System Description: LEO – A Resolution based Higher-Order Theorem Prover

Christoph Benzmüller

Fachbereich Informatik, Universität des Saarlandes
66041 Saarbrücken, Germany (www.ags.uni-sb.de/~chris)

Abstract. We present LEO, a resolution based theorem prover for classical higher-order logic. It can be employed as both an fully automated theorem prover and an interactive theorem prover. LEO has been implemented as part of the ΩMEGA environment [23] and has been integrated with the ΩMEGA proof assistant. Higher-order resolution proofs developed with LEO can be displayed and communicated to the user via ΩMEGA's graphical user interface LOUI. The LEO system has recently been successfully coupled with a first-order resolution theorem prover (Bliksem).

1 Introduction

Many of today's proof assistants such as Isabelle [22, 20], PVS [21], HOL [12], HOL-Light [13], and TPS [2, 3] employ classical higher-order logic (also known as Church's simple type theory) as representation and reasoning framework. One important motivation for the development of automated higher-order proof tools thus is to relieve the user of tedious interactions within these proof assistants by automating less ambitious (sub)problems.

In this paper we present LEO, an automated resolution based theorem prover for classical higher-order logic. LEO is based on extensional higher-order resolution which, extending Huet's constrained resolution [14, 15], proposes a goal directed, rule based solution for extensionality reasoning [6, 4, 5]. The main motivation for LEO is to serve as an automated subsystem in the mathematics assistance system ΩMEGA [23]. Additionally, LEO was intended to serve as a standalone automated higher-order resolution prover and to support the illustration and tutoring of extensional higher-order resolution. A previous system description of LEO has been published in [7]. Novel in this system description is the section on LEO's interaction facilities and its graphical user interface. We also provide more details on LEO's automation and point to some recent extensions.

This system description is structured as follows: In Sections 2 and 3 we briefly address LEO's connection with ΩMEGA and LEO's calculus. Section 4 presents the interactive theorem prover LEO and Section 5 the automated theorem prover LEO. In Section 6 we illustrate how LEO's resolution proofs can be inspected with ΩMEGA's graphical user interface LOUI. Some experiments with LEO are mentioned in Section 7 and Section 8 concludes the paper.

2 LEO is a Subsystem of ΩMEGA

LEO has been realized as a part of the ΩMEGA framework. This framework (and thus also LEO) is implemented in CLOS [25], an object-oriented extension of LISP. LEO is mainly dependent on ΩMEGA's term datastructure package KEIM. The KEIM package provides many useful data structures (e.g., higher-order terms, literals, clauses, and substitutions) and basic algorithms (e.g., application of substitution, subterm replacement, copying, and renaming). Thus, the usage of KEIM allows for a rather quick implementation of new higher or first-order theorem proving systems. In addition to the code provided by the KEIM-package, LEO consists of approximately 7000 lines of LISP code.

Amongst the benefits of the realization of LEO within the ΩMEGA framework are:

- Employing KEIM supported a quick implementation of LEO. Important infrastructure could be directly reused or had to be only slightly adapted or extended.
- An integration of LEO with the proof assistant and proof planner ΩMEGA was easily possible.
- LEO can easily be combined with other external systems already integrated with ΩMEGA. Combinations of reasoning systems are particularly well supported in ΩMEGA with the help of the agent based reasoning framework OANTS [8].
- LEO may retrieve and store theorems and definitions via ΩMEGA from MBASE, which is a structured repository of mathematical knowledge.
- LEO employs ΩMEGA's input language POST.

There are also drawbacks of LEO's realization as a part of ΩMEGA:

- LEO's latest version is only available in combination with the ΩMEGA package. Installation of ΩMEGA, however, is very complicated. Consequently, there is a conflict with the objective of providing a lean standalone theorem prover.
- The KEIM datastructures are neither very efficient nor are they optimized or easily optimizable with respect to LEO.

ΩMEGA and with it LEO can be download from http://www.ags.uni-sb.de/~omega.

3 LEO Implements Extensional Higher-Order Resolution

LEO is based on a calculus for extensional higher-order resolution. This calculus is described in [6, 4] and more recently in [5].

Extensionality treatment in this calculus is based on goal directed extensionality rules which closely connect the overall refutation search with unification by allowing for mutually recursive calls. This suitably extends the higher-order E-unification and E-resolution idea, as it turns the unification mechanism into a most general, dynamic theory unification mechanism. Unification may now

itself employ a Henkin complete higher-order theorem prover as a subordinated reasoning system and the theory under consideration (which is defined by the sum of all clauses in the actual search space) dynamically changes.

In order to illustrate LEO's extensional higher-order resolution approach we discuss the TPTP (v3.0.1 as of 20 January 2005, see http://www.tptp.org) example SET171+3. This problem addresses distributivity of union over intersection[1]

$$\forall A_{o\alpha}, B_{o\alpha}, C_{o\alpha}.A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

In a higher-order context we can define the relevant set operations as follows

$$\cup := \lambda Y_{o\alpha}, Z_{o\alpha}.(\lambda x_{\alpha}.x \in Y \vee x \in Z)$$

$$\cap := \lambda Y_{o\alpha}, Z_{o\alpha}.(\lambda x_{\alpha}.x \in Y \wedge x \in Z)$$

$$\in := \lambda Z_{\alpha}, X_{o\alpha}.(X\ Z)$$

After recursively expanding the definitions in the input problem, i.e., completely reducing it to first principles, LEO turns it into a negated unit clause. Unlike in standard first-order resolution, clause normalization is not a pre-process in LEO but part of the calculus. Internalized clause normalization is an important aspect of extensional higher-order resolution in order to support the (recursive) calls from higher-order unification to the higher-order reasoning process. Thus, LEO internally provides rules to deal with non-normal form clauses and this is why it is sufficient to first turn the input problem into a single, usually non-normal, negated unit clause. Then LEO's internalized clause normalization process can take care of subsequent normalization.

