The Dependently Typed Higher-Order Form for the TPTP World

Daniel Ranalter[1][0009−0006−2861−548X], Cezary Kaliszyk[2,1][0000−0002−8273−6059], Florian Rabe[3][0000−0003−3040−3655], and Geoff Sutcliffe[4][0000−0001−9120−3927]

[1] University of Innsbruck, Computational Logic, Austria
d.ranalter@gmail.com
[2] University of Melbourne, School of Computing, Australia
ckaliszyk@unimelb.edu.au
[3] University of Erlangen-Nuremberg, Computer Science, Germany
florian.rabe@fau.de
[4] University of Miami, Department of Computer Science, USA
geoff@cs.miami.edu

Abstract. Much of the current research and development in the field of automated reasoning builds on the infrastructure provided by the TPTP World. The TPTP language for logical formulae is central to the far-reaching adoption of the TPTP World. This paper introduces the Dependently Typed higher-order Form (DTF) of the TPTP language. It takes advantage of already established binders in the syntax, and is thus a minimally intrusive extension to the Typed Higher-order Form (THF). A starting set of over 100 problems is provided to exhibit the usefulness and incite interest in DTF. Some tools that are already able to reason about problems in the DTF language are discussed.

Keywords: Automated Theorem Proving · Dependent Types · Higher-Order Logic.

1 Introduction

The TPTP World [31] is a well-established infrastructure that supports research, development, and deployment of Automated Theorem Proving (ATP) systems. The TPTP language [27] is one of the keys to the success of the TPTP World. It has variants that support uniform expression of logical formulae across a wide range of logics. The TPTP language is used for writing both problems and solutions, which enables convenient communication between ATP systems and tools. The majority of modern ATP systems accept input in TPTP syntax. The TPTP language variants that form the basis for this work are the monomorphic and polymorphic typed higher-order forms (TH0 and TH1) [32,8] (see Section 2.1 for the background and further variants).

All the existing typed TPTP language variants are *simply* typed. However, there is a steady increase of interest in *dependently* typed systems, such as

Agda [3], Rocq [1,37], and Lean [9]. This interest extends to the SMT community, where the proposed version 3.0 of SMT-LIB is to include dependent types[5]. Dependent types allow for the elegant formulation of complex data structures, possibly even a direct encoding of correctness properties. This paper introduces the Dependently Typed higher-order Form (DTF) of the TPTP language.

While dependent types are frequently used in interactive theorem proving, Automated Theorem Proving (ATP) has yet to embrace dependent types. Rothgang et al. made first steps towards bringing ATP and dependent types together, by introducing dependently typed higher-order logic (DHOL) [17,18]. With only two minor extensions to the familiar syntax of Church-style HOL [6], DHOL makes dependent types easily accessible: HOL base types are extended into dependent base types that can take term arguments, and the function type $A \to B$ is changed into a dependent function type $\Pi x : A.B$. Originally DHOL did not allow quantifying over types or stating the equality of types, but a polymorphic version is in development.

As in FOL and HOL, DHOL allows arbitrary axioms that may constrain equality of terms in undecidable ways, and consequently DHOL's type checking is undecidable (see Section 3.2). To manage this complication Rothgang et al. provide an algorithm that reduces the well-formedness of a statement to a set of proof obligations. Thus theorem proving is needed to check the well-formedness of a problem's formulae, not just to prove the conjecture. Happily, typically that does not make it harder to prove the conjecture. To increase ATP support for DHOL, Rothgang et al. define a translation from well-typed DHOL to HOL that preserves provability in both directions, thereby making DHOL available for regular HOL ATP systems, albeit without leveraging DHOL's dependent types for more efficient proving. Furthermore, the translation introduces additional axioms capturing the constraints of the dependent types, thereby potentially complicating proof search. Several interactive theorem provers had previously employed the same idea, sacrificing decidable typing to gain the expressivity of dependent types, while keeping the general feel of the language simple. Most importantly, PVS [12] essentially contains DHOL as a fragment, but extends it beyond the capabilities of current *automated* provers. Mizar [38], using soft typing on top of first-order set theory, can also capture DHOL-like features.

A detail missing from the original formulation of DHOL was the choice operator. Ranalter et al. investigated the effects of losing the non-emptiness constraint in DHOL on Hilbert's choice in [16]. To this end, they extended the – to the authors knowledge – first native implementation of DHOL into the ATP system Lash, by Niederhauser et al. [11]. Their experiments strongly suggest that native reasoning in DHOL significantly outperforms reasoning on translated problems.

This work describes how DHOL is being integrated into the TPTP World, in a new TPTP language variant "Dependently Typed higher-order Form" (DTF), with monomorphic and polymorphic subvariants (DT0 and DT1). DTF requires only very minor changes to the familiar TPTP language syntax, mostly using existing notions for binders and application operators, thereby providing the ATP

[5] smt-lib.org/version3

community with the necessary foundations on which research into dependently typed automated reasoning can thrive. A set of over 100 problems in DTF, taken from several different sources, has been curated as an initial contribution to the TPTP problem library. The problems provide a spread of interesting formulations focusing on a variety of difficulty levels in proving the conjecture as well as in type checking.

Section 2 reviews the TPTP World and establishes the necessary background for DHOL, slightly generalizing the original DHOL definition to make it more suitable for TPTP. Section 3 introduces the new DTF form. Section 4 gives a short overview of the starting set of problems, and Section 5 introduces tools that already support the new form. Finally, Section 6 concludes and gives an outlook over future work.

2 Preliminaries

2.1 The TPTP World and Infrastructure

The TPTP World infrastructure includes the TPTP language [28], the TPTP problem library [25], the TSTP solution library [26], the SZS ontologies [24], the Specialist Problem Classes (SPCs) and problem difficulty ratings [29], SystemOnTPTP [23] and StarExec [22], and the CADE ATP System Competition (CASC) [30]. The problem library is a large collection of Thousands of Problems for Theorem Proving – hence the name. The problem library release v9.1.0 contains over 26000 problems from over 50 different domains, written in the TPTP language. The problems are categorized into Specialist Problem Classes according to their syntactic and logical status. The TSTP solution library is the result of running numerous ATP systems on the problems in that library and collecting their output. The TPTP and TSTP libraries provide the basis for assigning a difficulty rating to each problem, according to which ATP systems are able to solve the problem.

