Progress Report on LEO-II, an Automatic Theorem Prover for Higher-Order Logic*

Christoph Benzmüller[1,2], Larry Paulson[1], Frank Theiss[2], and Arnaud Fietzke[2]

[1]Computer Laboratory, The University of Cambridge, UK
[2]Computer Science, Saarland University, Saarbrücken, Germany

Abstract. LEO-II, a resolution based theorem prover for classical higher-order logic, is currently being developed in a one year research project at the University of Cambridge, UK, with support from Saarland University, Germany. We report on the current stage of development of LEO-II. In particular, we sketch some main aspects of LEO-II's automated proof search procedure, discuss its cooperation with first-order specialist provers, show that LEO-II is also an interactive proof assistant, and explain its shared term data structure and its term indexing mechanism.

1 Introduction

Automatic theorem provers (ATPs) based on the resolution principle, such as Vampire [26], E [27], and SPASS [31], have reached a high degree of sophistication. They can often find long proofs even for problems having thousands of axioms. However, they are limited to first-order logic. Higher-order logic extends first-order logic with lambda notation for functions and with function and predicate variables. It supports reasoning in set theory, using the obvious representation of sets by predicates. Higher-order logic is a natural language for expressing mathematics, and it is also ideal for formal verification. Moving from first-order to higher-order logic requires a more complicated proof calculus, but it often allows much simpler problem statements. Higher-order logic's built-in support for functions and sets often leads to shorter proofs. Conversely, elementary identities (such as the distributive law for union and intersection) turn into difficult problems when expressed in first-order form.

The LEO-II project develops a standalone, resolution-based higher-order theorem prover that is designed for fruitful cooperation with specialist provers for first-order and propositional logic. The idea is to combine the strengths of the different systems. On the other hand, LEO-II itself, as an external reasoner, wants to support interactive proof assistants such as Isabelle/HOL [24], HOL [16], and OMEGA [28] by efficiently automating subproblems and thereby reducing user effort.

LEO-II predominantly addresses higher-order aspects in its reasoning process with the aim to quickly remove higher-order clauses from the search space and

* The LEO-II project is funded by EPSRC under grant EP/D070511/1, "LEO-II: An Effective Higher-Order Theorem Prover."

to turn them into essentially first-order clauses which can then be refuted with a first-order prover. Furthermore, the project investigates whether techniques that have proved very successful in automated first-order theorem proving, such as shared data structures and term indexing, can be lifted to the higher-order setting. LEO-II is implemented in OCAML; it is the successor of LEO [9, 7], which was implemented in LISP and hardwired to the OMEGA proof assistant.

This paper is structured as follows: Sec. 2 presents some preliminaries. Sec. 3 illustrates LEO-II's main proof search procedure which is based on extensional higher-order resolution. The cooperation of LEO-II with other specialist provers is discussed in Sec. 4. LEO-II is also an interactive proof assistant as we will explain in Sec. 5. In Sec. 6 we address LEO-II's shared term data structures and term indexing. Sec. 7 mentions related work and Sec. 8 concludes the paper.

2 Preliminaries

LEO-II's logic LEO-II's logic is classical higher-order logic (Church's simple type theory [11]), which is a logic built on top of the simply typed λ-calculus. The set of simple types $\mathcal{T}$ is usually freely generated from basic types o and ι using the function type constructor $\rightarrow$. In LEO-II we allow an arbitrary but fixed number of additional base types to be specified.

For formulae we start with a set of (typed) variables (denoted by $X_\alpha, Y, Z, X^1_\beta, X^2_\gamma \ldots$) and a set of (typed) constants (denoted by $c_\alpha, f_{\alpha\rightarrow\beta}, \ldots$). The set of constants includes LEO-II's primitive logical connectives $\bot_o$, $\top_o$ $\neg_{o\rightarrow o}$, $\vee_{o\rightarrow o\rightarrow o}$, $\Pi_{(\alpha\rightarrow o)\rightarrow o}$ (abbreviated Π^α) and $=_{\alpha\rightarrow\alpha\rightarrow o}$ (abbreviated $=^\alpha$) for each type α. Other logical connectives can be defined in LEO-II in terms of the these primitive ones (as we will later see).

Formulae (or terms) are constructed from typed variables and constants using application and λ-abstraction. We use Church's dot notation so that **.** stands for a (missing) left bracket whose mate is as far to the right as possible (consistent with given brackets). We use infix notation $\mathbf{A} \vee \mathbf{B}$ for $((\vee\mathbf{A})\mathbf{B})$ and binder notation $\forall X_\alpha.\mathbf{A}$ for $(\Pi^\alpha(\lambda X_\alpha.\mathbf{A}_o))$.

The target semantics for LEO-II in the first place is Henkin semantics; see [8, 17] for further details. Thus, in theory LEO-II aims at a Henkin complete calculus which includes Boolean and functional extensionality as required, for instance, to prove

$$=^\iota =^{\iota\rightarrow\iota\rightarrow o} (\lambda X_\iota.\lambda Y_\iota.Y =^\iota X)$$

In practice, however, we sacrifice completeness and instead put an emphasize on coverage of the problems we are interested in.

Literals, Unification Constraints, and Clauses Here are examples of literals (α is an arbitrary type); they consist of a literal atom in [.]-brackets and a polarity T or F for positive or negative literal respectively:

$$[\mathbf{A}_o]^{=T} \qquad [\mathbf{B}_o]^{=F} \qquad [\mathbf{C}_\alpha =^\alpha \mathbf{D}_\alpha]^{=T} \qquad [\mathbf{F}_\alpha =^\alpha \mathbf{G}_\alpha]^{=F}$$

```
thf(reflexiv,definition,
    (reflexive := (^[R:$i>($i>$o)]: (![X:$i]: ((R @ X) @ X))))).
thf(symmetric,definition,
    (symmetric := (^[R:$i>($i>$o)]: (![X:$i,Y:$i]: ((R @ X) @ Y) => ((R @ Y) @ X))))).
thf(transitive,definition,
    (transitive := (^[R:$i>($i>$o)]: (![X:$i,Y:$i,Z:$i]:
                                        ((((R @ X) @ Y) & ((R @ Y) @ Z)) => ((R @ X) @ Z)))))).
thf(equiv_rel,definition,
    (equiv_rel := (^[R:$i>($i>$o)]: (reflexive @ R) & (symmetric @ R) & (transitive @ R)))).
thf(test,theorem,(?[R:$i>($i>$o)]: ~(equiv_rel @ R))).
```

Fig. 1. Example problem in THF syntax.

