MAX-PLANCK-INSTITUT FÜR INFORMATIK

Preprints of

Proceedings of GWAI–92

Editor: Hans Jürgen Ohlbach

MPI–I–92–232 July 1992

Im Stadtwald
66123 Saarbrücken
Germany

"Das diesem Bericht zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministers für Forschung und Technologie (Betreuungskennzeichen ITS 9102) gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor."

Contents

A Model Elimination Calculus with Built-in Theories

Peter Baumgartner
Universität Koblenz
Institut für Informatik
Rheinau 3-4
5400 Koblenz

Net: peter@infko.uucp

Abstract. *The* model elimination calculus *is a linear, refutationally complete calculus for first order clause logic. We show how to extend this calculus with a framework for* theory reasoning. *Theory reasoning means to separate the knowledge of a given domain or theory and treat it by special purpose inference rules. We present two versions of theory model elimination: the one is called* total theory model elimination *(which allows e.g. to treat equality in a rigid E-resolution style), and the other is called* partial theory model elimination *(which allows e.g. to treat equality in a paramodulation style).*

1 INTRODUCTION

The *model elimination calculus* (ME calculus) has been developed already in the early days of automated theorem proving ([Lov78b]). It is a linear, refutationally complete calculus for first order clause logic. In this paper, we will show how to extend model elimination with theory reasoning.

Technically, *theory reasoning* means to relieve a calculus from explicit reasoning in some domain (e.g. equality, partial orders) by taking apart the domain knowledge and treating it by special inference rules. In an implementation, this results in a universal "foreground" reasoner that calls a specialized "background" reasoner for theory reasoning. Theory reasoning comes in two variants ([Sti85]): *total* and *partial* theory reasoning. Total theory reasoning generalizes the idea of finding complementary literals in inferences (e.g. resolution) to a semantic level. For example, in theory resolution the foreground reasoner may select from some clauses the literal set $\{a < b, b < c, c < a\}$, pass it to the background reasoner (assume that $<$ is interpreted as a strict ordering, i.e. as a transitive and irreflexive relation) which in turn should discover that this set is contradictory. Finally the theory resolvent is built as in ordinary resolution by collecting the rest literals. The problem with total theory reasoning is that in general it cannot be predicted what literals and how many variants of them constitute a contradictory set. As a solution, partial theory reasoning tries to break the "big" total steps into more managable smaller steps. In the example, the background reasoner might be passed $\{a < b, b < c\}$, compute the logical consequence $a < c$ and return it as a new subgoal, called "residue", to the foreground reasoner. In the next step, the foreground reasoner might call the background reasoner with $\{a < c, c < a\}$ again, which detects a trivial contradiction and thus concludes this chain. It is this *partial* theory reasoning we are mostly interested in.

Theory reasoning is a very general scheme and thus has many applications, among them are *reasoning with taxonomical knowledge* as in the Krypton system ([BGL85]), *equality reasoning*

as by paramodulation or E-resolution, building in *theory-unification*, and building in the axioms of the "reachability" relation in the translation of modal logic to ordinary first order logic.

The advantages of theory reasoning, when compared with the naive method of supplying the theories's axioms as clauses, are the following: for the first, the theory inference system may be specially tailored for the theory to be reasoned with; thus higher efficiency can be achieved by a clever reasoner that takes advantage of the theories' properties. For the second, a lot of computation that is not relevant for the overall proof plan is hidden in the background. Thus proofs become shorter and are more compact, leading to better readability.

Of course, theory reasoning is not new. It was introduced by M. Stickel within the general, non-linear resolution calculus ([Sti85, Sti83]). Since then the scheme was ported to many calculi. It was done for matrix methods in [MR87], for the connection method in [Bib87, Pet90], and for the connection graph calculus in ([Ohl86, Ohl87]). In ([Bau92a]) we showed that total theory reasoning is compatible to ordering restrictions.

However there are significant differences between these works and the present one: for the first, model elimination is a *linear* calculus, which roughly means that an initially chosen goal clause is stepwisely processed until the refutation is found. Being a very efficient restriction, we want to keep it in our theory calculus. However none of the theory extensions of the above calculi makes use of linear restrictions. As a consequence we also need a new completeness proof and cannot use e.g. Stickel's proof. This new proof is our main result.

Another difference is our emphasis on partial theory reasoning. The completeness of the overall calculus depends from the completeness of the background reasoner for partial theory reasoning. Except Stickel ([Sti85]), the above authors do not supply sufficient completeness preserving criteria for the background reasoner. Again, since Stickel's calculus is nonlinear, his criteria cannot be applied in our case. Below we will define a reasonable criteria that meets our demands. This criteria also captures a treatment of equality by "linear paramodulation". Since linear paramodulation is complete ([FHS89]) we obtain as a corrollary the completeness of model elimination with paramodulation.

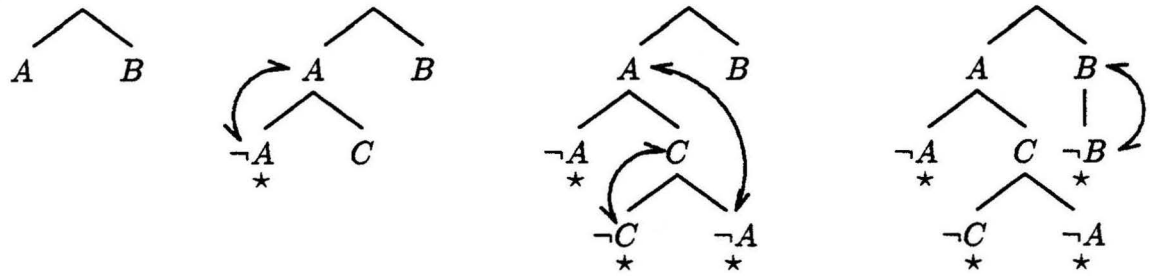
We will differ from Loveland's original ME calculus in two aspects: for the first, we have omitted some efficiency improvements such as factoring, and also we do not disallow inference steps that yield identical literals in a chain. This happens because in this paper we want to concentrate on the basic mechanisms of theory reasoning. We will adopt the efficiency improvements later. For the second we made a change in data structures: instead of *chains* we follow ([LSBB92]) and work in a tree-like setting in the tradition of analytic tableaux. This happens because we are mostly interested to implement our results in the SETHEO theorem prover(see also ([LSBB92])), which is based on that tableaux.

2 A Brief Introduction to Model Elimination

As mentioned above, we will follow the lines from [LSBB92] and define the inference rules as tree-transforming operators. Since we should not assume this format to be well-known, we will supply a brief and informal introduction.

In our format, model elimination can be seen as a restriction of semantic tableaux with unification for clauses (see [Fit90]). This restriction will be explained below. A *tableau* is, roughly, a tree whose nodes are labelled with literals in such a way that brother nodes correspond to a clause in the given clause set. A *refutation* is the construction of a tableau

where every branch is contradictory. For this construction we have to start with an initial tableau consisting of a single clause, and then repeatedly apply the inference rules *extension* and *reduction* until every branch is checked to be contradictory. Consider the unsatisfiable clause set $\{A \vee B, \neg A \vee C, \neg C \vee \neg A, \neg B\}$. and the following refutation:



1. Initial tableau with $A \vee B$
2. Extension with $\neg A \vee C$
3. Extension with $\neg C \vee \neg A$
4. Reduction
5. Extension with $\neg B$

In step 1, the tableau consisting of $A \vee B$ is built. In step 2 the branch ending in A is extended with $\neg A \vee C$ and marked with a $\star$ (such branches are called *closed*); in general, extension is only allowed if the *leaf* of the branch is complementary to a literal in the clause extended with. Step 3 is an extension step with $\neg C \vee \neg A$, and the branch ending in $\neg C$ is closed. Step 4 depicts the reduction inference: a branch, in this case $AC\neg A$, may be closed if the leaf is complementary to one of its ancestors. Finally, in step 5 the last open branch is closed by extension with $\neg B$.

Note that a closed branch contains complementary literals A and $\neg A$ and thus is unsatisfiable. If all branches are closed then the input clause set is unsatisfiable.

As usual, the ground case is lifted to the general case by taking variants of clauses, and establishing complementarism by means of a most general unifier. It should be noted that this unifier has to be applied to the entire tableau.

There is a close correspondance to linear resolution (see e.g. [CL73]): the set of open leafs corresponds to the near parent clause, extension corresponds to input resolution, and reduction corresponds to ancestor resolution. This correspondance also explains why model elimination is called "linear". If the restriction "the *leaf* (and not just any other literal in the branch) must be one of the complementary literals" is dropped, the calculus is no longer linear.