The most salient feature of the TPTP World for this work is the TPTP language. Originally the TPTP language supported only first-order clause normal form (CNF) [35]. Over time, more complex logics were added, starting with first-order form (FOF) in TPTP release v2.0.0 [25]. Releases v3.0.0 and v4.0.0 added monomorphic typed higher-order (TH0) [32] and monomorphic typed first-order (TF0) [34] forms to the mix respectively. These got extended to their polymorphic variants TF1 and TH1 in releases v5.0.0 [2] and v6.0.0 [8]. Release v7.0.0 of the TPTP started to include extended typed first-order form (TXF) [33] which extends the typed first-order form with conditionals, let expressions, and boolean terms. All the listed extensions to the TPTP are classical in nature. This changed with the addition of non-classical typed first-order form (NTF) in release v9.0.0 [21]. A general principle of the TPTP language is: "We provide the syntax, you provide the semantics". As such, there is no a priori commitment to any semantics for each of the language forms, although in almost all cases the intended logic and semantics are well known.

Problems and solutions are built from *annotated formulae* of the form

language(*name*, *role*, *formula*, *source*, *useful_info*)

The *language*s supported are `cnf` (clause normal form), `fof` (first-order form), `tff` (typed first-order form), and `thf` (typed higher-order form). The *role*, e.g., `axiom`, `lemma`, `conjecture`, defines the use of the formula. In a *formula*, terms and atoms follow Prolog conventions – functions and predicates start with a lowercase letter or are `'single quoted'`, and variables start with an uppercase letter. The language also supports interpreted symbols that either start with a `$`, e.g., the truth constants `$true` and `$false`, or are composed of non-alphabetic characters, e.g., integer/rational/real numbers such as 27, 43/92, -99.66. The logical connectives in the TPTP language are `!>`, `?*`, `@+`, `@-`, `!`, `?`, `~`, `|`, `&`, `=>`, `<=`, `<=>`, and `<~>`, for the mathematical connectives Π, Σ, choice (indefinite description), definite description, $\forall$, $\exists$, $\neg$, $\vee$, $\wedge$, $\Rightarrow$, $\Leftarrow$, $\Leftrightarrow$, and $\oplus$ respectively. Equality and inequality are expressed as the infix operators `=` and `!=`. The *source* and *useful_info* are optional.

2.2 Dependently Typed Higher-Order Logic

Dependently typed higher-order logic (DHOL) is an extension of Church's higher-order logic (HOL) [6] introduced by Rothgang et al. [17]. It takes the widely supported HOL and equips it with dependent types, i.e., types that take term arguments. As such, it is a classical and extensional type theory, as opposed to the theory used in Rocq [1,37], Lean [9], or others [3,13] that rely on an intensional type theory. Notable exceptions to this trend are PVS [19], NuPRL [7], and F* [36].

The extensionality of DHOL comes at the cost of making type checking undecidable because it must consider term equality, which may be subject to arbitrary axioms. Essentially, typing becomes undecidable if a type depends on a type for which equality is undecidable. This is because type checking t against type $a\ n$ must be done by inferring the type of t, say $a\ m$, and then checking $a\ m = a\ n$, and thus $m = n$. If all dependent type symbols depend only on types for which equality is decidable (e.g., the examples below where we only use natural numbers with Presburger arithmetic), type checking is decidable. Otherwise, e.g., when using types depending on natural numbers with Peano arithmetic, type checking is undecidable.

The gain of having judgmental and provable equality coincide is significant: It positions DHOL much closer to how mathematics is usually done in the context of ATP. The availability of dependent types allows the elegant definition of data structures such as lists of fixed-length, intervals of numbers, or vector spaces over some field. It also allows encoding constraints in the types, which can remove the need for lengthy and error-prone guards in programming and track invariants useful for theorem proving. The cost – which might seem steep at first glance – is mitigated by the ever-increasing performance of ATP systems, and the fact that in many cases the proof obligations resulting from type checking are much simpler than the original proving problem.

The changes to the TPTP syntax to accommodate DTF are small: the definition of the simple base type is changed to a type that can accept term arguments, and the simple function type $A \to B$ is changed to $\Pi x : A.B$. This makes it possible to let the result type of the function depend on the specific term of the argument.

Figure 1 gives the grammar of DHOL. A dependent base type a with arity n is written $a : \Pi x_1 : A_1, \cdots, x_n : A_n.\texttt{type}$, and it is a *simple* base type if $n = 0$. Declarations of this form are part of the theory against which the type checking procedure is performed. In addition to base type declarations, theories may declare constant symbols c and axioms $\triangleright F$. A context specifies typed variables and assumptions. Contexts are superficially similar to theories, but denote *local* declarations, and as such, do not contain type declarations. $\circ$ and $\bullet$ denote the empty theory and context respectively. The order in a theory or context matters because the well-typedness of declarations might depend on preceding axioms. Types, as they appear in statements and typing judgements, are either fully applied base types, (dependent) function types, or classical booleans o. Terms are built from variables/constants, lambda abstraction, application, and the usual connectives and quantifiers. Regular HOL can be recovered by omitting the **highlighted** elements – this is exactly the case when the arity of all base types is 0.

$$
\begin{array}{lll}
T, U & ::= \circ \mid T,\ a : (\Pi x : A.)^*\texttt{type} \mid T,\ c : A \mid T,\ \triangleright F & \text{theories} \\
\Gamma, \Delta & ::= \bullet \mid \Gamma,\ x : A \mid \Gamma,\ \triangleright F & \text{context} \\
A, B & ::= a\, t_1 \cdots t_n \mid \Pi x : A.B \mid o & \text{types} \\
t, u, F, G & ::= x \mid c \mid \lambda x : A.t \mid t\, u \mid \forall x : A.F \mid \exists x : A.F \mid F \Rightarrow G & \\
& \quad \mid\ F \wedge G \mid F \vee G \mid \bot \mid \top \mid \neg F \mid t =_A u & \text{terms (incl. formulae } F, G)
\end{array}
$$

