Polymorphism Meets DHOL

Rhea 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. We develop the syntax and semantics of polymorphic DHOL and extend the translation accordingly. We implement the translation in the logic-embedding tool and evaluate it on a range of TPTP formalizations. The logic-embedding tool, together with an off-the-shelf HOL theorem prover easily creates a PDHOL theorem prover for experimenting.

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.FSCD.2026.27

Related Version *Extended Version*: `http://arxiv.org/abs/2605.00295`

Supplementary Material *Software*: `https://zenodo.org/records/19931555`

Funding Rabe was supported by the FAUstairs project (see `https://www.faustairs.fau.de/`), funded by the Stiftung Innovation in der Hochschullehre under grant StIL:1001-3096. Kaliszyk and Ranalter were supported by the Renaissance Philanthropy grant DEEPER.

Acknowledgements We want to thank Alexander Steen for his work on the logic-embedding tool, which allowed for effective integration of our translation.

1 Introduction

Setting. Monomorphic dependently typed higher-order logic (DHOL) was introduced in [28, 29]. It combines the simplicity of higher-order logic (HOL) [5, 11], 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, 19, 9], 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 [14] but has the benefit of being close 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 [29], 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 [17].

Maybe surprisingly, even though the PER semantics of dependent types has been known for some time [1], it is not widely known and the proofs are complex. Indeed, as noticed during the preparation of [30], the literature contains examples of similar translations that were incorrectly assumed to be sound/complete. Therefore, [29] focused on the simplest possible language and left out a number of common language extensions for HOL.

Outline and Contribution. We introduce PDHOL as an extension of DHOL with shallow (ML-style, rank 1) polymorphism and subtype definitions. This greatly extends the expressivity of DHOL while retaining its simple definition and strong automation support.

To simplify the presentation, we split our exposition of PDHOL: Section 2 and 3 present the syntax and proof system for basic polymorphism, and we relegate the presentation of two advanced features to Section 6 and 7. In Section 4, we describe the translation from PDHOL to polymorphic HOL (PHOL) and show it is sound and complete.

PDHOL has been adopted as a TPTP standard in the form of the new DHF dialect [25]. Using its TPTP representation, we have implemented our translation as a PDHOL→PHOL logic-embedding. This allows using any off-the-shelf PHOL theorem prover as a PDHOL theorem prover with negligible overhead.

We apply PDHOL in Section 5 to give elegant formalizations of practically important problems, reaching a level that is typically only found in the rich languages of *interactive* provers and not in simple logics like PDHOL that provide *automated* proof support. For example, Section 5 shows how polymorphism and dependent types allow tracking key invariants in common polymorphic data structures, e.g., the dimensions and matrices or the black height of red-black trees. In addition to plain type variables (e.g., $\Pi_{\alpha:\mathtt{type}}\ldots$), PDHOL allows *dependent* type variables, i.e., type variables that depend on terms (e.g. $\Pi_{\alpha:\mathtt{nat}\to\mathtt{type}}\ldots$). We spell this out in Section 6 and use it for formalizations that require abstracting over *families* of types, such as heterogeneous lists. Finally, in HOL, the core feature for building conservative extensions of theories is the subtype definition principle: a fresh type is introduced and axiomatized to be isomorphic to a subtype of an existing type. In Section 7, we extend this feature to the dependently-typed case. We then show how dependent types, polymorphism, and subtype definitions together allow formalizing complex concepts such as the type of homomorphisms between algebraic structures.

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 adjusted to recover HOL as a fragment.

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

We use the usual notations for the logical connectives and binders, which are all definable [2], and we write $A \to B$ as usual. To avoid case distinctions, we will occasionally merge lists $\Pi_{\vec{\alpha}}\Pi_{\vec{x}:\vec{A}}$ of type and term variables into a single list Π_Δ for a context Δ. Similarly, we may merge the list $\vec{A}\ \vec{t}$ of type and term arguments into a single substitution δ. We also use the following abbreviations for sequences of expressions:

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

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

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

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

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 now wants 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 $\triangleright \Phi$. $\circ$ denotes the empty context.