Lovelands original chain-notation ([Lov78a]) with A- and B-literals can be seen as a linear notation for our tableaux. More precisely, the open branches can bijectively be mapped to a chain, where the leafs are B-literals and the inner nodes are A-literals. If in the tableau model elimination always the "rightmost" branch is selected for extension or reduction, then there exist corresponding inference steps in chain model elimination. See ([BF92]) for a detailled comparison.

3 THEORY UNIFIERS

A *clause* is a multiset of literals written as $L_1 \vee \ldots \vee L_n$. A *theory* $\mathcal{T}$ is a satisfiable set of clauses.[1] Concerning model theory it is sufficient to consider Herbrand-interpretations only, which assign a fixed meaning to all language elements short of atoms; thus we define a

[1]This restriction is motivated by the intended application of a Herbrand-Theorem, which only holds for universally quantified theories

(Herbrand-) interpretation to be any total function from the set of ground atoms to $\{true, false\}$. A *(Herbrand-) $\mathcal{T}$-interpretation* is an interpretation satisfying the theory $\mathcal{T}$. An interpretation (resp. $\mathcal{T}$-interpretation) I *satisfies* (resp. $\mathcal{T}$-satisfies) a clause set M iff I simultaneously assigns *true* to all ground instances of the clauses in M. ($\mathcal{T}$)-(un-)satisfiability and ($\mathcal{T}$-)validity of clause sets are defined on top of this notion as usual.

As with non-theory calculi the refutations should be computed at a most general level; this is usually achieved by most general unifiers. In the presence of theories however, unifiers need not be unique, and they are replaced by a more general concept:

Definition 3.1 Let $S = \{L_1, \ldots, L_n\}$ be a literal set. S is called *$\mathcal{T}$-complementary* iff the $\forall$-quantified disjunction $\forall(\overline{L_1} \vee \ldots \vee \overline{L_n})$ is $\mathcal{T}$-valid. By abuse of language, we will say that a substitution σ is a *$\mathcal{T}$-unifier* for S iff $S\sigma$ is $\mathcal{T}$-complementary. A "partial" variant is as follows: a pair (σ, R), where σ is a substitution and R is a literal, is a *$\mathcal{T}$-residue of S* iff $S\sigma \cup \{\overline{R}\}$ is minimal $\mathcal{T}$-complementary. *(End definition)*

There is a subtle difference between the $\mathcal{T}$-complementary of a literal set and the $\mathcal{T}$-unsatisfiability of S when S is read as a set of unit clauses. These notions are the same only for ground sets. Consider, for example, a language with at least two constant symbols a and b and the "empty" theory $\emptyset$. Then $S = \{P(x), \neg P(y)\}$ is, when read as a clause set, $\emptyset$-unsatisfiable, but S is not $\emptyset$-complementary, because the clause $P(x) \vee \neg P(y)$ is not $\emptyset$-valid (because the interpretation with $I(P(a)) = false$ and $I(P(b)) = true$ is no model). However, when applying the MGU $\sigma = \{x \leftarrow y\}$ to S the resulting set $S\sigma$ is $\emptyset$-complementary.

The importance of "complementary" arises from its application in inference rules, such as resolution, which have for soundness reasons be built on top of "complementarism", but not on "unsatisfiablitity". Since we deal with theory inference rules, we had to extend the usual notion of "complementarism" to "$\mathcal{T}$-complementarism". As an example consider the theory $\mathcal{E}$ of equality. Then $S = \{P(x), y = f(y), \neg P(f(f(a)))\}$ is $\mathcal{E}$-unsatisfiable but not $\mathcal{E}$-complementary. However with the $\mathcal{E}$-unifier $\sigma = \{x \leftarrow a, y \leftarrow a\}$, $S\sigma$ is $\mathcal{E}$-complementary. In this context it might be interesting to known that our notion of theory unifier generalizes the notion of *rigid E-unifier* ([GNPS90]) to more general theories than equality (see ([Bau92a]) for a proof).

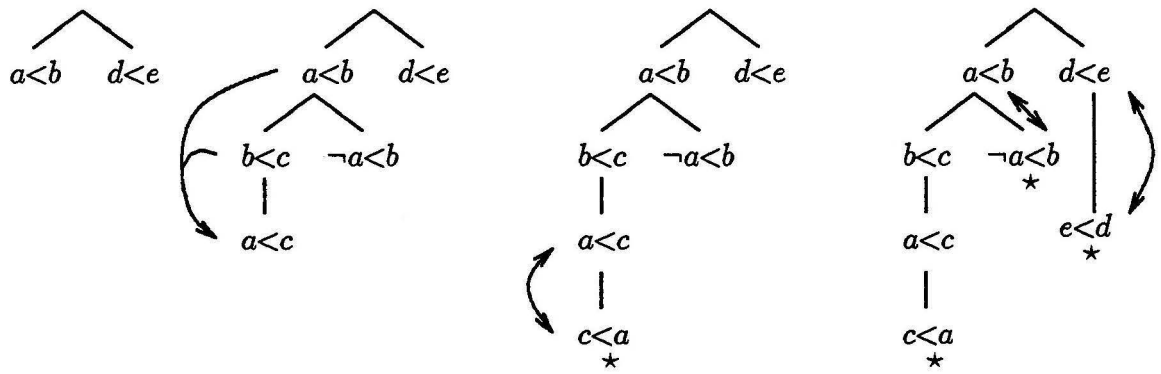
The semantics of a residue (L, σ) of S is given as follows: L is a logical consequence of $S\sigma$; operationally L is a new goal to be proved. For example let $S' = \{P(x), y = f(y)\}$. Then $(\{x \leftarrow y\}, P(f(y)))$ is an $\mathcal{E}$-residue of S', since $S'\{x \leftarrow y\} \cup \{\neg P(f(y))\} = \{P(y), y = f(y), \neg P(f(y))\}$ is minimal $\mathcal{E}$-complementary.

4 CALCULUS

Theory reasoning calculi require the computation of theory unifiers. Of course, any implementation of theory-unification in the traditional sense (see [Sie89]), e.g. AC-unification, performs a stepwise computation. Since we are interested in partial theory reasoning (see the introduction), this computation shall not remain hidden for the foreground reasoner; instead, intermediate results shall be passed back from the background reasoner to the foreground reasoner in the form of *residues*.

Let us informally describe this on the ground level with the aid of an example. Consider the clause set $S = \{a < b \vee d < e,\ b < c \vee \neg a < b,\ c < a,\ e < d\}$. S is unsatisfiable in the

theory of strict orderings ($<$ is transitive and irreflexive), and this is a theory model elimination proof:



1. Initial tableau with $a<b \vee d<e$
2. Partial extension with $b<c \vee \neg a<b$
3. Total extension with $c<a$
4. Reduction
5. Total extension with $e<d$

In step 1, the tableau consisting of $a < b \vee b < c$ is built. In step 2 the branch ending in $a < b$ is *partially extended* with the clause $b < c \vee \neg a < b$ and the *residue* $a < c$ (in this ground example no substitutions appear). The literals of the extending clauses which are relevant for the extension step, here solely $b < c$, are called *extending literals*. This step is sound, because $a < c$ is a logical consequence of its ancestors $a < b$ and $b < c$. The literals that semantically justify the inference step in this way are called the *key set* (here $\{a < b, b < c\}$). Since the branch resulting from this step is not contradictory, it is not closed (marked with a star). Besides partial extension, there exists another inference rule called *total extension*. Step 3. serves as an example: the extension with $c < a$ yields a theory-contradiction with $a < c$. Thus the branch may be closed. The ancestor literals that justify the total inference step, i.e. the contradictory set $\{c < a, a < c\}$ is also called a "key set", and $c < a$ is also called "extending literal". Step 4. is an ordinary reduction step, and step 5 is a total extension step again.

The inference steps are restricted in such a way that their key sets must consist a) of the leaf and b) possibly some other literals of the old branch, and c) of all extending literals. Condition c) implies that *all* new clauses are needed, and condition b) is the generalization of the condition "the *leaf* must be one of the complementary literals" in non-theory model elimination (section 2) to theory model elimination.