Fig. 1. The grammar of DHOL

The following example encodes the familiar notion of fixed-length lists. As prerequisites, we give the usual notion of natural numbers in a simple type `nat` and a simple type `char` of characters for the elements of the lists:

$$
\begin{array}{c}
\texttt{nat} : \texttt{type} \qquad 0 : \texttt{nat} \qquad \texttt{suc} : \texttt{nat} \to \texttt{nat} \qquad + : \texttt{nat} \to \texttt{nat} \to \texttt{nat} \\
\triangleright\ \forall n : \texttt{nat}.+\ 0\ n =_{\texttt{nat}} n \qquad \triangleright\ \forall n, m : \texttt{nat}.+\ (\texttt{suc}\ n)\ m =_{nat} \texttt{suc}\ (+\ n\ m) \\
\texttt{char} : \texttt{type} \qquad \texttt{a} : \texttt{char} \qquad \texttt{b} : \texttt{char} \qquad \ldots
\end{array}
$$

Then $\texttt{vec}\, n$ encodes the type of fixed-length lists of characters of length n:

$$
\begin{array}{c}
\texttt{vec} : \Pi n : \texttt{nat.type} \quad \texttt{nil} : \texttt{vec}\ 0 \quad \texttt{cons} : \Pi x : \texttt{nat.char} \to \texttt{vec}\ n \to \texttt{vec}\ (\texttt{suc}\ n) \\
++ : \Pi n, m : \texttt{nat.vec}\ n \to \texttt{vec}\ m \to \texttt{vec}\ (+\ n\ m)
\end{array}
$$

Dependent Connectives In DHOL it is desirable to make the binary connectives conjunction, implication, and disjunction dependent in the sense that the well-formedness of the second argument may assume the truth (for conjunction and

implication) or the falsity (for disjunction) of the first argument. Consider the statement $a =_A b \Rightarrow f\ a =_{B(a)} f\ b$. The well-formedness of the right-hand side requires the left-hand side as a premise. More precisely, $\Gamma \vdash F : o$ resp. $\Gamma \vdash F$ expresses that F is a well-formed resp. provable formula in context Γ. The definition of well-formed formulae is:

$$\begin{array}{lll} \Gamma \vdash F \Rightarrow G & \text{if} & \Gamma \vdash F \text{ and } \Gamma, \mathbf{\triangleright}\ F \vdash G \\ \Gamma \vdash F \wedge G & \text{if} & \Gamma \vdash F \text{ and } \Gamma, \mathbf{\triangleright}\ F \vdash G \\ \Gamma \vdash F \vee G & \text{if} & \Gamma \vdash F \text{ and } \Gamma, \mathbf{\triangleright}\ \neg F \vdash G \end{array}$$

where the **marked** parts make the connectives dependent. The usual natural deduction proof rules of implication and conjunction are the same as for the non-dependent versions. The proof rules for disjunction are adjusted as follows:

$$\frac{\Gamma \vdash F \quad \Gamma, \mathbf{\triangleright}\ \neg F \vdash G : o}{\Gamma \vdash F \vee G} \qquad \frac{\Gamma, \mathbf{\triangleright}\ \neg F \vdash G}{\Gamma \vdash F \vee G} \qquad \frac{\Gamma, \mathbf{\triangleright}\ F \vdash C \quad \Gamma, \mathbf{\triangleright}\ \neg F, \mathbf{\triangleright}\ G \vdash C}{\Gamma, \mathbf{\triangleright}\ F \vee G \vdash C}$$

As usual, it is possible to choose some connectives as primitives, from which the others are defined. Rothgang et al. choose equality and implication. Contrary to HOL, they included implication because they could not define the dependent binary connectives solely from equality. For the TPTP World, it is better not to choose primitive connectives – that choice should be left to the ATP system developers. Therefore DTF extends the work by Rothgang et al. to make all connectives primitive. ATP systems can choose which connectives to treat as abbreviations, but in doing so must take the dependent nature of the connectives into account. Note that dependent connectives break the commutativity of conjunction and disjunction. While seemingly disruptive, sacrificing commutativity in this way is common practice, e.g., for short-circuit evaluation of Boolean terms in programming languages. To clarify the impact on theorem proving, Table 1 summarizes typical proof rules for FOL and their status in DHOL. Roughly speaking, all rules that do not affect the order of subformulae remain sound, while the rest of the rules require the additional check to ensure the result remains well-formed. In particular, all rules needed to perform CNF or clause normal form transformations remain available.

Developing advanced calculi for DHOL is beyond the scope of this paper. However, for example, one way to generalize resolution is to store clauses as lists $[L_1, \ldots, L_n]$ where the well-formedness of each L_i may depend on $\neg L_j$ for $j < i$. Resolving $[A, \vec{L}]$ and $[\neg A, \vec{M}]$ to $[\vec{L}, \vec{M}]$ is sound if the resolvent is well-formed, i.e., if the well-formedness of the L_i resp. M_i does not depend on $\neg A$ resp. A.

Polymorphic DHOL. DHOL as presented in the previous section and [17] is monomorphic. ATP for polymorphic DHOL, as well as proofs of properties for such an extension of the calculus, is ongoing parallel work. Polymorphic logics are already available in the TPTP language, so it is natural to offer polymorphic DTF. All the polymorphic example problems considered so far use only shallow/rank-1 polymorphism in line with the existing polymorphic first- and higher-order forms for TPTP.

Table 1. Typical proof rules for FOL and their status in DHOL

Rule	**Holds in DHOL**
For disjunction and conjunction	
associativity	✓
commutativity	Only if both sides are well-formed
idempotence, e.g., $A \wedge X \wedge A \Leftrightarrow A \wedge X$	✓(Drop the *second* occurrence)
de Morgan laws	✓
distributivity of one over the other	✓
absorption, e.g., $A \wedge (A \vee B) \Leftrightarrow A$	✓
For implication	
$A \Rightarrow B \Leftrightarrow \neg A \vee B$	✓
$\neg(A \Rightarrow B) \Leftrightarrow A \wedge \neg B$	✓
$\neg(A \Rightarrow B) \Leftrightarrow \neg B \Rightarrow \neg A$	Only if both sides are well-formed
For quantifiers and equality	
all rules	✓
Common calculus rules	
classical reasoning	✓
weakening	✓
contraction	✓(Drop the *second* occurrence)
exchange	Only if still well-formed
cut	✓
resolution	Only if the clauses remain well-formed

Choice. Hilbert's choice operator has been part of HOL since its inception by Church [6]. As such, it is natural to include it in DHOL. This introduces some complications: Due to the usual non-emptiness constraint on types, the semantics of choice are clear in HOL. However, DHOL no longer abides by this constraint, requiring a design decision that affects well-typedness and provability. Experiments done in [16] suggest that the variant of choice dubbed "strong choice" results in more efficient automated reasoning. The eponymous characteristic of strong choice is the requirement that $\exists x : A.t$ needs to be true for $(\varepsilon x : A.t) : A$ to be well-typed. Such a requirement for typing fits well with DHOL in general, and as ATP is the main concern this is the variant of choice, as it were. The problem set described in Section 4 includes some examples supporting this variant.