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 $\checkmark$ in the corresponding place of a substitution. If we extended the formulation with a language of proofs, the $\checkmark$ would be replaced with an appropriate proof expression. $\bullet$ denotes the empty substitution. 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 this example, we also add two assumptions resulting in the context: $n : \texttt{nat}, m : \texttt{nat}, \triangleright n =_{\texttt{nat}} 2, \triangleright m =_{\texttt{nat}} 3$.

A well-formed substitution γ for this context is $2, \mathit{suc}\ 2, \checkmark, \checkmark$. $E[\gamma]$ would then be $\texttt{cons nat}\ 1\ 2\ (\texttt{cons nat}\ 0\ (\texttt{suc}\ 2)\ (\texttt{nil nat})) : \texttt{vec nat}\ 2$. The $\checkmark$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) $\triangleright \Phi$. . denotes the empty theory.

Contrary to contexts, declarations in theories may depend on arguments, and this is the mechanism how both monomorphic DHOL [29] 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. We exclude the remaining two cases: local assumptions as parameters of type/term symbols. These would effectively allow declaring partial functions and partial type symbols. We exclude those because, to our knowledge, they cannot be translated to HOL in a sound and complete way.

Types and Terms. Types $A, B, \ldots$ and terms $t, u, \ldots$ 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 $\Phi \Rightarrow \Psi$.

Note that equality of *terms* is a Boolean term. Equality of *types* is not – it will be a judgment in the type system.

Dependent implication means that in $F \Rightarrow G$, the well-formedness derivation for 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. We believe dependent implication is not definable from equality and therefore make it an additional primitive from which corresponding dependent variants of conjunction and disjunction can be defined. 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.

3 Inference System

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 $\rightarrow$. 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.

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 $a : \Pi_{\vec{\alpha}} \Pi_{\vec{x}:\vec{A}} \texttt{type}$ in a theory resp. $\alpha : \texttt{type}$ in a context. The main difference is that the

Name	Judgment	Intuition
theories	$\vdash T\ \texttt{Thy}$	T is a well-formed theory
contexts	$\vdash_T \Gamma\ \texttt{Ctx}$	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.

Well-formed theories T

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

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

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

Contexts Δ and substitutions δ for them

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

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

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

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

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

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

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

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

Figure 2 DHOL Rules for theories and contexts.

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 $\Delta, \alpha : \texttt{type}$ by adding a type substitute A for α.

Booleans: type formation, terms for equality and implication

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

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

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 [29].

Figure 3 DHOL Rules for Types and Terms.

The rules in Fig. 3 cover the rules for expressions. We only point out some specialties: 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 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. 6 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 $A^*\ \bar{t}\ \bar{t}$ captures the PDHOL-typing judgment $t : A$. Critically, term equality $s =_A t$ is translated to $A^*\ \bar{s}\ \bar{t}$.

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 3** (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 t**y**pe variables.*
- Γ^{ye} *is like* Γ *but contains only the type and t**e**rm 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 t**y**pe substitutes.*
- γ^{ye} *is like* γ *but contains only the type and t**e**rm substitutes.*
- *If* Γ *consists of 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 *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 $\bar{t}$ and a statement $A^*\ \bar{t}\ \bar{t}$. 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 must be removed if we want to bind $\overline{\Gamma}$ in a PHOL-type or term symbol declaration.

The abbreviations $\Gamma \to \texttt{type}$, $\Gamma \to A$, 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
α : `type`	α : `type`, $\alpha^* : \alpha \to \alpha \to o$, $\triangleright\ \mathrm{PER}(\alpha^*)$	A	$\overline{A}$, A^*, ✓
$x : A$	$x : \overline{A}$, $\triangleright\ A^*\ x\ x$	t	$\overline{t}$, ✓
$\triangleright\ \Phi$	$\triangleright\ \overline{\Phi}$	✓	✓

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 ✓ 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. 6.