Negative equation literals, such as our fourth example above, are also called a *unification constraints*. If both terms $\mathbf{F}_\alpha$ and $\mathbf{G}_\alpha$ in a unification constraint have a free variable at head position then we call it a *flex-flex unification constraint.* If only one of them has a free variable at head position we call it a *flex-rigid unification constraint.* Flex-flex and flex-rigid unification constraints may have infinitely many different solutions. For example, the following flex-rigid constraint (H is a variable and f and a are constants) has $H \longleftarrow \lambda x. \underbrace{f(f \ldots (f}_{n} x) \ldots)$ for all $n \geq 0$ amongst its solutions :

$$[H_{\iota\to\iota}(f_{\iota\to\iota}a_\iota) = f_{\iota\to\iota}(H_{\iota\to\iota}a_\iota)]^{=F}$$

The following clause consists of a positive literal and a unification constraint: $\mathcal{C} : [\mathbf{A}_o]^{=T}, [\mathbf{F}_\alpha =^\alpha \mathbf{G}_\alpha]^{=F}$. It corresponds to the formula $(\mathbf{F}_\alpha =^\alpha \mathbf{G}_\alpha) \Rightarrow \mathbf{A}_o$, which explains the name *unification constraint*: if we can equalize $\mathbf{F}_\alpha$ and $\mathbf{G}_\alpha$, for example, by unification with a unifier σ, then $\sigma(\mathbf{A}_o)$ holds.

An Example Problem in LEO-II's Input Syntax LEO-II employs a fragment of the new higher-order TPTP THF[1] syntax as its input and output syntax. We present an example problem in THF syntax in Fig. 1. It states that there exists a relation R (of type $(\iota \to \iota) \to o$) which is not an equivalence relation. In the notation of this paper this problem reads as follows:

$$reflexive \overset{def}{=} \lambda R_{(\iota\to\iota)\to o}.\forall X_\iota.(R\ X\ X) \tag{1}$$
$$symmetric \overset{def}{=} \lambda R_{(\iota\to\iota)\to o}.\forall X_\iota.\forall Y_\iota.(R\ X\ Y) \Rightarrow (R\ Y\ X) \tag{2}$$
$$transitive \overset{def}{=} \lambda R_{(\iota\to\iota)\to o}.\forall X_\iota.\forall Y_\iota.\forall Z_\iota.((R\ X\ Y) \wedge (R\ Y\ Z)) \Rightarrow (R\ X\ Z) \tag{3}$$
$$equiv\_rel \overset{def}{=} \lambda R_{(\iota\to\iota)\to o}.(reflexive\ R) \wedge (symmetric\ R) \wedge (transitive\ R) \tag{4}$$

$$\exists R_{(\iota\to\iota)\to o}.\neg(equiv\_rel\ R) \tag{5}$$

(1)–(4) are examples of definitions and (5) is the simple higher-order theorem which we want to prove. For this, LEO-II needs to generate a concrete instance for R and show that this instance is not an equivalence relation. Tow obvious candidates are inequality or the empty relation:

$$\{(x,y)|x \neq y\} \textit{ represented by } \lambda X_\iota.\lambda Y_\iota.\neg(X = Y) \tag{6}$$
$$\{(x,y)|false\} \textit{ represented by } \lambda X_\iota.\lambda Y_\iota.\bot \tag{7}$$

As we will see later, LEO-II finds the latter instance and then shows that it does not fulfill the reflexivity property.

A Note on Polymorphism Polymorphism and type inference is supported so far only partially in the LEO-II prover. Full support would add another dimension of complexity and non-determinism, as we will now briefly explain. Consider the following two operation tables

$\wedge_{o\to o\to o}$	$\top_o$	$\bot_o$
$\top_o$	$\top_o$	$\bot_o$
$\bot_o$	$\bot_o$	$\bot_o$

$\lambda F_{o\to o}\lambda G_{o\to o}\lambda X_o.(G\ (F\ X))$	$\lambda X_o.X_o$	$\lambda X_o.\top$
$\lambda X_o.X_o$	$\lambda X_o.X_o$	$\lambda X_o.\top$
$\lambda X_o.\top$	$\lambda X_o.\top$	$\lambda X_o.\top$

They are concrete instances of the following polymorphic (or schematic) table (where a is a type variable and where we assume that $A \neq B$):

$\circ_{\alpha\to\alpha\to\alpha}$	A_α	B_α
A_α	A_α	B_α
B_α	B_α	B_α

Using type variable α, we can easily formulate a theorem in polymorphic higher-order logic, expressing that there exist an instance of the latter table:

$$\begin{array}{l}\exists\alpha.\ \exists \circ_{\alpha\to\alpha\to\alpha}.\exists A_\alpha.\exists B_\alpha. \\ \quad A \neq B \wedge (\circ AA) = A \wedge (\circ AB) = B \wedge (\circ BA) = B \wedge (\circ BB) = B\end{array}$$

Negation and clause normalization reduces this theorem to the following clause (where $A, B, \circ$ are free variables):

$$\begin{array}{c}\mathcal{E}_1 : [A_\alpha = B_\alpha]^{=T},\ [(\circ_{\alpha\to\alpha\to\alpha} A_\alpha A_\alpha) = A_\alpha]^{=F},\ [(\circ_{\alpha\to\alpha\to\alpha} A_\alpha B_\alpha) = B_\alpha]^{=F}, \\ [(\circ_{\alpha\to\alpha\to\alpha} B_\alpha A_\alpha) = B_\alpha]^{=F},\ [(\circ_{\alpha\to\alpha\to\alpha} B_\alpha A_\alpha) = B_\alpha]^{=F}\end{array}$$

This clause consists of four flex-flex unification constraints and one positive flex-flex equation. Additionally, there is the type variable α. In order to refute this clause, the guessing of instances for the free type variable α in combination with a guessing of instances for the term variables seems unavoidable. This is why we want to avoid full polymorphism and concentrate for the moment only on one dimension of the problem, namely the instantiation of the free variables at term level.