Translation. In order to take advantage of the ATP systems available for regular HOL, Rothgang et al. define a dependency-erasure [17], and thereby a translation from DHOL into regular HOL. They also prove that this translation is sound and complete for well-typed DHOL problems. Due to this result, and the implementation of the translation into the preprocessor of the Leo-III theorem prover [20], there existed reasoning support for DHOL even before native DHOL reasoning was implemented in the Lash ATP system by Niederhauser et al. [11]. Information lost due to the erasure of term dependencies is captured in Partial Equivalence Relations (PERs) – symmetric and transitive relations on pairs of

terms – with the idea that the relation is reflexive exactly for those terms that were previously of the same dependent type. The translation is shown in Figure 2. The translation $\overline{t}$ of a term t is defined inductively on the structure of the terms. The erasure of one type declaration results in three erased declarations: the erased type, the PER constant and an axioms stating it's properties. The definition of the erasure on $\forall$- and $\exists$-quantified terms is notable as it uses a PER as guard on the argument. To see why, note that, e.g., $\forall x : A.t$ can be defined in terms of equality as $\lambda x : A.t =_{A\to o} \lambda x : A.\top$. The erasure creates a PER from this typed equality with the guarded input in the premise, and the erased term in the consequence of the implication as seen in the erasure of $\forall$.

theories

$$\begin{aligned}
\overline{\circ} &= \circ \\
\overline{T, D} &= \overline{T}, \overline{D} \\
\overline{c : A} &= c : \overline{A},\ \triangleright\, A^*\, c\, c \\
\overline{\triangleright\, F} &= \triangleright\, \overline{F}
\end{aligned}$$

contexts

$$\begin{aligned}
\overline{\bullet} &= \bullet \\
\overline{\Gamma, D} &= \overline{\Gamma}, \overline{D} \\
\overline{x : A} &= x : \overline{A},\ \triangleright\, A^*\, x\, x \\
\overline{\triangleright\, F} &= \triangleright\, \overline{F}
\end{aligned}$$

$$\begin{aligned}
&\overline{a : \Pi x_1 : A_1.\ \cdots\ \Pi x_n : A_n.\, \mathtt{type}} = \\
&\qquad a\ \mathtt{type} \\
&\qquad a^* : \overline{A_1} \to \cdots \to \overline{A_n} \to a \to a \to o \\
&\qquad \triangleright\, \forall x_1 : \overline{A_1}.\ \cdots \forall x_n : \overline{A_n}.\, \forall u, v : a.\, a^*\ x_1\ \cdots\ x_n\ u\ v \Rightarrow u =_a v
\end{aligned}$$

terms and types

$$\begin{aligned}
\overline{c} &= c & \overline{x} &= x \\
\overline{o} &= o & \overline{a\ t_1\ \ldots\ t_n} &= a \\
\overline{\Pi x : A.B} &= \overline{A} \to \overline{B} & \overline{\lambda x : A.t} &= \lambda x : \overline{A}.\, \overline{t} \\
\overline{\neg t} &= \neg \overline{t} & \overline{t\ u} &= \overline{t}\ \overline{u} \\
\overline{t \Rightarrow u} &= \overline{t} \Rightarrow \overline{u} & \overline{t =_A u} &= A^*\ \overline{t}\ \overline{u} \\
\overline{t \wedge u} &= \overline{t} \wedge \overline{u} & \overline{t \vee u} &= \overline{t} \vee \overline{u} \\
\overline{\bot} &= \bot & \overline{\top} &= \top \\
\overline{\forall x : A.t} &= \forall x : \overline{A}.\, A^*\, x\, x \Rightarrow \overline{t} & \overline{\exists x : A.t} &= \exists x : \overline{A}.\, A^*\, x\, x \wedge \overline{t}
\end{aligned}$$

PER for each type

$$\begin{aligned}
o^*\ t\ u &= t =_o u \\
(a\ t_1\ \ldots\ t_n)^*\ u\ v &= a^*\ \overline{t_1}\ \cdots\ \overline{t_n}\ u\ v \\
(\Pi x : A.B)^*\ t\ u &= \forall x, y : \overline{A}.\, A^*\, x\, y \Rightarrow B^*(t\, x)(u\, y)
\end{aligned}$$

Fig. 2. The translation from DHOL to HOL

As an example of erasure, consider the list of chars [a, b], represented by a term cons 1 a (cons 0 b nil) of type vec 2, where $0, \texttt{suc}\ 0, \texttt{suc}\ (\texttt{suc}\ 0), ...$ is abbreviated as $0, 1, 2, ...$ Applying the erasure gives cons a (cons b nil) of type vec. A predicate would be generated, establishing that this particular list is in the PER of vectors of length 2: $\texttt{vec}^*\ 2\ t\ t$ where t stands for cons a (cons b nil). While one might think that unary predicates would be sufficient as a type guard, PERs becomes necessary to express the typing and equality of higher-order functions: functions are well-typed if they map well-typed inputs to well-typed outputs, and they are equal if they agree on well-typed inputs.

3 DTF

After establishing the theoretic background, this section presents the realization of DHOL in the TPTP language. Syntax and semantics are given, as well as an exposition to the problem of type checking.