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 4.** We use the expression E from Ex. 2 to create E_2, 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., $\overline{\text{(}\texttt{vec nat}\ 2\text{)}}\ \overline{1}\ \overline{E}\ \overline{\text{(...)}}$. 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 contain contexts. This is where the abbreviations from Def. 3 are used:

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

▶ **Example 5.** 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 $\triangleright\, \Pi_\alpha \forall \alpha^* : \alpha \to \alpha \to o. \forall n : \texttt{nat}. PER(\alpha^*) \Rightarrow PER(\texttt{vec}^*\ \alpha\ \alpha^*\ n)$. 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 may 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 [29]: 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 $\overline{\delta}^{ye}$.

Properties of the Translation.

▶ **Theorem 6** (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 6 is straightforward and performed by induction over derivations.

▶ **Theorem 7** (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 $u =_{\Pi x:A \ s.A \ s} v$ between them would be ill-typed. The erased equality however is not only well-typed but provable.

The original proof for DHOL [29] 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 the extended preprint.In the remainder, we will call the combination of these properties *well-behavedness* of the translation.

5 Implementation and Case Studies

To evaluate PDHOL and obtain a theorem prover for it, we implemented our translation as part of the logic-embedding tool by Steen [31]. This enables discharging PDHOL proof obligations by existing PHOL ATPs. Our tool, as well as the dialect used to formalize this set of problems, has since been adopted as TPTP standard in the form of the new DHF dialect and the DT2H2X prover on SystemOnTPTP [25].

As outlined in Section 4, the translation requires the problem to be well-typed. Therefore, the use of a PHOL ATPs is only sound if we first obtain a type-checker for PDHOL. Because type-checking for PDHOL is undecidable, our tool transforms each conjecture F into $n + 1$ PHOL conjectures: n type-checking obligations (TCOs) that establish the well-typedness of the problem plus F itself.

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 53 problems to evaluate and test our implementation on. These include 18 problems that are polymorphic versions of the examples created in [17] for DHOL, and 17 problems that experiment with the impact of instantiating type variables in those conjectures. 11 more problems are lemmas that occur during an inductive proof that the reversal function on red-black trees is an involution (see below). Additionally, there are one formalization each of the zip function as used in functional programming and of finite sets. The remaining problems focus on the formalization capabilities of PDHOL as presented in Section 7, in the form of the group problem. This is not yet supported by the implementation.

Three PHOL ATP systems were combined with our translator to create PDHOL provers: Vampire v4.7 [26] in portfolio mode, Leo-III v1.6 [32] and Zipperposition v2.1 [7] in higher-order and portfolio mode. The reported times do not include problem translation time, which

amounted to 198±33ms. All experiments ran on an Intel Core i5-6200U CPU at 2.30GHz and 8 GB of RAM. All files and binaries, including the group example from Section 7, can be found in the supplementary material.

In the following, we give an overview of those problems. The names of the TPTP files as can be found in the supplements are given in parentheses, with asterisks acting as wildcards.

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` : $\Pi_\alpha \Pi_{n:\texttt{nat}} \texttt{vec}\,\alpha\,n \to \texttt{fin}\,n \to \alpha$ for safe access to list elements. The conjectures we investigated expressed properties of the append function – associativity (`list-app-assoc*`) and the identity of nil (`list-app-nil*`) – as well as the involution property of the reverse function (`list-rev-nil*`). These problems generated more complex TCOs than the red-black trees, due to the arithmetic needed to show that lengths of lists line up after appending. From this, matrices can then be represented by nesting vectors. We formalize, e.g., matrix transposition as `transpose` : $\Pi_\alpha \Pi_{m,n:\texttt{nat}} \texttt{vec}\,(\texttt{vec}\,\alpha\,m)\,n \to \texttt{vec}\,(\texttt{vec}\,\alpha\,n)\,m$, and formalize commutativity of matrix addition (`inst-poly-matrix-add-com`) and element-wise multiplication (`inst-poly-matrix-mul-com`).

This leverages both dependent types and polymorphism to concisely represent complex data structures in a way that guarantees their dimensional invariants.