Let us now come to a formal treatment. We are concerned with ordered, labelled trees with finite branching factor and finite length. A *branch b of length k* is a sequence $b = n_0 \circ n_1 \circ \cdots \circ n_k$ of nodes, where n_0 is the root, n_{i+1} is a son of n_i and n_k is a leaf, and a tree is represented as a multiset of branches. A *literal tree* is a tree whose nodes are labelled with literals, except the root, which remains unlabelled. For our purpose it is convenient to confuse a branch $n_0 \circ n_1 \circ \cdots \circ n_k$ with the sequence of its labels $L_1 \circ \cdots \circ L_k$ or with its literal set $\{L_1, \ldots, L_k\}$. A substitution is applied to a branch by applying it to its labels in the obvious way; similarly it is applied to a tree by application to all its branches. A literal tree T' is obtained from a literal tree T by *extension with a clause $L_1 \vee \ldots \vee L_n$ at a branch b* iff

$$T' = T - \{b\} \cup \{b \circ l_i \mid i = 1 \ldots n \text{ and } l_i \text{ is labelled with } L_i\}$$

In this case we also say that T' contains a clause $L_1 \vee \ldots \vee L_k$ rooted at b.

The term "to close a branch" means to attach an additional label "$\star$" to its leaf in order to indicate that the branch is proved to be $\mathcal{T}$ -complementary. A branch is *open* iff it is not labelled in that way.

Definition 4.1 ($\mathcal{T}$**-model elimination**) Let M be a clause set and $\mathcal{T}$ be a theory. An *initial model elimination tableau for M with top clause C* is a literal tree that results from extending the empty tree (the tree that contains only the empty branch) with the the clause C.

A *model elimination tableau (ME tableau) for M* is either an initial ME tableau or a literal tree obtained by a single application of one of the following inference rules to a ME tableau T:

Partial extension step: (cf. figure 1) Let $b = L_1 \circ \ldots \circ L_{k-1} \circ L_k$ be an open branch in T . Suppose there exist new variants $C_i = K_i^1 \vee \ldots \vee K_i^{m_i}$ $(i = 1 \ldots n)$ of clauses in M. These clauses are called the *extending clauses* and the sequence $K_1^1 \circ \cdots \circ K_n^1$ is called the *extending literals*.

In order to describe the appending of the extending clauses, we define the literal tree T_n and the "actual branch to extend", b_n, recursively as follows: if $n = 0$ then $T_0 := T$ and $b_0 := b$, else T_n is the literal tree obtained from T_{n-1} by extending with the clause C_n at b_{n-1} and $b_n := b_{n-1} \circ K_n^1$.

Let $\mathcal{K}$ be a subset of the literal set of b_n with $L_k, K_1^1, \ldots, K_n^1 \in \mathcal{K}$. Borrowing a notion from ([Sti85]), $\mathcal{K}$ is called the *key set*. If there exists a $\mathcal{T}$-residue (σ, R) of $\mathcal{L}$, then partial theory extension yields the tree T_n', where T_n' is obtained from $T_n\sigma$ by extension with the unit clause R at $b_n\sigma$.

Total extension step: This is similar to "partial extension step"; instead of appending a residue, the branch is closed. Let b, C_i, T_n and b_n and $\mathcal{K}$ as in "partial extension step". If there exists a $\mathcal{T}$-unifier σ for $\mathcal{K}$, and $\mathcal{K}\sigma$ is minimal $\mathcal{T}$-complementary, then total theory extension yields the literal tree $T_n\sigma$, and the branch $b_n\sigma \in T_n\sigma$ is closed.

A total extension step with $n = 0$ is also called **reduction step.**[2] A *derivation from M with top clause C and length n* is a finite sequence of ME tableaux $T_0, T_1, \ldots, T_n$, where T_0 is an initial tableau for M with top clause C, and for $i = 1 \ldots n$ T_i is the tableau obtained from T_{i-1} by one single application of one of the above inference rules with new variants of clauses from M. If additionally in T_n every branch is closed then this derivation is called a *refutation of M*. The *partial theory model elimination calculus (PTME-calculus)* consists of the inference rules "partial extension step" and "total extension step"; the *total theory model elimination calculus (TTME-calculus)* consists of the single inference rule "total extension step" *(End definition)*

The key sets $\mathcal{K}$ play the role of a semantical justification of each step. The condition $L_k \in \mathcal{K}$ generalizes the condition "the *leaf* must be one of the complementary literals" from non-theory model elimination (section 2).

In practice it is important that the key sets and residues may be restricted to some typical, syntactical form. For example, if the theory is equality, and the calculus shall be instantiated with "paramodulation", then in the ground case the key sets $\mathcal{K}$ in partial steps are of the form $\mathcal{K} = \{L[t], t = u\}$[3] or $\mathcal{K} = \{L[t], u = t\}$, and the residues are of the form $(\emptyset, L[t \leftarrow u])$; in

[2] This notion is kept for historical reasons

[3] $L[t]$ means that the term t occurs in the literal L, $L[t \leftarrow u]$ is the literal that results from replacing one occurence of t with u

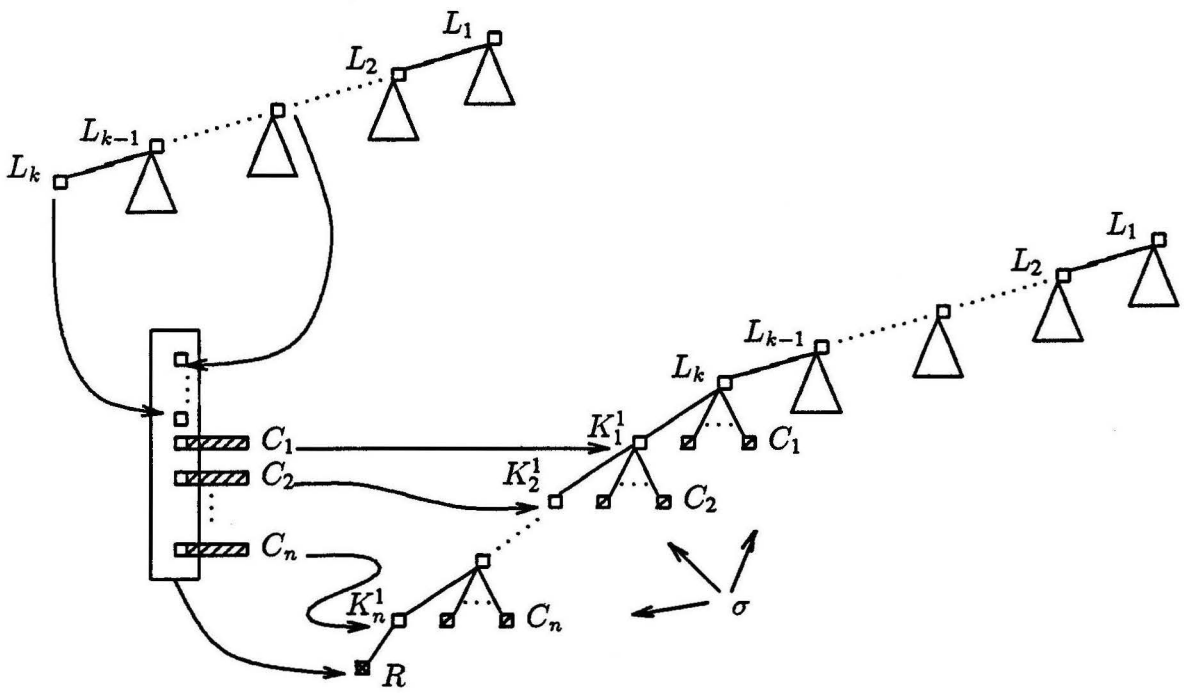


Figure 1: Partial extension step

total steps it suffices to restrict the key set to the form $\mathcal{K} = \{L, \overline{L}\}$ or $\mathcal{K} = \{\neg a = a\}$. In lifting these paramodulation steps to the first-order level, it is neccessary to allow instantiating before paramodulation (see e.g. [FHS89]). For example, if $\{P(x,x), a = b\}$ is a key set, then for completeness reasons it might be neccessary to instantiate first with $x \leftarrow f(y)$, which yields $P(f(y), f(y))$ and then paramodulate into y which finally yields $P(f(b), f(a))$. Since instantiating is not neccessary in a non-linear setting, this example also shows that inference systems for partial theory reasoning must *in principle* be designed differently than for non-linear calculi.

It should be noted that the *order* of the extending clauses is immaterial for completeness.

5 COMPLETENESS

Besides soundness, which is usually easy to prove, *(refutational) completeness* is the most important demand for a logic calculus. In order to establish such a result for partial theory reasoning, the theory reasoning component must be taken care of.

