Kalpa Publications in Computing

Volume 1, 2017, Pages 100–112

LPAR-21S: IWIL Workshop
and LPAR Short Presentations

Going Polymorphic – TH1 Reasoning for Leo-III*

Alexander Steen[1], Max Wisniewski[1], and Christoph Benzmüller[21]

[1] Freie Universität Berlin, Berlin, Germany
a.steen|m.wisniewski|c.benzmueller@fu-berlin.de
[2] University of Luxembourg, Luxembourg

Abstract

While interactive proof assistants for higher-order logic (HOL) commonly admit reasoning within rich type systems, current theorem provers for HOL are mainly based on simply typed λ-calculi and therefore do not allow such flexibility. In this paper, we present modifications to the higher-order automated theorem prover Leo-III for turning it into a reasoning system for rank-1 polymorphic HOL. To that end, a polymorphic version of HOL and a suitable paramodulation-based calculus are sketched. The implementation is evaluated using a set of polymorphic TPTP THF problems.

1 Introduction

Interactive proof assistants for higher-order logic such as Isabelle/HOL [32] are based on type systems that extend the simply typed λ-calculus with polymorphism, type classes and further type concepts. In contrast, automated theorem provers (ATP) for higher-order logic such as LEO-II [9] are still restricted to simply typed λ-calculus only. In particular, polymorphism has not yet found its way into such provers. Hence, middleware techniques, such as monomorphization [11], are still required to enable a fruitful interaction between these higher-order (HO) theorem proving systems.

In this paper, we present the theoretical foundations and technical details of the automated theorem prover Leo-III for polymorphic higher-order logic (HOL) with Henkin semantics and choice. The motivation is to turn Leo-III into a reasoner that is closer to Isabelle/HOL's native logic than other automated theorem provers and to better avoid the need for problem transformations. The progress of Leo-III's support for polymorphism has been strongly influenced by the recent development of the TH1 format for representing problems in rank-1 polymorphic HOL [26] extending the de facto standard THF syntax [35] for HOL.

The recent extension of TPTP's typed first-order format TFF to rank-1 polymorphism [12], yielding the so-called TF1 language, is closely related to the here examined TH1 logic. In fact, since TF1 is a fragment of TH1, Leo-III can also be employed for reasoning within TF1. There are increasingly many reasoning systems for first-order logic with rank-1 polymorphism, including Pirate, Zipperposition, CVC4 [5], Alt-Ergo [13] and further. However, in a higher-order setting there are, up to the author's knowledge, no stand-alone automated theorem provers

*This work has been supported by the DFG under grant BE 2501/11-1 (Leo-III).

that natively support polymorphism. The only exceptions here are HOL(y) Hammer [27] and the `tptp_isabelle` mode of Isabelle/HOL [32] that schedule proof tactics and use prover cooperation within HOL Light [22] and Isabelle/HOL, respectively.

The remainder of this paper is structured as follows: The syntax and semantics of rank-1 polymorphic higher-order logic is briefly sketched in §2. Subsequently, an adaption of Leo-III's paramodulation-based calculus to polymorphic HOL is presented. Its implementation and further underlying technical details are presented in §4. A preliminary evaluation of Leo-III's TH1 reasoning capabilities based on unpublished problems from TPTP version 7.0.0 is given in §5. Finally, §6 concludes the paper and sketches future work.

2 Polymorphic Higher-Order Logic

Simple type theory, also referred to as classical higher-order logic (HOL), is an expressive logic formalism that allows for higher-order quantification, that is, quantification over arbitrary set and function variables. It is based on the simply typed λ-calculus[1] and it has its origin in the work by Church [17].

Rank-1 polymorphic THF incorporates polymorphic types as well as explicit type abstraction and type application. In order to support these language features, a variation of HOL (simply called HOL in the following) based on a richer λ-calculus is introduced and considered throughout this paper. The system employed here corresponds to a restricted $\lambda 2$ system [3] and is also referred to as System F [20].

Types. Since polymorphic types must not appear left of a function type constructor, we can assume without loss of generality that all type quantifiers occur at outermost (prenex) position. This is reflected in the following two step definition of types. The non-polymorphic types, occurring within a polymorphic type's body, are called monotypes. Let $\mathcal{S}$ be a non-empty set of sort symbols. The set of monotypes $\mathcal{T}_{pre}$ is generated by the following abstract syntax ($\tau_i \in \mathcal{T}_{pre}$):

$$\begin{array}{rll} \tau_1, \ldots, \tau_n ::= & \zeta(\tau_1, \ldots, \tau_n) & \text{(Type application)} \\ \mid & \tau_1 \to \tau_2 & \text{(Abstraction type)} \\ \mid & \alpha & \text{(Type variable)} \end{array}$$

where $\zeta \in \mathcal{S}$ is a sort symbol (type constructor) of arity n and $\alpha \in \mathcal{V}^{\mathcal{T}}$ is a type variable. As a further restriction, type variables may only range over monotypes. Type constructors of arity zero are called base types. The set $\mathcal{T}$ of types is inductively defined as the least set satisfying

(i) if $\tau \in \mathcal{T}_{pre}$ is a monotype, then $\tau \in \mathcal{T}$, and

(ii) if $\tau \in \mathcal{T}$ is a type, then $(\forall \alpha.\ \tau) \in \mathcal{T}$ (called polymorphic type).

Types τ without free type variables are called *ground*.

For the semantics of types, we assume a universe $\mathcal{U}_0$ of sets with the usual properties [21] (often simply called $\mathcal{U}$ in the context of monomorphic HOL), where each element in $\mathcal{U}_0$ represents the denotation of a monotype, i.e. the set of elements having that particular type. Polymorphic types can be represented as products over $\mathcal{U}_0$ [37].