Another formalization leveraging the list basis, is our formalization of the zip function. Note that Haskell's default List package[1] includes distinct functions `zipN` for $N \in \{3, \ldots, 7\}$ arguments due to the lack of dependent types. Our formalization (`zip`) acts like an arbitrary dimensional zip function.

Adding a formalization of the finite set (`finite-type`) allows to have a unfailing getter function. This example is set up in a way that type-checks, but where the types do not make sense: They would allow getting an element of the empty list. Indeed, as can be seen in the results in Appendix B, Leo-III returns the SZS Status "Contradictory Axioms".

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 node 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 : $\Pi_\alpha \Pi_{b:o} \Pi_{n:\texttt{nat}} \texttt{type}$ 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.

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 [17, 24] we split the problem into smaller sub-problems, ensuring well-typedness.

The problem is split into a base case for red (`rbt-rev-invol-base-RTLeaf`, the prefix `rbt-rev-invol` will be omitted from here on) and black (`-base-BTLeaf`) 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`).

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

Next are the step cases: Again there is a step case for red (`-RTStep`) and for black (`-BTStep`) leaves. While the red step case is easy, the black is challenging due to the raised complexity of black nodes. While possible to prove problem `rbt-rev-invol-BTStep` directly, we found three sub-steps (`-subBTStep[1-3]`) and added them as axioms to help the ATP system along (`-composedBTStep`).

While the instanced induction scheme (`-indinst`) timed out, type checking was successful. Assuming the instanced induction allows to proof the main conjecture (`rbt-rev-invol`) 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 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 generated TCOs can be proven.

Notably, the red-black tree examples generate very few TCOs (4 in total) compared to the list examples, where the majority of examples generated 1+ TCOs. This is because most constraints, thanks to the efficient formulation provided by the dependent types, reduce to reflexivity.

Results and Discussion. The results of our experiments can be found in Appendix B. A significant number of statements can be proven. This is notably without any fine-tuning of the provers to the specific challenges translated PDHOL problems pose. This illustrates the viability of the translate-and-proof method and, in particular, shows that PDHOL retains automation support while providing a rich addition to expressivity.

Notably, the results support that undecidable type-checking is a price worth paying – no TCOs were left unproven. Frequently the TCOs were trivial and required no call to the ATP systems. While this will not be the case for every problem, it is promising to see that for these common data-structures type-checking is efficient.

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. [17], indicating that performance improvements can be obtained by building DHOL-specific and/or induction-aware provers, or splitting problems in a way that helps the ATP system with the induction arguments.

6 Dependent Type Variables

For simplicity, we have so far not described the feature of type variables depending on term variables as in α : `type`, $\beta : \alpha \to$ `type`. The following table shows the possible dependencies for variables in *contexts* akin to how we did it in Section 2 for *theories*:

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

Contrary to global declarations in theories, variable declarations must not bind type variables. This ensures that contexts are small and we can formulate the semantics without requiring a hierarchy of universes or similar. After adding the final feature of dependent type variables, the grammar of PDHOL becomes:

T	$::=$	$. \mid T, a : \Pi_\Gamma \texttt{type} \mid T, c : \Pi_\Gamma A \mid T, \triangleright \Pi_\Gamma F$			
K	$::=$	$\texttt{type} \mid \Pi_{x:A} K$	k	$::=$	$A \mid \lambda_{x:A} k$
Γ	$::=$	$\circ \mid \Gamma, \alpha : K \mid \Gamma, x : A \mid \Gamma, \triangleright F$	γ	$::=$	$\bullet \mid \gamma, k \mid \gamma, t \mid \gamma, \checkmark$
A, B	$::=$	$a\,\gamma \mid \alpha\, t\ \ldots\ t \mid \Pi_{x:A} B \mid o$	t, u	$::=$	$c\,\gamma \mid x \mid \lambda_{x:A} t \mid t\,u \mid t \Rightarrow u \mid t =_A u$

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.