In our concrete case, this normalization process leads to the following unit clause consisting of a (syntactically not solvable) unification constraint (here $B_{o\alpha}, C_{o\alpha}, D_{o\alpha}$ are Skolem constants and Bx is obtained from expansion of $x \in B$):

$$[(\lambda x_{\alpha}.Bx \vee (Cx \wedge Dx)) =^{?} (\lambda x_{\alpha}.(Bx \vee Cx) \wedge (Bx \vee Dx))]$$

Note that negated primitive equations are generally automatically converted by LEO into unification constraints. This is why $[(\lambda x_{\alpha}.Bx \vee (Cx \wedge Dx)) =^{?} (\lambda x_{\alpha}.(Bx \vee Cx) \wedge (Bx \vee Dx))]$ is generated, and not $[(\lambda x_{\alpha}.Bx \vee (Cx \wedge Dx)) = (\lambda x_{\alpha}.(Bx \vee Cx) \wedge (Bx \vee Dx))]^{F}$. Observe that we write $[.]^{T}$ and $[.]^{F}$ for positive and negative literals, respectively. This unification constraint has no 'syntactic' pre-unifier. It is solvable 'semantically' though with the help of the extensionality

[1] We use Church's notation $o\iota$, which stands for the functional type $\iota \to o$. The reader is referred to [1] for a more detailed introduction. In the remainder, o will denote the type of truth values and ι defines the type of individuals. Other small Greek letters will denote arbitrary types. Thus, $X_{o\alpha}$ (resp. its η-long form $\lambda y_{\alpha}.Xy$) is actually a characteristic function denoting the set of elements of type α for which the predicate associated with X holds. We use the square dot '.' as an abbreviation for a pair of brackets, where '.' stands for the left one with its partner as far to the right as is consistent with the bracketing already present in the formula.

principles. Thus, LEO applies its goal directed functional and Boolean extensionality rules which replace this unification constraint (in an intermediate step) by the negative literal (where x is a Skolem constant):

$$[(Bx \vee (Cx \wedge Dx)) \Leftrightarrow ((Bx \vee Cx) \wedge (Bx \vee Dx))]^F$$

This intermediate unit clause is not in normal form and subsequent normalization generates 12 clauses including the following four:

$$[Bx]^F \qquad [Bx]^T \vee [Cx]^T \qquad [Bx]^T \vee [Dx]^T \qquad [Cx]^F \vee [Dx]^F$$

This set is essentially of propositional logic character and trivially refutable by LEO. For the complete proof of the problem LEO needs less than one second (on a notebook with an Intel Pentium M processor 1.60GHz and 2 MB cache) and a total of 36 clauses is generated.

4 LEO is an Interactive Theorem Prover

LEO is an interactive theorem prover based on extensional higher-order resolution. The motivation for this is twofold. Firstly, the provided interaction features support interaction with the automation of this calculus. For this an automated proof attempt may be interrupted at any time and the system developer can then employ the interaction facilities to investigate the proof state and to perform some steps by hand. He may then again proceed with the automated proof search. Secondly, the interaction facilities can be employed for tutoring higher-order resolution in the classroom and they have in fact been employed for this purpose.

We illustrate below an interactive session with LEO within the ΩMEGA system. For this we assume that the ΩMEGA system has already been started. Interaction within ΩMEGA is supported in two ways. We may use the graphical user interface LOUI, or ΩMEGA's older Emacs based command line interface. Both interfaces can be started and used simultaneously. In this section we describe interaction only within the Emacs based interface. All commands could alternatively also be invoked via LOUI.

After starting ΩMEGA we are offered the following command line prompt in the Emacs interface:

```
OMEGA:
```

We assume that we have added the following problem, formalized in POST, to ΩMEGA's structured knowledge bass.

$$\exists Q_{o(o\iota)(o\iota)} . p_{oo}((a_{o\iota} m_\iota) \wedge ((b_{o\iota} m_\iota) \vee (c_{o\iota} m_\iota))) \Rightarrow p_{oo}((a_{o\iota} m_\iota) \wedge (Q_{o(o\iota)(o\iota)} c_{o\iota} b_{o\iota}))$$

The POST representation of this problem is

```
(th~defproblem little
         (in base)
         (constants (p (o o)) (a (o i)) (b (o i)) (c (o i)) (m i))
         (conclusion
          (EXISTS (lam (Q (o (o i) (o i)))
                     (IMPLIES (p (AND (a m) (OR (b m) (c m))))
                              (p (AND (a m) (Q c b))))))))
```

All problems have names and the name chosen for our problem is `little`. The `in`-slot in this problem definition specifies MBASE theories from which further knowledge, e.g., definitions and lemmata, is inherited. Examples of some MBASE theories are `typed-set, relation, function,` and `group`. Our very simple example problem is defined in the knowledge repository for theory `base`, and no further information is included. The assertion we want to prove is specified in the `conclusion`-slot and the typed constant symbols which occur in the assertion are declared in the `constants`-slot.

We now load all problems defined in theory `base` and then call the ΩMEGA command `show-problems` to display the names of all problems. '[...]' indicates that some less interesting output information from LEO has been deleted here for presentation purposes.

```
OMEGA: load-theory-problems base
[...]

OMEGA: show-problems
[...]
EMBEDDED
little
less-little
TEST
[...]
```

Next we initialize the ΩMEGA proof assistant with the proof problem `little` and display ΩMEGA's central proof object after initialization.

```
OMEGA: prove little
Changing to proof plan LITTLE-1

OMEGA: show-pds
              ...
LITTLE ()         ! (EXISTS [Q:(O (O I) (O I))]                  OPEN
                            (IMPLIES
                             (P (AND (A M) (OR (B M) (C M))))
                             (P (AND (A M) (Q C B)))))

OMEGA:
```