We do, however, provide some basic support for prefix-polymorphism in LEO-II. This means that we can define concepts using type variables if their instantiation can be uniquely determined in an application context by means of a very simple type inference mechanism.

An example is the following modified definition of *reflexive*, which generalizes from type $i to the type variable A:

```
reflexive := ^ [A:$type]: ^[R:A>(A>$o)]: (![X:A]: ((R @ X) @ X))
```

3 The Automatic Proof Search Procedure

We sketch some main aspects of LEO-II's automated proof search procedure.

Problem Initialization Given a problem in THF syntax consisting of a set of $n \geq 0$ assumptions $\mathbf{A}_1, \ldots, \mathbf{A}_n$ and a conjecture $\mathbf{C}$. Initialization in LEO-II creates the following initial clauses (usually they are not in clause normal form):

$$\mathcal{B}_1 : [\mathbf{A}_1]^{=T} \quad \ldots \quad \mathcal{B}_n : [\mathbf{A}_n]^{=T} \qquad \mathcal{B}_{n+1} : [\mathbf{C}]^{=F}$$

For our example problem in (1)–(5) we obtain the following clause

$$\mathcal{C}_1 : [\exists R_{(\iota\to\iota)\to o}\text{.}\neg(equiv\_rel\ R)]^{=F}$$

Definitions are not turned into clauses but kept implicit as rewriting rules.

Definition Unfolding A definition in LEO-II is of form $\mathbf{A} \overset{def}{=} \mathbf{B}$ where $\mathbf{Free}(\mathbf{B}) \subseteq \mathbf{Free}(\mathbf{A})$. We have already seen simple definition examples in (1)–(4). In particular, non-primitive logical connectives are introduced as definitions in LEO-II; for example:

$$\wedge_{o\to o\to o} \overset{def}{=} \lambda X_o \lambda Y_o.(X \vee Y) \qquad \supset_{o\to o\to o} \overset{def}{=} \lambda X_o \lambda Y_o.(\neg X \vee Y)$$
$$\Leftrightarrow_{o\to o\to o} \overset{def}{=} \lambda X_o \lambda Y_o.(X \supset Y) \wedge (Y \supset X) \qquad \forall X_\alpha.\mathbf{A}_o \overset{def}{=} \Pi^\alpha \lambda X_\alpha \mathbf{A}$$
$$\exists X_\alpha.\mathbf{A}_o \overset{def}{=} \neg\forall X_\alpha\ \neg\mathbf{A}$$

Currently, LEO-II simultaneously unfolds all definitions immediately after problem initialization (and thereby heavily benefits from the shared term data structures and indexing techniques as will be sketched in Sec. 6). Delayed and stepwise definition unfolding, which is required to successfully prove certain theorems [10], is future work. When applied to our example clause $\mathcal{C}_1$, definition unfolding generates clause 3 as depicted in Fig. 2.

Clause Normalization For clause normalization, LEO-II employs rules addressing the primitive logical connectives $\bot, \top, \neg, \vee, \Pi^\alpha$ and $=^\alpha$ for all types α. The rules for $=^\alpha$ apply functional and Boolean extensionality principles. Clause normalization is an integral part of LEO-II's calculus and it is not just applied in an initial phase such as in first-order theorem proving. Quite to the contrary, clause normalization is repeatedly required as we will later see.

Unfortunately, clause normalization as currently realized in LEO-II is quite naive. Future work therefore includes the development of a more efficient approach, for example, one based on the ideas of Flotter [32].

Fig. 2 shows the result of our running example after problem initialization, definition unfolding, and clause normalization. The clauses present in LEO-II's search state then are 13, 25, and 31. In our standard notation they read as follows (the V^i are all free variables):

$$\mathcal{C}_{15} : [V^1\ V^2\ V^2]^{=T} \qquad \mathcal{C}_{25} : [V^1\ V^5\ V^3]^{=T}, [V^1\ V^3\ V^5]^{=F}$$
$$\mathcal{C}_{31} : [V^1\ V^4\ V^7]^{=T}, [V^1\ V^4\ V^6]^{=F}, [V^1\ V^6\ V^7]^{=F}$$

```
1: (? [R:$i>($i>$o)] : (~ (equiv_rel @ R)))=$true
    ---  theorem(file('../problems/SIMPLE-MATHS-5.thf',[test]))
2: (? [R:$i>($i>$o)] : (~ (equiv_rel @ R)))=$false
    ---  neg_input 1
3: (~ (! [x0:$i>($i>$o)] : (~ (~ (~ ((~ (! [x1:$i] : ((x0 @ x1) @ x1))) |
   (~ (~ ((~ (! [x1:$i,x2:$i] : ((~ ((x0 @ x1) @ x2)) |
   ((x0 @ x2) @ x1)))) | (~ (! [x1:$i,x2:$i,x3:$i] :
   ((~ (~ ((~ ((x0 @ x1) @ x2)) | (~ ((x0 @ x2) @ x3))))) |
   ((x0 @ x1) @ x3))))))))))))))=$false
    ---  unfold_def 2
...
13: ((V_x0_1 @ V_x1_2) @ V_x1_2)=$true
    ---  cnf 11
25: ((V_x0_1 @ V_x2_5) @ V_x1_3)=$true | ((V_x0_1 @ V_x1_3) @ V_x2_5)=$false
    ---  cnf 23
31: ((V_x0_1 @ V_x1_4) @ V_x3_7)=$true | ((V_x0_1 @ V_x1_4) @ V_x2_6)=$false |
    ((V_x0_1 @ V_x2_6) @ V_x3_7)=$false
    ---  cnf 30
...
```

Fig. 2. Excerpt of the LEO-II proof protocol for our running example. Clause 13, 25, and 31 are the results of problem initialization, unfolding of definitions, and exhaustive clause normalization. The V_i are free variables.

Extensional Pre-Unification Pre-unification in LEO-II is based on the rules as presented in former work [6]. It is well known, that pre-unification is not decidable (remember our discussion of flex-rigid unification constraints in the beginning) which is why LEO-II introduces a depth limit for the generation of pre-unifiers. This clearly threatens completeness. Iterative deepening or dovetailing the generation of pre-unifiers with the overall proof search are possible ways out, however, so far we simply sacrifice completeness.