3.1 Syntax

The syntax of DTF requires almost no change to the existing TPTP syntax. The TPTP language already defines the !> binder for types. In the typed TPTP language variants it is currently used for only polymorphism, e.g.,

```
cons : !>[A: $tType]: ( A > ( list @ A ) > ( list @ A ) )
```

is a type declaration for a polymorphic cons. The TPTP syntax does not forbid listing terms in the types of such variable lists. This fact is used to unobtrusively extend TPTP by dependent types. A dependent type symbol declaration is written with m terms of n types as

$$\texttt{a : !>[}x_1 : A_1\texttt{, ..., }x_m : A_n\texttt{]:\$tType}$$

or alternatively

$$\texttt{a : }A_1\texttt{ > ... > }A_n\texttt{ > \$tType.}$$

Such types use the application operator @, to instantiate the terms to the dependent type:

$$\texttt{a @ }t_1\texttt{ @ ... @ }t_m\texttt{.}$$

In polymorphic problems, the variable list is prepended with the type variables, which may appear in the same binder. An example of a problem in DTF is shown in Figure 3.

```
thf(elem_type,type,    elem: $tType ).
thf(nat_type,type,     nat: $tType ).
thf(zero_type,type,    zero: nat ).
thf(suc_type,type,     suc: nat > nat ).
thf(plus_type,type,    plus: nat > nat > nat ).
thf(list_type,type,    list: nat > $tType ).
thf(nil_type,type,     nil: list @ zero ).
thf(cons_type,type,    cons:
    !>[N: nat] : (elem > (list @ N) > (list @ (suc @ N))) ).
thf(app_type,type,     app:
    !>[N: nat,M: nat] : ((list @ N) > (list @ M) >
                         (list @ (plus @ N @ M))) ).

thf(ax1,axiom,
    ! [N: nat] : ((plus @ zero @ N) = N) ).

thf(ax2,axiom,
    ! [N: nat,X: list @ N] : ((app @ zero @ N @ nil @ X) = X) ).

thf(plus_assoc,axiom,
    ! [M1: nat,M2: nat,M3: nat] :
      ( (plus @ M1 @ (plus @ M2 @ M3))
      = (plus @ ( plus @ M1 @ M2) @ M3)) ).

thf(list_app_assoc_base,conjecture,
    ! [M2: nat,L2: list @ M2,M3: nat,L3: list @ M3] :
      ( (app @ zero @ (plus @ M2 @ M3) @ nil @
          (app @ M2 @ M3 @ L2 @ L3))
      = (app @ (plus @ zero @ M2) @ M3 @
          (app @ zero @ M2 @ nil @ L2) @ L3)) ).
```

Fig. 3. The base case of associativity of append on fixed-length lists.

3.2 Type Checking

Due to equality reflection, type checking for DHOL is, in general, undecidable. Nevertheless, problems need to be well-typed, otherwise the translation outlined in Section 2.2 might not be sound. Type checking in DTF thus takes on a larger role than in other logics in the TPTP World.

While performing the usual type checking procedure in DHOL, obligations of the form $a\ t_1\ \cdots\ t_n \equiv a\ u_1 \cdots\ u_n$, are generated. These establish equality of the dependent base types applied to arguments $t_1 \cdots t_n, u_1 \cdots u_n$ of appropriate types. The type equality holds if all pairs t_i, u_i are equal, which depends on the available axioms. This can create interesting situations where a problem must include axioms that are not necessary for proving the conjecture itself, but are necessary for type checking it. The common example of fixed-length lists is one such example: the statement of the associativity of `append` is well-typed only if

addition on `nat` is associative, and thus requires including the defining equations of addition. To prove the problem only the defining equations of appending lists are needed.

The undecidability of type checking can lead to compromises. One such compromise is "shallow type checking". When a problem file is shallowly checked, only the simply typed skeleton of the problem is considered, i.e., term arguments to types as well as dependent functions are ignored. This collapses to type checking as is done on non-dependently typed problems, and is decidable. This form of type checking is sufficient to catch many careless mistakes in the formulation of problems, and provides a basic check of issues often found in human-written DHOL problems. Examples are: mismatches in the number of arguments of a base type or function, and egregious type mismatches. Shallow type checking provides a valuable sanity check for users, especially considering the complexity that problems in DHOL forms can reach.

3.3 Semantics

As for HOL, there are two kinds of semantics for DHOL: standard models are intuitive and are the ones that are usually used; non-standard (Henkin) models are a generalization that is needed for completeness. A full account is given in the forthcoming [15], which is summarized below. The rules of DHOL, as given by Rothgang et al., already define which formulae are theorems.

Standard Models. Given a theory T, a standard model $M \in [\![T]\!]$ is a tuple providing an interpretation for every declaration in T. Similarly, given a context Γ, an assignment $\alpha \in [\![\Gamma]\!]^M$ for Γ is a tuple providing an interpretation for every declaration in Γ. These induce the interpretation function $[\![-]\!]^M_\alpha$ (with α omitted if the context is empty), which is defined inductively for all the syntax. In particular, the possible components of a model are defined by induction on declarations:

- For a type symbol with arguments $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, a function $[\![\Gamma]\!]^M \to \mathcal{SET}$
- For a term symbol $c : A$, a value from $[\![A]\!]^M$
- For an axiom $\triangleright F$, a unique choice ✓ if $[\![M]\!]^F = 1$, and no choice otherwise

For the components of an assignment:

- For a term variable $x : A$, a value from $[\![A]\!]^M_\alpha$
- For an assumption $\triangleright F$, a unique choice ✓ if $[\![M]\!]^F_\alpha = 1$, and no choice otherwise

For types and terms, the model is defined by induction in the usual way, in particular

- $[\![o]\!]^M_\alpha = \{0, 1\}$
- $[\![\Pi x : A.B]\!]^M_\alpha$ is the set of functions f mapping every $u \in [\![A]\!]^M_\alpha$ to some $f(u) \in [\![B]\!]^M_{\alpha^u}$ where α^u extends α with the value u for x

General Models. The definition of general models generalizes the Henkin models from HOL by applying methods from categorical models of type theory. First,

akin to assignments for Γ, substitutions $\gamma : \Gamma \to \Delta$ as lists of terms or ✓ by induction on Γ are defined:

- For a term variable $x : A$, a term of type $\Delta \vdash A[\gamma]$
- For an assumption $\triangleright\ F$, the unique choice ✓ if $\Delta \vdash F[\gamma]$, and no choice otherwise

Equality of contexts and substitutions is defined by applying the existing equality judgments for types and terms component-wise. For every theory T, this yields the syntactic category $\overline{[\![T]\!]}$ of T-contexts and substitutions. A general model is then defined as any pushout-preserving contravariant functor $\Phi : \overline{[\![T]\!]} \to \mathcal{SET}$. From such a Φ, an interpretation function is extracted using $\Phi(x : A)$ as the interpretation of the type A and $\Phi(t)$ as the interpretation of the term $t : A$ (seen as a substitution $x : A \to \bullet$). These general models must further satisfy $\Phi(o) = \{0, 1\}$, and $\Phi \models F$ is defined as $\Phi(F) = 1$. Here the pushout-preservation essentially corresponds to the preservation of substitution, i.e., interpretation and substitution commute. The lack of any preservation of exponentials allows for a *non-compositional* interpretation of function types. This approach can be seen as a generalization of Henkin models, which also preserve substitution but do not need to interpret function types compositionally. Contrary to Henkin models, the interpretation of λ and application terms can also be non-compositional in these general models as long as substitution is preserved.

