A Calculus and a System Architecture for Extensional Higher-Order Resolution*

Christoph Benzmüller
Department of Mathematical Sciences
Carnegie Mellon University, Pittsburgh, USA
chrisb@cs.cmu.edu|chris@cs.uni-sb.de

June 2, 1997

Abstract

The first part of this paper introduces an extension for a variant of Huet's higher-order resolution calculus [Hue72, Hue73] based upon classical type theory (Church's typed λ-calculus [Chu40]) in order to obtain a calculus which is complete with respect to Henkin models [Hen50]. The new rules connect higher-order pre-unification with the general refutation process in an appropriate way to establish full extensionality for the whole system. The general idea of the calculus is discussed on different examples.

The second part introduces the LEO system which implements the discussed extensional higher-order resolution calculus. This part mainly focus on the embedding of the new extensionality rules into the refutation process and the treatment of higher-order unification.

1 Introduction

Many mathematical problems can be expressed shortly and elegantly in higher order logic whereas they often lead to unnatural and inflated formulations in first-order logic, e.g., when coding them into axiomatic set theory.

On the other hand automated theorem proving in higher-order logic is much more complicated than in first-order because of additional challenging problems like the undecidability and complexity of higher-order unification, the need for a special treatment of predicate variables (e.g., by primitive substitution) or the handling of equality and extensionality. Nonetheless, proof systems using higher-order logic like TPS [ABI+96, AINP90], PVS [ORS92] or HOL [GM93]

*This work was supported by the HOTEL project of the Deutsche Forschungsgemeinschaft (DFG) and the 'Studienstiftung des deutschen Volkes'.

have demonstrated that automated higher-order theorem proving is feasible in practice (e.g., see theorems proved by TPS discussed in [ABI+96]). But higher-order theorem provers still have not reached the same power in handling huge and deep search spaces as advanced first-order provers. Among the most successful first-order provers are resolution style provers like OTTER [McC94], the Boyer-Moore theorem prover [BKM] or the recently developed system SPASS [WGR96].

Unfortunately many of the advanced techniques introduced in first-order systems, like sort systems [SS89], indexing [Gra95] or rewriting techniques are either not fully examined and adapted to the higher-order setting or are still not used in many systems. And even if the resolution approach was very successful in first-order, there is still no higher-order prover known to the author which uses a resolution style calculus.

Therefore the LEO[1] theorem prover for classical higher-order logic builds upon a variant of Huet's higher-order resolution approach [Hue72, Hue73] and tries to adapt as many as possible of the advanced first-order techniques (sorts, indexing, equality, strategies) to the higher-order setting. Hence LEO wants to demonstrate that higher-order resolution can be an alternative to other higher-order approaches.

Two of the main problems for higher-order resolution are the undecidability of higher-order unification and the need for a special treatment of predicate variables. To handle the first problem LEO uses, like TPS, higher-order pre-unification [Hue75] which is sufficient within a refutation approach. Because of the undecidability of higher-order unification (and pre-unification) LEO delays unification in resolution and factorization rules by adding unification constraints to the resulting clauses. To tackle the second problem LEO uses a primitive substitution rule which allows instantiating any predicate variable occurring at head position in a literal by a general binding that imitates a logical connective or quantifier.

Concerning the adaption of techniques from first-order resolution to higher-order resolution some important cornerstones have been established by the sorted higher-order resolution calculus described in [Koh94a, Koh94b] or the higher-order indexing techniques examined in [Kle97]. Indeed, the treatment of equality and extensionality is still a challenging problem for all higher-order theorem provers. In fact there is no automated higher-order theorem prover known to the author which can effectively and without using the 'right' extensionality axioms handle full extensionality and which is complete with respect to Henkin models [Hen50]. Thus easy looking examples like $\forall p_{o\to o}, a_o, b_o \,.\, p(a \wedge b) \Rightarrow p(b \wedge a)$ or $\forall p_{o\to o}, a_o, b_o \,.\, (pa \wedge pb) \Rightarrow p(b \wedge a)$ can often not be proven automatically by existing systems.

In the first part of this paper we introduce an extension of the Huet style cal-

[1] **Logical Engine for Omega.** The LEO project is strongly connected to the OMEGA project [BCF+97] and LEO's main intention is to become a powerful subsystem of OMEGA.

culus introduced in [Koh94a, Koh94b][2] in order to establish full extensionality. Although a formal proof for this conjecture is still lacking, the different examples discussed in this paper provide strong evidence for the Henkin completeness of the extended resolution calculus.

LEO wants to reach a considerable power on its own, but this prover is mainly intended to be used as a subsystem and logical engine for the OMEGA proof development environment [BCF+97, HKK+94b]. Within the OMEGA system, LEO will be called on specific higher-order subproblems or it will be used within other components, e.g., the proof planner. As a subsystem of OMEGA, LEO will have to cope mainly with subproblems which presume an appropriate and powerful treatment of equality and extensionality in higher-order logic.

2 Preliminaries

We consider a higher-order logic based on Church's simply typed lambda calculus [Chu40] and choose the set of basetypes $\mathcal{BT}$ to consists of the types ι and o, where o denotes the set of truth values and ι the set of individuals. The set of all types $\mathcal{T}$ is inductively defined over $\mathcal{BT}$ and the type constructor $\rightarrow$. We assume that our signature Σ contains a countably infinite set of variables $\mathcal{V}_\tau$ and constants $\mathcal{C}_\tau$ for every type $\tau \in \mathcal{T}$. Additionally we postulate the existence of the logical connectives $\neg_{o\rightarrow o}$, $\vee_{o\rightarrow o\rightarrow o}$, $\Pi_{(\alpha\rightarrow o)\rightarrow o}$ (in short Π^α) and $=_{\alpha\rightarrow\alpha\rightarrow o}$ (in short $=^\alpha$) for every type $\alpha \in \mathcal{T}$ in Σ. These constants denote their intuitive semantical counterparts.

The remaining logical connectives are defined as abbreviations of the given ones $(\mathbf{A} \wedge \mathbf{B} \doteq \neg(\neg\mathbf{A} \vee \neg\mathbf{B}), \mathbf{A} \Rightarrow \mathbf{B} \doteq \neg\mathbf{A} \vee \mathbf{B}, \mathbf{A} \equiv \mathbf{B} \doteq (\mathbf{A} \Rightarrow \mathbf{B}) \wedge (\mathbf{B} \Rightarrow \mathbf{A}), \forall X_\alpha.\mathbf{A}_o \doteq \Pi_{(\alpha\rightarrow o)\rightarrow o}(\lambda X_\alpha.\mathbf{A}), \exists X_\alpha.\mathbf{A}_o \doteq \neg\forall X_\alpha.\neg\mathbf{A}_o)$.

If the type of a symbol is uniquely determined by the given context we avoid its explicit mention.

To ease readability we assume right-associativity for the type constructor $\rightarrow$ and left-associativity of function application ($A_{\alpha\rightarrow\beta\rightarrow\gamma}B_\alpha C_\beta \doteq ((A_{\alpha\rightarrow\beta\rightarrow\gamma}B_\alpha)$ $C_\beta)$). Sometimes we abbreviate function applications by $h_{\alpha_1\rightarrow\cdots\rightarrow\alpha_n\rightarrow\beta}\overline{\mathbf{U}^n_{\alpha_n}}$ which stands for $(\cdots(h_{\alpha_1\rightarrow\cdots\rightarrow\alpha_n\rightarrow\beta}\mathbf{U}^1_{\alpha_1})\cdots\mathbf{U}^n_{\alpha_n})$. A dot "." occurring in a λ-term stands for a left bracket whose mate is as far to the right as consistent with all other brackets and the construction of the term. We allow further to avoid brackets in every case, where the construction of an expression is uniquely determined by the context.

Different from many other notational conventions we write variables in upper case letters and constants in lower case variables. As metavariables for λ-terms we use bold capital letters.

[2] [Koh94a, Koh94b] introduced a sorted variant for Huet's higher-order resolution approach. Note that we are not interested in sorts here and therefore we discuss an extension of the unsorted fragment of the calculus introduced [Koh94a, Koh94b].

We introduce α-, β- and η-*conversion* rules and refer to [Bar84]. It is also well known for the typed λ-calculus (e.g., [Bar84]) that each λ-term can be transformed into a unique β-, η- or $\beta\eta$-*normal form*.

The definitions of *free* and *bound* variables, *substitutions* and the *application of substitution* are according to [Bar84].

When we speak of a *Skolem term* s_α *for a clause* C and $\{X^1_{\alpha^1} \cdots X^n_{\alpha^n}\}$ is the set of free variables occuring in C, then s_α is an abbreviation for the term $(f^n_{\alpha^1 \to \ldots \to \alpha^n \to \alpha} X^1 \cdots X^n)$, where f is a new constant from $\mathcal{C}_{\alpha^1 \to \ldots \to \alpha^n \to \alpha}$ and n specifies the number of necessary arguments[3] for f.

For a general introduction to higher-order unification and especially for the definition of a set of *general bindings* $\mathcal{GB}^h_\gamma$ for a type γ and a (head-)constant h we refer to [GS89].