LEO-II also provides a unification rule for Boolean extensionality which transforms unification constraints between terms of type o back into proof problems. Consider, for example, clause $\mathcal{D}_1$ below, which consists of exactly one unification constraint (a negated equation between two syntactically non-unifiable abstractions). Note how $\mathcal{D}_1$ is translated by functional and Boolean extensionality into the propositional-like clauses $\mathcal{D}_3$ and $\mathcal{D}_4$ (p, q are constants of type $\iota \longrightarrow o$ and s is a constant of type ι):

$$\begin{array}{rl} & \mathcal{D}_1 : [(\lambda X_\iota\text{.}(pX) \wedge (qX)) = (\lambda X_\iota\text{.}(qX) \wedge (pX))]^{=F} \\ \mathit{Func}\ \mathcal{D}_1 : & \mathcal{D}_2 : [((ps) \wedge (qs)) = (qs) \wedge (ps))]^{=F} \\ \mathit{Bool}\ \mathcal{D}_2 : & \mathcal{D}_3 : [((ps) \wedge (qs))]^{=F}, [(qs) \wedge (ps))]^{=F} \\ \mathit{Bool}\ \mathcal{D}_2 : & \mathcal{D}_4 : [((ps) \wedge (qs))]^{=T}, [(qs) \wedge (ps))]^{=T} \end{array}$$

This example also illustrates why clause normalization is not only a preliminary phase in LEO-II: $\mathcal{D}_3$ and $\mathcal{D}_4$ are non-normal clauses generated by LEO-II's extensional pre-unification approach and they need to be subsequently normalized. Note also how the interplay of functional and Boolean extensionality turns our initially higher-order problem (semantic unifiability of two abstractions) into a propositional one. This aspect is picked up again in Sec. 4, where we address

```
LEO-II> prove
3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 [1] 33 34
35 36 37 38 39 40
Eureka --- Thanks to Corina!
Here are the empty clauses
 [
41:[ 0:<$false = $true>-w(1)-i() ]-mln(1)-w(1)-i(sim 35)-fv([ ])
]
0.01373: Total Reasoning Time
LEO-II (Proof Found!)> show-derivation 41
**** Beginning of derivation protocol ****
[...]
15: ((V_x0_1 @ V_x1_2) @ V_x1_2)=$true  ---  cnf 13
35: ($false)=$true  ---  prim-subst (V_x0_1-->lambda [V25]: lambda [V26]: false) 15
41: ($false)=$true  ---  sim 35
**** End of derivation protocol ****
LEO-II (Proof Found!)>
```

Fig. 3. Second excerpt of the LEO-II proof protocol for our running example; see also Fig. 2. From Clause 13 (reflexivity) we derive a contradiction (the clause 41). The key step is the guessing of an appropriate relation via primitive substitution in clause 35.

cooperation of LEO-II with specialist provers for fragments of higher-order logic, such as first-order and propositional logic.

Resolution, Factorization, and Primitive Substitution Resolution, factorization and primitive substitution are driving the main proof search of LEO-II. Respective rules are presented in [6]. Unlike in first-order theorem proving, resolution and factorization do not employ unification directly as a filter, but instead introduce unification constraints. These unification constraints are processed subsequently with the extensional pre-unification rules.

Primitive substitution is needed in higher-order resolution theorem proving for completeness reasons. We illustrate the importance of primitive substitution with the help of our example clauses $\mathcal{C}_{11}$, $\mathcal{C}_{25}$, and $\mathcal{C}_{31}$ expressing that the relation represented by variable V^1 is reflexive, symmetric, and transitive. Remember that these clauses have been derived from (the negation of) our example problem (1)–(5). In order to prove theorem (5) we need to refute the clause set $\{\mathcal{C}_{11}, \mathcal{C}_{25}, \mathcal{C}_{31}\}$. Without guessing a candidate relation for V^1, LEO-II cannot find a refutation. Guessing instantiations of free relation variables, such as V^1, occurring at head positions in literals is the task of the primitive substitution rule. In fact, LEO-II proposes the instance $\lambda X_\iota . \lambda Y_\iota . \bot$, see (7) above. Then it quickly finds a contradiction to reflexivity; see the proof protocol excerpt in Fig. 3.

Simplification, Rewriting, Subsumption, Heuristic Control, etc. Simplification, rewriting, and subsumption are still at comparably early stage of development. However, we believe that with the help of our term indexing mechanism (see Sec. 6) we shall be able to develop efficient solutions soon. Intelligent heuristic control, for example, based on term orderings, is also future work.

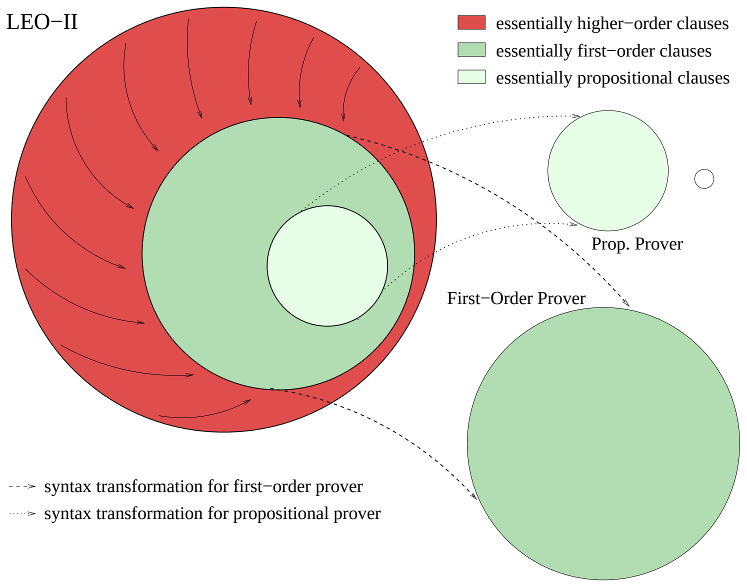
LEO−II
essentially higher−order clauses
essentially first−order clauses
essentially propositional clauses
Prop. Prover
First−Order Prover
syntax transformation for first−order prover
syntax transformation for propositional prover

Fig. 4. Illustration of LEO-II's cooperation with first-order provers.