Models for Polymorphic DHOL. As mentioned above, a rank-1 polymorphic variant of DHOL is being developed in parallel work. It is straightforward to extend standard models to polymorphic DHOL. The syntax of binding a type variable corresponds to abstracting over an arbitrary set on the semantic side. In particular, the interpretation of a polymorphic term/type symbol with n type variables takes n sets as arguments. Polymorphic axioms correspond to universal quantification over sets. The definition of syntactic category and general models is expected to carry over to polymorphic DHOL as well. This has not been investigated in detail.

4 Problem Dataset

Over 100 problems in DTF format have been collected for addition to the TPTP problem library. Their classification is presented in Table 2 and discussed here (with 36 problems just for testing DHOL prover features omitted). The number of problems in each class is given in the last column. The problems concern several domains that can benefit from dependent types. While [17] shows DHOL to be sound and complete, the strength of the existing automation for this foundation (discussed in Section 5) still needs to be improved. For this reason, some of the harder problems were broken down into simpler subproblems that can be proven independently. Some list properties that require both induction and reasoning with dependent types are an instance of this. For example, the fact that list append is associative, `ListAppAssoc`, is split into three subproblems, showing the particular induction scheme, the proof of the base case, and the step

case. These three subproblems are easier to prove than their combined version, which is also included. Some problems benefit from intermediate lemmas, e.g. the instantiation of the inductive step case. These are found in the "Lemmas" categories of Table 2.

One of the simplest classes of examples are lists that depend on their length (also called vectors, for example in the Rocq library). As the list libraries of most interactive theorem provers are substantial, it is relatively easy to experiment with many properties of dependently typed lists. Such properties include the aforementioned associativity of `append`, corollaries of this statement, or involution statements about the `reverse` function. Some of these list examples are extended to their polymorphic generalizations, which are in the "Polymorphic" categories.

The idea of expressing well-known but sometimes challenging properties extends to several other algebraic data types, such as matrices that have fixed dimensions, and lists of lists. Red-black trees are a well-known data structure for balanced trees where the invariant can be expressed using dependent types, and again several problems concerning this type are included. The `Fin` type present in several proof libraries has been manually recreated, and some problems about these are in the ROCQ category of Table 2. The collection includes the five examples from category theory that were originally presented in [17], slightly reformatted to match the TPTP syntax. To make use of the choice operator [16], several problems about dependent higher-order Skolemization are included. Choice is also used in a function definition with no fixed point, and conjectures establishing this are presented in the "no FP" category. Finally, several simple tests to evaluate the ability of provers to perform native DHOL inferences are provided.

Some of the dependent HOL problems are more interesting from a proof perspective – the deep type checking is there only to make sure the problem is well-formed. For example, for all the dependent list problems, the type checking obligations are there mostly to make sure no incorrect calls are being made, but they are relatively straightforward to discharge. It is the proof that requires more logical reasoning. Other problems, while relatively straightforward in terms of proving, are harder to type check. This is because it is possible to use dependent types to encode important properties and invariants in the type system.

5 Tools

This section discusses the tools capable of processing problems in DTF format.

5.1 The Logic Embedding Tool

The Leo-III [20] prover includes the *Logic Embedding Tool*, which has been extended to support polymorphic DTF. The tool implements the erasure presented in Section 2.2, and incorporates the polymorphic extension. The tool can generate both the type checking obligations and the translated problem separately.

Problem Type	Problem Category	Problem Count
Monomorphic Complete	Category theory	5
	Choice basic	11
	Choice list	3
	Choice no fixed point	10
	List app assoc	3
	List app assoc corollary	1
	List app nil	4
	List of lists	1
	List reversal involution	1
	List reversal inv lemma	3
	Matrices	5
	ROCQ	3
Monomorphic Lemmas	Choice no fixed point	10
	List app assoc	5
	List app assoc corollary	5
	List reversal involution	5
	List reversal inv lemma	11
Polymorphic Complete	List app assoc poly	3
	List app nil poly	4
	List reversal involution poly	1
	Red-black tree	3
Polymorphic Lemmas	List app assoc poly	14
	List reversal involution poly	13
	Red-black tree	9

Table 2. The categories of the DTF problems.

This makes it possible to translate DTF problems into THF problems (that do not have dependent types). The embedding tool is available as NTFLET in SystemB4TPTP [6]. The embedding tool enables the use of existing higher-order ATP systems for solving DTF problems, by pipelining the output from NTFLET to a THF ATP system of the user's choosing. This has been implemented as the DT2H2X ATP systems, available in SystemOnTPTP [7].

5.2 DLash

The Lash prover [5] is a partial reimplementation of the tableaux calculus of Satallax [4], using a central term representation with perfect sharing. This design facilitated the implementation of the DLash extension of Lash, which handles DTF [11]. In addition to the erasure implementation, DLash can process monomorphic dependently typed higher-order logic with choice. As with the Logic Embedding Tool, type checking and proving can be requested separately. DLash, like Satallax, includes a strategy language used to build so-called modes.

[6] tptp.org/cgi-bin/SystemB4TPTP
[7] tptp.org/cgi-bin/SystemOnTPTP

The current version includes 36 dedicated modes for dependent types, tailored to specific problem types. DLash is available in SystemOnTPTP [8].