Then we initialize LEO with this problem. Here we choose the default settings (suggested in the '`[]`'-brackets) as input parameters for LEO. Each tactic (here `EXT-INPUT-RECURSIVE`) determines a specific flag setting of LEO. This flag setting is displayed after initialization.

```
OMEGA: leo-initialize
NODE (NDLINE) Node to prove with LEO: [LITTLE]
TACTIC (STRING) The tactic to be used by LEO: [EXT-INPUT-RECURSIVE]
THEORY-LIST (SYMBOL-LIST) Theories whose definitions will be expanded: [()]
Expanding the Definitions....
Initializing LEO....
```

```
Applying Clause Normalization....
============== variable settings ================
  Value(LEO*F-FO-ATP-RESOURCE) = 0
  Value(LEO*F-COOPERATE-WITH-FO-ATP) = NIL
  Value(LEO*F-TACTIC) = EXT-INPUT-RECURSIVE
  Value(LEO*F-VERBOSE-HALF) = NIL
  Value(LEO*F-VERBOSE) = NIL
  Value(LEO*F-WEIGHT-AGE-INT) = 4
  Value(LEO*F-SOS-TYPE) = TOSET
  Value(LEO*F-USABLE-TYPE) = INDEX
  Value(LEO*F-CLAUSE-LENGTH-RESTRICTION) = NIL
  Value(LEO*F-SAVE-FO-CLAUSES) = T
  Value(LEO*F-SUBSUM-MATCH-RESSOURCE) = NIL
  Value(LEO*F-SOS-SUBSUMTION) = NIL
  Value(LEO*F-BACKWARD-SUBSUMTION) = T
  Value(LEO*F-FORWARD-SUBSUMTION) = T
  Value(LEO*F-PARAMODULATION) = NIL
  Value(LEO*F-REMOVE=EQUIV-NEG) = NIL
  Value(LEO*F-REMOVE=EQUIV-POS) = NIL
  Value(LEO*F-REMOVE=LEIBNIZ-NEG) = NIL
  Value(LEO*F-REMOVE=LEIBNIZ-POS) = NIL
  Value(LEO*F-EXT-DECOMPOSE-ONLY) = T
  Value(LEO*F-EXT-UNICONSTRCLS-ONLY) = NIL
  Value(LEO*F-EXTENSIONALITY-NUM) = 6
  Value(LEO*F-EXT-INPUT-TREATMENT-RECURSIVE) = T
  Value(LEO*F-EXT-INPUT-TREATMENT) = NIL
  Value(LEO*F-EXTENSIONALITY) = T
  Value(LEO*F-NO-FLEX-UNI) = NIL
  Value(LEO*F-UNI-RESSOURCE) = 5
  Value(LEO*F-PRIM-SUBST-TYPES) = NIL
  Value(LEO*F-PRIMITIVE-SUBSTITUTION) = T
  Value(LEO*F-FACTORIZE) = T
  Value(LEO*F-AUTO-PROOF) = NIL
  Value(LEO*F-MAIN-COUNTER) = 0
 =========== end variable settings =============

OMEGA:
```

During initialization LEO first recursively expands defined symbols occurring in the assumptions or the assertion with respect to the specified theories (here we have none). Then it negates the assertion, turns it into a negated unit input clause and subsequently normalizes it with its internalized normalization rules. The resulting clauses are put either into LEO's set of support (if they stem from the assertion) or the usable set (if they stem from assumptions; here we have none). LEO clauses have unique names starting with `cl` and followed by an automatically created number. In '`( )`'-brackets further clause specific information follows and the '`{ }`'-brackets contain the clause literals. Negative literals have a leading `-` and positive literals a leading `+`. Type information is usually not displayed. All symbols starting with `dc` are free variables.

```
OMEGA: show-clauses
     ================= BEGIN ================================
The set of support
     cl3(1.5|1):{(-(p (and (a m) (dc2 c b))))}
     cl4(1.5|1):{(+(p (and (a m) (or (b m) (c m)))))}
The set of usable clauses
     ================= END ==================================
```

LEO offers a list of commands, for instance, to apply calculus rules, to manipulate and maintain the proof state, or to display information.

```
OMEGA: show-commands leo-interactive
[...]
DECOMPOSE:              Applies decomposition on a clause.
DELETE-CLAUSE:          Deletes a clause from the current environment.
END-REPORT:             Closes the report stream.
EXECUTE-LOG:            Reads a log file stepwise and eventuelly stores
                        some of its commands in a new log file.
EXIT:                   Leave the current command interpreter top level.
EXT:                    Applies extensionality rule on a clause.
[...]
FACTORIZE:              Applies factorization rule on a clause.
[...]
GUI-PROOF:              Displays the LEO proof of node in the GUI.
[...]
LEO-PROVE:              Prove with default parameter-settings.
NEW-LOG:                Sets the log mode and the log file name to the
                        given path name.
PARA:                   Applies paramodulation rule on two clauses.
PRE-UNIFIERS:           Computes the pre-unifiers of a clause.
PRE-UNIFY:              Applies pre-unification on a clause.
PRIM-SUBST:             Applies primitive substitution rule on a clause.
[...]
READ-LEO-PROBLEM:       Read a file, which contains a POST
                        representation of a problem, and transforms this
                        problem into clause normal form.
[...]
RESOLVE:                Applies resolution rule on two clauses.
SAVE-CLAUSE:            Save a clause for use after termination of LEO
                        under its clausename.
[...]
SET-FLAG:               Sets a global flag
SET-TACTIC:             Sets the tactic.
SHOW-CLAUSE:            Displays a clause, determined by name.
SHOW-CLAUSES:           Displays the two clause sets: the set of support
                        (LEO*G-SOS) and the set of usable clauses
                        (LEO*G-USABLE).
[...]
PROJECT:                Applies projection rule on a clause.
SHOW-DERIVATION:        Displays a linearized derivation of a the clause.
SHOW-FLAGS:             Shows the global flags
SHOW-LEO-PROBLEM:       Displays the given problem in POST.
SHOW-LEO-PROOF:         Displays the linearized LEO proof
SHOW-TACTICS:           Shows the tactics.
SHOW-VARS:              Shows the global vars.
START-REPORT:           Opens the tex and html report streams.
STEP-LOG:               Reads a log file stepwise and eventuelly stores
                        some of its commands in a new log file.
SUBSUMES:               Determines whether a clause subsumes another clause.
[...]
WRITE-DERIVATION:       Writes the derivation of a clause in a file.
WRITE-LOUIDERIVATION:   Writes the derivation of a clause in LOUI format
                        in a file.
WRITE-LOUIPROOF:        Writes the proof in LOUI format in a file.
WRITE-PROOF:            Writes the proof in a file.

OMEGA:
```

We apply the resolution rule to the clause `cl3` and `cl4` on literal positions 1 and 1 respectively. This results in the clause `cl5` which consists only of a unification constraint. In this display unification constraints are presented as negated equations (on the datastructure level they are distinguished from them as already mentioned in Section 3). Provided that we can solve the unification constraint `cl5`, we have found an empty clause and we are done.

```
OMEGA: resolve
```

```
NAME1 (STRING) of a clause: cl3
NAME2 (STRING) of a clause: cl4
POSITION1 (INTEGER) in clause 1: 1
POSITION2 (INTEGER) in clause 2: 1
 Clause 1: cl3(1.5|1):{(-(p (and (a m) (dc2 c b))))}.
 Clause 2: cl4(1.5|1):{(+(p (and (a m) (or (b m) (c m)))))}.
 Res(CL3[1],CL4[1]):
    (cl5(1|2):{(-(= (p (and (a m) (dc37 c b)))
                    (p (and (a m) (or (b m) (c m))))))}).

OMEGA:
```