LEO-II **is incomplete (and probably always will be)** As has been pointed out already, LEO-II sacrifices completeness in various respects for pragmatic reasons. When our current prototype development has sufficiently progressed, our first interest will be to adapt LEO-II to particular problem classes and to make it strong for them. We have a particular interest in optimizing the reasoning in LEO-II towards quick transformation of essentially higher-order clauses into essentially first-order or propositional ones as illustrated before (see $\mathcal{D}_1 - \mathcal{D}_4$). Then, as we will further discuss in Sec. 4, we want to cooperate with specialist reasoners for efficient refutation of these subsets of clauses. In summary, we are rather interested in the strengths of the cooperative approach than in isolated completeness of LEO-II and both aspects may even turn out to be in conflict with each other.

4 Cooperation with Specialist Reasoners

LEO-II tries to operate predominantly on essentially higher-order clauses with the goal to reduce them to sets of essentially first-order or essentially propositional ones. Ideally, the latter sets grow until they eventually become efficiently refutable by an automated specialist reasoner. Fig. 4 graphically illustrates this idea.

We also illustrate the idea by a slight modification of our running example. Instead of proving (5) from (1)–(4) we now want to prove (8) from (1)–(4):

$$(equiv\_rel\ \ =) \tag{8}$$

Compared to (5) this is now an even simpler problem and it is clearly not higher-order anymore. In fact, definition unfolding and clause normalization immediately turns this problem into a trivially refutable set of equations containing not a single free variable, see the clauses 26, 28, 38, 39, 40, and 41 in Fig. 3. In the general case, however, the set of essentially first-order clauses and essentially propositional clauses generated by LEO-II this way may easily become very large. Since LEO-II's proof search is not tailored for efficient first-order or propositional reasoning it may therefore fail to prove them. Therefore, a main objective of LEO-II is to fruitfully cooperate with first-order reasoners.

In Fig. 5 we illustrate the cooperative proof search approach by showing an excerpt of a LEO-II session in which the generated essentially first-order clauses 26, 28, 38, 39, 40, and 41 are passed to prover E [27] for refutation.

Here are some general remarks on our approach:

- The idea is to cooperate with different specialist reasoners for fragments of higher-order logic. Currently we are experimenting with the first-order systems E and SPASS. Nevertheless our idea is generic and LEO-II may later support also other fragments, with propositional logic and monadic second-order logic as possible candidates.
- As has been shown by Hurd [18] and Meng/Paulson [22] the particular syntax translations used to convert essentially higher-order clauses to essentially first-order ones may have a strong impact on the efficiency of the specialist provers. Thus, we later want to support a wide range of such syntax transformations. Currently, the transformations supported in LEO are based on Kerber's work [19] and on the work of Hurd [18]. For communication with first-order provers we use the TPTP FOF syntax [29].
- The specialist reasoners will run in parallel to LEO-II (cf. [5, 3, 2, 4]) and they will incrementally receive new clauses from LEO-II belonging to their fragment. Once they find a refutation, they report this back to LEO-II, which then reports that a proof has been found. However, since this has not been fully realized yet we still employ a sequential approach.
- We are interested in producing proof objects that are verifiable by the proof assistant LEO-II is working for, in the first place, Isabelle/HOL. LEO-II already produces detailed proof objects in (extended) TSTP format. The results of the specialist reasoners, however, can currently not be translated back into LEO-II (sub-)proofs. This is future work.

5 LEO-II as an Interactive Proof Assistant

LEO-II provides an interactive mode in which user and system can interact to produce proofs in simple type theory. So far LEO-II offers 49 commands, roughly half of which support interactive proof construction. The others are of general nature and support tasks like reading a problem from a file or inspection of LEO-II's search state.

```
1: (equiv_rel @ (^ [X:$i,Y:$i] : (X = Y)))=$true
   ---  theorem(file('../problems/SIMPLE-MATHS-3.thf',[thm]))
2: (equiv_rel @ (^ [X:$i,Y:$i] : (X = Y)))=$false
   ---  neg_input 1
3: (~ ((~ (! [x0:$i,x1:$i] : (x0 = x0))) | (~ (~ ((~ (! [x0:$i,x1:$i] : ((~ (x0 = x1)) |
   (x1 = x0)))) | (~ (! [x0:$i,x1:$i,x2:$i] : ((~ (~ ((~ (x0 = x1)) | (~ (x1 = x2))))) |
   (x0 = x2)))))))))=$false
    ---  unfold_def 2
[...]
26: (sk_x0_1 = sk_x0_1)=$false | (sk_x1_4 = sk_x0_3)=$false | (sk_x0_5 = sk_x2_9)=$false
    ---  cnf 24
[...]
28: (sk_x0_1 = sk_x0_1)=$false | (sk_x0_3 = sk_x1_4)=$true | (sk_x0_6 = sk_x2_10)=$false
    ---  cnf 25
[...]
38: (sk_x1_4 = sk_x0_3)=$false | (sk_x0_1 = sk_x0_1)=$false | (sk_x0_5 = sk_x1_7)=$true
    ---  cnf 35
39: (sk_x1_4 = sk_x0_3)=$false | (sk_x0_1 = sk_x0_1)=$false | (sk_x1_7 = sk_x2_9)=$true
    ---  cnf 34
40: (sk_x0_3 = sk_x1_4)=$true | (sk_x0_1 = sk_x0_1)=$false | (sk_x0_6 = sk_x1_8)=$true
    ---  cnf 37
41: (sk_x0_3 = sk_x1_4)=$true | (sk_x0_1 = sk_x0_1)=$false | (sk_x1_8 = sk_x2_10)=$true
    ---  cnf 36

LEO-II> call-fo-atp e

*** File /tmp/atp_in written; it contains translations of the FO-like clauses in LEO-II's
search space into FOTPTP FOF syntax. Here is its content: ***

 fof(leo_II_clause_26,axiom,((~ (sk_x1_4 = sk_x0_3)) | ((~ (sk_x0_5 = sk_x2_9)) |
   (~ (sk_x0_1 = sk_x0_1))))).
 fof(leo_II_clause_28,axiom,((~ (sk_x0_6 = sk_x2_10)) | ((sk_x0_3 = sk_x1_4) |
   (~ (sk_x0_1 = sk_x0_1))))).
 fof(leo_II_clause_38,axiom,((~ (sk_x1_4 = sk_x0_3)) | ((sk_x0_5 = sk_x1_7) |
   (~ (sk_x0_1 = sk_x0_1))))).
 fof(leo_II_clause_39,axiom,((sk_x1_7 = sk_x2_9) | ((~ (sk_x1_4 = sk_x0_3)) |
   (~ (sk_x0_1 = sk_x0_1))))).
 fof(leo_II_clause_40,axiom,((sk_x0_6 = sk_x1_8) | ((sk_x0_3 = sk_x1_4) |
   (~ (sk_x0_1 = sk_x0_1))))).
 fof(leo_II_clause_41,axiom,((sk_x1_8 = sk_x2_10) | ((sk_x0_3 = sk_x1_4) |
   (~ (sk_x0_1 = sk_x0_1))))).
*** End of file /tmp/atp_in ***

*** Calling the first order ATP E on  /tmp/atp_in ***

*** Result of calling first order ATP E on  /tmp/atp_in ***

# Proof found!
# SZS status: Unsatisfiable
# Initial clauses:                     : 6
# Removed in preprocessing             : 0
# Initial clauses in saturation        : 6
# Processed clauses                    : 11
# ...of these trivial                  : 0
# ...subsumed                          : 0
# ...remaining for further processing  : 11
[...]
# Current number of unprocessed clauses: 0
# ...number of literals in the above   : 0
# Clause-clause subsumption calls (NU) : 1
# Rec. Clause-clause subsumption calls : 1
# Unit Clause-clause subsumption calls : 0
# Rewrite failures with RHS unbound    : 0

*** End of file /tmp/atp_out ***
LEO-II>
```