A *type interpretation* $\mathfrak{I}$ over $\mathcal{S}$ is a function mapping each type constructor $s \in \mathcal{S}$ of arity n to an appropriate denotation in $\mathcal{U}_0^n \to \mathcal{U}_0$. A *type variable assignment* $\mathfrak{g}$ maps type variables α

[1] For a detailed discussion of typed λ-calculi, we refer to the literature [4].

to an element $\mathfrak{g}(\alpha) \in \mathcal{U}_0$. Finally, the denotation $\|\tau\|^{\mathfrak{I},\mathfrak{g}}$ of a type $\tau \in \mathcal{T}$ with respect to a type interpretation $\mathfrak{I}$ and a type variable assignment $\mathfrak{g}$ is given by

$$\begin{aligned}
\|\zeta(\tau_1,\ldots,\tau_n)\|^{\mathfrak{I},\mathfrak{g}} &= \mathfrak{I}(\zeta)\left(\|\tau_1\|^{\mathfrak{I},\mathfrak{g}},\ldots,\|\tau_n\|^{\mathfrak{I},\mathfrak{g}}\right)\\
\|\tau \to \nu\|^{\mathfrak{I},\mathfrak{g}} &= \|\nu\|^{\mathfrak{I},\mathfrak{g}\|\tau\|^{\mathfrak{I},\mathfrak{g}}}\\
\|\alpha\|^{\mathfrak{I},\mathfrak{g}} &= \mathfrak{g}(\alpha)\\
\|\forall\alpha.\,\tau\|^{\mathfrak{I},\mathfrak{g}} &= \Pi_{\mathfrak{a}\in\mathcal{U}_0}\|\tau\|^{\mathfrak{I},\mathfrak{g}[\mathfrak{a}/\alpha]}
\end{aligned}$$

We assume that $\mathcal{S}$ consists of at least two elements $\{o,\iota\} \subseteq \mathcal{S}$, both of arity zero, where o and ι denote the type of Booleans and some non-empty domain of individuals, respectively, and are mapped to distinguished sets $\mathfrak{I}(o) = \{\mathrm{T},\mathrm{F}\}$ and $\mathfrak{I}(\iota) \neq \emptyset$.

Terms. The set of HOL terms Λ is given by the following grammar ($\tau,\nu \in \mathcal{T}$):

$$\begin{array}{lll}
s,t ::= X_\tau \in \mathcal{V} \mid c_\tau \in \Sigma & & \text{(Variable / Constant)}\\
\quad \mid (\lambda x_\tau.\, s_\nu)_{\tau\to\nu} \mid (s_{\tau\to\nu}\ t_\tau)_\nu & & \text{(Term abstraction / application)}\\
\quad \mid (\Lambda\alpha.\, s_\tau)_{\forall\alpha.\,\tau} \mid (s_{\forall\alpha.\,\tau}\ \nu)_{\tau[\nu/\alpha]} & & \text{(Type abstraction / application)}\\
\quad \mid (\forall\alpha.\, s_o)_o & & \text{(Type quantification)}
\end{array}$$

where c_τ denotes a typed constant from the signature Σ, $X_\tau \in \mathcal{V}$ is a variable and $\alpha \in \mathcal{V}^{\mathcal{T}}$ is a type variable. The type of a term is explicitly stated as a subscript but may be dropped for legibility reasons if obvious from the context. Note that, for using extrinsic typing [33, p. 111], each term $t \in \Lambda$ is inherently well-typed. Terms s_o of type o are called *formulae*. As a further restriction, only non-polymorphic types are allowed as argument to type applications.

Note that type quantification $\forall\alpha.\, s_o$ is explicitly included to the syntax of HOL terms due to the rank-1 polymorphism restrictions. In a more liberal type system, it can be regarded as defined by $\forall\alpha.\, s_o \equiv (\Lambda\alpha.\, s_o = \Lambda\alpha.\, T)$.

Σ is chosen to consist at least of the primitive logical connectives for disjunction, negation, and (polymorphic) equality, universal quantification and choice. Hence, we have $\{\vee_{o\to o\to o}, \neg_{o\to o}, =_{\forall\alpha.\,\alpha\to\alpha\to o}, \Pi_{\forall\alpha.\,(\alpha\to o)\to o}, \iota_{\forall\alpha.\,(\alpha\to o)\to\alpha}\} \subseteq \Sigma$. Binder notation is used whenever reasonable, i.e. by convention the term $\Pi_{\forall\alpha.\,(\alpha\to o)\to o}\ \tau\ (\lambda X_\tau.\, s_o)$ is abbreviated by $\forall X_\tau.s_o$. Also, type applications and infix notation are implicit whenever clear from the context, so $s_\tau = t_\tau$ is short for $(=_{\forall\alpha.\,\alpha\to\alpha\to o}\ \tau\ s_\tau\ t_\tau)$. The remaining logical connectives can be defined as usual, e.g. $s \wedge t := \neg(\neg s \vee \neg t)$ or $T := \neg\forall X_o.\, X$.

The semantics of monomorphic HOL can be found in the literature [7, 19, 21]. We briefly sketch the most important adaptions for rank-1 polymorphic HOL.

Let $\mathcal{S}$ be a set of sort symbols and $\mathfrak{I}$ a type interpretation over $\mathcal{S}$. An *interpretation* is a pair $\mathcal{M} = (\mathfrak{I},\mathcal{I})$ where $\mathcal{I} = (\mathcal{I}_\mathfrak{a})_{\mathfrak{a}\in\mathcal{U}_0}$ is a family of functions mapping each constant c_τ to its denotation. Given a variable assignment g and a type variable assignment $\mathfrak{g}$, we define the *valuation* $\|s_\tau\|^{\mathcal{M},g,\mathfrak{g}}$ of a HOL term s_τ to its *value* in $\|\tau\|^{\mathfrak{I},\mathfrak{g}}$ by

