, b : \overline{B}.\forall x : \overline{A}, y : \overline{B}.t\ x\ y \Leftrightarrow A^*\ a\ x \wedge B^*\ b\ y$$

This translation is more involved and the treatment of equality may stress theorem provers. But it enables using HOL-equality as a necessary condition for DHOL equality, which hints at optimization potential.

7.2 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 conduct a similar analysis for declaration in a *context*. The following table shows the possible combinations:

	declaration of		
depending on	type symbol	term symbol	global 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 $\Rightarrow$

Contrary to theories, contexts must be small in the sense that declarations must not depend on type variables. This ensures that we can formulate the semantics without requiring a hierarchy of universes or similar. And just like for theories several combinations are problematic or definable.

But one combination remains open: type variables could depend on term variables. This would allow declaring, e.g., $\alpha : \mathtt{type}$, $\beta : \alpha \to \mathtt{type}$. Dependent type variables present no fundamental problems but require a more complex presentation. Therefore, we have relegated the discussion of this feature to a separate section. A simple practical example that requires this is heterogeneous lists.

▶ **Example 11.** 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

$$\begin{aligned}
&\mathtt{Hlist} : \Pi_{n:\mathtt{nat}}(\mathtt{fin}\ n \to \mathtt{type}) \to \mathtt{type}\\
&\mathtt{hlist} : \Pi_{n:\mathtt{nat}}\Pi_{L:\mathtt{fin}\ n \to \mathtt{type}}(\mathtt{fin}\ n \to L\ n) \to \mathtt{Hlist}\ n\ L\\
&\mathtt{hget} : \Pi_{n:\mathtt{nat}}\Pi_{L:\mathtt{fin}\ n \to \mathtt{type}}\mathtt{Hlist}\ n\ L \to \Pi_{i:\mathtt{fin}\ n} L\ i
\end{aligned}$$

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

Grammar and Rules We can allow dependent type variables by using the following grammar:

$$\begin{array}{lll}
T & ::= & . \mid T, a : \Pi_\Gamma \mathtt{type} \mid T, c : \Pi_\Gamma A \mid T, \triangleright \Pi_\Gamma F\\
K & ::= & \mathtt{type} \mid \Pi_{x:A} K\\
k & ::= & A \mid \lambda_{x:A} k\\
\Gamma & ::= & \circ \mid \Gamma, \alpha : K \mid \Gamma, x : A \mid \Gamma, \triangleright F\\
\gamma & ::= & \bullet \mid \gamma, k \mid \gamma, t \mid \gamma, \checkmark\\
A, B & ::= & a\,\gamma \mid \alpha\, t\ \dots\ t \mid \Pi_{x:A} B \mid o\\
t, u & ::= & c\,\gamma \mid x \mid \lambda_{x:A} t \mid t\,u \mid t \Rightarrow u \mid t =_A u
\end{array}$$

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

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

Adjusting Rule CtxType and Rule SubType 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 $\triangleright F$ as parameters of theory declarations. This is not necessary for dependent type variables and whether to allow or forbid is a remaining degree of freedom in the language design. But no matter the design choice, the notation becomes much simpler if we allow it at least in the grammar and the rules. 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.

Translation We need to adjust three cases in the translation:

DHOL		Translation
type variable declaration	$\alpha : \Pi_\Delta \texttt{type}$ where $\Delta = \vec{x} : \vec{A}$	$\alpha\ :\ \texttt{type}$ $\alpha^* : \overline{\Delta}^{ye} \to \alpha \to \alpha \to o$, $\triangleright \overline{\Delta} \Rightarrow \mathrm{PER}(\alpha^*\ \vec{x})$
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

We extended Dependently-typed Higher-order Logic by polymorphism. In addition to extending the syntax and the rules, we proposed a translation to HOL which gives an efficient automated reasoning procedure for the calculus. We demonstrated the practical usability of the language and its automation by encoding several automated reasoning problems that combine higher-order reasoning with dependent types and polymorphism and showing that the translation can solve many of them.

Future work includes improving the foundation's automation capabilities. We primarily want to do this by dedicated automated reasoning rules for polymorphic DHOL similar to Niederhauser et al. [13]. Furthermore, a larger case study involving dependently typed examples from interactive proof assistant systems like Coq and Agda would allow checking if polymorphic DHOL constitutes a useful intermediate language for proof assistants to call ATP services. There is also a strong similarity between our PERs and the interpretations of types as relations for dependent types by Bernardy et al. [2]. Investigating the relationship between them might prove interesting.

Finally, in parallel work by colleagues, DHOL is being extended with subtyping. Now that both developments have stabilized, we want to join the developments. While just merging the language extensions is presumably straightforward, their combination 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 soundness/completeness proofs and the design of a (sub)type-checking algorithm will be in that case.

References

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

2 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.

3 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.

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

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

6 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.

7 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.

8 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.

9 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.

10 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.

11 C. Kaliszyk, F. Rabe, and G. Sutcliffe. TH1: The TPTP Typed Higher-Order Form with Rank-1 Polymorphism. In P. Fontaine, S. Schulz, and J. Urban, editors, *Workshop on Practical Aspects of Automated Reasoning*, pages 41–55, 2016.

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

13 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.

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

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

16 F. Rabe. How to Identify, Translate, and Combine Logics? *Journal of Logic and Computation*, 27(6):1753–1798, 2017.

17 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.

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

19 C. Rothgang, C. Rabe, and C. Benzmüller. Theorem proving in dependently-typed higher-order logic – extended preprint, 2023. doi:10.48550/arXiv.2305.15382.

20 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.

21 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.

22 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.

23 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.

24 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 Soundness

The proof given for the soundness of the translation is taken from Rothgang et al. [19] 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 challenge regarding soundness of the translation 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-translation 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 12.** *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.

A.1 Type-wise injectivity of the translation

Compared to the original formulation of Rothgang et al. [20] 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 13.** *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 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. ◀

$$
\begin{aligned}
norm[\overline{t}] &:= t \\
norm[norm[s]] &:= norm[s] \\
norm[A^*\ s] &:= \lambda y : \overline{A}.A^*\ s\ y \\
norm[A^*] &:= \lambda x : \overline{A}.\lambda y : \overline{A}.A^*\ x\ y \\
norm[{\color{red}c\ \delta^{ye}}] &:= {\color{red}c\ \delta^{ye}} \\
norm[x] &:= x \\
norm[f\ t] &:= norm[f]\ norm[t] \\
norm[\lambda x : C.t] &:= \lambda x : C.norm[t] \\
norm[s =_{\overline{A}} t] &:= A^*\ s\ t \\
norm[s \Rightarrow t] &:= norm[s] \Rightarrow norm[t] \\
norm[\forall x : \overline{A}.A^*\ x\ x\ \Rightarrow G] &:= \forall x, y : \overline{A}.A^*\ x\ y \Rightarrow G \\
\text{If } F \text{ not of shape } A^*\ \_\ \_ \Rightarrow \_ \text{ or } \forall x' : \overline{A}.A^*\ x\ x' \Rightarrow \_ : & \\
norm[\forall x : \overline{A}.F] &:= norm[\forall x : \overline{A}.A^*\ x\ x \Rightarrow F]
\end{aligned}
$$

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

A.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 [19]. $sRed(t)$ 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 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 14.** *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*

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

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 :

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

Figure 6 Definition of $sRed(t)$.

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 15 (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 13) 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 [19] 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. ◀

A.3 Translation of HOL proofs into DHOL proofs

Finally, we show the soundness of the translation. As previously, we give the general outline of the proof and refer to [19] for details.

Proof. According to Lemma 14 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 [19]. ◀