Fig. 5. For problem (8) we immediately generate a set of essentially first-order clauses which are here refuted by the first-order prover E.

As illustrated before, the user may also interactively call external specialist reasoners from LEO-II. Thus, in interactive mode, LEO-II is a lean but nevertheless fully equipped proof assistant for simple type theory. The main difference to systems such as HOL4, Isabelle/HOL, and OMEGA is that LEO-II's base calculus is extensional higher-order resolution, which is admittedly not very well suited for developing complex proofs interactively. However, for training students in higher-order resolution based theorem proving, for debugging, and for presentation purposes our interactive mode may turn out to be very useful.

LEO-II in particular provides support to investigate the proof state, the internal data representation (shared terms), and term index. For example, the command `termgraph-to-dot` generates a graphical representation of LEO-II's shared term data structure, which is a directed acyclic graph, in the DOT syntax [15] which can be processed by the program `dot` [15] in order obtain, for example, the ps-representation as given in Fig. 6.

With the command `analyze-index` we may request useful statistical information about our term data structure, such as the term sharing rate. We may, for instance, analyze how proof search modifies the term graph and changes the term sharing rate when applied to our running example from Fig. 2. For this, we call `analyze-index` first after loading the problem and then again after successfully proving it:

```
LEO-II> read-problem-file ../problems/SIMPLE-MATHS-5.thf
LEO-II> analyze-index
[...]
------------- The Termset Analysis -------------
[...]
Sharing rate: 8 nodes with 7 bindings
Average sharing rate:                              0.875 bindings per node
Average term size:                                 2.75
Average number of supernodes:                      2.25
Average number of supernodes (symbols):            2.66666666667
Average number of supernodes (nonprimitive terms): 1.5
Rate of term occurrences PST size / term size:     0.440298507463
Rate of symbol occurrences PST size / term size:   0.510204081633
Rate of bound occurrences PST size / term size:    0.636363636364
------------- End Termset Analysis -------------
LEO-II> prove
[...]
Eureka --- Thanks to Corina!
[...]
LEO-II (Proof Found!)> analyze-index
[...]
------------- The Termset Analysis -------------
[...]
Sharing rate: 232 nodes with 325 bindings
Average sharing rate:                              1.40086206897 bindings per node
Average term size:                                 11.0689655172
Average number of supernodes:                      7.76293103448
Average number of supernodes (symbols):            10.6060606061
Average number of supernodes (nonprimitive terms): 5.71505376344
Rate of term occurrences PST size / term size:     0.228137695104
Rate of symbol occurrences PST size / term size:   0.386256667713
Rate of bound occurrences PST size / term size:    0.504699009398
------------- End Termset Analysis -------------
```

6 Shared Term Data Structures and Term Indexing

Operations on terms in Leo-II are supported by term indexing (see also [30]). Key features of Leo-II's term indexing are the representation of terms in a perfectly shared graph structure and the indexing of various structural properties, such as the occurrence of subterms and their position.

Term sharing is widely employed in first-order theorem proving [26, 27, 31]: syntactically equal terms are represented by a single instance. For Leo-II, we have adapted this technique to the higher-order case. We use de Bruijn-notation [12] to avoid blurring of syntactical equality by α-conversion.

A shared representation of terms has multiple benefits. The most obvious is the instant identification of all occurrences of a term or subterm structure. Furthermore, it allows an equality test of syntactic structures at constant cost, which allows the pruning of structural recursion over terms early in many operations. Finally, it allows for 'tabling of term properties' (i.e., the memorization of term properties with the help of tables) at reasonable cost, as the extra effort spent on term analysis is compensated by the reusability of the results.

The indexing approach of Leo-II has a strong focus on structural aspects. It differs in this respect from the approach by Pientka [25], which is based on a discrimination tree and which allows for perfect filtering on the basis of higher order pattern unification. In contrast, we are particularly interested also in more relaxed search criteria, such as subterm occurrences or head symbols.

Equality and Occurrences The basis of Leo-II's data structure for terms is the shared representation of all occurrences of a syntactical structure by exactly one instance. This invariant is preserved by all operations on the index, such as insertion of new terms to the index. An example of a shared term representation, called term graph, is shown in Fig. 6.

Leo-II's term graph is implemented in a data structure based on hashtables. Based on the invariant of a perfectly shared representation of terms in the graph, the check for the existence of a given syntactical structure in the index is reduced to a few hashtable lookups. Our representation in particular reduces equality checks for terms in the index to a single pointer comparison. In addition, the index provides information on the structure of terms by indexing subterm occurrences. From Leo-II'S interactive interface, such information can be accessed by the user using the command `inspect-node` and `inspect-symbol`.