We ask LEO to compute the pre-unifiers for this unification constraint.

```
OMEGA: pre-unifiers
NAME (STRING) of a clause: cl5
 The clause: cl5(1|2):{(-(= (p (and (a m) (dc37 c b)))
                            (p (and (a m) (or (b m) (c m))))))}.
 pre-unifiers: ({(dc37 --> [lam ?h45 ?h46.(or (?h46 m) (?h45 m))])}
                {(dc37 --> [lam ?h45 ?h46.(or (?h46 m) (c m))])}
                {(dc37 --> [lam ?h45 ?h46.(or (b m) (?h45 m))])}
                {(dc37 --> [lam ?h45 ?h46.(or (b m) (c m))])}).

OMEGA:
```

Pre-unification of clause `cl5` first generates these four pre-unifiers and then subsequently applies them to `cl5`. Instantiation of different pre-unifiers usually leads to different result clauses. In our simple case here, however, we obtain four copies of the empty clause.

```
OMEGA: pre-unify cl5
 The clause: cl5(1|2):{(-(= (p (and (a m) (dc37 c b)))
                            (p (and (a m) (or (b m) (c m))))))}.
 Result: (cl14(0|3):{NIL} cl15(0|3):{NIL} cl16(0|3):{NIL} cl17(0|3):{NIL}).

OMEGA:
```

We now display the derivation of clause `cl14` in the Emacs interface. `cl14` is the clause we obtain by application of the first pre-unifier from above. The employed pre-unifier (see clause `cl7`) is displayed in non-idempotent form. After applying pre-unifiers LEO subsequently normalizes the resulting clauses. This is why we have this superfluous looking normalization step from `cl7` to `cl14`.

```
OMEGA: show-derivation cl14
================= clauses ==================
 =============
Clause cl2 is #<Justified by ((Input))> :
 cl2(20|0):{(+(not (exists [lam dc-20735.
  (implies (p (and (a m) (or (b m) (c m))))
           (p (and (a m) (dc-20735 c b))))]))}
 =============
================== proof ===================
Clause cl2 is #<Justified by ((Input))> :
 cl2(20|0):{(+(not (exists [lam dc-20735.
  (implies (p (and (a m) (or (b m) (c m))))
           (p (and (a m) (dc-20735 c b))))]))}
 =============
Clause cl3 is #<Justified by ((CNF)) on (cl2)> :
 cl3(1.5|1):{(-(p (and (a m) (dc-20738 c b))))}
 =============
Clause cl2 is #<Justified by ((Input))> :
 cl2(20|0):{(+(not (exists [lam dc-20735.
```

```
  (implies (p (and (a m) (or (b m) (c m))))
           (p (and (a m) (dc-20735 c b))))]))))}
 =============
Clause cl4 is #<Justified by ((CNF)) on (cl2)> :
 cl4(1.5|1):{(+(p (and (a m) (or (b m) (c m)))))}
 =============
Clause cl5 is #<Justified by
((RES 1 1) (RENAMING {(dc2 --> dc37)})) on (cl3 cl4)> :
 cl5(1|2):{(-(= (p (and (a m) (dc37 c b)))
                (p (and (a m) (or (b m) (c m))))))}
 =============
Clause cl7 is #<Justified by
 ((UNI {(?h113 --> [lam ?h122 ?h123.m])
        (?h107 --> [lam ?h120 ?h121.m])
        (?h101 --> [lam ?h111 ?h112.(?h111 (?h113 ?h111 ?h112))])
        (?h100 --> [lam ?h105 ?h106.(?h106 (?h107 ?h105 ?h106))])
        (dc37 --> [lam ?h98 ?h99.(or (?h100 ?h98 ?h99)
                                     (?h101 ?h98 ?h99))])})
  (RENAMING {})) on (cl5)> :
 cl7(0|2):{NIL}
 =============
Clause cl14 is #<Justified by ((CNF)) on (cl7)> :
 cl14(0|3):{NIL}
 =============
========== clauses in proof: 7 ============

OMEGA:
```

We now slightly modify our example problem and obtain a much harder one.

$$\exists Q_{o(o\iota)(o\iota)}.\, p_{oo}((a_{o\iota}m_\iota) \wedge ((b_{o\iota}m_\iota) \vee (c_{o\iota}m_\iota))) \Rightarrow p_{oo}((Q_{o(o\iota)(o\iota)}c_{o\iota}b_{o\iota}) \wedge (a_{o\iota}m_\iota))$$

In POST this problem is represented as

```
(th~defproblem less-little
         (in base)
         (constants (p (o o)) (a (o i)) (b (o i)) (c (o i)) (m i))
         (conclusion
          (EXISTS (lam (Q (o (o i) (o i)))
                     (IMPLIES (p (AND (a m) (OR (b m) (c m))))
                              (p (AND (Q c b) (a m))))))))
```

While problem `little` can still be solved by simple higher-order to first-order transformational approaches, this is not easily the case for the modified problem `less-little` since extensionality reasoning is required. The syntactic difference to `little`, however, is small. We only switched the two inner conjuncts in the right hand side of the implication. We now first load the problem, then initialize LEO and then display LEO's initial proof state.

```
OMEGA: prove less-little
Changing to proof plan LESS-LITTLE-1

OMEGA: show-pds
              ...
LESS-LITTLE ()         ! (EXISTS [Q:(O (O I) (O I))]           OPEN
                           (IMPLIES
                            (P (AND (A M) (OR (B M) (C M))))
                            (P (AND (Q C B) (A M)))))

OMEGA: leo-initialize
NODE (NDLINE) Node to prove with LEO: [LESS-LITTLE]
TACTIC (STRING) The tactic to be used by LEO: [EXT-INPUT-RECURSIVE]
```

```
THEORY-LIST (SYMBOL-LIST) Theories whose definitions will be expanded: [()]
Expanding the Definitions....
[...]

OMEGA: show-clauses
     ================ BEGIN ================================
The set of support
     cl3(1.5|1):{(-(p (and (dc-304 c b) (a m))))}
     cl4(1.5|1):{(+(p (and (a m) (or (b m) (c m)))))}
The set of usable clauses
     ================ END ==================================

OMEGA:
```

Again we resolve between clauses `cl3` and `cl4`.

```
OMEGA: resolve cl3 cl4 1 1
Clause 1: cl3(1.5|1):{(-(p (and (dc-304 c b) (a m))))}.
Clause 2: cl4(1.5|1):{(+(p (and (a m) (or (b m) (c m)))))}.
Res(CL3[1],CL4[1]): (cl5(1|2):{(-(= (p (and (dc1573 c b) (a m)))
                                    (p (and (a m) (or (b m) (c m))))))}).

OMEGA:
```