The grammar above describes the arguments of global declarations as a single context (as in Π_Γ) rather than a list of type variables followed by a list of term variables (as in $\Pi_{\vec{\alpha}} \Pi_{\vec{x}:\vec{A}}$). This is necessary because term variables $x : A$ may now occur in a type variable declaration $\alpha : A \to \texttt{type}$. Therefore, we can no longer assume that all type variables are bound before all term variables. The typing rules for theories remain unchanged except that we to modify the syntactic restrictions on the context in Rule ThyType, Rule ThyCon, and Rule ThyAss. For the same reason as for global declarations in theories, we do not allow local assumptions in Γ.

▶ **Example 8.** 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 $\alpha : \texttt{fin}\ n \to \texttt{type}$ that provides the types of the n components. This uses finite types $\texttt{fin}\ n = \{0, \ldots, n-1\}$, which we can declare by

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

Then heterogeneous lists can be declared as

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

Note how type and term variables alternate, e.g., in the declaration of `hlist`.

The translation rules for declarations in a theory remain unchanged except for generalizing the syntactic restrictions on the contexts and substitutions. We only need to adjust the three cases below for the translation of type variables. The invariants are the same.

	PDHOL	PHOL
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

7 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 [12]. 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

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

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 , or (ii) translate it to a PHOL definition and then apply the usual PHOL elaboration. Consequently, the translation remains well-behaved.

▶ **Example 9** (Algebraic Structures)**.** Let us assume we have already formalized types for algebraic structures, such as a type $\mathtt{group} : \Pi_\alpha \mathtt{type}$ with (among others) a selector $\mathtt{op} : \Pi_\alpha \alpha \to \alpha \to \alpha$. (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 $\mathtt{ishom} : \Pi_{\alpha,\beta}\, \mathtt{group}\,\alpha \to \mathtt{group}\,\beta \to (\alpha \to \beta) \to o$, and use that to define the

polymorphic and dependent type of group homomorphisms `hom` by

$$\texttt{ishom}\,\alpha\,\beta\,G\,H\,m =_o \forall x, y : \alpha.m\,(\texttt{op}\,G\,x\,y) =_\beta \texttt{op}\,H\,(m\,x)\,(m\,y)$$
$$\texttt{hom} := \lambda_{\alpha:\texttt{type},\,\beta:\texttt{type},\,G:\texttt{group}\,\alpha,\,H:\texttt{group}\,\beta}(\alpha \to \beta)|\texttt{ishom}\,\alpha\,\beta\,G\,H$$

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:

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

This example illustrates the expressiveness of the PDHOL language while assuaging lingering doubts about undecidable type checking: Although its a complex formulation we cannot proof directly, similar to examples in Section 5, the TCOs are discharged easily.

8 Conclusion and Related 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 [16], 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 [11, 18, 13] and automated [33, 26, 34]. 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 [20] provides a combination of dependent types and shallow polymorphism similar to PDHOL. It combines native decision procedures with interaction and automation but it is too expressive for easy translation into standard ATP tools. [22] gives a model-theoretical semantics of DHOL that includes polymorphism. 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 [8] 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: [10] experimented with a translation of LF into hereditary Harrop formulas, a simply-typed meta-logic. Similarly, Jacobs and Melham [15] give a translation of dependent type theory into higher-order logic. Both translate dependent types to unary predicates that serve as type guards.

Such translations are not necessarily sound and complete. This is because 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. This motivated the use of PERs, which subsume refinements and quotients, and 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 [23]. Another example are parametricity arguments in polymorphic type theories such as [3]. For example, recently, [21] 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. [17] 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 [27]. 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 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 Simon Cruanes. *Extending Superposition with Integer Arithmetic, Structural Induction, and Beyond. (Extensions de la Superposition pour l'Arithmétique Linéaire Entière, l'Induction Structurelle, et bien plus encore)*. PhD thesis, École Polytechnique, Palaiseau, France, 2015. URL: https://tel.archives-ouvertes.fr/tel-01223502.

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

9 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. doi:10.1007/978-3-319-21401-6_26.

10 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.

11 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.

12 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.

13 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. doi:10.1007/BFB0031814.

14 M. Hofmann. *Extensional Constructs in Intensional Type Theory*. CPHC/BCS distinguished dissertations. Springer, 1997.

15 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.