5.3 MMT

MMT [14] is a logical framework designed to formalize and manage large collections of interconnected formal systems and their libraries, using modular theory graphs. A particular application of MMT is rapid prototyping [10], and it was the tool originally used to develop and prototype DHOL. The MMT/DHOL implementation offers reconstruction of omitted types and implicit arguments as well as parsing against user-defined notations. It can be used to interactively author and type check DHOL problems and export them in TPTP format. It uses the PER translation, and calls the Leo-III prover to discharge the resulting proof obligations. MMT is mostly useful for developing formalizations, rather than proving TPTP conjectures. Therefore, it does not provide a TPTP import at this point, but provides additional evidence of the well-typedness DTF problems.

5.4 TPTP Systems

As discussed in Section 2.1, TPTP includes several generic tools capable of processing problems and solutions. For DTF problems:

- `TPTP4X` pretty-prints DTF problems and solutions, and offers various transformations/augmentations of problems.
- `BNFParser` produces the abstract syntax tree from parsing a DTF problem.
- `Leo-III-STC` validates the syntax and types of DTF problems.
- `ProblemStats` outputs various syntactic measures for problems.

All these tools are available in SystemB4TPTP [9]. For DTF proofs:

- `ProofStats` outputs various syntactic measures for DAG-structured proofs.
- `IDV` provides interactive viewing of proofs from DTF problems.

All these tools are available in SystemOnTSTP [10].

6 Conclusion

This paper has described DTF, the dependently typed higher-order form of the TPTP language. It responds to the growing interest in dependently typed automated reasoning as exemplified by the number of TPTP problems and tools that have cropped up in the short time since DHOL was first described. It can be

[8] tptp.org/cgi-bin/SystemOnTPTP
[9] tptp.org/cgi-bin/SystemB4TPTP
[10] tptp.org/cgi-bin/SystemOnTSTP

seen as pushing the boundary of automated theorem proving towards language features that have previously been found only in interactive provers.

DHOL problems sometimes used differing standards, which defeated the uniformity advantage that the TPTP language provides. This work unifies them, and provides over a 100 problems from different domains, benefiting from the use of dependent types. We hope that the availability of dependent types in the TPTP will stimulate research into dependently typed automated theorem proving, by making it easier to exchange and compare results. Extending existing systems with support for DTF, and improving the performance of the systems that already exist, will be important next steps. In particular, the extension of superposition-based theorem proving to dependent types is a tantalizing goal.

Acknowledgements: The authors thank Johannes Niederhauser and Colin Rothgang for granting access to their DHOL problems that are in the problem dataset. This work was supported by the ERC PoC grant no. 101156734 "FormalWeb3".

References

1. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer (2004)
2. Blanchette, J.C., Paskevich, A.: TFF1: the TPTP typed first-order form with rank-1 polymorphism. In: Bonacina, M.P. (ed.) Proc. 24th International Conference on Automated Deduction. LNCS, vol. 7898, pp. 414–420. Springer (2013). https://doi.org/10.1007/978-3-642-38574-2_29, https://doi.org/10.1007/978-3-642-38574-2_29
3. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda – a functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Proc. 22nd International Conference on Theorem Proving in Higher Order Logics. LNCS, vol. 5674, pp. 73–78 (2009). https://doi.org/10.1007/978-3-642-03359-9_6
4. Brown, C.E.: Satallax: An automatic higher-order prover. In: Gramlich, B., Miller, D., Sattler, U. (eds.) Proc. 6th International Joint Conference on Automated Reasoning. LNAI, vol. 7364, pp. 111–117 (2012). https://doi.org/10.1007/978-3-642-31365-3_11
5. Brown, C.E., Kaliszyk, C.: Lash 1.0 (system description). In: Blanchette, J., Kovács, L., Pattinson, D. (eds.) Proc. 11th International Joint Conference on Automated Reasoning. LNAI, vol. 13385, pp. 350–358 (2022). https://doi.org/10.1007/978-3-031-10769-6_21
6. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5**(2), 56–68 (1940). https://doi.org/10.2307/2266170
7. Constable, R., Allen, S., Bromley, H., Cleaveland, W., Cremer, J., Harper, R., Howe, D., Knoblock, T., Mendler, N., Panangaden, P., Sasaki, J., Smith, S.: Implementing Mathematics with the Nuprl Development System. Prentice-Hall (1986)
8. Kaliszyk, C., Sutcliffe, G., Rabe, F.: TH1: the TPTP typed higher-order form with rank-1 polymorphism. In: Fontaine, P., Schulz, S., Urban, J. (eds.) Proc. 5th Workshop on Practical Aspects of Automated Reasoning. CEUR Workshop Proceedings, vol. 1635, pp. 41–55. CEUR-WS.org (2016), https://ceur-ws.org/Vol-1635/paper-05.pdf