In our viewpoint, the computation of the residues in partial extension steps, and the computation of the substitutions in total extension steps should be described by a "theory calculus" with two respective inference rules: the one derives from a key set a residue, and the other derives from a key set a theory-unifier. In order to establish the completeness of the overall calculus, the theory calculus itself must be "complete". However, in order to formulate this we do not want to fix to a certain calculus; instead we will use the following abstract characterization.

Definition 5.1 Let $\mathcal{T}$ be a theory and S_1 be a ground $\mathcal{T}$-unsatisfiable literal set. Suppose that some $L_0 \in S_1$ is contained in some minimal $\mathcal{T}$-unsatisfiable subset of S_1. The theory $\mathcal{T}$ is

called *acceptable* iff there exist finite sequences

$$\begin{array}{llll} S_1, & \ldots\ S_n, & S_{n+1} & \text{and} \\ K_1, & \ldots\ K_n, & K_{n+1} & \text{and} \\ (L_1,\sigma_1), & \ldots\ (L_n,\sigma_n), & \sigma_{n+1} & \end{array}$$

such that for $i = 1 \ldots n$:

1. $L_{i-1} \in K_i$, $K_i \subseteq S_i$ and (L_i, σ_i) is a $\mathcal{T}$-residue of K_i.
2. $S_{i+1} := S_i\sigma_i \cup \{L_i\}$.
3. $L_n \in K_{n+1}$, $K_{n+1} \subseteq S_{n+1}$, and $K_{n+1}\sigma_{n+1}$ is minimal $\mathcal{T}$-complementary.
4. For every[4] $K'_i \leq K_i$ there exists a residue $(L'_i\sigma'_i)$ such that $L'_i \leq L_i$ and $\sigma'_i \leq \sigma_i$.
5. For every $K'_{n+1} \leq K_{n+1}$ there exists a $\mathcal{T}$ unifier $\sigma'_{n+1} \leq \sigma_{n+1}$ for K'_{n+1}.

An acceptable theory is called *acceptable and total* iff $n = 0$ for all L_0 and S_1 satisfying the precondition. *(End definition)*

The idea behind items 1. – 3. is to stepwisely modify an initially chosen goal L_0, leading to $L_1, L_2, \ldots, L_n$ until a contradiction is obvious (The presence of unifiable and syntactical complementary literals might be such a case, or, in equational reasoning, the presence of a literal $\neg s = t$ where s and t are unifiable). The K_is have the same meaning as the key sets in the definition of theory model elimination. In the first chain of the introductory example in the previous section, $n = 1$, $S_1 = \{a < b, b < c, c < a\}$, $L_0 = a < b$, $K_1 = \{a < b, b < c\}$, $(\sigma_1, L_1) = (\emptyset, a < c)$, $S_2 = \{a < b, b < c, c < a, a < c\}$, $K_2 = \{c < a, a < c\}$ and $\sigma_2 = \emptyset$. This strategy can also be roughly explained in linear resolution terminology: L_1 plays the role of the top clause, and the L_{i+1} are derived from the near parent L_i and a collection of far parent clauses $K_i \subseteq S_i$.

Note that although S_1 is a ground set, substitutions are involved. This is, because $\forall$-quantified variables might be introduced in the residues and thus the S_j $(j > 1)$ might no longer be ground. Items 4. is the lifting requirements for the residues, and item 5. is the lifting requirement for the concluding unifier.

Now we can turn to completeness of theory model elimination. In an attempted model elimination refutation it is essential to pick a suitable clause for the initial tableau. For example, if $S = \{A, B, \neg B\}$ then no proof can be found when the initial tableau is built from A. What we need is expressed in the completeness theorem:

Theorem 5.1 (Completeness of partial theory model elimination) *Let $\mathcal{T}$ be an acceptable theory and M be a $\mathcal{T}$-unsatisfiable clause set. Let $C \in M$ be such that C is contained in some minimal $\mathcal{T}$-unsatisfiable subset of M. Then there exists a PTME refutation of M with top clause C.*

The completeness of TTME follows as a corollary of PTME if the theory reasoner can find a $\mathcal{T}$-unifier in one step, or, more technically:

[4] $K' \leq K$ iff $\exists\delta : K'\delta = K$ and $\sigma' \leq \sigma$ iff $\exists\delta : \sigma'\delta|dom(\sigma) = \sigma$

Corollary 1 *Let M and C as in the previous theorem and $\mathcal{T}$ be an acceptable and total theory. Then there exists a TTME refutation of M with top clause C.*

The completeness proof employs the standard technique of proving first the ground case and then lifting to the variable case. Thus we apply a theory-version of the Skolem-Herbrand-Gödel theorem. Such a theorem only holds for universally quantified formula. This fact explains our restriction to *clausal* theories.

Since the main difficulties are in the ground proof, we will omitt lifting here. For the ground proof we need the following notion: a clause $C \in M$ in an $\mathcal{T}$-unsatisfiable clause set M is called *essential in M* iff $M - \{C\}$ is not $\mathcal{T}$-unsatisfiable. Note that not every $\mathcal{T}$-unsatisfiable clause set has an essential literal, e.g. $\{A, \neg A, B, \neg B\}$ is $\emptyset$-unsatisfiable but deleting any element results in a still $\emptyset$-unsatisfiable set. However every literal in a *minimal* $\mathcal{T}$-unsatisfiable set is essential.

Lemma 5.1 *Let M be a $\mathcal{T}$-unsatisfiable ground clause set, with $L, L_1 \vee \ldots \vee L_n \in M$ being essential. Define $M_i = M - \{L_1 \vee \ldots \vee L_n\} \cup \{L_i\}$ (for $i = 1 \ldots n$). Then every M_i is $\mathcal{T}$-unsatisfiable and the clauses L and L_i are essential in some M_j ($1 \leq j \leq n$).*

This lemma is needed in the proof of the following ground completeness lemma:

Lemma 5.2 *Let $\mathcal{T}$ be a theory and M be a $\mathcal{T}$-unsatisfiable ground clause set with essential clause C. Then there exists a TTME refutation of M with top clause C.*

Proof. For convenience some terminology is introduced: if we speak of "replacing a clause C_1 in a derivation by a clause C_2" we mean the derivation that results from replacing some specific occurence of C_1 by C_2, which must be a superset of $C1$, in every tableau in the derivation. By a "derivation of a clause $L_1 \vee \ldots \vee L_n$" we mean a derivation that ends in a tableau which in turn contains n open branches ending in brother leafs $L_1, \ldots, L_n$. Furthermore, if D_1 is a derivation of a clause C and D_2 is a derivation with top clause C, then by "appending D_1 and D_2" we mean the derivation that results from extending D_1 with the inferences of D_2 in order, where one specific occurence of C in D_1 takes the role of the top clause C in D_2.

Let $k(M)$ denote the number of occurences of literals in M minus the number of clauses in M ($k(M)$ is called the *excess literal parameter* in ([AB70])). Now we prove the claim by induction on $k(M)$.

In the induction base $k(M) = 0$. Then M must be a set of unit clauses, i.e. a literal set. Now set $L_0 = C$ and consider definition (5.1). The sequences defined there can be mapped to a refutation as follows: the initial tableau consists of L_0. The $\mathcal{T}$-residues $(\sigma_1, L_1), \ldots, (\sigma_n, L_n)$ of the respective sets $K_1, \ldots, K_n$ are mapped to n partial extension steps as follows: in step i choose as the key set K_i, as extending literals $K_i - \{L_{i-1}\}$, and as residue (σ_i, L_i). A final total extension step with key set K_{n+1} and extending literals $K_{n+1} - \{L_n\}$ and substitution σ_{n+1} yields the desired refutation.

To complete the induction assume now that $k(M) > 0$ and that the result holds for sets M' with $k(M') < k(M)$. We need a further case analyses.

Case 1: C is a non-unit clause of the form $C = L \vee R_1 \vee \ldots \vee R_n$. Define

$$M'_L = (M - \{C\}) \cup \{L\} \quad (1)$$

$$M'_R = (M - \{C\}) \cup \{R_1 \vee \ldots \vee R_n\} \quad (2)$$

Both M'_L and M'_R are unsatisfiable, since otherwise a model for one of them were a model for M, which contradicts the assumption that M is unsatisfiable. Find a minimal $\mathcal{T}$-unsatisfiable $M_L \subset M'_L$ that contains L. Such a set must exist, because otherwise $M'_L - \{L\} \subset M$ were $\mathcal{T}$-unsatisfiable, and with $C \notin M'_L$ it follows that C is not essential in M. Since M_L is minimal $\mathcal{T}$-unsatisfiable L is essential in M_L. Since $k(M_L) < k(M)$ we can apply the induction hypothesis and obtain a refutation D_L of M_L with top clause L. We may assume that D_L is in the following normal form: in every extension or reduction step, L does not occur as an extending clause. Such a normal form can always be achieved, since L is the top clause, and thus in every step the extending clause L can be replaced by the ancestor clause L.