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

17 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.

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

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

20 S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, pages 748–752. Springer, 1992. doi:10.1007/3-540-55602-8_217.

21 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. doi:10.1145/3571235.

22 F. Rabe. Semantics for Dependently-Typed HOL. In A. Biere, C. Lutz, and S. Negri, editors, *Automated Reasoning*. Springer, 2026. to appear.

23 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.

24 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. doi:10.29007/2V8H.

25 Daniel Ranalter, Cezary Kaliszyk, Florian Rabe, and Geoff Sutcliffe. The dependently typed higher-order form for the TPTP world. In René Thiemann and Christoph Weidenbach, editors, *Frontiers of Combining Systems - 15th International Symposium, FroCoS 2025, Reykjavik, Iceland, September 29 - October 1, 2025, Proceedings*, volume 15979 of *Lecture Notes in Computer Science*, pages 287–305. Springer, 2025. doi:10.1007/978-3-032-04167-8_16.

26 A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AI Communications*, 15:91–110, 2002. URL: http://content.iospress.com/articles/ai-communications/aic259.

27 C. Rothgang and F. Rabe. Subtyping in dependently-typed higher-order logic. In R. Thiemann and C. Weidenbach, editors, *Frontiers of Combining Systems - 15th International Symposium, FroCoS 2025, Reykjavik, Iceland, September 29 - October 1, 2025, Proceedings*, volume 15979 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2025. doi:10.1007/978-3-032-04167-8_6.

28 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. doi:10.1007/978-3-031-38499-8_25.

29 C. Rothgang, F. Rabe, and C. Benzmüller. Dependently-Typed Higher-Order Logic. *ACM Transactions on Computational Logic*, 27(1), 2025.

30 M. Southern. *A Framework for Reasoning about LF Specifications*. PhD thesis, University of Minnesota, 2021.

31 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. URL: https://ceur-ws.org/Vol-3201/paper13.pdf.

32 A. Steen and C. Benzmüller. The higher-order prover Leo-III. In G. De Giacomo, A. Catalá, B. Dilkina, M. Milano, S. Barro, A. Bugarín, and J. Lang, editors, *European Conference on Artificial Intelligence*, pages 2937–2938. IOS Press, 2020. doi:10.3233/FAIA200462.

33 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.

34 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. doi:10.1007/S10817-021-09613-Z.

A Red-Black Tree in PDHOL

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

$$\texttt{color} : \texttt{type} \quad \texttt{black} : \texttt{color} \quad \texttt{red} : \texttt{color}$$

$$\texttt{nat} : \texttt{type} \quad 0 : \texttt{nat} \quad \texttt{suc} : \texttt{nat} \to \texttt{nat}$$

$$\texttt{tree} : \Pi_\alpha \Pi_{c:\texttt{color},n:\texttt{nat}} \texttt{type} \quad \texttt{leaf} : \Pi_\alpha \texttt{tree}\ \alpha\ \texttt{black}\ 0$$

$$\texttt{red\_tree} : \Pi_\alpha \Pi_{n:\texttt{nat}} \texttt{tree}\ \alpha\ \texttt{black}\ n \to \alpha \to \texttt{tree}\ \alpha\ \texttt{black}\ n \to \texttt{tree}\ \alpha\ \texttt{red}\ n$$

$$\texttt{black\_tree} : \Pi_\alpha \Pi_{c_1:\texttt{color},c_2:\texttt{color},n:\texttt{nat}} \texttt{tree}\ \alpha\ c_1\ n \to \alpha \to \texttt{tree}\ \alpha\ c_2\ n \to \texttt{tree}\ \alpha\ \texttt{black}\ (\texttt{suc}\ n)$$

$$\texttt{rev} : \Pi_\alpha \Pi_{c:\texttt{color},n:\texttt{nat}} \texttt{tree}\ \alpha\ c\ n \to \texttt{tree}\ \alpha\ c\ n$$

$$\triangleright\ \forall \alpha : \texttt{type}. \texttt{rev}\ \alpha\ \texttt{black}\ 0\ (\texttt{leaf}\ \alpha) = \texttt{leaf}\ \alpha$$