9. de Moura, L., Ullrich, S.: The Lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Proc. 28th International Conference on Automated Deduction. LNAI, vol. 12699, pp. 625–635 (2021). https://doi.org/10.1007/978-3-030-79876-5_37
10. Müller, D., Rabe, F.: Rapid Prototyping Formal Systems in MMT: Case Studies. In: Miller, D., Scagnetto, I. (eds.) Logical Frameworks and Meta-languages: Theory and Practice. pp. 40–54 (2019)
11. Niederhauser, J., Brown, C.E., Kaliszyk, C.: Tableaux for automated reasoning in dependently-typed higher-order logic. In: Benzmüller, C., Heule, M.J.H., Schmidt, R.A. (eds.) Proc. 24th International Joint Conference on Automated Reasoning. pp. 86–104. LNAI (2024). https://doi.org/10.1007/978-3-031-63498-7_6
12. Owre, S., Shankar, N.: The formal semantics of PVS. Tech. Rep. SRI-CSL-97-2, SRI International (1997)
13. Pfenning, F., Schürmann, C.: System description: Twelf – a meta-logical framework for deductive systems. In: Ganzinger, H. (ed.) Proc. 16th International Conference on Automated Deduction. LNAI, vol. 1632, pp. 202–206 (1999). https://doi.org/10.1007/3-540-48660-7_14
14. Rabe, F.: A Modular Type Reconstruction Algorithm. ACM Transactions on Computational Logic **19**(4), 1–43 (2018)
15. Rabe, F.: Model theory for dependently-typed higher-order logic (2024), under review, see https://kwarc.info/people/frabe/Research/rabe_dholmodels_24.pdf
16. Ranalter, D., Brown, C.E., Kaliszyk, C.: Experiments with choice in dependently-typed higher-order logic. In: Bjørner, N.S., Heule, M., Voronkov, A. (eds.) Proc. 25th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 100, pp. 311–320. EasyChair (2024). https://doi.org/10.29007/2V8H, https://doi.org/10.29007/2v8h
17. Rothgang, C., Rabe, F., Benzmüller, C.: Theorem proving in dependently-typed higher-order logic. In: Pientka, B., Tinelli, C. (eds.) Proc. 29th International Conference on Automated Deduction. LNAI, vol. 14132, pp. 438–455 (2023). https://doi.org/10.1007/978-3-031-38499-8_25
18. Rothgang, C., Rabe, F., Benzmüller, C.: Dependently-typed higher-order logic – extended preprint (2025). https://doi.org/10.48550/arXiv.2305.15382, under review
19. Rushby, J., Owre, S., Shankar, N.: Subtypes for specifications: Predicate subtyping in PVS. IEEE Transactions on Software Engineering **24**(9), 709–720 (1998). https://doi.org/10.1109/32.713327
20. Steen, A., Benzmüller, C.: Extensional higher-order paramodulation in Leo-III. Journal of Automated Reasoning **65**, 775–807 (2021). https://doi.org/10.1007/s10817-021-09588-x
21. Steen, A., Fuenmayor, D., Gleißner, T., Sutcliffe, G., Benzmüller, C.: Automated reasoning in non-classical logics in the TPTP world. In: Konev, B., Schon, C., Steen, A. (eds.) Proc. 8th Workshop on Practical Aspects of Automated Reasoning. CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022), https://ceur-ws.org/Vol-3201/paper11.pdf
22. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: a Cross-Community Infrastructure for Logic Solving. In: Demri, S., Kapur, D., Weidenbach, C. (eds.) Proceedings of the 7th International Joint Conference on Automated Reasoning. pp. 367–373. No. 8562 in Lecture Notes in Artificial Intelligence (2014)
23. Sutcliffe, G.: SystemOnTPTP. In: McAllester, D. (ed.) Proceedings of the 17th International Conference on Automated Deduction. pp. 406–410. No. 1831 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2000)

24. Sutcliffe, G.: The SZS Ontologies for Automated Reasoning Software. In: Sutcliffe, G., Rudnicki, P., Schmidt, R., Konev, B., Schulz, S. (eds.) Proceedings of the LPAR Workshops: Knowledge Exchange: Automated Provers and Proof Assistants, and the 7th International Workshop on the Implementation of Logics. pp. 38–49. No. 418 in CEUR Workshop Proceedings (2008)
25. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. The FOF and CNF Parts, v3.5.0. Journal of Automated Reasoning **43**(4), 337–362 (2009)
26. Sutcliffe, G.: The TPTP World - Infrastructure for Automated Reasoning. In: Clarke, E., Voronkov, A. (eds.) Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning. pp. 1–12. No. 6355 in Lecture Notes in Artificial Intelligence, Springer-Verlag (2010)
27. Sutcliffe, G.: The Logic Languages of the TPTP World. Logic Journal of the IGPL **31**(6), 1153–1169 (2023)
28. Sutcliffe, G., Schulz, S., Claessen, K., Van Gelder, A.: Using the TPTP Language for Writing Derivations and Finite Interpretations. In: Furbach, U., Shankar, N. (eds.) Proceedings of the 3rd International Joint Conference on Automated Reasoning. pp. 67–81. No. 4130 in Lecture Notes in Artificial Intelligence, Springer (2006)
29. Sutcliffe, G., Suttner, C.: Evaluating General Purpose Automated Theorem Proving Systems. Artificial Intelligence **131**(1-2), 39–54 (2001). https://doi.org/{10.1016/S0004-3702(01)00113-8}
30. Sutcliffe, G.: The CADE ATP system competition - CASC. AI Magazine **37**(2), 99–101 (2016). https://doi.org/10.1609/AIMAG.V37I2.2620, https://doi.org/10.1609/aimag.v37i2.2620
31. Sutcliffe, G.: Stepping Stones in the TPTP World. In: Benzmüller, C., Heule, M.J.H., Schmidt, R.A. (eds.) Proc. 24th International Joint Conference on Automated Reasoning. LNAI, vol. 14739, pp. 30–50 (2024)
32. Sutcliffe, G., Benzmüller, C.: Automated reasoning in higher-order logic using the TPTP THF infrastructure. Journal of Automated Reasoning **3**(1), 1–27 (2010). https://doi.org/10.6092/ISSN.1972-5787/1710, https://doi.org/10.6092/issn.1972-5787/1710
33. Sutcliffe, G., Kotelnikov, E.: TFX: the TPTP extended typed first-order form. In: Konev, B., Urban, J., Rümmer, P. (eds.) Proc. 6th Workshop on Practical Aspects of Automated Reasoning. CEUR Workshop Proceedings, vol. 2162, pp. 72–87. CEUR-WS.org (2018), https://ceur-ws.org/Vol-2162/paper-07.pdf
34. Sutcliffe, G., Schulz, S., Claessen, K., Baumgartner, P.: The TPTP typed first-order form with arithmetic. In: Bjørner, N.S., Voronkov, A. (eds.) Proc. 18th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. LNCS, vol. 7180, pp. 406–419. Springer (2012). https://doi.org/10.1007/978-3-642-28717-6_32, https://doi.org/10.1007/978-3-642-28717-6_32
35. Sutcliffe, G., Suttner, C.B.: The TPTP problem library - CNF release v1.2.1. Journal of Automated Reasoning **21**(2), 177–203 (1998). https://doi.org/10.1023/A:1005806324129, https://doi.org/10.1023/A:1005806324129
36. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: Bodik, R., Majumdar, R. (eds.) Proc. 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 256–270 (2016). https://doi.org/10.1145/2837614.2837655
37. The Rocq Development Team: The Rocq reference manual – release 9.0.0. https://coq.inria.fr/doc/V9.0.0/refman (2024)

38. Trybulec, A., Blair, H.: Computer Assisted Reasoning with MIZAR. In: Joshi, A. (ed.) Proceedings of the 9th International Joint Conference on Artificial Intelligence. pp. 26–28. Morgan Kaufmann (1985)