Our calculus will be defined on *clauses* which are disjunctions of *literals* (e.g., $[q_{\alpha \to o} X_\alpha]^T \vee [p_{\alpha \to o} X_\alpha]^F \vee [c_\alpha = X_\alpha]^F$). For literals we differentiate between *pre-literals* and *proper literals*. A *pre-literal* consists of an arbitrary λ-term $\mathbf{N}_o$ with type o and a polarity T or F stating if this literal is positive or negative (e.g., $[\forall X_\alpha . p_{\alpha \to o} X]^F$). We call a literal *proper* iff it contains no logical constant beside $=$ at head position.

We further differentiate between *positive literals*, *negative literals* and *unification constraints*. Unification constraints are special negative literals and their atoms are equations. We sometimes call unification constraints also *unification pairs*, especially if we want to focus on the two terms - the left hand side and the right hand side of the equation - defining the unification problem.

A clause C is in *clause normal form* iff it is a *proper clause* which means that all literals of C have to be proper. A clause which is not in clause normal form is called *pre-clause*. Clauses which consists only of unification constraints are called *almost empty*.

Any unification constraint $U \doteq [X_\alpha = \mathbf{N}_\alpha]^F$ or $U \doteq [\mathbf{N}_\alpha = X_\alpha]^F$ is *solved* iff X_α is not free in $\mathbf{N}_\alpha$. In this case X is called the *solved variable* of U.

Let $C \doteq L^1 \vee \cdots \vee L^n \vee U^1 \vee \cdots \vee U^m$ be a clause with unification constraints $U^1 \vee \cdots \vee U^m$ $(1 \leq m)$. Then a disjunction $U^{i_1} \vee \cdots \vee U^{i_k}$ $(i_j \in \{1, \cdots, m; 1 \leq j \leq k)$ of solved unification constraints occuring in C is called *solved for* C iff for every $U^{i_j} (1 \leq j \leq k)$ holds: the solved variable of U^{i_j} does not occur free in any of the U^{i_l} for $l \neq j; 1 \leq l \leq k$.

Note that each solved set of unification constraints E for a clause C can be associated with a substitution subst_E which is uniquely determined by the solved variables of E and their dedicated unification partners.

For a discussion of *standard models*, *Henkin models* and *general models* see [Hen50]. For a discussion of the role of Leibniz equality within general models see also [And72].

The *functional extensionality principle* says that two functions are equal iff they are equal on all arguments, which can be formulated by the following

[3] See discussion of skolemization in subsection 3.1.

basetypes	ι, o, α, β
functional types	$\alpha \to \beta, \alpha, \beta, \gamma, \delta$
variables	$X_\alpha, M_{\iota\to\iota}, P_o, Q_{\alpha\to o}$
constants	$a_\iota, b_\iota, m_{\iota\to\iota}, f_{\alpha\to\beta}, p_o$
applications	$(p_{\iota\iota\to o} a_\iota X_\iota), (h_{\alpha_1\to\cdots\to\alpha_n\to\beta} \overline{\mathbf{U}^n_{\alpha_n}})$
abstractions	$(\lambda X_\alpha . f_{\alpha\to\beta} X), (\lambda X_\alpha . \lambda P_{\alpha\to o} . PX)$
metavariables for ho-terms	$\mathbf{A}_\iota, \mathbf{B}_o, \mathbf{M}_{\alpha\to\beta}$
positive literals	$[\forall X_\alpha . Q_{\alpha\to o} X]^T, [c_\alpha = f_{\alpha\to\alpha} X_\alpha]^T$
negative literals	$[p_{\alpha\to o} X_\alpha]^F, [(\lambda X_\alpha . \lambda P_{\alpha\to o} . PX) b_\alpha Q_{\alpha\to o}]^F$
unification constraints	$[c_\alpha = f_{\alpha\to\alpha} X_\alpha]^F, [P_o = \exists X_\beta . \neg p_{\beta\to o} X]^F$
proper literals	$[Q_{\alpha\to o} a_\alpha]^T, [p_o]^F, [P_o = \exists X_\beta . \neg p_{\beta\to o} X]^F$
pre-literals	$[\forall X_\alpha . Q_{\alpha\to o} X]^T, [p_o \Rightarrow q_o]^F$
proper clauses	$[p_o]^T \vee [p_{\alpha\to o} X_\alpha]^F \vee [c_\alpha = X_\alpha]^F$
pre-clauses	$[\forall P_o . P]^T \vee [p_{\alpha\to o} X_\alpha \Rightarrow q_o]^F \vee [p_o]^T$
metavar. for clauseparts	C, D
metavar. for uni. constr. parts	E

Figure 1: Examples for some notational conventions

schematic λ-term: $\forall M_{\alpha\to\beta} . \forall N_{\alpha\to\beta} . (\forall X . (MX) = (NX)) \equiv (M = N)$. This term is schematic with respect to the (arbitrary) types α and β. The *extensionality principle for truth values* states that on the set of truth values equality and equivalence relation coincide: $\forall P_o . \forall Q_o . (P = Q) \equiv (P \equiv Q)$. Note that in Henkin models both principles are valid.

Figure 1 provides some examples for the introduced notational conventions.

3 The calculus

In this section we will introduce the calculus forming the theoretical foundation of the LEO-System. The basis of this calculus is given by the unsorted fragment of the Huet style calculus discussed in [Koh94a, Koh94b]. We will introduce an extension for this fragment in order to obtain Henkin completeness. Note that we assume commutativity and associativity of $\vee$ for all our rules.

3.1 Clause normalization

Clause normalization is very similar to the first-order case, except for the treatment of skolemization. If skolemization is handled in the intuitive first-order way one can prove the axiom of choice [Mil83]. A solution due to [Mil83] is to associate with each Skolem constant the minimum number of arguments the constant has to be applied to. As we assume that the reader is familiar with the rules for clause normalization from first-order logic we do not introduce these set of rules here and assume that each given higher-order proof problem

$\mathcal{P}$ can be transformed by these rules into a set of clauses $\mathcal{CNF}(\mathcal{P})$. A more detailed discussion of clause normalization in higher-order logic can be found in [Koh94a, Koh94b].

We allow clause normalization to be applied also on pre-clauses like $[p_{\iota\to o}a_\iota]^F \vee [P_{\iota\to o}b_\iota \wedge q_o]^T$. The result is a set of proper clauses. For the above pre-clause we get $[p_{\iota\to o}a_\iota]^F \vee [P_{\iota\to o}b_\iota]^T$ and $[p_{\iota\to o}a_\iota]^F \vee [q_o]^T$.

We introduce the following convention: clause normalization replaces in every given input problem each equality symbol $=^\alpha$ by its Leibniz definition $\lambda X_\alpha Y_\alpha . \forall P_{\alpha\to o} . PX \Rightarrow PY$. Therefore a result of applying clause normalization to any given set of formulas results in a set of clauses containing no equality symbol and therefore no unification constraint.

Clause normalization of partial-clauses differs from this and equality symbols in unification constraints are never replaced but remain unchanged instead.

3.2 Higher-order pre-unification

For this section we refer to the (pre-)unification rules discussed in [Koh94a, Koh94b]. But deviating from the rules presented there we are not interested in sorts here and we do not introduce extra-logical variable conditions[4]. Additionally we lift the pre-unification rules as shown in figure 2 on clauses level. They work directly on the unification constraint parts of the clauses. For all the rules in figure 2 we assume the symmetry of $=$.

The first four rules (α, η, *Dec* and *Triv*) in figure 2 define the deterministic part of higher-order unification, namely simplification. Rule α eliminates the top λ-binder of both sides in the unification constraint and replaces the variables of the λ-binders by a Skolem term[5] s_α for this clause.

The η-rule is applied in cases when only one hand side is a λ-abstraction[6]. The λ-abstracted term is again modified by removing the λ-binder and by substituting a Skolem term s_α for the bound variable X. The non-abstracted side is simply applied on s_α.

[4] [Koh94a, Koh94b] introduces these extra-logical variable conditions, for example, to treat higher-order skolemization in a special way. See also footnote 5.

[5] In [Koh94a, Koh94b] instead of a Skolem term s_α a new special variable $\mathbf{Z}^0$ is introduced. This is because the author wants to avoid an infinitive set of constants in the signature and the construction of Skolem terms. In an extra-logical form (variable conditions) he takes care of the relationship between the different variables occuring in each clause and therefore he can judge based upon this variable condition which substitutions are legal in a given context and which are not. Thus each new symbol introduced by one of the inference rules in [Koh94a, Koh94b] becomes a variable, either of positive character (usual new variables), negative character (usual new constant) or of a special 0-character (close to Skolem terms, but weaker in some sense since they are not allowed being bound against any other variable). Consequently with each new introduced variable the extra-logical variable condition has to be updated. Here we are not interested in such an extra-logical approach and the reason we especially introduce a Skolem term here will be clarified in subsection 3.4 when we discuss the extensionality rule *Func*.