Now the resulting clause `cl5` is not pre-unifiable. Its unification constraint has no 'syntactic' solution.

```
OMEGA: pre-unifiers cl5
The clause: cl5(1|2):{(-(= (p (and (dc1573 c b) (a m)))
                           (p (and (a m) (or (b m) (c m))))))}.
pre-unifiers: NIL.

OMEGA: pre-unify cl5
The clause: cl5(1|2):{(-(= (p (and (dc1573 c b) (a m)))
                           (p (and (a m) (or (b m) (c m))))))}.
Result: NIL.

OMEGA:
```

However, there is a semantic solution which we find by application of Leo's combined extensionality treatment. This first decomposes the unification constraint and then applies Boolean extensionality (beforehand it usually tries to exhaustively apply functional extensionality, which is not applicable here).

```
OMEGA: decompose cl5
The clause: cl5(1|2):{(-(= (p (and (dc1500 c b) (a m)))
                           (p (and (a m) (or (b m) (c m))))))}.
Decomposed: (cl6(1|3):{(-(= (and (dc1500 c b) (a m))
                            (and (a m) (or (b m) (c m)))))}).

OMEGA: ext cl6
The clause: cl6(1|3):{(-(= (and (dc1500 c b) (a m))
                           (and (a m) (or (b m) (c m)))))}
Result: (cl14(4.5|4):{(+(a m) +(dc-4682 c b))}
         cl13(6.0|4):{(+(c m) +(b m) +(dc-4681 c b))}
         cl12(1.5|4):{(+(a m))}
         cl11(4.5|4):{(+(c m) +(b m) +(a m))}
         cl10(6.0|4):{(-(b m) -(a m) -(dc-4680 c b))}
         cl9(6.0|4):{(-(c m) -(a m) -(dc-4679 c b))}).

OMEGA:
```

This set of resulting (first-order like) clauses is refutable and Leo can find a refutation. Unfortunately Leo is very bad at first-order reasoning (since it does

not employ optimizations and implementation tricks that are well known in the first-order community) and therefore the refutation of this clause set is not very efficient. This motivates a cooperation with first-order provers; see Sections 5 and 7 for further details.

5 LEO is an Automated Theorem Prover

LEO is first and foremost an automated theorem prover for classical higher-order logic. Within ΩMEGA it can be applied to prove (sub)problems automatically. Below we first reinitialize ΩMEGA with the problem `less-little` and then call LEO in its standard flag setting to it.

```
OMEGA: prove less-little
Changing to proof plan LESS-LITTLE-7

OMEGA: show-pds
             ...
LESS-LITTLE ()         ! (EXISTS [Q:(O (O I) (O I))]              OPEN
                          (IMPLIES
                           (P (AND (A M) (OR (B M) (C M))))
                           (P (AND (Q C B) (A M)))))


OMEGA: call-leo-on-node
NODE (NDLINE) Node to prove with LEO: [LESS-LITTLE]
TACTIC (STRING) The tactic to be used by LEO: [EXT-INPUT-RECURSIVE]
SUPPORTS (NDLINE-LIST) The support nodes: [()]
TIME-BOUND (INTEGER) Time bound for proof attempt: [100]
THEORY-LIST (SYMBOL-LIST) Theories whose definitions will be
expanded: [(ALL)]
DEFS-LIST (SYMBOL-LIST) Symbols whose definitions will not be
expanded: [(= DEFINED EQUIV)]
INSERT-FLAG (BOOLEAN) A flag indicating whether a partial result will
be automatically inserted.: [()]
Looking for expandable definitions ....
Initializing LEO....
Applying Clause Normalization....
[...]
Start proving ...
Loop: (100sec left)
 #1 (SOS 2 USABLE 0 EXT-QUEUE 0 FO-LIKE 4)
(99sec left)
 #2 (SOS 1 USABLE 1 EXT-QUEUE 0 FO-LIKE 4)
[...]
(94sec left)
 #29 (SOS 72 USABLE 12 EXT-QUEUE 0 FO-LIKE 98)
(91sec left)
 #30 (SOS 111 USABLE 13 EXT-QUEUE 0 FO-LIKE 138)

Total LEO time: 8446
**** proof found (next clause nr: cl599) *****
; cpu time (non-gc) 7,600 msec user, 0 msec system
; cpu time (gc)     750 msec user, 10 msec system
; cpu time (total)  8,350 msec user, 10 msec system
; real time  8,451 msec
; space allocation:
;  8,909,213 cons cells, 3,568 symbols, 122,908,112 other bytes,
; 0 static bytes

OMEGA:
```

LEO's Architecture and Main Loop

LEO's basic architecture adapts the set of support approach. The four cornerstones of LEO's architecture (see Fig. 1) are:

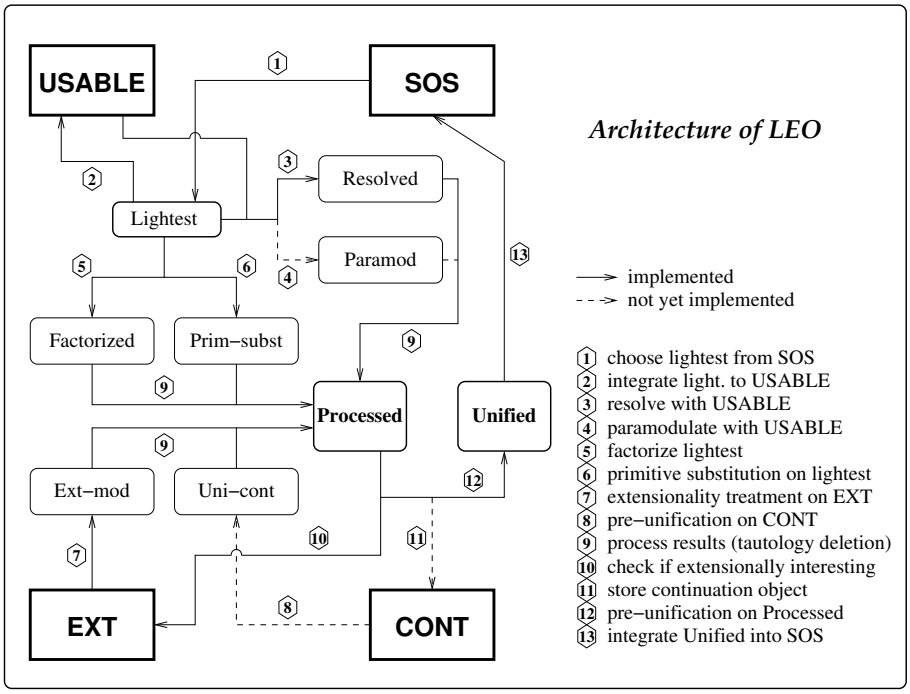

Fig. 1. LEO's main architecture (the 'dotted lines' indicates functionalities which are usually disabled or not fully available yet)

`USABLE` The set of all usable clauses, which initially only contains clauses that are assumed to be satisfiable, i.e., the clauses stemming from the assumptions of the theorem to prove.