$$\triangleright\ \forall \alpha : \texttt{type}, x : \alpha, n : \texttt{nat}, c_1 : \texttt{color}, c_2 : \texttt{color}, t_1 : \texttt{tree}\ \alpha\ c_1\ n, t_2 : \texttt{tree}\ \alpha\ c_2\ n.$$
$$\texttt{rev}\ \alpha\ \texttt{black}\ (\texttt{suc}\ n)\ (\texttt{black\_tree}\ \alpha\ n\ c_1\ c_2\ t_1\ x\ t_2) =$$
$$\texttt{black\_tree}\ \alpha\ n\ c_2\ c_1\ (\texttt{rev}\ \alpha\ c_2\ n\ t_2)\ x\ (\texttt{rev}\ \alpha\ c_1\ n\ t_1)$$

$$\triangleright\ \forall \alpha : \texttt{type}, x : \alpha, n : \texttt{nat}, t_1 : \texttt{tree}\ \alpha\ \texttt{black}\ n, t_2 : \texttt{tree}\ \alpha\ \texttt{black}\ n.$$
$$\texttt{rev}\ \alpha\ \texttt{red}\ n\ (\texttt{red\_tree}\ \alpha\ n\ t_1\ x\ t_2) =$$
$$\texttt{red\_tree}\ \alpha\ n\ (\texttt{rev}\ \alpha\ \texttt{black}\ n\ t_2)\ x\ (\texttt{rev}\ \alpha\ \texttt{black}\ n\ t_1)$$

From this various in-between steps, e.g. base cases and instanced induction schemes, were shown. In the interest of readability these results, as well as the induction axiom, are omitted here. Taking everything together, this allows us to proof the following involution lemma:

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

B Experimental Results

TCOs with the entry "trivial" were equalities that could be dismissed by reflexivity. "TO" represents a timeout, CA the SZS Status ContradictoryAxioms. `-TI` stands for *T*ype *I*nstantiated and represent our experiments where type arguments are applied to polymorphic axioms and conjectures.