[6] Note that this rule is superfluous if we presuppose $\alpha\beta$-normal forms. But since we do not do that this rule is necessary, for example, to show the η-equality of two terms.

$$\frac{C \vee [(\lambda X_\alpha . \mathbf{A}) = (\lambda Y_\alpha . \mathbf{B})]^F \quad s_\alpha \text{ Skolem term for this clause}}{C \vee [[s/X]\mathbf{A} = [s/Y]\mathbf{B}]^F}\ \alpha$$

$$\frac{C \vee [(\lambda X_\alpha . \mathbf{A}) = \mathbf{B}]^F \quad s_\alpha \text{ Skolem term for this clause}}{C \vee [[s/X]\mathbf{A} = (\mathbf{B}s)]^F}\ \eta$$

$$\frac{C \vee [h\overline{\mathbf{U}^n} = h\overline{\mathbf{V}^n}]^F}{C \vee [\mathbf{U}^1 = \mathbf{V}^1]^F \vee \ldots \vee [\mathbf{U}^n = \mathbf{V}^n]^F}\ Dec \qquad \frac{C \vee [\mathbf{A} = \mathbf{A}]^F}{C}\ Triv$$

$$\frac{C \vee [F_\gamma \overline{\mathbf{U}^n} = h\overline{\mathbf{V}}]^F \quad \mathbf{G} \in \mathcal{GB}^h_\gamma}{C \vee [F = \mathbf{G}]^F \vee [\mathbf{G}/F]([F\overline{\mathbf{U}} = h\overline{\mathbf{V}}]^F)}\ Flex - Rigid$$

$$\frac{C \vee [F_\gamma \overline{\mathbf{U}^n} = H_\delta \overline{\mathbf{V}}]^F \quad \mathbf{G} \in \mathcal{GB}^{g_\delta}_\gamma \text{ for some constant } g_\delta}{C \vee [F = \mathbf{G}]^F \vee [\mathbf{G}/F]([F\overline{\mathbf{U}} = H\overline{\mathbf{V}}]^F)}\ Flex - Flex$$

Figure 2: Higher-order (pre-)unification rules

Decomposition is analogous to the first-order case and the rule *Triv* allows to remove reflexivity pairs. Rule *Dec* will be discussed again in connection with the extensionality rules in section 3.4.

Unfortunately the *Flex-Rigid* and *Flex-Flex* rules for higher order unification are not deterministic. Both rules have to deal with the problem of function or predicate variables at head position. In the flex-rigid case it is sufficient to instantiate this headvariable by a most general binding specified by the type of the headvariable itself and the rigid head. Fortunately the sets of general bindings are finite for any type and any rigid head and consequently the *Flex-Rigid* rule is only finitely branching. In contrast to this the *Flex-Flex* rule is infinitely branching since there are infinitely many constant functions which can be substituted for both heads and which make both terms equal. Additionally the *Flex-Rigid* rule introduces redundancies and it is no longer the case that every successful branch of the unification search tree leads to a distinct solution.

The intention of a theorem prover like LEO which uses refutation principle is not to enumerate all unifiers for a given unification problem but to seek for one possible instantiation of a given problem which leads to the contradiction. This makes it possible to use pre-unification instead of general unification and thereby to avoid the usage of the *Flex-Flex* rule in LEO.

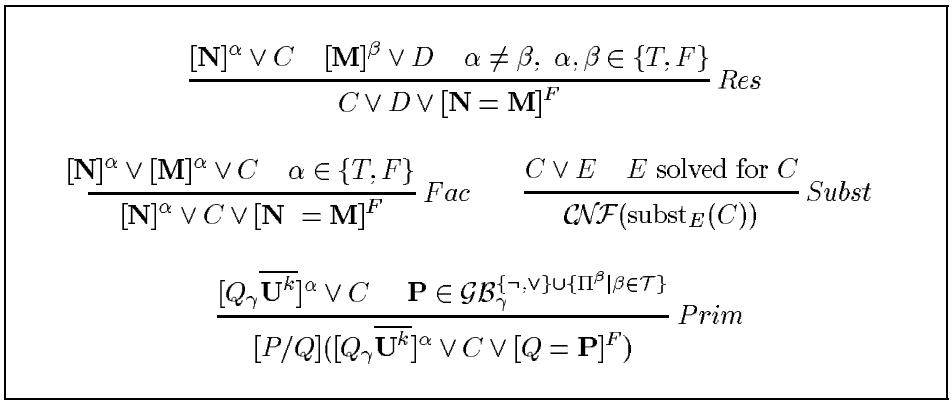

Figure 3: Higher-order resolution rules

The four simplification rules together with the *Flex-Rigid* rule and the convention that *flex-flex* pairs can be viewed as pre-solved define the calculus $\mathcal{PREUNI}$.

3.3 Higher-order resolution

The core of the higher-order resolution calculus is given by the three rules shown in figure 3. As in first-order we introduce the resolution and factorization rules *Res* and *Fac*. But instead of solving the unification problems immediately within a rule application we delay their solution and incorporate them explicitly in form of unification constraints in the resulting clauses. Note that the resolution rule as well as the factorization rule is allowed operating on unification constraints.

Delaying the unification problems until we reach an almost empty clause is nice in theory but does not work in practice as the search space explodes. Instead we are interested in solving the unification constraints as soon as possible. Most important is that with premature unification all clauses with an unsolvable unification constraint can be filtered out[7]. Additionally unification might specialize our clauses and probably instantiate flexible literals with constant heads and hence decrease the search space. Therefore rule *Subst*[8] allows applying the substitution subst_E determined by a disjunction of unification constraints E which are solved for $C \vee E$ back to the other literals in C. Since applying a substitution subst_E might result in pre-clause C' we have to apply clause normalization in order to obtain a proper clause.

[7] As we will see later this solution is too strong if we want to be complete in Henkin models since an unsolvable unification constraint might be solvable by using the extensionality rules.

[8] Note that this rule is more of practical interest, since theoretically in can be avoided.

To find a refutation for a given problem we might have to instantiate some predicate variables at head positions of some literals in the given clauses by certain formulas. But unfortunately these instantiations can not be generally determined by pre-unification within the refutation process. Therefore we introduce the primitive substitution rule *Prim*. This rule permits to instantiate each headvariable Q_γ of a flexible literal with a general binding $\mathbf{P}$ of type γ which imitates one of the logical constants in $\{\neg, \vee\} \cup \{\Pi^\beta | \beta \in \mathcal{T}\}$. By substituting $\mathbf{P}$ for Q in a clause C we obtain a pre-clause on which we have to apply clause normalization to get proper clauses. Note that the rule *prim-subst* is infinitely branching.

3.4 Extensionality

The set of rules introduced so far is not able to deal with extensionality in general and as a consequence examples like E1-E5 are not provable without using additional axioms for functional extensionality and/or extensionality on truth values. We want to emphasize that these problems are not specific for the resolution approach and that other higher-order theorem provers will find them at least very difficult or tricky.

E1 $a_o \equiv b_o \Rightarrow (\forall P_{o\to o}.Pa \Rightarrow Pb)$

E2 $\forall P_{o\to o}.P(a_o \wedge b_o) \Rightarrow P(b \wedge a)$

E3 $\forall P_{o\to o}.(Pa_o \wedge Pb_o) \Rightarrow P(b \wedge a)$

E4 $(\forall X_\iota.\forall P_{\iota\to o}.(P(m_{\iota\to\iota}X) \Rightarrow P(n_{\iota\to\iota}X))) \Rightarrow (\forall Q_{(\iota\to\iota)\to o}.Q(\lambda X_\iota.mX) \Rightarrow Q(\lambda X_\iota.nX))$

E5 $(\forall X_\iota.\forall P_{\iota\to o}.P(m_{\iota\to\iota}X) \Rightarrow P(n_{\iota\to\iota}X)) \Rightarrow (\forall Q_{(\iota\to\iota)\to o}.Qm \Rightarrow Qn)$

In Problems E1,E2,E4 and E5 we have used Leibniz definition of equality to remove the intuitive equality symbols. E1 formulates the extensionality property for truth values: if a_o is equivalent to b_o then a_o is equal to b_o ($a_o \equiv b_o \Rightarrow a = b$)[9]. E2 states that any property which holds for $a_o \wedge b_o$ also holds for $b_o \wedge a_o$ (or simply that $a_o \wedge b_o = b \wedge a$). E3 says, that any predicate $P_{o\to o}$ which coincidently holds for a_o and b_o also holds for their conjunction. E4 can be interpreted as an instance of the ξ-rule[10] ($(\forall X_\iota.m_{\iota\to\iota}X = n_{\iota\to\iota}X) \Rightarrow (\lambda X_\iota.mX) = (\lambda X_\iota.nX)$). E5 is an instance of the functional extensionality axiom for type $\iota \to \iota$ ($\forall X_\iota.(m_{\iota\to\iota}X) = (n_{\iota\to\iota}X) \Rightarrow m = n$)[11].