$$\begin{aligned}
\|c_\tau\|^{\mathcal{M},g,\mathfrak{g}} &= \mathcal{I}_{\|\tau\|^{\mathfrak{I},\mathfrak{g}}}(c_\tau)\\
\|X_\tau\|^{\mathcal{M},g,\mathfrak{g}} &= g(X_\tau)\\
\|\lambda X_\tau.\, s_\nu\|^{\mathcal{M},g,\mathfrak{g}} &= z \in \|\tau\|^{\mathfrak{I},\mathfrak{g}} \longmapsto \|s\|^{\mathcal{M},g[z/X_\tau],\mathfrak{g}}\\
\|s_{\tau\to\nu}\ t_\tau\|^{\mathcal{M},g,\mathfrak{g}} &= \|s_{\tau\to\nu}\|^{\mathcal{M},g,\mathfrak{g}}\left(\|t_\tau\|^{\mathcal{M},g,\mathfrak{g}}\right)\\
\|\Lambda\alpha.\, s_\tau\|^{\mathcal{M},g,\mathfrak{g}} &= \mathfrak{a} \in \mathcal{U}_0 \longmapsto \|s_\tau\|^{\mathcal{M},g,\mathfrak{g}[\mathfrak{a}/\alpha]}\\
\|s_{\forall\alpha.\,\tau}\ \tau\|^{\mathcal{M},g,\mathfrak{g}} &= \|s_{\forall\alpha.\,\tau}\|^{\mathcal{M},g,\mathfrak{g}}\left(\|\tau\|^{\mathfrak{I},\mathfrak{g}}\right)\\
\|\forall\alpha.\, s_o\|^{\mathcal{M},g,\mathfrak{g}} &= \mathrm{T} \text{ iff } \|s_\tau\|^{\mathcal{M},g,\mathfrak{g}[\mathfrak{a}/\alpha]} \equiv \mathrm{T} \text{ for every } \mathfrak{a} \in \mathcal{U}_0
\end{aligned}$$

$$\begin{array}{rcll}
\langle \tau \doteq \tau :: U, \sigma\rangle & \longrightarrow & \langle U, \sigma\rangle & \text{(TyDelete)}\\
\langle \alpha \doteq \tau :: U, \sigma\rangle & \longrightarrow & \langle U\{\tau/\alpha\}, \sigma \circ \{\tau/\alpha\}\rangle & \text{(TyBind)}\\
 & & \text{if } \alpha \notin fv(\tau) & \\
\langle \zeta(\overline{\tau_i}) \doteq \zeta(\overline{\nu_i}) :: U, \sigma\rangle & \longrightarrow & \overline{\langle \tau_i \doteq \nu_i :: U, \sigma\rangle} & \text{(TyDecomp)}\\
\langle \tau_1 \to \tau_2 \doteq \nu_1 \to \nu_2 :: U, \sigma\rangle & \longrightarrow & \langle \tau_1 \doteq \nu_1 :: \tau_2 \doteq \nu_2 :: U, \sigma\rangle & \text{(TyFunDecomp)}
\end{array}$$

Figure 1: Type unification rules

We assume that the primitive logical connectives are assigned their usual denotation. Note that both term abstraction and type abstraction evaluate to a function. The first one, $\lambda X_\tau.\, s_\nu$, denotes a function that maps each element d from the denotation set of τ to the value of s where X is substituted for d in s. The same holds for $\Lambda\alpha.\, s_\tau$ except that the function now maps elements of the type universe $\mathcal{U}_0$ (types themselves) to the appropriate valuation of s. The above definition yields so-called *standard models*. As a consequence of Gödel's Incompleteness Theorem, HOL with standard semantics (that is, with respect to standard models) is necessarily incomplete. However, if we assume a more generalized notion of HOL models, called general models or Henkin models, a meaningful notion of completeness can be achieved [23]. For Henkin models, we do not require $\|\tau \to \nu\|^{\mathfrak{I},\mathfrak{g}}$ to denote the complete set of functions from τ to ν, but rather a subset of those functions such that every HOL term denotes (i.e. $\|.\|^{\mathcal{M},g,\mathfrak{g}}$ is a total function). We assume Henkin semantics in the following. Validity and consequence are defined as usual.

HOL augmented with type operators and quantification over types was studied by Melham [29]. The here presented variant of HOL is more similar to the one used by HOL2P [37] or HOL-Omega [24] since it moreover includes universally quantified types. However, as opposed to HOL2P, we do not support type operator variables.

3 Calculus

The proof search of Leo-III is guided by a refutation-based calculus which uses the fact that $A_1, \ldots, A_n \vdash C$ if and only if $S = \{A_1, \ldots, A_n, \neg C\}$ is inconsistent. To that end, the set S consisting of axioms and the negated conjecture is transformed into an equisatisfiable set S' of clauses in (equational) clausal normal form. Then, $A_1, \ldots, A_n \vdash C$ is valid if the empty clause $\square$ can be derived from S' or, equivalently, if $\square \in \overline{S'}$ where $\overline{S'} \supseteq S'$ is a set of clauses closed under the inferences of the calculus.

A method for saturating a given set of HO clauses is resolution [6] as e.g. employed by LEO-II [9]. In first-order theorem proving, superposition [2] – a further restricted form of paramodulation [31] – is probably the most successful calculus, which improves naive resolution not only by a dedicated handling of equality, but also by using ordering constraints to restrict the number of possible inferences. However, currently no generalization for superposition, paramodulation or even ordered resolution for HOL exist since finding appropriate term orderings for higher-order terms is rather involved.

Leo-III employs a higher-order paramodulation calculus with specialized rules for treatment of extensionality, choice and defined equalities.

Type Unification Central to polymorphic calculi is the notion of *type unification*. Analogous to the unification of two terms s and t that returns, if existent, a substitution σ such that

$\boxed{\mathcal{INF}}$

$$\text{(Para)}\ \frac{\mathcal{C} \vee [l \simeq r]^{\mathfrak{tt}} \qquad \mathcal{D} \vee [s \simeq t]^{\alpha}}{\big(\mathcal{C} \vee \mathcal{D} \vee [s[l]_\pi \simeq t]^{\alpha} \vee [(s|_\pi)_\tau \simeq r_\nu]^{\mathfrak{ff}}\big)\sigma}\ \text{if } \sigma = mgu(\tau, \nu)$$

$$\text{(EqFac)}\ \frac{\mathcal{C} \vee [l \simeq r]^{\alpha} \vee [s \simeq t]^{\alpha}}{\big(\mathcal{C} \vee [l \simeq r]^{\alpha} \vee [l_\tau \simeq s_\nu]^{\mathfrak{ff}} \vee [r \simeq t]^{\mathfrak{ff}}\big)\sigma}\ \text{if } \sigma = mgu(\tau, \nu)$$

$$\text{(PrimSubst)}\ \frac{\mathcal{C} \vee [X_{\overline{\tau_i} \to o}\ \overline{t^i_{\tau_i}}]^{\alpha} \qquad P \in \mathcal{AB}^{\{\neg, \vee\} \cup \{\Pi\ \tau \mid \tau \in \mathcal{T}_{pre}\}}_{\overline{\tau_i} \to o}}{\big(\mathcal{C} \vee [X_{\overline{\tau_i} \to o}\ \overline{t^i_{\tau_i}}]^{\alpha}\big)\{P/X\}}$$