We exemplarily apply the command `inspect-symbol` after automatically proving our running example to the variable symbol `V_x0_1`:

```
LEO-II> read-problem-file ../problems/SIMPLE-MATHS-5.thf
LEO-II> prove
[...]
Eureka --- Thanks to Corina!
[...]
LEO-II (Proof Found!)> inspect-symbol V_x0_1
Inspecting:
  node 161: V_x0_1
Type:
  $i>($i>$o)
Structure:
  symbol V_x0_1
Parents:
 - as function term:
  node 180: V_x0_1 @ (sk_x2_4 @ V_x0_1)
  node 174: V_x0_1 @ (sk_x1_6 @ V_x0_1)
  node 164: V_x0_1 @ (sk_x1_1 @ V_x0_1)
  node 178: V_x0_1 @ (sk_x2_7 @ V_x0_1)
  node 168: V_x0_1 @ (sk_x1_3 @ V_x0_1)
 - as argument term:
  node 182: sk_x3_5 @ V_x0_1
  node 173: sk_x1_6 @ V_x0_1
  node 163: sk_x1_1 @ V_x0_1
  node 176: sk_x2_7 @ V_x0_1
  node 167: sk_x1_3 @ V_x0_1
  node 170: sk_x2_4 @ V_x0_1
 total: 11 parents
Occurs in terms indexed with role:
  node 165: (V_x0_1 @ (sk_x1_1 @ V_x0_1)) @ (sk_x1_1 @ V_x0_1)
   (in Clause:25/0 max neg)
  node 171: (V_x0_1 @ (sk_x1_3 @ V_x0_1)) @ (sk_x2_4 @ V_x0_1)
   (in Clause:33/1 max pos)
  node 177: (V_x0_1 @ (sk_x1_6 @ V_x0_1)) @ (sk_x2_7 @ V_x0_1)
   (in Clause:28/2 max pos)
  node 179: (V_x0_1 @ (sk_x2_7 @ V_x0_1)) @ (sk_x1_6 @ V_x0_1)
   (in Clause:25/2 max neg)
  node 183: (V_x0_1 @ (sk_x2_4 @ V_x0_1)) @ (sk_x3_5 @ V_x0_1)
   (in Clause:35/1 max pos)
  node 184: (V_x0_1 @ (sk_x1_3 @ V_x0_1)) @ (sk_x3_5 @ V_x0_1)
   (in Clause:25/1 max neg)
 total: 6 terms
LEO-II (Proof Found!)>
```

The index first provides information on the direct relation of `V_x0_1` to other terms in the index: `V_x0_1` occurs as function term or argument in the terms represented by the 11 parent nodes shown above. This information is maintained in cascaded hashtables and provides the necessary information to preserve the index' single instance representation. For example, when a term `sk_x1_6 @ V_x0_1` is inserted to the index in the displayed state, the existence of a node representing this term can be checked by two hashtable lookups: first all terms that have variable `V_x0_1` as argument are looked up, then a second hashtable lookup tests whether there is among these a term with function term `sk_x1_6`. Here, in fact, we already have a node in the index representing this term, namely node 173.

The single instance representation allows for indexing of term properties, such as the occurrence in clause literals or in other relevant positions: In the given proof state, the variable `V_x0_1` occurs in 6 clause literals, where three of them are literals of clause 25:

```
25: ((V_x0_1 @ (sk_x1_1 @ V_x0_1)) @ (sk_x1_1 @ V_x0_1))=$false |
    ((V_x0_1 @ (sk_x1_3 @ V_x0_1)) @ (sk_x3_5 @ V_x0_1))=$false |
    ((V_x0_1 @ (sk_x2_7 @ V_x0_1)) @ (sk_x1_6 @ V_x0_1))=$false
```

A term graph representing these three literals is shown in Fig. 6. Node 161, representing variable `V_x0_1`, is shared by the nodes 165, 184 and 179, as it is reachable from all of them. Furthermore, all occurrences within a single term are also represented by a single node. The graph shows node 161 occurring in six different positions in the three literals, both as a function term of some

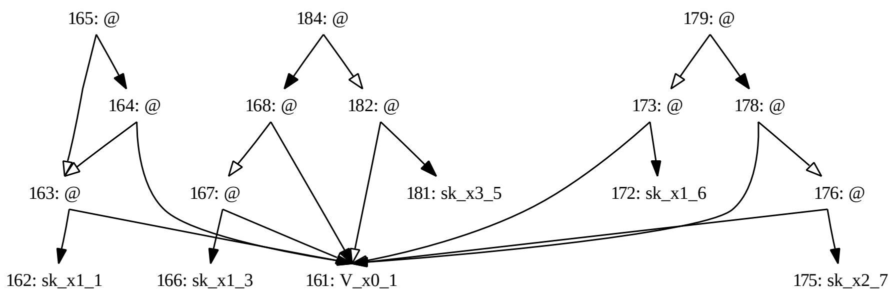

Fig. 6. A graph representation of the literals in clause 25. The number at the beginning of each node label is the node's unique identifier in the internal representation of LEO-II term data structure the term graph. Nodes labeled by `@` are application nodes where the edge with the filled arrow points to the function and the edge with the empty arrow points to the argument. Abstraction nodes are not present in this graph.

application (shown by filled arrowheads) and as argument term (shown by blank arrowheads). However, not only primitive terms such as symbols are shared, but also non-primitive terms, that is, applications and abstractions. An example is node 163 (`sk_x1_1 @ V_x0_1`), which is shared by nodes 164 (`V_x0_1 @ (sk_x1_1 @ V_x0_1)`) and 165 (`(V_x0_1 @ (sk_x1_1 @ V_x0_1)) @ (sk_x1_1 @ V_x0_1)`).

Using the Index In LEO-II, the information provided by the index is used to guide a number of operations both at term level as well as at calculus level. In addition to speeding up standard operations in the proving procedure, the indexing mechanism allows us to address problems in a different way. For example, it helps avoiding a naive, sequential checking of many clause and literal properties. Instead, the checking process is reversed and terms in the index having a particular property are first identified and then the relevant clauses are selected via hashtable lookups. Global unfolding of definition is already in LEO-II this way. Unfortunately this is not the case yet for many other important components, including simplification, rewriting, and subsumption.