By the same argumentation as for L, the clause $R_1 \vee \ldots \vee R_n$ is essential in M_R, and since $k(M_R) < k(M)$ we can apply the induction hypothesis again and obtain a refutation D_R of M_R with top clause $R_1 \vee \ldots \vee R_n$.

Now replace in D_R every occurence of the clause $R_1 \vee \ldots \vee R_n$ by $L \vee R_1 \vee \ldots \vee R_n$. Call this derivation D'_R. D'_R is derivation of several occurences of a clause L from M with top clause $L \vee R_1 \vee \ldots \vee R_n$. Now append D'_R with D_L as many times until all these occurences of the clause L are closed. Since D_L is in the above normal form, the clause L is no longer used in this final derivation. Thus we obtain the desired refutation of M.

Case 2: C is a unit clause $C = K$. Since $k(M) > 0$, M contains a non-unit clause D. We distinguish two cases. In the first (and trivial) case, D is not essential in M. Thus $M' = M - \{D\}$ is $\mathcal{T}$-unsatisfiable. But C is still essential in M', because otherwise it were not essential in M either. Since $k(M') < k(M)$ the refutation as claimed exists by the induction hypothesis.

In the other case D is essential in M. By lemma (5.1) D contains a literal L such that $M_L = (M - \{D\}) \cup \{L\}$ is $\mathcal{T}$-unsatisfiable, and K and L are essential in M_L. It holds that $k(M_L) < k(M)$. Thus by the induction hypothesis there exists a refutation D_K of M_L with top clause K. D is of the form $D = L \vee R_1 \vee \ldots \vee R_n$. Since L is essential in M_L and $k(M_L) < k(M)$ there exists by the induction hypothesis a refutation D_L of M_L with top clause L. As for D_L in case 1 above, the set D_L here can be assumed to be in the same normal form, i.e. L is not used as an extending clause in any inference step.

Let $M_R = (M - \{D\}) \cup \{R_1 \vee \ldots \vee R_n\}$. By the same argumentation as in case 1, M_R is $\mathcal{T}$-unsatisfiable and $R_1 \vee \ldots \vee R_n$ is essential in M_R, and by the induction hypothesis there exists a refutation D_R of M_R with top clause $R_1 \vee \ldots \vee R_n$.

Now we can put things together. First replace in D_K every occurence of the clause L by D. The result is a derivation D'_K of several occurences of a clause $R_1 \vee \ldots \vee R_n$ from M with top clause K. Now append D'_K with D_R as many times until all occurences of $R_1 \vee \ldots \vee R_n$ in D'_K are closed. Since $R_1 \vee \ldots \vee R_n$ may be used in D_R several times, the result is a refutation D''_K of $M \cup \{R_1 \vee \ldots \vee R_n\}$ with top clause K. In order to turn D''_K into a refutation of M first replace in D''_K every occurence of the clause $R_1 \vee \ldots \vee R_n$ by D. This results in a derivation D'''_K of several occurences of the clause L from M. In order to turn this into a refutation, append D'''_K with D_L as many times until all occurences of L are closed. Since D_L is in the above normal form, the clause L is no longer used in this final derivation. Thus we obtain the desired refutation of M. □

6 CONCLUSIONS

We have developed a partial and a total variant of the model elimination calculus for theory reasoning and proved their completeness. For this purpose we gave a sufficient completeness criterion for the theory reasoning system, but left open the question how such a system can be obtained from a given theory. This is currently being investigated ([Bau92b]). Finally I would like to thank U. Furbach for reading an earlier draft of this paper.

REFERENCES

[AB70] R. Anderson and W. Bledsoe. A linear format for resolution with merging and a new technique for establishing completeness. *J. of the ACM*, 17:525–534, 1970.

[Bau91] P. Baumgartner. A Model Elimination Calculus with Built-in Theories. Fachbericht Informatik 7/91, Universität Koblenz, 1991.

[Bau92a] P. Baumgartner. An Ordered Theory Resolution Calculus. In *Proc. LPAR '92*, 1992. (To appear).

[Bau92b] P. Baumgartner. Completion for Linear Deductions. (in preparation), 1992.

[BF92] P. Baumgartner and U. Furbach. Consolution as a Framework for Comparing Calculi. (in preparation), 1992.

[BGL85] R. Brachman, V. Gilbert, and H. Levesque. An Essential Hybrid Reasoning System: Knowledge and Symbol Level Accounts of Krypton. In *Proc. IJCAI*, 1985.

[Bib87] W. Bibel. *Automated Theorem Proving*. Vieweg, 2nd edition, 1987.

[CL73] C. Chang and R. Lee. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, 1973.

[FHS89] Ulrich Furbach, Steffen Hölldobler, and Joachim Schreiber. Horn equational theories and paramodulation. *Journal of Automated Reasoning*, 3:309–337, 1989.

[Fit90] M. Fitting. *First Order Logic and Automated Theorem Proving*. Texts and Monographs in Computer Science. Springer, 1990.

[GNPS90] J. Gallier, P. Narendran, D. Plaisted, and W. Snyder. Rigid E-unification: NP-Completeness and Applications to Equational Matings. *Information and Computation*, pages 129–195, 1990.

[Lov78a] D. Loveland. *Automated Theorem Proving - A Logical Basis*. North Holland, 1978.

[Lov78b] D. W. Loveland. Mechanical Theorem Proving by Model Elimination. *JACM*, 15(2), 1978.

[LSBB92] R. Letz, J. Schumann, S. Bayerl, and W. Bibel. SETHEO: A High<-Performace Theorem Prover. Journal of Automated Reasoning, 1992.

[MR87] N. Murray and E. Rosenthal. Theory Links: Applications to Automated Theorem Proving. *J. of Symbolic Computation*, 4:173–190, 1987.

[Ohl86] Hans Jürgen Ohlbach. The Semantic Clause Graph Procedure – A First Overview. In *Proc GWAI '86*, pages 218–229. Springer, 1986. Informatik Fachberichte 124.

[Ohl87] Hans Jürgen Ohlbach. Link Inheritance in Abstract Clause Graphs. *Journal of Automated Reasoning*, 3(1):1–34. 1987.

[Pet90] U. Petermann. Towards a connection procedure with built in theories. In *JELIA 90*. European Workshop on Logic in AI, Springer, LNCS, 1990.

[Sie89] Jörg H. Siekmann. Unification Theory. *Journal of Symbolic Computation*, 7(1):207–274, January 1989.

[Sti83] M.E. Stickel. Theory Resolution: Building in Nonequational Theories. SRI International Research Report Technical Note 286, Artificial Intelligence Center, 1983.

[Sti85] M. E. Stickel. Automated deduction by theory resolution. *Journal of Automated Reasoning*, pages 333–356, 1985.

A New Sorted Logic

Christoph Weidenbach[1]
Max-Planck-Institut für Informatik
Im Stadtwald
D-6600 Saarbrücken 11, Germany
email: weidenb@mpi-sb.mpg.de

Abstract: We present a sound and complete calculus for an expressive sorted first-order logic. Sorts are extended to the semantic and pragmatic use of unary predicates. A sort may denote an empty set and the sort structure can be created by making use of the full first-order language. Technically spoken, we allow sort declarations to be used in the same way than ordinary atoms. Therefore we can compile every first-order logic formula into our logic.

The extended expressivity implies an extended sorted inference machine. We present a new unification algorithm and show that the declarations the unification algorithm is built on have to be changed dynamically during the deduction process. Deductions in the resulting resolution calculus are very efficient compared to deductions in the unsorted resolution calculus. The approach is a conservative extension of the known sorted approaches, as it simplifies to the known sorted calculi if we apply the calculus to the much more restricted input formulas of these calculi.