[9] This is the interesting direction of the extensionality principle for truth values.
[10] See [Bar84].
[11] This is the interesting direction of the functional extensionality principle.

$$\frac{C \vee [\mathbf{M}_{\alpha\to\beta} = \mathbf{N}_{\alpha\to\beta}]^F \quad s_\alpha \text{ new Skolem term for this clause}}{C \vee [\mathbf{M}s = \mathbf{N}s]^F}\ Func$$

$$\frac{C \vee [\mathbf{M}_o = \mathbf{N}_o]^F}{\mathcal{CNF}(C \vee [\mathbf{M}_o \equiv \mathbf{N}_o]^F)}\ Equiv$$

$$\frac{C \vee [\mathbf{M}_\alpha = \mathbf{N}_\alpha]^F \quad \alpha \in \mathcal{BT},\ p_{\alpha\to o} \text{ new Skolem term for this clause}}{\begin{array}{c} C \vee [p\mathbf{M}]^T \\ C \vee [p\mathbf{N}]^F \end{array}}\ Leib$$

Figure 4: Extensionality rules

LEO shall especially deal with equality- and extensionality problems and hence should be able to solve such elementary problems like E1-E5 very fast. Therefore our goal is to find an extension of the given resolution calculus which on the one hand introduces full extensionality and on the other hand is useful for an implementation. Surely, the introduction of axioms for functional extensionality and and the extensionality axiom for truth values can solve the problem in theory but this will lead to an explosion of the search space which can not be handled very well in practice. Instead we do not change the purely negative resolution calculus by introducing axioms but introduce the rules shown in figure 4 for dealing with extensionality.

The first rule *Func* reflects the functional extensionality property but in a negative way: if two functions are not equal then there exists an argument s_α on which these functions differ. To ensure soundness s_α has to be a new Skolem term which contains all the free variables occuring in the given clause.

We are interested in adding Skolem terms to arbitrary terms of functional type with this rule, especially if the unification constraint is not unifiable. But note that we already introduced two rules – α and η in simplification (see figure 2) – which are very similar to this one. Therefore we can restrict this rule here to the case were $\mathbf{N}$ and $\mathbf{M}$ are non-abstractions. Or, to turn it around, we can remove the α and η rules from simplification if we consider the rule *Func* as purely typed-based and apply β-reduction to both hand sides of the modified unification constraint.

The second rule *Equiv* allows to replace each negated equality on type o by an equivalence. Therefore this rule reflects the extensionality property for truth values but like *Func* in a negative way: if two formulas are not equal then they

are also not equivalent.

The third rule *Leib* just instantiates the equality symbol by its Leibniz definition. Thereby we obtain a pre-clause $C \vee [\forall P_{\alpha\to o}.P\mathbf{M}_\alpha \Rightarrow P\mathbf{N}_\alpha]^F$ and by applying clause normalization we get two proper clauses $C \vee [pM]^T$ and $C \vee [pN]^F$ where $p_{\alpha\to o}$ is a new Skolem term.

We want to point out that the necessity of rule *Equiv* in connection with the pre-unification rules for dealing with extensionality is also discussed in [Koh95]. But the rules introduced there are not sufficient for full extensionality and, e.g., examples E4 and E5 are not provable.

Note that none of three new extensionality rules introduces any flexible literal and even better, they introduce no new free variable at all.

As mentioned before the new rules strongly connect the unification part of our calculus with the resolution part. In some sense they make the unification part extensional since they allow to modify unification problems which are not solvable by pre-unification alone in an extensional appropriate way and to translate them back into usual literals.

We will illustrate this idea by our five examples:

E1 $a_o \equiv b_o \Rightarrow (\forall P_{o\to o}.Pa \Rightarrow Pb)$

Clause normalization leads to (p is a new Skolem term):

c1: $\boxed{[pa]^T}$ *c2:* $\boxed{[pb]^F}$ *c3:* $\boxed{[a]^T \vee [b]^F}$ *c4:* $\boxed{[b]^T \vee [a]^F}$

By resolving *c1* against *c2* we get:

c5: $\boxed{[(pa) = (pb)]^F}$

Even if this is a non-unifiable unification constraint we can apply decomposition rule *Dec*:

c6: $\boxed{[a = b]^F}$

This is still a unification constraint of type o and we can apply rule *Equiv*:

c7: $\boxed{[a]^T \vee [b]^T}$ *c8:* $\boxed{[a]^F \vee [b]^F}$

The rest of the proof is obvious and we get the contradiction by the clauses *c3*, *c4*, *c7* and *c8*.

□ (LEO can find this proof within 0.5 sec. on a Sparc Workstation Ultra)

E2 $\forall P_{o\to o}.P(a_o \wedge b_o) \Rightarrow P(b \wedge a)$

Clause normalization to leads to (p is a new Skolem term):

$c1$: $\boxed{[p(a \wedge b)]^T}$ $c2$: $\boxed{[p(b \wedge a)]^F}$

By resolving $c1$ against $c2$ we get:

$c3$: $\boxed{[p(a \wedge b) = p(b \wedge a)]^F}$

On $c3$ we apply rule *Dec*:

$c4$: $\boxed{[(a \wedge b) = (b \wedge a)]^F}$

Rule *Equiv* is applicable and we get:

$c5$: $\boxed{[a]^F \vee [b]^F}$ $c6$: $\boxed{[a]^T}$ $c7$: $\boxed{[b]^T}$ $c8$: $\boxed{[a]^T \vee [b]^T}$

Resolving $c6$ and $c7$ against $c5$ leads to the contradiction.
□ (LEO can find this proof within 0.3 sec. on a Sparc Workstation Ultra)

E3 $\forall P_{o \to o}.(Pa_o \wedge Pb_o) \Rightarrow P(a \wedge b)$

By clause normalization we get (p is a new Skolem term):

$c1$: $\boxed{[pa]^T}$ $c2$: $\boxed{[pb]^F}$ $c3$: $\boxed{[p(a \wedge b)]^F}$

We resolve between clauses $c3$ and $c1$ and between $c3$ and $c2$:

$c4$: $\boxed{[p(a \wedge b) = pa]^F}$ $c5$: $\boxed{[p(a \wedge b) = pb]^F}$

We can apply the decomposition rule *Dec*:

$c6$: $\boxed{[(a \wedge b) = a]^F}$ $c7$: $\boxed{[(a \wedge b) = b]^F}$

Rule *Equiv* applied on $c6$ leads to:

$c8$: $\boxed{[a]^F \vee [b]^F}$ $c9$: $\boxed{[a]^T \vee [b]^T}$ $c10$: $\boxed{[a]^T}$

Similar for $c7$ we get:

$c11$: $\boxed{[a]^F \vee [b]^F}$ $c12$: $\boxed{[a]^T \vee [b]^T}$ $c13$: $\boxed{[b]^T}$

The rest of the refutation proof is obvious: Resolve $c10$ and $c13$ against $c8$ (or $c11$).
□ (LEO can find this proof within 0.5 sec. on a Sparc Workstation Ultra)

E4 $(\forall X_\iota.\forall P_{\iota\to o}.(P(m_{\iota\to\iota}X) \Rightarrow P(n_{\iota\to\iota}X))) \Rightarrow (\forall Q_{(\iota\to\iota)\to o}.Q(\lambda X_\iota.mX) \Rightarrow Q(\lambda X_\iota.nX))$

By clause normalization we get (q is a new Skolem term.):

$$c1: \boxed{[P(mX)]^F \vee [P(nX)]^T}$$

$$c2: \boxed{[q(\lambda X.mX)]^T} \qquad c3: \boxed{[q(\lambda X.nX)]^F}$$

Unfortunately the idea to resolve *c2* and *c3* immediately against *c1* does not lead to successful refutation. The resulting unification constraints are not solvable. Therefore we choose another way and resolve between *c2* and *c3*:

$$c4: \boxed{[q(\lambda X_\iota.mX) = q(\lambda X_\iota.nX)]^F}$$

The decomposition rule *Dec* is applicable:

$$c5: \boxed{[(\lambda X_\iota.mX) = (\lambda X_\iota.nX)]^F}$$

With rule *Func* we can add a witness s_ι (skolem term) for the inequality of these two functions and by β-reduction we get:

$$c6: \boxed{[ms_\iota = ns_\iota]^F}$$

With rule *Leib* we can derive the clauses *c7* and *c8* (note the difference in the types of constant $p_{\iota\to o}$ and $q_{(\iota\to\iota)\to o}$ above):

$$c7: \boxed{[p_{\iota\to o}(ms)]^T} \qquad c8: \boxed{[p_{\iota\to o}(ns)]^F}$$

We made a detour to the pre-unification part of the calculus and modified the clauses *c2* and *c3* in an extensionally appropriate way and *c2* and *c3* have now their counterparts in *c7* and *c8*. But in contrast to *c2* and *c3* the new clauses can successfully be resolved against *c1*.
□ (LEO can find this proof within 2.5 sec. on a Sparc Workstation Ultra)

E5 $(\forall X_\iota.\forall P_{\iota\to o}.P(m_{\iota\to\iota}X) \Rightarrow P(n_{\iota\to\iota}X)) \Rightarrow (\forall Q_{(\iota\to\iota)\to o}.Qm \Rightarrow Qn)$

By clause normalization we get (q is a new Skolem term.):

$$c1: \boxed{[P(mX)]^F \vee [P(nX)]^T} \qquad c2: \boxed{[qm]^T} \qquad c3: \boxed{[qn]^F}$$

In analogy to example E4 and resolve *c2* against *c3*:

$$c4: \boxed{[qm = qn]^F}$$

Decomposition rule *Dec* is applicable:

$$c5: \boxed{[m = n]^F}$$

With rule *Func* we can add witnesses for the disequality of these two functions:

$$c6: \boxed{[ms = ns]^F}$$

The rest of the proof – as the proof so far – is similar to example E4.
□ (LEO can find this proof within 2.5 sec. on a Sparc Workstation Ultra)

Note the order in which the extensionality rule were applied in the above examples. For a practical implementation these examples suggest the following *extensionality treatment* of unification constraints: First decompose the unification constraint as much as possible. Then use rule *Func* to add as many arguments as possible to both hand sides of the resulting unification constraints. And last use rule *Leib* and/or *Equiv* to finish the extensionality treatment. In this sense the above rules can be combined to form only one rule *Ext-Treat*.