$\boxed{\mathcal{EXT}}$

$$\text{(FuncExt)}\ \frac{\mathcal{C} : \mathcal{D} \vee [s_{\tau \to \nu} \simeq t_{\tau \to \nu}]^{\alpha} \qquad a_\tau = arg^\tau_{\mathcal{C}}(\alpha)}{\mathcal{D} \vee [s\ a \simeq t\ a]^{\alpha}} \qquad \text{(BoolExt)}\ \frac{\mathcal{C} \vee [s_o \simeq t_o]^{\alpha}}{\mathcal{C} \vee [s_o \Leftrightarrow t_o]^{\alpha}}$$

$\boxed{\mathcal{UNI}}$

$$\text{(Decomp)}\ \frac{\mathcal{C} \vee [f(\overline{\tau_i})\overline{s_i} \simeq f(\overline{\nu_i})\overline{t_i}]^{\mathfrak{ff}}}{\big(\mathcal{C} \vee \overline{[s_i \simeq t_i]}^{\mathfrak{ff}}\big)\sigma}\ \text{if } \sigma = mgu\big(\overline{\tau_i \doteq \nu_i}\big)$$

$$\text{Triv}\ \frac{\mathcal{C} \vee [A \simeq A]^{\mathfrak{ff}}}{C} \qquad \text{Subst}\ \frac{\mathcal{C} \vee [X \simeq \mathbf{A}]^{\mathfrak{ff}} \qquad X \text{ does not occur in } A}{C\{A/X\}}$$

$$\text{FlexRigid}\ \frac{\mathcal{C} \vee [F_{\overline{\tau^n} \to \nu}\overline{\mathbf{U}^n} \simeq h_{\overline{\tau'^m} \to \nu}\overline{\mathbf{V}^m}]^{\mathfrak{ff}} \qquad G \in \mathcal{AB}^{\{h\}}_{\overline{\tau^n} \to \nu}}{\mathcal{C} \vee [F_{\overline{\tau^n} \to \nu}\overline{\mathbf{U}^n} \simeq h_{\overline{\tau'^m} \to \nu}\overline{\mathbf{V}^m}]^{\mathfrak{ff}} \vee [F = G]^{\mathfrak{ff}}}$$

Figure 2: Polymorphic Higher-Order Paramodulation Calculus

$s\sigma \equiv t\sigma$, type unification between two types τ and ν yields a substitution σ such that $\tau\sigma \equiv \nu\sigma$, if such a substitution exists. Type unification is given as a rewrite procedure given by the rewrite rules displayed in Fig. 1. Here, the type unification problem is represented as a tuple $\langle U, \sigma\rangle$ where U is a list of unsolved type equations (the double colon concatenates list entries, the empty list is denoted []) and σ is a type substitution. Since unification on types is essentially a first-order unification problem, it is decidable and unitary, i.e. gives a unique most general unifier if one exists. More precisely, if $\tau, \nu \in \mathcal{T}_{pre}$ are monotypes, τ and ν are unifiable if and only if $\langle[\tau \doteq \nu], \emptyset\rangle \longrightarrow^\star \langle[], \sigma\rangle$. In particular, it then holds that $mgu(\tau, \nu) := \sigma$ is a most general unifier for τ and ν.

Intuitively, when a monomorphic calculus would require two terms to have the same type, in a polymorphic equivalent we would check whether the two terms' types are unifiable and apply the calculus rule on the type unified clause resp. literal.

Higher-Order Paramodulation The polymorphic higher-order paramodulation calculus, denoted $\mathcal{PEP}$, is given by the union $\mathcal{PEP} = \mathcal{INF} \cup \mathcal{EXT} \cup \mathcal{UNI} \cup \mathcal{CNF}$ of the primary

$$\text{(CNFForall)}\ \frac{\mathcal{C} \vee [\forall X_\alpha.\, s_o]^{\mathfrak{tt}} \qquad Y_\alpha \text{ is fresh for } \mathcal{C}}{\mathcal{C} \vee [s_o\{Y/X\}]^{\mathfrak{tt}}}$$

$$\text{(CNFExists)}\ \frac{\mathcal{C} \vee [\forall X_\alpha.\, s_o]^{\mathfrak{ff}} \qquad sk_\alpha \text{ is new Skolem term}}{\mathcal{C} \vee [s_o\{sk/X\}]^{\mathfrak{ff}}}$$

$$\text{(CNFTyForall)}\ \frac{\mathcal{C} \vee [\forall \alpha.\, s_o]^{\mathfrak{tt}} \qquad X \text{ fresh type variable for } \mathcal{C}}{\mathcal{C} \vee [s_o\{X/\alpha\}]^{\mathfrak{tt}}}$$

$$\text{(CNFTyExists)}\ \frac{\mathcal{C} \vee [\forall \alpha.\, s_o]^{\mathfrak{ff}} \qquad \tau \text{ is new Skolem type}}{\mathcal{C} \vee [s_o\{\tau/\alpha\}]^{\mathfrak{ff}}}$$

$$\text{(OrTrue)}\ \frac{\mathcal{C} \vee [A \vee B]^{\mathfrak{tt}}}{\mathcal{C} \vee [A]^{\mathfrak{tt}} \vee [B]^{\mathfrak{tt}}}$$

$$\text{(Neg)}\ \frac{\mathcal{C} \vee [\neg A]^{\alpha}}{\mathcal{C} \vee [A]^{\overline{\alpha}}}$$

$$\text{(OrFalse)}\ \frac{\mathcal{C} \vee [A \vee B]^{\mathfrak{ff}}}{\mathcal{C} \vee [A]^{\mathfrak{ff}} \qquad \mathcal{C} \vee [B]^{\mathfrak{ff}}}$$

Figure 3: Clause Normalization $\mathcal{CNF}$

inference rules $\mathcal{INF}$, extensionality rules $\mathcal{EXT}$, unification rules $\mathcal{UNI}$, and clause normal form rules $\mathcal{CNF}$. We briefly discuss some of the involved inference rules.