`SOS` The set of support, which initially only contains the clauses belonging to the negated assertion.

`EXT` The set of all extensionally interesting clauses, i.e., heuristically determined clauses which are stored for extensionality treatment. Initially this set is empty. See Step 12 of the main loop description below.

`CONT` The set of all continuations created by the higher-order pre-unification algorithm when reaching the pre-unification search depth limit. The idea is to support continuations of interrupted pre-unification attempts at a later time. (This store is not activated yet in LEO)

LEO's main loop (see Fig. 1) consists of the steps 1–13 as described below (the **Initialize** step is applied only at the very beginning of the proof attempt and is not part of the main loop). This loop, whose data-flow is graphically illustrated in Figure 1, is executed until an empty clause, i.e., a clause consisting no literals or only of flex-flex-unification constraints[2], is detected.

LEO employs a higher-order subsumption test that is, apart from the technical details, very similar to the ones employed in first-order provers. Instead of first-order matching, the criteria for comparing the single literals is higher-order simplification matching, i.e., matching with respect to the deterministic higher-order simplification rules. It is theoretically possible to develop and employ a much stronger subsumption filter in LEO. This is future work. However, a perfect extensional higher-order subsumption filter, which would be the ideal case, is not feasible since extensional higher-order unification is undecidable.

Initialize The specified assumptions and the assertion are pre-clausified, i.e., the assumptions become positive unit (pre-)clauses and the assertion becomes a negative unit (pre-)clause. Definitions are expanded (with respect to the specified theories) and the pre-clauses are normalized. Within the clause normalization process the positive primitive equations are usually replaced by respective Leibniz equations. Negative primitive equations are not expanded but immediately encoded as extensional unification constraints. Furthermore, identical literals are automatically factorized. The assumption clauses are passed to `USABLE` and the assertion clause to `SOS`.

Step 1 (Choose Lightest) LEO chooses the lightest (wrt. to a clause ordering), i.e., topmost, clause from SOS. If this clause is a pre-clause, i.e., not in proper clause normal form, then LEO applies clause normalization to it and integrates the resulting proper clauses into `SOS`. Depending on the flag-setting forward and/or backward subsumption is applied; see also step 13).

Step 2 (Insert to `USABLE`) LEO inserts the lightest clause into USABLE while employing forward and/or backward subsumption depending on LEO's overall flag-setting.

Step 3 (Resolve) The lightest clause is resolved against all clauses in `USABLE` and the results are stored in `RESOLVED`.

Step 4 (Paramodulate) Paramodulation is applied between all clauses in `USABLE` and the lightest clause, and the results are stored in `PARAMOD`. (This step is currently not activated in LEO; currently Leibniz equality is globally employed instead of primitive equality.)

Step 5 (Factorize) The lightest clause is factorized and the resulting clauses are stored in `FACTORIZED`.

Step 6 (Primitive Substitution) LEO applies the primitive substitution principle to the lightest clause. The particular logical connectives to be imitated in this step are specified by a flag. The resulting clauses are stored in `PRIM-SUBST`.

[2] A flex-flex-unification constraint has topmost free variables in each of the two terms to be unified. Flex-flex-constraints are always solvable.

Step 7 (Extensionality Treatment) The heuristically sorted store `EXT` contains extensionally interesting clauses (i.e., clauses with unification constraints that may have additional pre-unifiers, if the extensionality rules are taken into account). LEO chooses the topmost clause and applies the compound extensionality treatment to all extensionally interesting literals.

Step 8 (Continue Unification) The heuristically sorted store `CONT` contains continuations of interrupted higher-order pre-unification attempts from the previous loops (cf. step 10). If the actual unification search depth limit (specified by a flag, whose value can be dynamically increased during proof attempts) allows for a deeper search in the current loop, then the additional search for unifiers will be performed. The resulting instantiated clauses are passed to `UNI-CONT` and the new continuations are sorted and integrated into `CONT`. (This step is currently not activated in LEO)

Step 9 (Collect Results) In this step LEO collects all clauses that have been generated within the current loop from the stores `RESOLVED, PARAMOD, FACTORIZED, PRIM-SUBST, EXT-MOD`, and `UNI-CONT`, eliminates obvious tautologies, and stores the remaining clauses in `PROCESSED`.

Step 10 (Pre-Unify) LEO tries to pre-unify the clauses in `PROCESSED`. Thus, it applies the pre-unification rules exhaustively, thereby spanning a unification tree until it reaches the unification search depth limit specified by a special flag. The unification search depth limit specifies how many subsequent flex-rigid-branchings may at most occur in each path through the unification search tree. The pre-unified, i.e., instantiated, clauses are passed to `UNIFIED`. The main idea of this step is to filter out all those clauses with syntactically non-solvable unification constraints (modulo the allowed search depth limit). But note that there are exceptions, which are determined in steps 11 and 12. That means that not all syntactically non-unifiable clauses are removed from the search space as this would, e.g., also remove the extensionally interesting clauses.

Step 11 (Store Continuations) Each time a pre-unification attempt in step 10 is interrupted by reaching the unification search depth limit, a respective continuation is created. This object stores the state of the interrupted unification search process, i.e., it contains the particular unification constraints as given at the point of interruption together with the remaining literals of the clause in focus and some information on the interrupted unification process. Continuations allow the prover to continue the interrupted unification process at any later time. The set of all such continuations is integrated in the sorted store `CONT`. (This step is not activated yet in LEO.)

Step 12 (Store Extensionally Interesting Clauses) In the pre-unification process in step 10 LEO analyzes the unification pairs in focus in order to estimate whether this unification constraint and thus this clause is extensionally interesting, i.e., probably solvable with respect to both extensionality principles. All extensionally interesting clauses are passed to `EXT`, which is heuristically sorted. While inserting the clauses into `EXT` forward and/or backward subsumption is applied in order to minimize the number of clauses in this store.

Step 13 (Integrate to SOS) In the last step LEO integrates all pre-unified clauses in `UNIFIED` into the sorted store `SOS`. Forward and/or backward subsumption is employed depending on the flag-setting.

6 Visualizing LEO Derivations in LOUI

ΩMEGA's graphical user interface LOUI [24] usually displays ΩMEGA proof objects in multiple modalities: a graphical map of the proof tree, a linearized presentation of the proof nodes with their formulae and justifications, and a term browser. Display of type information is optional and is determined by a switch in LOUI.