3.5 Soundness and Completeness

The pre-unification rules in figures 2 are discussed in [Koh94a, Koh94b][12].

The same holds for the soundness of the resolution rules in figure 3.

Concerning the soundness of the additional extensionality rules and concerning the completeness of the whole introduced calculus with respect to Henkin models we will state the following remarks.

It is very important for the soundness of the whole system to make sure that higher-order skolemization is sound. In contrast to [Koh94a, Koh94b] we use a skolemization technique which is similar to first-order skolemization but which introduces some additional restrictions as discussed in [Mil83]. This additional restrictions are necessary since otherwise some instances of the axiom of choice are provable [And73]. The soundness of this skolemization technique is discussed in [Mil83].

For the rules *Leib* and *Equiv* the soundness is obvious since if we assume Henkin semantics it is on the one hand allowed to replace each primitive equality symbol by its Leibniz definition and on the other hand to replace an equality on type o by an equivalence. For rule *Equiv* we have further to ensure the soundness of the clause normalization rules $\mathcal{CNF}$(see [Koh94a, Koh94b]). And for rule *Leib*

[12] There are slight differences since we do not deal with sorts here and do not introduce an extra-logical treatment of skolemization.

note that $p_{\alpha\to o}$ is a new Skolem term for the given clause. Rule *Func* looks very dangerous and if we are not careful enough and allow an arbitrary constant to be applied on both hand sides of the equations then we would immediately lose soundness and we could prove invalid formulas like $\lambda X_\alpha.X_\alpha = \lambda X_\alpha.Y_\alpha$. But by skolemization with respect to the free variables in the clause we can avoid this and soundness is guaranteed.

In contrast to soundness the completeness proof is a rather difficult task. But even if a detailed proof is still lacking the author has worked out a proof sketch that uses the idea of abstract consistency properties. This proof technique was introduced by Smullyan [Smu63] and adapted for general models by Andrews [And71]. Here we need a further extension for Henkin semantic as described in [Koh93, Koh94a]. With a slightly modification in the definition of extensional abstract consistency properties in [Koh94a, Koh94b] the proof seems to be straightforward. But first we have to ensure that the necessary modification in the definition of extensional abstract consistency properties is sound.

Further evidence for the Henkin completeness of the introduced calculus is given by the examples discussed so far.

4 The LEO System

The introduced calculus is implemented in the LEO theorem prover for classical higher-order logic. This implementation demonstrates that higher-order resolution can be an alternative to other approaches, e.g., the mating method [And76, And81]. And it demonstrates strongly that higher-order resolution might be an appropriate approach to embed full extensionality which is still a challenging problem for all automated higher-order theorem provers known to the author.

In this section we will suggest and discuss possible strategies for an extensional higher-order resolution theorem prover. Whereas the first implementation of LEO was strongly oriented on the standard first-order set of support strategy as used by OTTER [McC94] and did not deal with extensionality, the current architectures (Versions 07 and 08) take more and more higher-order specific demands into account and integrate the extensionality rules.

Before we discuss the basic datastructures of LEO and continue with the discussion of the strategies, we will focus on the two main problems of LEO's implementation: the handling and integration of the pre-unification and extensionality rules.

4.1 The embedding of pre-unification

When shall we apply pre-unification? Delaying it until we reach clauses consisting only of unification constraints would lead to an enormous search space

explosion and we could probably never solve any non trivial problem. Instead we are strongly interested in pre-unifying newly derived clauses as soon as possible in order to prevent this explosion.

Thus we combine all our pre-unification rules and the rule *Subst* to one new rule *Pre-Unify*. This new rule can be used in every loop of the refutation process in order to filter out all non-unifiable clauses and to instantiate the unifiable clauses with their pre-unifiers. But unfortunately higher-order unification (and higher-order pre-unification) is undecidable and we have to limit the search space for pre-unification since otherwise our general refutation procedure may run into an dead end. Hence when applying this rule to any given clause the search for further pre-unifiers stops as soon as the specified limit[13] is reached. The result of the application is a (possibly empty) set of pre-unified clauses with respect to the given search depth. Note that we have to take care of the *flex-flex* pairs generated within pre-unification. These *flex-flex* pairs have to be added as unification constraints to the particular result clauses.

Unfortunately by this search space restriction for pre-unification we lose completeness. A solution is to store information about each aborted pre-unification process in a special continuation which can be activated again in a later stage of the refutation process. The pre-unification algorithm provided by KEIM allows to create such continuations and to proceed with the interrupted search for pre-unifiers by activating this continuations again.

4.2 The embedding of extensionality

The idea of the extensionality rules is to modify literals by a detour to the unification part of the calculus. Unfortunately this is opposed to the newly introduced rule *Pre-Unify* since the intention of this rule is to filter out all non-unifiable clauses even if among them are some clauses which are interesting for an extensionality treatment[14]. A second less serious problem is that the extensionality treatment uses decomposition rule *Dec*, which is no longer separately available since all pre-unification rules are combined into rule *Pre-Unify*.

The latter problem can easily be solved and in analogy to the rule *Pre-Unify* we combine all rules necessary for extensionality treatment to one rule *Ext-Treat*. Note that we have already discussed the idea of an extensionality treatment at the end of subsection 3.4.

To tackle the first problem – the conflict between rule *Pre-Unify*, which tries to eliminate all non-unifiable clauses from search space and the rule *Ext-Treat* which wants to work on certain non-unifiable clauses – we have to decide very carefully about the order we want to apply both rules. In LEO this is solved as follows: before applying rule *Pre-Unify* rule on a clause, LEO verifies if this clause is suited for an extensionality treatment. If so, this clause is put into a

[13] This limit is specified by a flag of LEO.

[14] See for example clause *c4* in Example E4 (or E5) in subsection 3.4.

set of interesting extensionality objects EXT from which it can be chosen later in the search process to be extensionally processed.

Why does LEO not immediately apply rule *Ext-Treat* to an extensionality interesting clause? This is because we want to introduce a mechanism which allows us to delay and control the use of the extensionality rules in order to prevent the search space from a flooding of extensionally processed clauses. Additionally we want to have a mechanism to prevent redundancy. To be more concrete about that, suppose that the following clauses are given:

$$c1: \boxed{L^1 \vee L^2 \vee [q_{(\iota\to\iota)\to o} f_{\iota\to\iota} = q_{(\iota\to\iota)\to o} g_{\iota\to\iota}]^F}$$

$$c2: \boxed{L^2 \vee [r_{(\iota\to\iota)\to o} f_{\iota\to\iota} = r_{(\iota\to\iota)\to o} g_{\iota\to\iota}]^F}$$

By our extensionality treatment applied on *c1* we get:

$$c3: \boxed{L^1 \vee L^2 \vee [p_{\iota\to o}(fs_\iota)]^T} \qquad c4: \boxed{L^1 \vee L^2 \vee [p_{\iota\to o}(gs_\iota)]^F}$$

Similar we obtain for *c2*:

$$c5: \boxed{L^2 \vee [p'_{\iota\to o}(fs'_\iota)]^T} \qquad c6: \boxed{L^2 \vee [p'_{\iota\to o}(gs'_\iota)]^F}$$

Since p, p', s and s' are all new Skolem terms it is obvious that every proof which can be found with *c3* and *c4* – both are derived from *c1* – can also be found with clauses *c5* and *c6*, which are derived from *c2*. But none of these clauses subsumes any other and especially *c2* does not subsume *c1*.

What we need is a special filter for extensionality objects in the sense of a special 'extensionality subsumption' which works on the set EXT and filters out all redundant candidates. In the current implementation LEO uses a very weak filter which applies usual subsumption after decomposing the clauses as much as possible. Note that after decomposing *c1* and *c2*, *c2* indeed subsumes *c1*.

But which clauses are interesting for an extensionality treatment and should be put into EXT? It is obvious that not every non-unifiable clause is immediately a candidate for an extensionality treatment. For example the non-unifiable clause

$$[P_{\iota\to o} X_\iota]^T \vee [a_\iota = f_{\iota\to\iota} b_\iota]^F$$

is certainly not extensionally interesting. And on the other hand there are certain clauses among the unifiable or undecidable clauses which are well suited for an extensionality treatment, e.g., the clause

$$[P_{o\to o} a_o = \neg Q_{o\to o} a_o]^F \vee [Q_{o\to o} a_o = R_{o\to o} a_o]^F \vee$$
$$[R_{o\to o} a_o = \neg S_{o\to o} a_o]^F \vee [S_{o\to o} a_o = P_{o\to o} a_o]^F$$

This pre-unification problem can not be solved by pre-unification rules alone[15]. Instead pre-unification runs into a dead end and creates by using rule *Flex-Rigid* the same problem[16] again and again, but does not classify this clause as non-unifiable. Even so this clause is very interesting for an extensionality treatment.