Keywords: Automated Reasoning, Sorted Reasoning, Sorted Unification

1 Introduction

It is widely accepted that the introduction of "sorts", e.g. see [11, 8, 13, 3, 4, 1] in first-order logic results in a more efficient resolution calculus and a more natural representation of problems. Although the second statement depends on the personal taste, there are many examples in the literature, where a sorted formalization of a problem leads to shorter proofs and a less branching search space. In all of the approaches mentioned above, sorts correspond to unary predicates in unsorted first-order logic. For example in the logic of [13] the sort "declaration" $f(x_S) \leq S$ is logically equivalent to the clause $S(x) \Rightarrow S(f(x))$.

[1]This research was supported by the ESPRIT project MEDLAR (3125) of the European Community

This simple example shows the advantages of a sorted formalization: less literals and less resolution possibilities. In particular, for this simple example there are infinitely many (self) resolution possibilities in the unsorted version, whereas there are no possible resolution steps in the sorted version.

The logic presented in this paper follows the lines of [11, 8, 13]. It extends the work of Schmidt-Schauß [8] in three directions

- We allow conditioned declarations, i.e. declarations can be used as ordinary atoms, e.g. the conditioned declaration $i(x_P, y_\top) \leqslant P \Rightarrow y_\top \leqslant P$ expresses that if x is a theorem and y a proposition and x implies y, then y is a theorem. The sort P denotes the set of all theorems and the function i denotes implication (see Example 2.4).

- Sorts may denote empty sets, i.e. we also have to consider interpretations where the empty set is assigned to a sort.

- We have a fixed sort symbol $\top$ which denotes the topsort "Any".

Our way of reasoning is the same as in [11, 8]. We choose a set of declarations which is fixed in [11, 8] from the beginning but dynamic in our approach. We exploit the declarations in a sorted unification algorithm and modify standard deduction rules (e.g. resolution) by considering sorted unifiers only.

Compared to our own work [13] we have simplified the semantics by considering total functions only and we have replaced the unsorted unification algorithm by a sorted version.

The difference between our approach and the work of [3, 4, 1] is that we do not separate formulas containing declarations only from the rest of the formulas. This allows for a more efficient calculus, because we incorporate all declarations in sorted reasoning, whereas [3, 4, 1] do not consider declarations occurring in the formula part (in fact [4] does not allow for such declarations) in the sorted reasoning process. Therefore they need more and less restrictive inference rules. Besides they only give algorithms for simple or elementary [8] sort structures, whereas our approach applies to arbitrary sort structures.

Because of the generality of our approach, we are able to compile every formula of first-order logic into a formula of our sorted logic, i.e. in general we get a smaller set of formulas (but never larger) and are able to apply our much more efficient calculus. In subsection 2.3 we will give the foundations of this compilation.

We have tried to keep the theory in this paper simple. Technical lemmas and proofs have been skipped. They can be found in [12]. Instead, we provide key examples in order to justify our results. The paper now first explains the syntax and semantics of the logic and relates the logic to unsorted first-order logic. Then we introduce our sorted unification algorithm and the rules of the resolution calculus. We end with a solution to an example (see Example 2.4) and some conclusions.

2 The Logic $\mathcal{L}_S$

2.1 Syntax

A signature $\Sigma := (\mathbf{S}, \mathbf{V}, \mathbf{F}, \mathbf{P})$ contains a set of sort symbols in addition to the usual sets $\mathbf{V}$ of variable symbols, $\mathbf{F}$ of function symbols and $\mathbf{P}$ of predicate symbols. The fixed sort $\top$ is always in $\mathbf{S}$ and the 2-place predicate $\leqslant$ is always in $\mathbf{P}$. Variables are indexed with their sort, e.g. x_S, y_W.

Terms and atoms are built in the usual unsorted way with the addition that if t is a term and S a sort symbol, then $t \leqslant S$ is an atom. Atoms of the form $t \leqslant S$ are called *declarations*. Atoms of the form $t \leqslant \top$ are forbidden, because they are not useful and complicate the theory. From atoms we construct literals and formulas using the logical connectives. A clause is a set of literals which is interpreted as the universal closure of the disjunction of the literals.

The set of all terms is denoted by $\mathbf{T}_\Sigma$. Substitutions are total functions $\sigma: \mathbf{V} \to \mathbf{T}_\Sigma$ such that the set $DOM(\sigma) := \{x_S \mid \sigma(x_S) \neq x_S\}$ is finite. The application of substitutions can be extended to terms, formulas, and clauses in the usual way. With $COD(\sigma)$ we denote the codomain of σ, $COD(\sigma) := \sigma(DOM(\sigma))$. Finally, we assume that there is a function V which maps terms, formulas and sets of such objects to their variables and a function *Sorts* which maps terms, formulas and sets of such objects to the sorts assigned to the variables occurring in these objects.

2.2 Semantics

In order to define the semantics for $\mathcal{L}_S$, we have to assign a set of objects from the non-empty universe to every sort symbol and a total function to every function symbol. $\leqslant$ is interpreted as the membership relation and we interpret predicates and the logical connectives as in unsorted first-order logic.

Let Σ be a signature. An *interpretation* $\Im$ consists of a non-empty carrier set $\mathbf{A}$, a total function $f_\Im: \mathbf{A}^n \to \mathbf{A}$ for every function symbol $f \in \mathbf{F}_n$, a set $S_\Im \subseteq \mathbf{A}$ for every sort S, and $\top_\Im := \mathbf{A}$. The interpretation $\Im$ assigns an element of $S_\Im$ to every variable x_S. An interpretation $\Im\{x_S/a\}$ is like $\Im$ except that it maps x_S to a, if $a \in S_\Im$. Furthermore, $\Im$ assigns an n-ary relation $P_\Im \subseteq \mathbf{A}^n$ to every n-place predicate symbol P and the membership relation to $\leqslant$, such that for every formula $\mathcal{F}$, $t \in \mathbf{T}_\Sigma$, $S \in \mathbf{S}$, and $P \in \mathbf{P}_n$:

- $\Im \models P(t_1, \ldots, t_n)$ iff $(\Im_h(t_1), \ldots, \Im_h(t_n)) \in P_\Im$.
- $\Im \models t \leqslant S$ iff $\Im_h(t) \in S_\Im$
- $\Im \models \forall x_S\, \mathcal{F}$ iff for all $a \in S_\Im$, $\Im\{x_S/a\} \models \mathcal{F}$
- $\Im \models \exists x_S\, \mathcal{F}$ iff there exists an $a \in S_\Im$, $\Im\{x_S/a\} \models \mathcal{F}$
- The logical connectives are interpreted in the usual way.

2.3 Relativization

In this subsection we show how to transform formulas of $\mathcal{L}_\mathcal{S}$ into unsorted first-order logic. We can use the equivalences

$$\forall x_S \, \mathcal{F} \Leftrightarrow \forall x_\top \, (x_\top \leqslant S \Rightarrow \mathcal{F})$$
$$\exists x_S \, \mathcal{F} \Leftrightarrow \exists x_\top \, (x_\top \leqslant S \ \wedge \ \mathcal{F})$$

in order to eliminate variables of sorts different from $\top$, then erase the sorts of the variables and transform each atom $t \leqslant S$ into $S(t)$. The unsorted first-order formula obtained is logically equivalent to the sorted formula [12].

If we use these steps in reverse order, we can also translate every unsorted first order logic formula in our logic. For arbitrary formulas this process might not be useful, but it works fine for clauses.

2.4 Example: Condensed Detachment

Condensed detachment is an inference rule that combines modus ponens and instantiation [7]. The general technique is to code various problems into the term structure of first-order logic. Thus we only need one predicate symbol P expressing "is a theorem". Solving examples of this form with an automated theorem prover can be arbitrarily hard [6]. The example presented in the following is from the two-valued sentential calculus where the 2-place function i denotes implication and the 1-place function n negation. We start with an unsorted formalization.

(1) $\forall x, y \, (P(i(x,y)) \ \wedge \ P(x) \Rightarrow P(y))$
(2) $\forall x, y, z \, P(i(i(x,y), i(i(y,z), i(x,z))))$
(3) $\forall x \, P(i(i(n(x), x), x))$
(4) $\forall x, y \, P(i(x, i(n(x), y)))$
(5) $\forall x \, P(i(x,x))$

We will prove that $((1) \ \wedge \ (2) \ \wedge \ (3) \ \wedge \ (4)) \Rightarrow (5)$. But first we compile the unsorted formalization into our sorted logic. As there is only one 1-place predicate symbol P, we select P and turn it into a sort, attach the sort $\top$ (any) to all variables and rewrite atoms of the form $P(t)$ to $t \leqslant P$.