An equation is a pair $s \simeq t$ of HOL terms, where $\simeq$ is assumed to be symmetric. A literal ℓ is a signed equation, written $[s \simeq t]^\alpha$ where $\alpha \in \{\mathfrak{tt}, \mathfrak{ff}\}$ represents the polarity of the literal. Non-equality predicates (terms s_o of type o) are represented as literals $[s_o \simeq \mathrm{T}]^\alpha$ and may simply be written $[s_o]^\alpha$. $s|_\pi$ is the subterm of s at position π (where positions are defined as usual), and $s[r]_\pi$ denotes the term created by replacing the subterm of s at position π by r. A clause $\mathcal{C}$ is a multiset of literals, denoting its disjunction. For brevity, if $\mathcal{C}$ and $\mathcal{D}$ are clauses and ℓ is a literal, we write $\mathcal{C} \vee \ell$ and $\mathcal{C} \vee \mathcal{D}$ for the (multiset) union $\mathcal{C} \uplus \{\ell\}$ and $\mathcal{C} \uplus \mathcal{D}$, respectively.

The calculus rules of $\mathcal{PEP}$ without clause normalization are shown in Fig. 2. The rules do not differ greatly from the monomorphic $\mathcal{EP}$ calculus [6] equivalents, except that type unification constraints are required to be solved prior to rule application. Hence, all literals $[s \simeq t]^\alpha$ remain invariantly well-typed (in particular, both sides of a literal have identical types).

Since term unification in HOL is undecidable, the unification constraints are encoded as negative equality literals – as seen in (Para) and (EqFac) – which may again be subject of further inferences. Intuitively, paramodulation is a conditional rewriting step that is justified if the unification tasks can be solved.

In HOL, the additional rule (PrimSubst) is required for completeness. Primitive substitutions guess the top-level logical structure of the instantiated term, while further decisions on P are delayed. The instantiations $\mathcal{AB}^H_{\overline{\tau_i}\to\nu}$ are called approximating bindings for type $\overline{\tau_i} \to \nu$ and head symbols $h \in H$ [8]. A term $t \in \mathcal{AB}^H_{\overline{\tau_i}\to\nu}$ is of the form $\lambda\overline{X^i_{\tau_i}}.\, h\left(Y_1\overline{X_i}\right)\ldots\left(Y_m\overline{X_i}\right)$ where $h_{\mu_1\to\ldots\to\mu_m\to\mu} \in H$ and the Y_i are fresh variables of appropriate type.

Extensionality aspects of HOL with Henkin semantics are dealt with on the calculus level (cf. (FuncExt) and (BoolExt)) rather than postulating corresponding extensionality axioms. In the context of the extensionality rule (FuncExt), the definition of $arg(.)$ is given by $arg^\tau_{\mathcal{C}}(\mathfrak{tt}) = X_\tau$, where X is a fresh variable wrt. $\mathcal{C}$, and $arg^\tau_{\mathcal{C}}(\mathfrak{ff}) = \left(sk\ \overline{tyfv(\mathcal{C})}\ \overline{fv(\mathcal{C})}\right)_\tau$ is a Skolem term, consisting of a fresh uninterpreted constant symbol sk of appropriate type applied to the free type variables $tyfv(\mathcal{C})$ and the free term variables $fv(\mathcal{C})$ of the clause $\mathcal{C}$. The augmented definition of Skolem terms is analogously used within clause normalization.

$\mathcal{UNI}$ defines further treatment of unification literals. The rules are based on Huet's pre-unification [25] augmented with type unification rules. The only notable difference to its

(1)	$[\forall\alpha.\,\exists X_\alpha\,.\,p_{\forall\beta.\beta\to o}\;\alpha\;X]^{\mathfrak{tt}}$	Axiom
(2)	$[\exists Y_{\zeta(i)}.\,p_{\forall\beta.\beta\to o}\;\zeta(i)\;Y]^{\mathfrak{ff}}$	Neg. Conj.
(3)	$[p\;\alpha\;(sk_{\forall\beta.\beta\to\alpha}\alpha)]^{\mathfrak{tt}}$	CNF 1
(4)	$[p\;\zeta(i)\;Y_{\zeta(i)}]^{\mathfrak{ff}}$	CNF 2
(5)	$[T\simeq T]^{\mathfrak{ff}}\vee[p\;\alpha\;(sk_{\forall\beta.\beta\to\alpha}\alpha)\simeq p\;\zeta(i)\;Y_{\zeta(i)}]^{\mathfrak{ff}}$	Para 3,4
(6)	$[sk_{\forall\beta.\beta\to\zeta(i)}\;\zeta(i)\simeq Y_{\zeta(i)}]^{\mathfrak{ff}}$	Triv, Decomp 5
(7)	$\square$	Subst 6

Figure 4: Example for a proof in $\mathcal{PEP}$

monomorphic equivalent is in rule (Decomp) where type arguments $\overline{\tau_i}$ and $\overline{\nu_i}$ to the function head symbols are eagerly solved and applied to the resulting new unification literals as well as to the remaining literals of the clause.

Clause Normalization The clause normalization rules $\mathcal{CNF}$ (Fig. 3) are in most cases identical to the monomorphic setting. However as noted above, Skolem terms now need to respect the free type variables of a clause as well. Two new rules are added: (CNFTyForall) and (CNFTyExists) where prefix type quantifications are normalized analogously to quantifiers on the term level: Universally quantified type variables become free type variables of the clause, whereas existentially quantified types variables are replaced by type witnesses, i.e. fresh type constants. Skolem types need to consider existing free type variables of the clause as well, hence witness types $\zeta(\overline{\alpha_i})$ are, in general, fresh type constructors ζ of arity $|tyfv(\mathcal{C})| = n$ applied to the free type variables $tyfv(\mathcal{C}) = \overline{\alpha_i}$ of clause $\mathcal{C}$.

Example As a short example to illustrate the polymorphic extensions of the calculus, we present a proof for $\forall\alpha.\,\exists X_\alpha.\,p_{\forall\beta.\beta\to o}\;\alpha\;X\vdash\exists Y_{\zeta(i)}.\,p_{\forall\beta.\beta\to o}\;\zeta(i)\;Y_{\zeta(i)}$, which is a theorem in Henkin semantics, in Fig. 4.