Thus to decide which clauses should be put into EXT, LEO uses a predicate *extensionally interesting* which defines a clause to be extensionally interesting if one of its unification constraints (also called unification pairs) is. And an unification pair is extensionally interesting if its terms fulfill one of the following conditions [17]:

1. At least one of the two terms has a logical connective at head position.
2. At least one of the two terms is a λ-abstraction.
3. Both terms are unequal constants of functional type.
4. If the decomposition rule is applicable and the common head symbol p is not a Skolem constant introduced by rule *Leib*, then one of the decomposed unification constraints has to be extensionally interesting.

Summing up we can describe the embedding of the extensionality treatment as follows: On all new derived clauses in each loop of the refutation process we first apply our predicate *extensionally interesting* and decide which clauses should be put into the set EXT for later usage. The integration of the extensionally interesting clauses is done with respect to a special filter which is a generalization of usual subsumption. Then the rule *Pre-Unify* can be used to eliminate all non-unifiable clauses from the set of derived clauses before we integrate them into the set of support SOS.

4.3 The implementation platform and the basic datastructures

LEO is implemented in the object-oriented extension CLOS of COMMON LISP and is based on the platform KEIM [HKK+94a, Nes94] which provides most of the required datastructures to implement a higher-order theorem prover. Building upon this platform LEO introduces the following special datastructures:

[15] The pre-unification rules do not know about the fact that we are interested in Henkin models here, where the set of truth values consists of exactly two elements. This knowledge is brought into the calculus by the additional extensionality rules. One solution for this unification problem within Henkin models is: $P, S \leftarrow \lambda X_o.X_o$; $Q, R \leftarrow \lambda X_o.\neg X_o$; $a \leftarrow T$.

[16] Modulo variable renaming.

[17] This conditions might not be sufficient and it seems to be interesting to examine this question in detail.

- Literals[18]
  The datastructure for literals provide slots for the polarity, the atom, the weight and the clause-position information. The weight of a (usual) literal is given by the number of symbols occuring in its atom. By the clause-position information, which is a pair consisting of a position and a pointer to a certain clause, each literal is related to exactly those clause in which it occurs.

- Unification constraints
  Unification constraints are special literals. They are differed from usual literals to assign them a very low weight. This is necessary because otherwise, for example, a unit clause with many symbols in its unification constraints would get a very high weight and would be chosen very late as lightest clause from the set of support. Thus in the current implementation the weight of unification constraints is fixed at 0.

- Extensionality literals
  Like unification constraints extensionality literals are treated specially. During the search process such literals are generated within the rule *Leib*. Since it might be appropriate to prefer clauses obtained by a *Leib*-step in the refutation process or to put them at a disadvantage, LEO allows treating extensionality literals special and for example to assign them a certain weight. In the current implementation they get the constant weight 1.

- Clauses
  Clauses provide the following slots: positive literals, negative literals, unification constraints, weight, age, parent1, parent2, justification and some slots for storing additional information concerning the clauses status with respect to the rule *Prim-Subst* and the extensionality rules. The weight of a clause is computed by summing up the weights of its literals and the age is given by adding 1 to the maximum age of its parents. Input clauses get the age 0. Very important in connection with the use of an indexing-mechanism (see [Kle97] [Gra95]) is that the clause-position info in each of the literals has to be correct.

LEO is based upon four cornerstones: the set of support (SOS), the usable set (USABLE) – we assume that the standard set of support strategy is known to the reader – the set of extensionality objects (EXT) and a set of continuations for higher-order pre-unification (CONT).

- SOS
  The set of support stores all clauses which are waiting to be chosen as

[18] KEIM already provides datastructures for literals and clauses, but they are too weak for our purpose. Therefore we introduce special datastructures which are realized as CLOS subclasses of the KEIM-literals and -clauses.

lightest clause in one of the next loops. These clauses are either directly connected to the negated input theorem or they are derived clauses from a previous loop in the search process. The set of support is implemented as an ordinary list-structure. But especially in connection with subsumption it seems to be appropriate to implement this set similar to the usable set basing upon literal indices.

- USABLE
  The usable set contains all elements which either reflect the input axioms or which have been chosen as a lightest clause before. The clauses of the usable set are resolved in each loop against the lightest clause. Consequently the usable set should be kept as small as possible since its number of elements intensely influences the amount of derived clauses in each step. Thus LEO uses subsumption to filter out all redundant clauses. But subsumption[19] itself is a very time expensive algorithm and to increase its speed the usable set is implemented in LEO as a pair of two literal indices: one literal index for the positive literals and another one for the negative literals. This makes it possible for LEO to use indexing techniques as introduced in [Gra95] for first-order logic and adapted in [Kle97] for higher order logic to determine which literal in the index unifies or matches with which literal of a given clause, e.g., the lightest clause. By doing this for all literals of a given clause and with the help of the particular substitutions as well as the very important clause-position information in each literal it is possible to implement a faster even not satisfyingly fast subsumption algorithm. Additionally these literal indices can be used to determine possible resolution partners. But note that as soon as we are interested in extensionality we want to allow resolution-steps also on non-unifiable literals.

- EXT
  As already motivated LEO stores those clauses which are interesting for a extensionality treatment in the set of extensionally interesting objects EXT in order to prevent them from deletion by the pre-unification filter. The set EXT is implemented as an ordinary list. When a new clause is inserted into this list a special extensionality subsumption filter is used to remove redundant extensionality objects. Choosing objects from EXT can be handled analogous to choosing clauses from the set of support.

- CONT
  Every time pre-unification algorithm stops because of the given search depth limit a continuation is passed back to the main process. This con-

[19] The currently used higher-order subsumption algorithm is only a very unprecise filter and the author believes that the question of a fast and precise subsumption algorithm should be examined.

tinuation is put into the set of continuations CONT[20] from which it can be chosen later to activate the search process again with new search depth resources. The list of continuations can be implemented as a simple queue.

There are several additional clause storing objects: the set of resolved clauses (Resolved), the set of paramodulated clauses (Paramod), the set of factorized clauses (Factorized), the set of primitive substituted clauses (Prim-subst), the set of extensionally modified clauses (Ext-mod) , the set of continued pre-uni clauses (Uni-cont), the set of processed clauses (Processed) and finally the set of unified clauses (Unified). They are all used as clause buffers within each loop and in the current implementation they are all realized by simple list-structures.

4.4 LEO's strategies

LEO is still in an experimental stage where neither the principle architecture and strategy nor the adjustment of LEO's flags are optimized.

In this section we discuss two experimental strategies, LEO07 and LEO08, which differ mainly in the integration and use of pre-unification. Both strategies have different advantages and disadvantages and both are extensions of the standard first-order set of support strategy. Whereas the first strategy uses pre-unification and subsumption intensely to keep the amount of clauses in the search space as low as possible the second strategy tries to avoid the time-expensive pre-unification as much as possible.

The performance of each of these strategies is further adjustable by a set of flags which we will not discuss here in detail.

A graphical overview to the strategies LEO07 and LEO08 is given by the figures 5 and 6.

4.4.1 The refutation strategy LEO07

By reading an input problem the given formulas are normalized with the $\mathcal{CNF}$-algorithm and the resulting clauses are put either into the set of support or into the set of usable depending if they are connected to the theorem or to an axiom. Initially the set of extensionality objects EXT and the set of continuations CONT are empty.

In each loop of the refutation process LEO chooses a lightest clause from SOS. The criterion for this selection is either the weight of the clauses or their age. A flag defines in which loops LEO switches between these two selection criteria. After selection the lightest clause is put into USABLE and by flags one can decide if this should be done with respect to forward and/or backward subsumption. Next the lightest clause is resolved against all clauses in this set

[20] These ideas are not fully implemented yet. Nonetheless, the KEIM pre-unification package already provides continuations.