(1) $\forall x_\top, y_\top \, (i(x_\top, y_\top)) \leqslant P \ \wedge \ x_\top \leqslant P) \Rightarrow y_\top \leqslant P$
(2) $\forall x_\top, y_\top, z_\top \, i(i(x_\top, y_\top), i(i(y_\top, z_\top), i(x_\top, z_\top))) \leqslant P$
(3) $\forall x_\top \, i(i(n(x_\top), x_\top), x_\top) \leqslant P$
(4) $\forall x_\top, y_\top \, i(x_\top, i(n(x_\top), y_\top)) \leqslant P$
(5) $\forall x_\top \, i(x,x) \leqslant P$

So far the changes have not affected the structure of the formulas, only the syntax. The only candidate for applying the relativization backwards is the literal $x_\top \not\leqslant P$ in the first clause. As we have $P \sqsubseteq \top$ ($\sqsubseteq$ denotes the subsort relation, P is a subsort of $\top$ by definition of $\top$. In general the subsort relation is decided with respect to a set of declarations, see Section 3.1. The compilation process can be automatized [12].), we change the sort of the variable $x_\top$ into x_P, delete the literal, negate formula (5), compute clause normal form (see [13]) and finally result in the compiled clause set

(1) $\{i(x_P, y_\top)) \not\leqslant P, y_\top \leqslant P\}$
(2) $\{i(i(x_\top, y_\top), i(i(y_\top, z_\top), i(x_\top, z_\top))) \leqslant P\}$
(3) $\{i(i(n(x_\top), x_\top), x_\top) \leqslant P\}$
(4) $\{i(x_\top, i(n(x_\top), y_\top)) \leqslant P\}$
(5) $\{i(a, a) \not\leqslant P\}$

It seams that the effect of the compilation is quiet small (in fact for other examples we can save more literals), because we have saved one literal only. But we will show that the proof using the compiled clause set is unique whereas the proof using the original unsorted clause set requires search in an infinite search space.

Note that the problem cannot be represented as sort information in any other known sorted logic [11, 8, 3, 4, 1], i.e. the occurring declarations can not be processed by their sorted inference mechanisms. Therefore applying these sorted calculi to the example is like applying the unsorted resolution calculus.

3 The Calculus $\mathcal{CL}_S$

We will present a resolution based calculus for $\mathcal{L}_S$. In [12] we have shown how to transform sentences of $\mathcal{L}_S$ into clauses. So our starting point here is a set of clauses CS.

The difference between a sorted and an unsorted calculus is the different processing of unary predicates. The corresponding atoms are either viewed as declarations or the name of the predicate is attached as a sort to a variable. For example, the unsorted clause $\{\neg S(x), P(x, a)\}$ can be compiled to the sorted clause $\{P(x_S, a)\}$. If we want to instantiate a sorted variable, we have to check the well sortedness of the instantiation. Therefore we need a different (sorted) unification algorithm. For example, in order to perform a resolution step between the literals $P(x_S, a)$ and $\neg P(a, a)$ we must guarantee that a has sort S. The declaration $a \leqslant S$ would provide the necessary information. Thus the unification algorithm checks well sortedness of assignments to variables with respect to a set $\mathbf{L}_\leqslant$ of declarations.

In our approach the interesting question is which declarations we must consider during unification. Note that declarations may not only occur in unit clauses (e.g. $\{a \leqslant S\}$) but also mixed with other literals (e.g. $\{a \leqslant S, P(a, a)\}$ or $\{a \leqslant S, a \leqslant T\}$).

In the calculi of [11, 8], declarations occurred only in the declaration part of the logic. This corresponds to unit clauses containing a declaration as their literal in $\mathcal{L}_S$. We will see that it is not sufficient to only consider declarations occurring in unit clauses during unification. Instead, we must also consider declarations occurring in non unit clauses in order to obtain a complete calculus. The following example demonstrates this fact.

Consider the clause set $CS = \{\{P(x_A)\}, \{P(x_B)\}, \{\neg P(a)\}, \{\neg P(b)\}, \{a \leqslant A, b \leqslant B\}\}$. If we only take unit declarations into account, we have $\mathbf{L}_\leqslant = \emptyset$ and therefore all sorts are considered to be empty. There is no well sorted resolution step, but CS is unsatisfiable.

We will now develop the necessary notions for the presentation of our unification algorithm. The example shows that we have to consider declarations which occur in non unit clauses, i.e. declarations which are "conditioned". Thus we introduce the notion of a *conditioned declaration* (*conditioned term*, *conditioned substitution*) which is a pair $(t \leqslant S, C)$ $((t, C), (\sigma, C))$ where C is a set of literals.

3.1 The Unification Algorithm

Definition 3.1 (Conditioned Well Sorted Terms) Let $\mathbf{L}_{\leqslant}$ be a set of conditioned declarations. Then the set of conditioned well sorted terms (we use the abbreviation "cws." for conditioned well sorted) $\mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$ of sort S is recursively defined by:

- for every variable $x_S \in \mathbf{V}$, $(x_S, \emptyset) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$
- for every conditioned declaration $(t \leqslant S, C) \in \mathbf{L}_{\leqslant}$, $(t, C) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$
- for every conditioned well sorted term $(t, C) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$ $(S \neq \top)$, substitution $\sigma := \{x_{S_1} \mapsto t_1, \ldots, x_{S_n} \mapsto t_n\}$, with $(t_i, C_i) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S_i}$, $(\bigcup C_i) \subseteq D$ for some finite set of literals D, $(\sigma(t), \sigma(C) \cup D) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$
- for every term t we have $(t, \emptyset) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},\top}$

We define $\mathbf{T}_{\mathbf{L}_{\leqslant},S} := \{(t, \emptyset) \mid (t, \emptyset) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}\}$. With $\mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},gr,S}$ we denote the set of cws. ground terms of sort S. Obviously $\mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},gr,S} = \{(t, C) \mid (t, C) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$ and t is ground $\}$. Thus we always have $\mathbf{T}_{\mathbf{L}_{\leqslant},gr,\top} = \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},gr,\top} \neq \emptyset$, $\mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},gr,S} \subseteq \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$, and $\mathbf{T}_{\mathbf{L}_{\leqslant},S} \subseteq \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$.

Often we are not interested in the condition part of a conditioned object. We say that $t \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$, if there is a set of literals C such that $(t, C) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S}$. Similarly we say that a declaration $(t \leqslant S) \in \mathbf{L}_{\leqslant}$, if there is a set of literals C such that $(t \leqslant S, C) \in \mathbf{L}_{\leqslant}$ and (if not necessary) we do not mention the conditioned part of a conditioned substitution.

A cws. substitution $\sigma^{\mathbf{c}} = (\sigma, C)$ (with respect to a set of declarations $\mathbf{L}_{\leqslant}$) consists of a substitution σ and a finite set of literals C such that for every $x_{S_i} \in DOM(\sigma)$, $(\sigma(x_{S_i}), C_i) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S_i}$ and $(\bigcup C_i) \subseteq C$.

For the unification algorithm it is useful to define a binary relation $\sqsubseteq$ which denotes the subsort relationship. If S and T are sorts, we define $S \sqsubseteq T$ iff there exists a variable x_S with $x_S \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},T}$.

We use the standard notations [9, 8] for the presentation of our unification algorithm: a *unification problem* Γ is a set of equations $\Gamma = \{t_1 = s_1, \ldots, t_n = s_n\}$. A substitution σ *solves* a unification problem $\Gamma = \{t_1 = s_1, \ldots, t_n = s_n\}$, iff $\sigma(t_1) = \sigma(s_1), \ldots, \sigma(t_n) = \sigma(s_n)$. A unification problem Γ is called *solved*, iff $\Gamma = \{x_{S_1} = t_1, \ldots, x_{S_n} = t_n\}$, each x_{S_i} is a variable, $x_{S_i} \notin V(t_j)$, and $x_{S_i} \neq x_{S_j}$ for every i and j and $t_i \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leqslant},S_i}$.