	Time to prove				Time to prove		
File name	Vampire	Leo	Zipper	File name	Vampire	Leo	Zipper
rbt-rev-invol	13.48	TO	TO	list-app-assoc	13.37	TO	TO
↳TCO	trivial	trivial	trivial	↳TCO	0.02	0.70	0.35
-base	33.01	TO	TO	-TI	TO	TO	TO
↳TCO	trivial	trivial	trivial	↳TCO	0.02	0.66	0.38
-base-BTLeaf	0.08	TO	TO	-TI_alt	14.66	TO	TO
↳TCO	trivial	trivial	trivial	↳TCO	0.02	0.81	0.36
-base-RTLeaf	8.03	TO	TO	-base	5.18	TO	TO
↳TCO	trivial	trivial	trivial	↳TCO	0.02	0.65	<0.01
-base-lemma	18.01	TO	TO	-base-TI	10.83	TO	TO
↳1.TCO	0.02	0.64	<0.01	↳TCO	0.02	0.47	<0.01
↳2.TCO	0.02	0.67	<0.01	-indinst	TO	TO	TO
-subBTStep1	0.46	2.99	TO	↳1.TCO	0.02	0.54	<0.01
↳TCO	trivial	trivial	trivial	↳2.TCO	0.02	0.72	1.81
-subBTStep2	0.22	47.48	TO	↳3.TCO	0.02	0.90	1.29
↳TCO	trivial	trivial	trivial	↳4.TCO	0.02	0.72	1.71
-subBTStep3	0.08	2.80	2.32	-indinst-TI	TO	TO	TO
↳TCO	trivial	trivial	trivial	↳1.TCO	0.02	0.56	<0.01
-composedBTStep	10.62	TO	TO	↳2.TCO	0.02	0.80	1.60
↳TCO	trivial	trivial	trivial	↳3.TCO	0.02	0.78	1.34
-BTStep	35.49	TO	TO	↳4.TCO	0.02	0.76	1.70
↳TCO	trivial	trivial	trivial	-indstep1	10.56	TO	TO
-RTStep	5.67	TO	TO	↳1.TCO	0.02	0.56	0.37
↳TCO	trivial	trivial	trivial	↳2.TCO	0.02	1.93	0.69
-indinst	TO	TO	TO	-indstep1-TI	TO	TO	TO
↳TCO	trivial	trivial	trivial	↳1.TCO	0.02	0.63	0.50
bad_tree	TO	TO	TO	↳2.TCO	0.02	1.43	0.49
↳1.TCO	TO	TO	TO	-indstep2	10.58	TO	TO
↳2.TCO	TO	TO	TO	↳1.TCO	0.02	0.64	0.40
list-app-nil	1.39	TO	TO	↳2.TCO	0.02	0.67	0.68
↳TCO	0.02	0.65	<0.01	indstep2-TI	TO	TO	TO
-base	0.02	TO	2.18	↳1.TCO	0.02	0.64	0.39
↳TCO	0.02	0.63	<0.01	↳2.TCO	0.02	1.20	0.39
-indinst	TO	TO	TO	-indstep3	0.4	TO	TO
↳1.TCO	0.02	0.60	<0.01	↳1.TCO	0.02	0.65	0.33
↳2.TCO	0.02	0.64	<0.01	↳2.TCO	0.02	8.60	0.09
↳3.TCO	0.02	0.67	<0.01	↳3.TCO	0.02	8.65	0.61
↳4.TCO	0.02	0.72	<0.01	indstep3-TI	6.72	TO	TO
-indstep	5.27	TO	TO	↳1.TCO	0.02	0.62	0.35
↳1.TCO	0.02	0.55	<0.01	↳2.TCO	0.02	1.92	0.02
↳2.TCO	0.02	0.52	<0.01	↳3.TCO	0.02	1.41	0.40
list-rev-invol	11.15	TO	TO	-indstep4	1.43	2.69	TO
↳TCO	trivial	trivial	trivial	↳TCO	0.01	0.58	0.01
-TI	8.55	TO	TO	-indstep4-TI	6.72	TO	TO
↳TCO	trivial	trivial	trivial	↳TCO	0.02	0.53	0.02
-step1	0.09	TO	TO	-indstep5	TO	TO	TO
↳TCO	0.02	0.82	0.25	↳1.TCO	0.02	0.70	0.35
-step1-TI	6.60	TO	TO	↳2.TCO	0.03	1.22	0.74
↳TCO	0.02	0.75	0.19				
-step2	5.40	TO	TO				
↳1.TCO	0.02	0.56	0.35				
↳2.TCO	0.02	2.26	<0.01				

-step2-TI	6.63	TO	TO
↳1.TCO	0.02	0.80	0.29
↳2.TCO	0.01	0.61	<0.01
-step3	5.31	TO	TO
↳1.TCO	0.02	0.49	<0.01
↳2.TCO	0.02	0.37	<0.01
-step3-TI	6.60	TO	TO
↳1.TCO	0.02	0.56	<0.01
↳2.TCO	0.02	0.60	<0.01
-step4	0.18	TO	TO
↳TCO	0.02	0.56	<0.01
-step4-TI	6.60	TO	TO
↳TCO	0.02	0.56	<0.01
-step5	0.02	TO	1.12
↳TCO	trivial	trivial	trivial
-step5-TI	6.62	TO	TO
↳TCO	trivial	trivial	trivial
-step5-TI_alt	0.03	TO	1.27
↳TCO	trivial	trivial	trivial

-indstep5-TI	TO	TO	TO
↳1.TCO	0.02	0.71	0.35
↳2.TCO	0.02	1.45	0.41
indstep5-TI_alt	0.18	TO	TO
↳1.TCO	0.02	0.63	0.48
↳2.TCO	0.02	0.86	0.42
inst-poly-matrix-add-com	TO	TO	TO
↳TCO	trivial	trivial	trivial
inst-poly-matrix-mul-com	TO	TO	TO
↳TCO	trivial	trivial	trivial
finite-type	TO	CA	TO