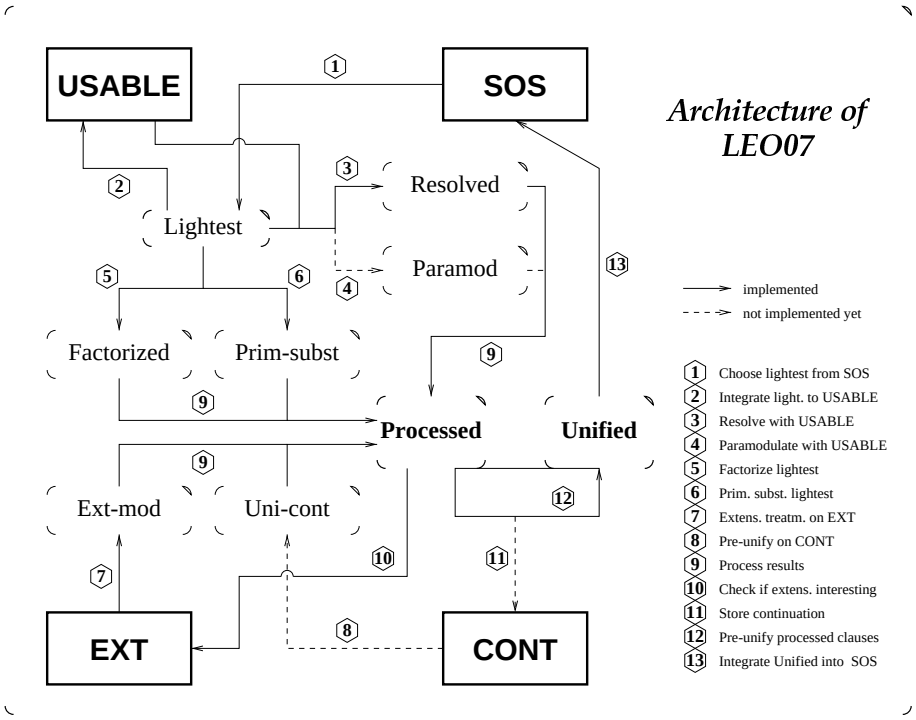

Figure 5: The refutation strategy LEO07

and the resolvents are collected in the set Resolved. Additionally paramodulation rule can be used to derive all paramodulants between the lightest clause and USABLE[21]. Now factorization and primitive substitution is applied on the lightest clause. The results are stored in the sets Factorized and Prim-subst. In the next step LEO operates on EXT and CONT. As previously mentioned it might not be appropriate to do this in every loop since this could increase the search space very fast. Instead LEO operates on extensionality clauses and continuations only in each n-th respectively m-th loop, whereas n and m are specified by flags. In the n-th loop LEO chooses the lightest extensionality object from EXT and applies the rule *Ext-Treat* to it. Analogously in the m-th loop one object is chosen from CONT and the interrupted pre-unification process is being continued until the specified search depth limit is reached again[22]. The resulting clauses of the extensionality treatment are put into the set Ext-mod

[21] Paramodulation is not fully implemented yet. The question of how to embed rules for a primitive treatment of equality will be examined in the next phase of the LEO project.

[22] The operation on the continuations is not fully implemented yet.

and those resulting from the continuation of pre-unification are inserted into the set Uni-cont. This finishes the deriving phase.

Now LEO is interested in eliminating as much redundant clauses as possible from the newly derived clauses. For this LEO can apply different filter, e.g., tautology[23], to the different sets of derived clauses before he integrates the remaining clauses in the set Processed. Next LEO decides which clauses in Processed are extensionally interesting and puts them into EXT for a later extensionality treatment.

The set of processed clauses then becomes pre-unified and thereby all non-unifiable clauses are eliminated. The result of the pre-unification step is a set of instances (Unified) for the clauses in Processed. Note that the clauses in Unified may still contain unification constraints – namely flex-flex pairs. Every time the unification process is stopped because of the search depth limit the resulting continuation is integrated in the set of continuations[24].

The last but very time consuming step if subsumption is activated is the integration of the set of unified clauses into the set of support.

4.4.2 The refutation strategy LEO08

The main difference between the strategy of LEO07 and LEO08 (see figure 6) concerns the integration and usage of pre-unification. The aim of LEO08 is to avoid the very expensive pre-unification algorithm as much as possible. Whereas in LEO07 all clauses in Processed become pre-unified before they are integrated into the set of support, LEO08 does not filter out the non-unifiable clauses from Processed. Instead pre-unification is applied on the lightest clause at the beginning of each loop. If the chosen lightest clause is pre-unifiable all the resulting clauses are stored in Unified. If instead the lightest clause is not pre-unifiable LEO08 throws this clause away and chooses the next clause from SOS. LEO proceeds with the clauses in Unified as the clauses of interest in this loop of the refutation process.

4.5 Experiences with LEO07 and LEO08

Since the aim of strategy LEO08 is to avoid as much pre-unification steps as possible, subsumption is used very rarely, e.g., only to keep USABLE as small as possible.

The examples discussed in this paper are solved faster by LEO08 but note that we are still dealing with very simple problems. The set of support in LEO08 increases very fast and therefore LEO08 should not be able to find very deep and complicated proofs. Additionally, the success of this strategy strongly depends on the criteria used to choose the lightest clause from SOS. Only if

[23] In the current implementation LEO uses only a very weak tautology filter.
[24] Not fully implemented yet.

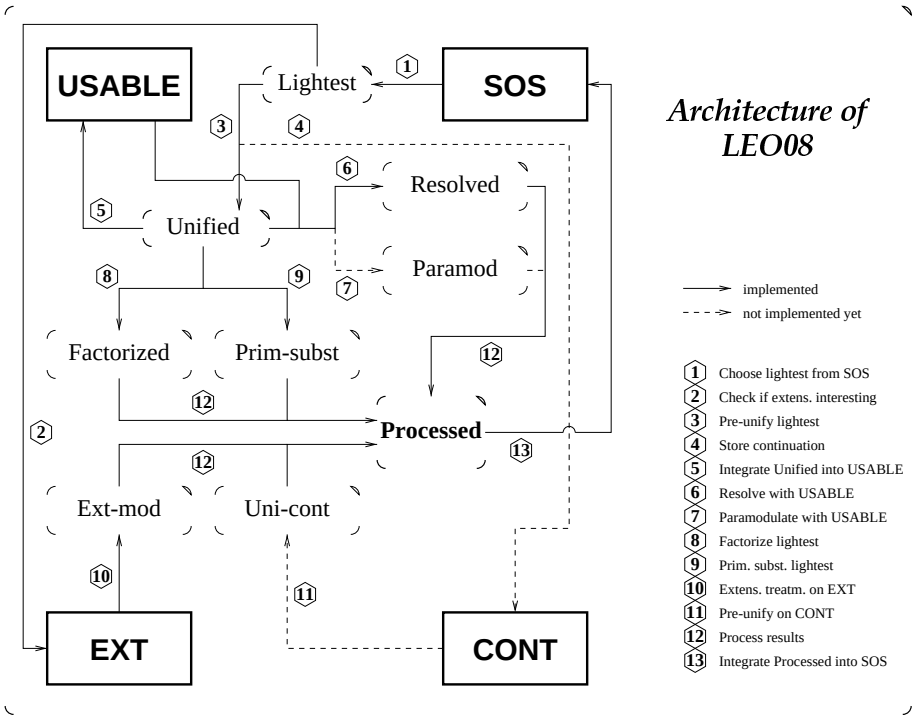

Figure 6: The refutation strategy LEO08

the interesting clauses can be found very early among all clauses in SOS this strategy has some advantages.

LEO07 is obviously the right approach for proving more complicated theorems. But when running more complicated examples with fully enabled subsumption LEO spends most of the time for pre-unification and subsumption.

Therefore the efficiency of the pre-unification and subsumption algorithms highly influences the efficiency of LEO and consequently any improvement in these algorithms will be very important for the LEO project.

4.6 The importance of rule *Leib*

Every refutation which uses the results of the rule *Leib* can possibly be done without this rule by resolving against the extensional modified unification constraint instead[25]. For example the application of rule *Leib* in the proof of exam-

[25]This idea is due to Frank Pfenning (Dep. of Computer Science, Carnegie Mellon University, Pittsburgh, USA). He suggested to introduce 'primitive equality' and to replace every

ple E4 can be replaced by an immediate resolution step between clause *c1* and *c6*:

$$c7: \boxed{[P(mX)]^F \vee [P(nX) = (ms = ns)]^F}$$

And by pre-unification ($P \leftarrow \lambda Y_\iota.(ms = Y)$ and $X \leftarrow s$) we immediately get the empty clause.

However, there are two reasons why rule *Leib* seems to be very appropriate. First the completeness proof with respect to Henkin models seems to be more complicated without rule *Leib*. The second reason is a practical one and very important. If we avoid rule *Leib*, we will get an additional control problem since we would have to allow clauses with non-unifiable unification constraints in our search space. An example is clause *c6* above. Note that in our current implementation either *c6* can never become an element of the set of support (in LEO07) because of the pre-unification filter or *c6* can not be pre-unified after being chosen as lightest clause (in LEO08).

Thus by avoiding rule *Leib* we can not use pre-unification in this simple way to prevent any non-unifiable clause from being put into SOS (in LEO07) or from being worked upon (in LEO08). If we use rule *Leib* instead then any extensionally modified unification constraint gets transformed back into non-unification literals and there is no need to treat the resulting clauses in a special way.

5 Conclusion and further work

5.1 Conclusion

In the first part of this paper we have discussed an extension of Huet's higher-order resolution approach and motivated the completeness of the extended calculus with respect to Henkin models. Aside from [Koh95][26] this is the first approach known to the author which tries to embed full extensionality in a higher-order refutation calculus without using axioms. The idea of the novel extensionality rules is to combine the power of classical higher-order logic in an appropriate way with the pre-unification rules in order to obtain full extensionality and Henkin completeness. On different and for current higher-order theorem prover still challenging examples we have demonstrated the practical

unification constraint between λ-terms of functional type by an equation of primitive type. Note that this is also the intention of rule *Func* within the extensionality treatment discussed in section 3.4.