First the axiom (1) and the negated conjecture (2) are transformed into clause normal form. Here, (CNFTyForall) and (CNFExists) are applied to (1). After application of (CNFTyForall) to (1), α becomes a free (type) variable to the clause. Consequently, the Skolem term $sk_{\forall\beta.\beta\to\alpha}\alpha$ introduced by (CNFExists) is dependent on α. Next, a paramodulation with (3) on (4) is performed at root position ϵ. Since the types of both involved terms coincide (both have type o), the type unification yields an identity substitution. In the result, clause (5), the first literal represents the result of the conditional rewriting step whereas the second literal is the resulting unification constraint that is added by (Paramod). In the application of (Decomp) on (5), the types α and $\zeta(i)$ are unified – using $\sigma = \{\zeta(i)/\alpha\}$ as unifier – before the argument is transformed into a new unification task. In the last step, Y is bound to the Skolem term using (Subst). Since this results in an empty clause $\square$, the original problem is proven.

Completeness At this point, a completeness proof of $\mathcal{PEP}$ with respect to the polymorphic Henkin semantics as sketched in §2 is ongoing work.

In the development of Leo-III, we naturally aim at providing a general purpose prover, hence the underlying calculus should be complete. However, as a primary goal of this work, we intend to develop a theorem proving system that performs well in practice even if its known not to be complete on a theoretical level. We hence tend to accept an underlying calculus as long as it is "complete enough" for practical employment. This approach is also reflected by ongoing developments in the ATP community: Since the development of Gandalf [36], mode scheduling has been a major improvement to the overall success of ATPs. In mode scheduling,

theorem proving systems run different configurations (i.e. vary parameters that influence the search space traversal) within a single invocation using a limited resource scheduling strategy. Here, some modes may intentionally sacrifice completeness due to empirical evidence that the restrictions perform reasonably well on (some subset of) problems. Current HO ATP such as Satallax [14] and TPS use this approach as well.

In the context of HOL automation, there is additionally a natural limit on the completeness of concrete implementations. Not only that higher-order unification is undecidable, there is also the practical issue on how to restrict the primitive substitution rule (PrimSubst). Theoretically, (PrimSubst) may blindly instantiate free (top-level) variables of literals. However, in a practical scenario, one would restrain from unrestricted enumeration of instantiations since it further increases the search-space (in addition to the usual explosion known from first-order theorem proving). It is currently not known whether there are reasonable restrictions to the (PrimSubst) rule which preserves completeness.

4 Implementation

Leo-III is implemented in Scala and based on the associated system platform LEOPARD [39]. The latter provides a reusable framework and infrastructure for higher-order deduction systems, consisting of fundamental generic data structures for typed λ-terms, indexing means, a generic agent-based blackboard architecture [38], parser, and proof printer. Leo-III makes heavy use of these data structures and support means, and, on top of them, implements its specific calculus rules, control heuristics, the proof procedure and further functionality.

In this section, the most important adjustments required for turning Leo-III into a TH1-compliant ATP system are discussed. This of course includes, but is not limited to, proper augmentation of the implemented calculus rules and logical operations as sketched in §3. However, since Leo-III was designed from the beginning as an HOL reasoning system with support for polymorphism, only minor technical issues needed to be resolved for supporting reasoning within TH1. Below we briefly survey Leo-III's existing data structures for polymorphic term languages, and than we present some further TH1-related implementation details. The main reasoning loop (cf. further below) did not need any adjustments at all.

Term and type data structures The existing term and type data structures are already expressive enough to represent TH1 problems. Although currently not exploited, these data structures can capture full System F whose unrestricted use would, however, render the system inconsistent [18]. Nevertheless, conservative extensions to higher-rank polymorphism would already be supported on a syntactical basis.

Leo-III uses a so-called spine notation [16] that imitates the structure of first-order terms in a higher-order setting. Here, terms are either type abstractions, term abstractions or applications of the form $f \cdot (s_1; s_2; \ldots; s_n)$ where the head f is either a constant symbol, a bound variable or a complex term and the spine $(s_1; s_2; \ldots; s_n)$ is a linear list of arguments that are, again, spine terms. Note that if a term is β-normal, f cannot be a complex term.[2]

Additionally, the term representation employs explicit substitutions [1]. In a setting of explicit substitutions, substitutions are part of the term language and can thus be postponed and composed before being applied to the term. In the context of polymorphism, term normalization traversals carry two (i.e. one additional) substitutions, where the first represents a term substitution and the other a type substitution.

[2] The notion of β-normality for spine terms involves are spine merging rule (Nil) given by $f \cdot (s_1; s_2) \cdot (s_3; s_4) \to_{Nil} f \cdot (s_1; s_2; s_3; s_4)$. β-normalization remains confluent and strongly normalizing in this setting [16].

The term data structure moreover uses a locally nameless representation both at the type and term level, that extends De Bruijn indices [15] to (bound) type variables [28]. The definition of De Bruijn indices for type variables is analogous to the one for term variables. One of the most important advantages of nameless representations over representations with explicit variable names is that α-equivalence is reduced to syntactical equality, i.e. two terms are α-equivalent if their nameless representation is equal. In conjunction with perfect term sharing, α-equivalence checking is reduced to pointer comparison and hence possible in constant time (analogously for types).

TH1 parsing The syntactical parser of Leo-III is implemented using ANTLR4, which compiles a given grammar definition to actual code (here: Java code) that can be used as a library. The parser of Leo-III supports every common TPTP language, including CNF, FOF, TF0, TF1, TH0 and TH1, and is freely available.[3]

In a second step, the concrete syntax tree (CST) produced by the ANTLR-generated parser is translated into an abstract syntax tree (AST) representation. That is essentially motivated by the contract that while concrete parser implementations may change over time (and produce different CSTs), the AST representation remains stable. Consequently, the remaining system is decoupled from the parsing process and tedious (and error-prone) transformation methods converting the AST into λ-terms are only implemented once and thus increasingly reliable.

In final step, the AST representation of the input problem is interpreted and thereby translated into the above sketched term representation. Here, the only technical complication is the translation of *implicitly polymorphic* symbols of the TPTP language: Consider, for example, the THF (sub-)term `l = r` where `l` and `r` have the same type τ. The very same equality symbol may of course occur between terms of arbitrary type and is hence polymorphic. It is additionally *implicitly* polymorphic since the equality symbol is not passed a type argument as an analogous HOL formula $(=_{\forall\alpha.\,\alpha\to\alpha\to o}\ \tau\ l\ r)$ would require. Consequently, the processing operation needs to scan for implicit polymorphic symbols defined by the TPTP language and apply a type argument manually to the corresponding polymorphic constant.