LOUI can be used to display proofs of other systems as well. LEO's connection to LOUI supports the graphical presentation of extensional higher-order resolution derivations and proofs with the gain for the user that the structure of interactively or automatically created derivations becomes more transparent as is possible in the linearized display shown in Section 4.

Display of external proofs is supported by LOUI via a specific interface language. In order to display LEO's resolution proofs a simple mapping of the internal proof state in LEO into this interface language is required. Fig. 2 illustrates the LOUI visualization of the automatically generated LEO proof for problem `less-little` from Section 5, and Fig. 3 displays part of the respective interface language representation of this proof.

7 Experiments with LEO

LEO has successfully been applied to different higher-order examples. For example, in [4] LEO's performance on simple examples about sets has been investigated. One example is the already addressed TPTP problem SET171+3, i.e., distributivity of union over intersection. Despite their simplicity such examples are often non-trivial for automated first-order theorem provers. More details on this discussion can be found in [9]. Further, proof examples have been investigated in [5].

In [9] a cooperation of LEO with a first-order theorem prover (we used the automated theorem prover Bliksem [11] since this was already well integrated in the OMEGA framework) has been proposed and investigated. Thus, LEO has been slightly extended so that it now constantly accumulates a bag of first-order like clause. First-order like clauses do not contain any 'real' higher-order subterms (such as a λ-abstraction or embedded equations), and are therefore suitable for treatment by a first-order ATP or even a propositional logic decision procedure after appropriate transformation. We use the transformation mapping as also employed in TRAMP [18], which has been previously shown to be sound and is based on [17]. Essentially, it injectively maps expressions such as $P(f(a))$ to expressions such as $@^1_{\text{pred}}(P, @^1_{\text{fun}}(f, a))$, where the @ are new first-order operators describing function and predicate application for particular types and arities.

Fig. 2. LOUI usually displays ΩMEGA proof plans and ΩMEGA natural deduction proofs. In addition it can be employed to display LEO's resolution proofs. Here we display the derivation of clause `cl7` from example problem `less-little`.

```
[...]
 insertNode(grounded none "cl3" ["cl3""cl2"]
  "([NOT (O O)] ([P (O O)] ([AND (O O O)] ([DC-11438 (O (O I) (O I))]
    [C (O I)] [B (O I)]) ([A (O I)] [M I])))"
  "((CNF))" ["cl2"] false)
 insertNode(grounded none "cl4" ["cl4""cl2"]
  "([P (O O)] ([AND (O O O)] ([A (O I)] [M I]) ([OR (O O O)] ([B (O I)]
    [M I]) ([C (O I)] [M I])))"
  "((CNF))" ["cl2"] false)

 insertNode(grounded none "cl5" ["cl5""cl4""cl3""cl2"]
  "([NOT (O O)] ([= (O O O)] ([P (O O)] ([AND (O O O)] ([A (O I)] [M I])
    ([OR (O O O)] ([B (O I)] [M I]) ([C (O I)] [M I]))))) ([P (O O)]
    ([AND (O O O)] ([dc2762 (O (O I) (O I))] [C (O I)] [B (O I)])
    ([A (O I)] [M I])))))"
  "((RES 1 1) (RENAMING {(dc2705 --> dc2762)}))" ["cl4""cl3"] false)
 insertNode(grounded none "cl6" ["cl6""cl5""cl4""cl3""cl2"]
  "([NOT (O O)] ([= (O O O)] ([AND (O O O)] ([A (O I)] [M I]) ([OR (O O O)]
    ([B (O I)] [M I]) ([C (O I)] [M I]))) ([AND (O O O)] ([dc2762 (O (O I)
    (O I))] [C (O I)] [B (O I)]) ([A (O I)] [M I])))))"
  "((DEC (1)))" ["cl5"] false)
[...]
```

Fig. 3. Part of the LOUI interface language representation of the LEO proof for problem `less-little` as communicated to LOUI.

The injectivity of the mapping guarantees soundness, since it allows each proof step to be mapped back from first-order to higher-order.

Whenever LEO creates a new clause it checks whether this is a first-order like clause, i.e., whether it is in the domain of the employed transformational mapping. If this is the case, a copy of it is passed to the store of first-order like clauses. In each loop of LEO's search procedure a fast first-order prover can now be applied to the set of first-order like clauses to find a refutation. In this case an overall proof has been found. The experiments in [9] show that this is a very promising approach to combining the benefits of higher-order and first-order theorem provers. Whereas this cooperative approach can solve the problem `less-little` only slightly faster than LEO alone, many examples in [9] show that there are often significant improvements possible.

```
OMEGA: prove less-little
Changing to proof plan LESS-LITTLE-10

OMEGA: call-leo-on-node
NODE (NDLINE) Node to prove with LEO: [LESS-LITTLE]
TACTIC (STRING) The tactic to be used
by LEO: [EXT-INPUT-RECURSIVE]fo-atp-cooperation
SUPPORTS (NDLINE-LIST) The support nodes: [()]
TIME-BOUND (INTEGER) Time bound for proof attempt: [100]
THEORY-LIST (SYMBOL-LIST) Theories whose definitions will be
expanded: [(ALL)]
DEFS-LIST (SYMBOL-LIST) Symbols whose definitions will not be
expanded: [(= DEFINED EQUIV)]
INSERT-FLAG (BOOLEAN) A flag indicating whether a partial result
will be automatically inserted.: [()]
Looking for expandable definitions ....
Initializing LEO....
Applying Clause Normalization....
[...]
```

```
Start proving ...
Loop: (100sec left)
 #1 (SOS 2 USABLE 0 EXT-QUEUE 0 FO-LIKE 4)
(99sec left)
 #2 (SOS 1 USABLE 1 EXT-QUEUE 0 FO-LIKE 4)
(98sec left)
[...]
(96sec left)
 #9 (SOS 3 USABLE 4 EXT-QUEUE 0 FO-LIKE 12)
[...]
 Calling bliksem process 22267 with time resource 50sec .
 PARSING BLIKSEM OUTPUT ...
 Bliksem has found a saturation.
[...]
(96sec left)
 #10 (SOS 10 USABLE 5 EXT-QUEUE 0 FO-LIKE 20)
[...]
(94sec left)
 #21 (SOS 39 USABLE 9 EXT-QUEUE 0 FO-LIKE 60)
 Calling bliksem process 22454 with time resource 50sec .
 bliksem Time Resource in seconds:
 PARSING BLIKSEM OUTPUT ...
 Bliksem has found a proof.
Bliksem's time:
; cpu time (non-gc) 0 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  0 msec user, 0 msec system
; real time  174 msec
; space allocation:
;  416 cons cells, 0 symbols, 15,720 other bytes, 8936 static bytes
Input Clauses: 75
clauses generated:      37
(94sec left)
 #22 (SOS 53 USABLE 10 EXT-QUEUE 0 FO-LIKE 75)
Total LEO time: 7262
**** proof found (next clause nr: cl323) *****
; cpu time (non-gc) 5,650 msec user, 20 msec system
; cpu time (gc)     490 msec user, 0 msec system
; cpu time (total)  6,140 msec user, 20 msec system
; real time  7,273 msec
; space allocation:
;  5,218,155 cons cells, 1,886 symbols, 82,231,456 other bytes,
;  49536 static bytes
```