[26] [Koh95] extends a higher-order tableaux calculus in order to obtain extensionality and therefore a tableaux rule which is analogous to rule *Equiv* is introduced to be used in connection with the other pre-unification rules. Unfortunately the pre-unification rules η and α introduce a special variable Z^0 (instead of a skolem term s like in our calculus) which can not be substituted for any other variable. Hence the resulting calculus is to weak to establish full extensionality (e.g., examples E4 and E5 can not be proven in this calculus).

fitness of the extensional resolution calculus and shown that they can be solved very easily within this approach. In some remarks we have pointed out that the calculus might still contain theoretical redundancies and that there exist different ideas for a further improvement.

In the second part we have introduced the LEO system which implements the introduced extensional higher-order resolution calculus. We have discussed the main ideas concerning the undecidability problem of higher-order pre-unification and the integration of the extensionality rules. We have further sketched the basic datastructures and suggested two slightly different refutation strategies (LEO07 and LEO08) which are both extensions of the standard first-order set of support strategy.

The general aim of this paper was to demonstrate that higher-order resolution can be a reasonable approach and that it provides a suited basis for embedding full extensionality.

5.2 Further work

Since the LEO project is a very young project, there are many open tasks.

Among these the elaboration of the formal proofs has a very high priority and in connection with this proofs the theoretical necessity of rule *Leib* should be examined in detail.

Another important problem is the theoretical and practical examination of higher order subsumption. As subsumption is strongly connected to unification/matching it might be worth to examine how much profit a change of the datastructures (e.g., usage of explicit substitutions) can bring. A more technical problem is the embedding of the higher-order indexing techniques [Kle97] as their integration in the current implementation is rather poor yet.

A very interesting question concerns the the connection between the infinitely branching rule *Prim-subst* and the new extensionality rules. By generalizing the resolution and factorization rules in an appropriate way it seems to be possible to avoid the primitive substitution rule in many examples which usually presuppose the usage of this rule. Therefore the interesting connection between the extensionality rules and the primitive substitution rule should be examined in theory.

Very important for the practical progress of LEO is the refinement of the discussed architecture and strategies or the development of new and possibly much better strategies. Additionally LEO should be applied on larger and more complicated examples.

LEO is intended to become a powerful subsystem of OMEGA. Therefore many questions will arise concerning the integration of LEO into an interactive proof development system like OMEGA. E.g., similar to the translation between the mating calculus and the ND-calculus from TPS to ETPS [Pfe87], it will be necessary to translate the resolution proofs (with extensionality) back into the ND-calculus used by the OMEGA-system.

5.3 Acknowledgements

I am deeply indebted to Peter Andrews, Frank Pfenning and the researchers and students around the TPS project at Carnegie Mellon University for the new insights into higher-order logic they brought to me and the time they spent in discussing the ideas of this paper with me. My special thanks to Peter Andrews, Michael Kohlhase and Jörg Siekmann who made my stay in this exceptional research environment possible. Lastly I want to thank the 'Studienstiftung des Deutschen Volkes' for providing the necessary financial support without insisting on many paperwork.

References

[ABI+96] Peter B. Andrews, Matthew Bishop, Sunil Issar, Dan Nesmith, Frank Pfenning, and Hongwei Xi. TPS: A theorem-proving system for classical type theory. *Journal of Automated Reasoning*, 16:321–353, 1996.

[AINP90] Peter B. Andrews, Sunil Issar, Dan Nesmith, and Frank Pfenning. The TPS theorem proving system. In Mark Stickel, editor, *Proceedings of the 10th Conference on Automated Deduction*, number 449 in LNCS, Kaiserslautern, Germany, 1990.

[And71] Peter B. Andrews. Resolution in type theory. *Journal of Symbolic Logic*, 36(3):414–432, 1971.

[And72] Peter B. Andrews. General models and extensionality. *Journal of Symbolic Logic*, 37(2):395–397, 1972.

[And73] Peter B. Andrews, 1973. letter to Roger Hindley dated January 22, 1973.

[And76] Peter B. Andrews. Refutations by matings. *IEEE Trans. Comp.*, C-25(8):801–807, 1976.

[And81] Peter B. Andrews. Theorem proving via general matings. *Journal of the Association for Computing Machinery*, 28(2):193–214, April 1981.

[Bar84] H. P. Barendregt. *The Lambda Calculus*. North Holland, 1984.

[BCF+97] C. Benzmüller, L. Cheikhrouhou, D. Fehrer, A. Fiedler, X. Huang, M. Kerber, M. Kohlhase, K. Konrad, E. Melis, A. Meier, W. Schaarschmidt, J. Siekmann, and V. Sorge. ΩMEGA: Towards a mathematical assistant. In William McCune, editor, *Proceedings of the 14th Conference on Automated Deduction*, LNAI, Townsville, Australia, 1997. Springer Verlag.

[BKM] R. S. Boyer, M. Kaufmann, and J. S. Moore. The boyer-moore theorem prover and its interactive enhancement. *Computers and Mathematics with Applications*.

[Bun94] Alan Bundy, editor. *Proceedings of the 12th Conference on Automated Deduction*, number 814 in LNAI, Nancy, France, 1994. Springer Verlag.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[GM93] M. J. C. Gordon and T. F. Melham. *Introduction to HOL – A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Gra95] Peter Graf. *Term Indexing*. PhD thesis, Universitaet des Saarlandes, Saarbruecken, Germany, July 1995.

[GS89] Jean H. Gallier and Wayne Snyder. Complete sets of transformations for general E-unification. *Theoretical Computer Science*, (67):203–260, 1989.

[Hen50] Leon Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15(2):81–91, 1950.

[HKK+94a] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Dan Nesmith, Jörn Richts, and Jörg Siekmann. Keim: A toolkit for automated deduction. In Bundy [Bun94], pages 807–810.

[HKK+94b] Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Erica Melis, Daniel Nesmith, Jörn Richts, and Jörg Siekmann. Ω-MKRP a proof development environment. In Bundy [Bun94], pages 788–792.

[Hue72] Gérard P. Huet. *Constrained Resolution: A Complete Method for Higher Order Logic*. PhD thesis, Case Western Reserve University, 1972.

[Hue73] Gérard P. Huet. A mechanization of type theory. In *Proceedings of the Third International Joint Conference on Artificial Intelligence*, pages 139–146, 1973.

[Hue75] Gérard P. Huet. An unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Kle97] Lars Klein. Indexing für Terme höherer Stufe. Master's thesis, FB 14 Informatik, Universität des Saarlandes, Saarbrücken, Germany, 1997.

[Koh93] Michael Kohlhase. A unifying principle for extensional higher-order logic. Technical Report 93–153, Dept. of Mathematics, Carnegie Mellon University, 1993.

[Koh94a] Michael Kohlhase. Higher-order order-sorted resolution. Seki Report SR-94-1, Fachbereich Informatik, Universität des Saarlandes, 1994.

[Koh94b] Michael Kohlhase. *A Mechanization of Sorted Higher-Order Logic Based on the Resolution Principle*. PhD thesis, Universität des Saarlandes, 1994.

[Koh95] Michael Kohlhase. Higher-order tableaux. In R. Hähnle P. Baumgartner and J. Posegga, editors, *Theorem Proving with Analytic Tableaux and Related Methods*, volume 918 of *Lecture Notes in Artificial Intelligence*, pages 294–309, 1995.

[McC94] W. W. McCune. Otter 3.0 reference manual and guide. Technical Report ANL-94-6, Argonne National Laboratory, Argonne, Illinois 60439, USA, 1994.

[Mil83] Dale Miller. *Proofs in Higher-Order Logic*. PhD thesis, Carnegie-Mellon University, Pittsburgh Pa.,USA, 1983.

[Nes94] Dan Nesmith, editor. KEIM-Manual. Version 1.2, 1994. Universität des Saarlandes, Germany.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: a prototype verification system. In D. Kapur, editor, *Proceedings of the 11th Conference on Automated Deduction*, volume 607 of *LNCS*, pages 748–752, Saratoga Spings, NY, USA, 1992. Springer Verlag.

[Pfe87] F. Pfenning. *Proof Transformations in Higher-Order Logic*. PhD thesis, Carnegie-Mellon University, Pittsburgh Pa.,USA, 1987.

[Smu63] Raymond M. Smullyan. A unifying principle for quantification theory. *Proc. Nat. Acad Sciences*, 49:828–832, 1963.

[SS89] Manfred Schmidt-Schauß. *Computational Aspects of an Order-Sorted Logic with Term Declarations*, volume 395 of *LNAI*. Springer Verlag, 1989.

[WGR96] Christoph Weidenbach, Bernd Gaede, and Georg Rock. Spass & flotter, version 0.42. In M.A. McRobbie and J.K Slaney, editors, *Proceedings of the 13th Conference on Automated Deduction*, number 1104 in LNAI, New Brunswick, NJ, USA, 1996. Springer Verlag.