Reasoning loop Leo-III implements, besides from using specialist agents, a stand-alone reasoning loop similar to the given-clause algorithm of E [34]. During the adaption to TH1 reasoning, this loop did not need any adjustments. This is primarily due to the strict separation of the loop, the state, the control layer (invoked by the loop) and the actual rule implementation (hidden from the loop). The changes to the calculus in §3 mainly affect the control layer and the rule implementation.

A simplified version of the saturation loop is given by

```
1  U := preprocess(input)
2  P := ∅
3  while(U ≠ ∅)
4    g := selectBest(U)
5    if (g = □) Theorem
6    else
7      U := U ∪ generate(g, P)
8      P := P ∪ {g}
9    endif
10 end
```

[3] The single grammar file containing all TPTP language specifications can be found at the Leo-III web site under http://inf.fu-berlin.de/~lex/leo3/#downloads.

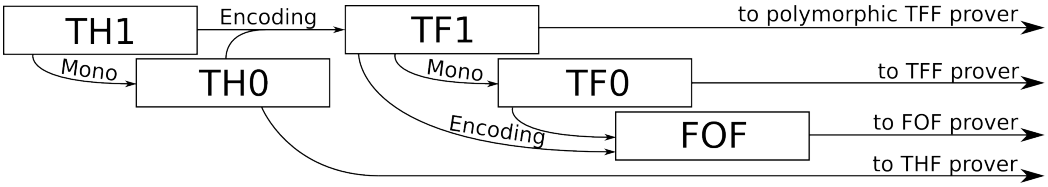

Figure 5: Cooperation schemes with external proving systems

After initial preprocessing of the input (line 1) the loop starts by selecting an unprocessed clause (line 4) and checking if a proof has already been found (line 5). The difference compared to the monomorphic implementation in `generate` (line 7) is that type unification is applied before applying the calculus rules. Here the generation applies simplification, unification and clausification before inserting the results into the unprocessed set. For a more thorough discussion of the implemented saturation procedure, we refer to the system description of Leo-III Version 1.1 [10].

Prover cooperation Although Leo-III is a stand-alone ATP, it heavily relies on cooperation with external reasoning systems, in particular with first-order automated theorem provers. The cooperation is thereby enabled by an encoding mechanism that translates HO clauses to first-order equivalents. Unlike its predecessor LEO-II, which encoded HO clauses to untyped first-order logic, the default translation scheme submits (polymorphically or monomorphically) typed first-order clauses to the external provers.

The employment of such cooperation for polymorphically typed HO clauses is straightforward due to the nature of the existing encoding framework: In Leo-III, HO clauses translation consists of various independent steps that can be flexibly combined (see Fig 5). As a first step, HO proof obligations are converted into polymorphically typed first-order clauses by eliminating all higher-order constructs such as anonymous functions, higher-order variables and constants [30]. Subsequently, the resulting polymorphic first-order clauses are converted to monomorphic first-order clauses by heuristic monomorphization or directly to untyped formulae using polymorphic encoding schemes [11]. The monomorphically typed first-order problems may be reduced to untyped first-order formulas using appropriate type encodings [11] as well. Of course, each intermediate result may already be given to an appropriate external theorem proving system, preferably early in the translation process to minimize encoding clutter.

In this setting, polymorphic higher-order clauses can either (a) be translated directly into polymorphically typed first-order problems (and are then handled as usual) or (b) undergo heuristic monomorphization, yielding monomorphic HO clauses. In the latter case, these monomorphized HO clauses can then, in turn, undergo the same above encoding procedure or be directly sent to external HO reasoners.

At the current state, external prover cooperation for TH1 is functional albeit experimental. In particular for large problem inputs (e.g. produced by Isabelle/HOL's Sledgehammer tool) the current monomorphization procedure needs to be improved.

5 Preliminary Evaluation

With the development of the TH1 format, Geoff Sutcliffe started collecting corresponding problems planned for release in TPTP version 7.0.0. The set of problems we tested our prover on

consisted of 666 TH1 problems, all of them theorems. Since Leo-III does not support arithmetic, 220 problem using TPTP interpreted arithmetic functions were removed from this set. The remaining 446 problems are reasonable for a first, preliminary evaluation of the above described calculus implementation. Each problem was given a timeout of 60 seconds on a Intel Core 2 Duo E8400 CPU @ 3.00 GHz with 2 GB RAM. Using internal reasoning only, Leo-III was able to prove 61 problems to be a theorem. It is important to note that most higher-order theorem provers gain their reasoning strength through first-order or SAT prover cooperation. All the above theorems have been found through internal reasoning alone. As soon as a more robust cooperation scheme (i.e. a more sophisticated monomorphization procedure) is implemented, this number will potentially increase.

The only other systems available at SystemOnTPTP that support TH1 are Isabelle/HOL using its `tptp_isabelle` mode and HOL(y) Hammer [27]. In contrast to those, Leo-III outputs proof objects for each theorem found using a TH1 variant of TSTP output. Unfortunately, we were not able to properly use HOL(y) Hammer, which returned `Unknown` in all cases. Also, there were some problems (e.g. DAT230^1.p) which could not be proved by Isabelle/HOL but could by Leo-III. A more thorough performance comparison is future work.

6 Summary and Further Work

In this paper we presented the calculus and implementation details of Leo-III's adjustments for reasoning within polymorphic higher-order logic. Due to the previously existing data structures of Leo-III, the overall adaption efforts for supporting the novel TH1 format were comparably low. The main adjustments to be made, are the implementation of a type unification, as well as a strict application of type unifiers to clauses prior to inference rule application. Hence, the most important step for developing a general purpose polymorphic HO-ATP is the design of appropriate data structures.

A major share of Leo-III's reasoning strength comes from corporation with first-order provers. Since the cooperation is not yet available for polymorphic formulae, a success rate of roughly 15% seems fair considering a straight-forward implementation of the calculus. Moreover, we are able to prove some problems neither HOL(y)-Hammer nor Isabelle/HOL could prove.