7 Related Work

The integration of reasoners and reasoning strategies was pioneered in the TEAMWORK system [14], which realizes the cooperation of different reasoning strategies, and the TECHS system [13], which realizes a cooperation between a set of heterogeneous first-order theorem provers. Related is also the work of Meier [20], Hurd [18], and Meng/Paulson [21, 23]. They realize interfaces between proof assistants (OMEGA, HOL, and Isabelle/HOL) and first-order theorem provers. All these approaches pass essentially first-order clauses to first-order theorem provers after appropriate syntax transformations. The main difference to the work presented here is that LEO-II calls first-order provers from within automatic higher-order proof search.

The project LEO-II was strongly inspired by encouraging previous work on LEO and the agent based OANTS framework (cf. [5, 3, 2, 4]).

8 Concluding Remarks

The LEO-II project has been under way since October 2006. As this document illustrates the LEO-II system has since developed comparably fast with respect to both its interactive and its automatic mode. Within short time we have produced more than 12000 lines of OCAML code (partly as OCAML beginners). While still facing several 'Kinderkrankheiten' we have now entered the highly fascinating theorem prover development phase in which first theorems can already be proven automatically although some crucial features, in particular wrt. heuristic guidance layer, are still missing or have to be further investigated and developed.

References

1. The thf syntax (hotptp). `http://www.cs.miami.edu/~tptp/TPTP/Proposals/THFSyntaxBNF.html`.
2. C. Benzmüller, M. Jamnik, M. Kerber, and V. Sorge. Experiments with an Agent-Oriented Reasoning System. In *In Proc. of KI 2001*, volume 2174 of *LNAI*, pages 409–424. Springer, 2001.
3. C. Benzmüller and V. Sorge. Oants – an open approach at combining interactive and automated theorem proving. In *Proceedings of the Calculemus Symposium 2000*, St. Andrews, Schottland, August 2000. A.K.Peters.
4. C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Can a higher-order and a first-order theorem prover cooperate? In *Proc. of LPAR'05*, volume 3452 of *LNCS*, pages 415–431. Springer, 2005.
5. C. Benzmüller, V. Sorge, M. Jamnik, and M. Kerber. Combined reasoning by automated cooperation. *Journal of Applied Logic*, 2007. To appear.
6. C. Benzmüller. *Equality and Extensionality in Automated Higher-Order Theorem Proving.* PhD thesis, Universität des Saarlandes, 1999.
7. C. Benzmüller. System description: Leo – a resolution based higher-order theorem prover. In *Proc. of LPAR-05 Workshop: Empirically Successfull Automated Reasoning in Higher-Order Logic (ESHOL)*, Montego Bay, Jamaica, 2005. Avalaible from http://arxiv.org/abs/cs/0601042.
8. C. Benzmüller, C.E. Brown, and M. Kohlhase. Higher order semantics and extensionality. *Journal of Symbolic Logic*, 69:1027–1088, 2004.
9. C. Benzmüller and M. Kohlhase. System description: LEO — a higher-order theorem prover. In *Proc. of CADE-15*, volume 1421 of *LNAI*, pages 139–143. Springer, 1998.
10. M. Bishop and P. B. Andrews. Selectively instantiating definitions. In *Proc. of CADE-15*, volume 1421 of *LNAI*, pages 365–380. Springer, 1998.
11. A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

12. N.G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.
13. J. Denzinger and D. Fuchs. Cooperation of Heterogeneous Provers. In *In Proc. IJCAI-16*, pages 10–15. Morgan Kaufmann, 1999.
14. J. Denzinger and M. Fuchs. Goal oriented equational theorem proving using team work. In *In Proc. of KI-94*, volume 861 of *LNAI*. Springer, 1994.
15. E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience*, 30(11):1203–1233, 2000.
16. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, New York, NY, USA, 1993.
17. L. Henkin. Completeness in the theory of types. *Journal of Symbolic Logic*, 15:81–91, 1950.
18. J. Hurd. An LCF-style interface between HOL and first-order logic. In *Proc. of CADE-18*, volume 2392 of *LNAI*, pages 134–138. Springer, 2002.
19. M. Kerber. *On the Representation of Mathematical Concepts and their Translation into First-Order Logic*. PhD thesis, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, Germany, 1992.
20. A. Meier. TRAMP: Transformation of Machine-Found Proofs into Natural Deduction Proofs at the Assertion Level. In *In Proc. of CADE-17*, number 1831 in LNAI. Springer, 2000.
21. J. Meng and L.C. Paulson. Experiments on supporting interactive proof using resolution. In *In Proc. of IJCAR 2004*, volume 3097 of *LNCS*, pages 372–384. Springer, 2004.
22. J. Meng and L.C. Paulson. Experiments on supporting interactive proof using resolution. In David Basin and Michaël Rusinowitch, editors, *Automated Reasoning — Second International Joint Conference, IJCAR 2004*, LNAI 3097, pages 372–384. Springer, 2004.
23. J. Meng, C. Quigley, and L.C. Paulson. Automation for interactive proof: first prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.
24. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
25. B. Pientka. Higher-order substitution tree indexing. In *Proc. of ICLP*, volume 2916 of *LNCS*, pages 377–391. Springer, 2003.
26. A. Riazanov and A. Voronkov. The design and implementation of Vampire. *AICOM*, 15(2-3):91–110, jun 2002.
27. S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3):111–126, 2002.
28. J. Siekmann, C. Benzmüller, and S. Autexier. Computer supported mathematics with omega. *Journal of Applied Logic*, 4(4):533–559, 2006.
29. G. Sutcliffe and C.B. Suttner. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning*, 21(2):177–203, 1998.
30. F. Theiss and C. Benzmüller. Term indexing for the LEO-II prover. In *IWIL-6 workshop at LPAR 2006: The 6th International Workshop on the Implementation of Logics*, Pnom Penh, Cambodia, 2006.
31. C. Weidenbach, et al. Spass version 2.0. In *Proc. of CADE-18*, volume 2392 of *LNAI*, pages 275–279. Springer, 2002.
32. C. Weidenbach, B. Gaede, and G. Rock. Spass & flotter version 0.42. In *Proc. of CADE-13*, pages 141–145. Springer, 1996.