8 Related Work and Conclusion

There are only very few automated theorem provers available for higher-order logic. TPS [2, 3], which is based on the mating search method, is the oldest and probably still the strongest prover in this category. The extensionality reasoning of TPS has recently been significantly improved by Brown in his PhD thesis [10].

Related to the cooperation approach is the work of Hurd [16] which realizes a generic interface between higher-order logic and first-order theorem provers. It is similar to the solution previously achieved by TRAMP [18] in ΩMEGA. Both approaches pass essentially first-order clauses to first-order theorem provers and then translate their results back into higher-order. More recent related work on the cooperation of ISABELLE with the first-order theorem prover VAMPIRE is presented in [19]. Further related work is OTTER-λ (see http://mh215a.cs.sjsu.edu/), which extends first-order logic with λ-notation.

LEO has initially been implemented as a demonstrator system for extensional higher-order resolution in the context of the author's PhD thesis [4]. The experiments carried out with LEO so far, in particular, its recent combination with a fast first-order theorem prover, have been very promising, and they motivate further work in this direction. This is particularly true since interactive proof assistants based on higher-order logic are recently gaining increasing attention in formal methods.

During the implementation and later during the experiments many shortcomings of LEO have been identified by the author. These shortcomings are both of theoretical and of practical nature. Altogether this calls for a proper reimplementation of LEO. This reimplementation should ideally be independent of ΩMEGA in order to provide a lean and easy to install and use automated higher-order theorem to the community.

References

1. P. Andrews. *An Introduction to mathematical logic and Type Theory: To Truth through Proof*. Number 27 in Applied Logic Series. Kluwer, 2002.
2. P. B. Andrews, M. Bishop, and C. E. Brown. System description: TPS: A theorem proving system for type theory. In *Conference on Automated Deduction*, pages 164–169, 2000.
3. P.B. Andrews, M. Bishop, S. Issar, D. Nesmith, F. Pfenning, and H. Xi. TPS: A theorem proving system for classical type theory. *Journal of Automated Reasoning*, 16(3):321–353, 1996.
4. C. Benzmüller. *Equality and Extensionality in Higher-Order Theorem Proving*. PhD thesis, Universität des Saarlandes, Germany, 1999.
5. C. Benzmüller. Comparing approaches to resolution based higher-order theorem proving. *Synthese*, 133(1-2):203–235, 2002.
6. C. Benzmüller and M. Kohlhase. Extensional higher-order resolution. In *Proc. of CADE-15*, number 1421 in LNAI. Springer, 1998.
7. C. Benzmüller and M. Kohlhase. LEO – a higher-order theorem prover. In *Proc. of CADE-15*, number 1421 in LNAI. Springer, 1998.
8. C. Benzmüller and V. Sorge. OANTS – An open approach at combining Interactive and Automated Theorem Proving. In *Proc. of Calculemus-2000*. AK Peters, 2001.
9. C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Can a higher-order and a first-order theorem prover cooperate? In F. Baader and A. Voronkov, editors, *Proceedings of the 11th International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 3452 of *LNAI*, pages 415–431. Springer, 2005.
10. C. E. Brown. *Set Comprehension in Church's Type Theory*. PhD thesis, Dept. of Mathematical Sciences, Carnegie Mellon University, USA, 2004.
11. H. de Nivelle. *The Bliksem Theorem Prover, Version 1.12*. Max-Planck-Institut, Saarbrücken, Germany, 1999. http://www.mpi-sb.mpg.de/~bliksem/manual.ps.
12. M. Gordon and T. Melham. *Introduction to HOL – A theorem proving environment for higher order logic*. Cambridge University Press, 1993.
13. J. Harrison. The hol light theorem prover.
14. G.P. Huet. *Constrained Resolution: A Complete Method for Higher Order Logic*. PhD thesis, Case Western Reserve University, 1972.

15. G.P. Huet. A mechanization of type theory. In Donald E. Walker and Lewis Norton, editors, *Proc. of the 3rd International Joint Conference on Artificial Intelligence (IJCAI73)*, pages 139–146, 1973.
16. J. Hurd. An LCF-style interface between HOL and first-order logic. In *Automated Deduction — CADE-18*, volume 2392 of *LNAI*, pages 134–138. Springer, 2002.
17. M. Kerber. *On the Representation of Mathematical Concepts and their Translation into First Order Logic*. PhD thesis, Universität Kaiserslautern, Germany, 1992.
18. A. Meier. TRAMP: Transformation of Machine-Found Proofs into Natural Deduction Proofs at the Assertion Level. In *Proc. of CADE-17*, number 1831 in LNAI. Springer, 2000.
19. J. Meng and L. C. Paulson. Experiments on supporting interactive proof using resolution. In *Proc. of IJCAR 2004*, volume 3097 of *LNCS*, pages 372–384. Springer, 2004.
20. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Number 2283 in LNCS. Springer, 2002.
21. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in LNCS, pages 411–414, New Brunswick, NJ, 1996. Springer.
22. L. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer, 1994.
23. J. Siekmann, C. Benzmüller, A. Fiedler, A. Meier, I. Normann, and M. Pollet. Proof development in OMEGA: The irrationality of square root of 2. In *Thirty Five Years of Automating Mathematics*. Kluwer, 2003.
24. J. Siekmann, S. Hess, C. Benzmüller, L. Cheikhrouhou, A. Fiedler, H. Horacek, M. Kohlhase, K. Konrad, A. Meier, E. Melis, M. Pollet, and V. Sorge. LOUI: Lovely OMEGA user interface. *Formal Aspects of Computing*, 11:326–342, 1999.
25. G.L. Steele. *Common Lisp: The Language, 2nd edition*. Digital Press, Bedford, Massachusetts, 1990.