For future work, the main focus lies on the stable cooperation with first-order provers. Also combination of different type encodings and monomorphization techniques need to be analyzed. A proof of completeness for $\mathcal{PEP}$ is work in progress.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Levy. Explicit substitutions. In *Proc. of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 31–46, New York, NY, USA, 1990. ACM.

[2] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4(3):217–247, 1994.

[3] H. P. Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[4] H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013.

[5] C. Barrett et al. Cvc4. In *Computer aided verification*, pages 171–177. Springer, 2011.

[6] C. Benzmüller. Extensional higher-order paramodulation and RUE-resolution. In H. Ganzinger, editor, *Automated Deduction - CADE-16, 16th International Conference on Automated Deduction, Trento, Italy, July 7-10, 1999, Proceedings*, number 1632 in LNCS, pages 399–413. Springer, 1999.

[7] C. Benzmüller, C. Brown, and M. Kohlhase. Higher-order semantics and extensionality. *Journal of Symbolic Logic*, 69(4):1027–1088, 2004.

[8] C. Benzmüller and D. Miller. Automation of higher-order logic. In D. M. Gabbay, J. H. Siekmann, and J. Woods, editors, *Handbook of the History of Logic, Volume 9 — Computational Logic*, pages 215–254. North Holland, Elsevier, 2014.

[9] C. Benzmüller, L. C. Paulson, N. Sultana, and F. Theiß. The higher-order prover LEO-II. *Journal of Automated Reasoning*, 55(4):389–404, 2015.

[10] C. Benzmüller, A. Steen, and M. Wisniewski. Leo-III version 1.1 (system description). In T. Eiter and D. Sands, editors, *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR) — Short Papers*, Kalpa Puplications, Maun, Botswana, 2017. EasyChair. To appear.

[11] J. C. Blanchette, S. Böhme, A. Popescu, and N. Smallbone. Encoding monomorphic and polymorphic types. *Logical Methods in Computer Science*, 12(4), 2016.

[12] J. C. Blanchette and A. Paskevich. TFF1: the TPTP typed first-order form with rank-1 polymorphism. In M. P. Bonacina, editor, *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, volume 7898 of *LNCS*, pages 414–420. Springer, 2013.

[13] F. Bobot et al. The Alt-Ergo automated theorem prover, 2008.

[14] C.E. Brown. Satallax: An automatic higher-order prover. In *Automated Reasoning*, volume 7364 of *LNCS*, pages 111–117. Springer Berlin Heidelberg, 2012.

[15] N.G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[16] I. Cervesato and F. Pfenning. A linear spine calculus. *J. Logic and Computation*, 13(5):639–688, 2003.

[17] A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.

[18] T. Coquand. A new paradox in type theory. In *Proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science*, pages 7–14. Elsevier, 1994.

[19] D. Gallin. *Intensional and Higher-Order Modal Logic: With applications to Montague semantics*. North-Holland Mathematics Studies. Elsevier Science, 2011.

[20] J.Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[21] M. J. C. Gordon and A. M. Pitts. The HOL logic and system. In J. Bowen, editor, *Towards Verified Systems*, volume 2 of *Real-Time Safety Critical Systems*, chapter 3, pages 49–70. Elsevier Science B.V., 1994.

[22] John Harrison. Hol light: An overview. In *International Conference on Theorem Proving in Higher Order Logics*, pages 60–66. Springer, 2009.

[23] L. Henkin. Completeness in the theory of types. *J. Symb. Log.*, 15(2):81–91, 1950.

[24] Peter V. Homeier. The hol-omega logic. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 244–259, Berlin, Heidelberg, 2009. Springer-Verlag.

[25] G. P. Huet. A unification algorithm for typed lambda-calculus. *Theor. Comput. Sci.*, 1(1):27–57, 1975.

[26] C. Kaliszyk, G. Sutcliffe, and F. Rabe. TH1: the TPTP typed higher-order form with rank-1 polymorphism. In P. Fontaine, S. Schulz, and J. Urban, editors, *Proceedings of the 5th Workshop on Practical Aspects of Automated Reasoning*, volume 1635 of *CEUR Workshop Proceedings*, pages 41–55. CEUR-WS.org, 2016.

[27] C. Kaliszyk and J. Urban. HOL (y) hammer: Online ATP service for HOL Light. *Mathematics*

in Computer Science, 9(1):5–22, 2015.

[28] A. J. Kfoury, S. Ronchi Della Rocca, J. Tiuryn, and P. Urzyczyn. Alpha-conversion and typability. *Inf. Comput.*, 150(1):1–21, 1999.

[29] T. F. Melham. The HOL logic extended with quantification over type variables. *Formal Methods in System Design*, 3(1-2):7–24, 1993.

[30] Jia Meng and Lawrence C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1):35–60, 2008.

[31] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier, 2001.

[32] T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[33] B.C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[34] S. Schulz. E - a brainiac theorem prover. *AI Commun.*, 15(2,3):111–126, August 2002.

[35] G. Sutcliffe and C. Benzmüller. Automated reasoning in higher-order logic using the TPTP THF infrastructure. *J. Formalized Reasoning*, 3(1):1–27, 2010.

[36] T. Tammet. A resolution theorem prover for intuitonistic logic. In M. A. McRobbie and J. K. Slaney, editors, *Automated Deduction - CADE-13, 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, July 30 - August 3, 1996, Proceedings*, volume 1104 of *Lecture Notes in Computer Science*, pages 2–16. Springer, 1996.

[37] N. Völker. HOL2P – a system of classical higher order logic with second order polymorphism. In *Theorem Proving in Higher Order Logics*, pages 334–351. Springer, 2007.

[38] M. Wisniewski and C. Benzmüller. Is it reasonable to employ agents in theorem proving? In J. van den Heerik and J. Filipe, editors, *Proceedings of the 8th International Conference on Agents and Artificial Intelligence*, volume 1, pages 281–286, Rome, Italy, 2016. SCITEPRESS – Science and Technology Publications, Lda.

[39] M. Wisniewski, A. Steen, and C. Benzmüller. LeoPARD - A generic platform for the implementation of higher-order reasoners. In M. Kerber et al., editors, *Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA, July 13-17, 2015, Proceedings*, volume 9150 of *LNCS*, pages 325–330. Springer, 2015.