Algorithm 3.2 (The Sorted Unification Algorithm) The input of the algorithm is a unification problem Γ which is changed by the following thirteen rules until it is solved or the problem is found to be unsolvable:

(1) Tautology
$$\frac{\{x_S = x_S\} \cup \Gamma}{\Gamma}$$

(2) Decomposition
$$\frac{\{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\} \cup \Gamma}{\{t_1 = s_1, \ldots, t_n = s_n\} \cup \Gamma}$$

(3) Application
$$\frac{\{x_S = t\} \cup \Gamma}{\{x_S = t\} \cup \{x_S \mapsto t\}\Gamma}$$

if x_S is a variable, t a non-variable term and $x_S \notin V(t), x_S \in V(\Gamma)$

(4) Orientation
$$\frac{\{t = x_S\} \cup \Gamma}{\{x_S = t\} \cup \Gamma}$$

if x_S is a variable and t a non-variable term

(5) Clash
$$\frac{\{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)\} \cup \Gamma}{\text{STOP.FAIL}}$$

if $f \neq g$

(6) Cycle
$$\frac{\{x_S = t\} \cup \Gamma}{\text{STOP.FAIL}}$$

if $x_S \in V(t)$

(7) Empty Sort
$$\frac{\Gamma}{\text{STOP.FAIL}}$$

if there exists a variable $x_S \in V(\Gamma)$ such that $\mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leq}, gr, S} = \emptyset$

(8) Sorted Fail VV
$$\frac{\{x_S = y_T\} \cup \Gamma}{\text{STOP.FAIL}}$$

if $S \neq \top$, $T \neq \top$, $y_T \notin \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leq}, S}$ and $x_S \notin \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leq}, T}$ and there is no common subsort of S and T and there are no conditioned declarations $(f(s_1, \ldots, s_n) \leq S') \in \mathbf{L}_{\leq}$, $S' \sqsubseteq S$ and $(f(t_1, \ldots, t_n) \leq T') \in \mathbf{L}_{\leq}$, $T' \sqsubseteq T$

(9) Sorted Fail VT
$$\frac{\{x_S = f(t_1, \ldots, t_n)\} \cup \Gamma}{\text{STOP.FAIL}}$$

if $S \neq \top$, $x_S \notin V(f(t_1, \ldots, t_n))$, $f(t_1, \ldots, t_n) \notin \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_{\leq}, S}$ and there is no declaration $(f(s_1, \ldots, s_n) \leq S') \in \mathbf{L}_{\leq}$, $S' \sqsubseteq S$

(10) Weakening VT
$$\frac{\{x_S = f(t_1, \ldots, t_n)\} \cup \Gamma}{\{x_S = f(t_1, \ldots, t_n)\} \cup \{t_1 = s_1, \ldots, t_n = s_n\} \cup \Gamma}$$

if $S \neq \top$, $x_S \notin V(f(t_1, \ldots, t_n))$, $f(t_1, \ldots, t_n) \notin \mathbf{T}_{\mathbf{L}_{\leq}, S}$ and there is a conditioned declaration $(f(s_1, \ldots, s_n) \leq S') \in \mathbf{L}_{\leq}$, $S' \sqsubseteq S$

(11)	Subsort	$\frac{\{x_S = y_T\} \cup \Gamma}{\{y_T = x_S\} \cup \Gamma}$ if $x_S \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_\leqslant, T}$ and $y_T \notin \mathbf{T}_{\mathbf{L}_\leqslant, S}$
(12)	Common Subsort	$\frac{\{x_S = y_T\} \cup \Gamma}{\{x_S = z_V\} \cup \{y_T = z_V\} \cup \Gamma}$ if $S \neq \top$, $T \neq \top$, $x_S \notin \mathbf{T}_{\mathbf{L}_\leqslant, T}$ and $y_T \notin \mathbf{T}_{\mathbf{L}_\leqslant, S}$ and V is a maximal sort with $V \sqsubseteq S$ and $V \sqsubseteq T$
(13)	Weakening VV	$\frac{\{x_S = y_T\} \cup \Gamma}{\{x_S = f(\vec{s})\} \cup \{y_T = f(\vec{t})\} \cup \{s_1 = t_1, \ldots, s_n = t_n\} \cup \Gamma}$ if $S \neq \top$, $T \neq \top$, $y_T \notin \mathbf{T}_{\mathbf{L}_\leqslant, S}$ and $x_S \notin \mathbf{T}_{\mathbf{L}_\leqslant, T}$ and there are conditioned declarations $(f(s_1, \ldots, s_n) \leqslant S') \in \mathbf{L}_\leqslant$, $S' \sqsubseteq S$ and $(f(t_1, \ldots, t_n) \leqslant T') \in \mathbf{L}_\leqslant$, $T' \sqsubseteq T$ and $S' \not\sqsubseteq T'$ and $T' \not\sqsubseteq S'$ where $\vec{s} = s_1, \ldots, s_n$ and $\vec{t} = t_1, \ldots, t_n$

Every declaration $t \leqslant T$ taken from $\mathbf{L}_\leqslant$ must be completely renamed with new variables before it is used in a unification step. The rules (1)-(6) implement the unsorted Robinson unification algorithm.

In order to compute a cws. substitution from a solved unification problem, we have to do the following. Let $\Gamma = \{x_{S_1} = t_1, \ldots, x_{S_n} = t_n\}$ be the solved unification problem, then $\sigma := \{x_{S_1} \mapsto t_1, \ldots, x_{S_n} \mapsto t_n\}$ is the corresponding unconditioned unifier. $\sigma^{\mathbf{c}} := (\sigma, C_1 \cup \ldots \cup C_n)$ is a most general cws. unifier, if we have $(t_i, C_i) \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_\leqslant, S_i}$. Therefore a solved unification problem may lead to several (but only finitely many if $\mathbf{L}_\leqslant$ is finite) cws. mgu's.

The presented algorithm is a complete and correct unification algorithm. Complete means that for every cws. ground substitution $\tau^{\mathbf{c}} = (\tau, C)$ solving a unification problem Γ (where $DOM(\tau)$ is restricted to $V(\Gamma)$)the unification algorithm computes a cws. substitution $\sigma^{\mathbf{c}}$ such that there exists a cws. ground substitution $\lambda^{\mathbf{c}}$ with $\lambda^{\mathbf{c}}(\sigma^{\mathbf{c}}) = \tau^{\mathbf{c}}$. As the presented unification algorithm is an extension of the unification algorithm of Schmidt-Schauß [8] it may produce infinitely many unifiers and the problem whether two terms can be unified is undecidable. Questions of the form $t \in \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_\leqslant, S}$, $\mathbf{T}^{\mathbf{c}}_{\mathbf{L}_\leqslant, gr, S} = \emptyset$ are decidable in quasi-linear time [12].

3.2 The Inference Rules

In order to apply the inference rules, we need a function *Empty* which computes from a set of sorts a set of sets of literals (conditions), such that each set of literals guarantees the sorts to be non empty. We assume that *Empty* computes a minimal, finite set of conditions with respect to set inclusion. This can be done by constructing appropriate ground terms according Definition 3.1.

Definition 3.3 (Factorization Rule) Let $C = \{L_1, \ldots, L_n\}$ be a clause, $\{L_i, L_j\} \subseteq C$ $(i \neq j)$, and $\sigma^{\mathbf{c}} = (\sigma, D)$ a cws. mgu of $\{L_i, L_j\}$, i.e. $\sigma(L_i) = \sigma(L_j)$, $E \in Empty(Sorts(C) \setminus Sorts(\sigma(C)))$, then

$$\sigma(C) \cup D \cup E$$

is called a *factor* of C.

Definition 3.4 (Resolution Rule) Let C_1 and C_2 be two clauses with no variables in common, $L \in C_1$ and $K \in C_2$. We define $C_1' := C_1 \setminus \{L\}$ and $C_2' := C_2 \setminus \{K\}$. If there exists a cws. mgu $\sigma^c = (\sigma, D)$ of L and K such that $\sigma(L)$ and $\sigma(K)$ become two complementary literals, $E \in Empty(Sorts(C) \setminus Sorts(\sigma(C_1' \cup C_2')))$, then

$$\sigma(C_1' \cup C_2') \cup D \cup E$$

is a *resolvent* of C_1 and C_2.

The correctness of the rules follows immediately from their form and the correctness of the unification algorithm [12].

Now the remaining question is which declarations we have to take into account for the unification algorithm in order to preserve completeness. The general idea is to reduce the number of declarations, because this restricts the applicability of inference rules, whence the search space. The previous example shows that we have to consider at least one of the literals $\{a \leqslant A,\ b \leqslant B\}$. In general we must choose one declaration from each clause containing declarations only. We call such clauses *declaration clauses*. Choosing the declarations has to be done dynamically during the derivation process, i.e. if we have derived a new declaration clause, we have to extend the well sorted terms using one of these declarations.

Theorem 3.5 (Completeness Theorem) Let CS be a clause set. We choose $\mathbf{L}_{\leqslant} := \{(t_i \leqslant S_i, C_i')\}$ such that for each declaration clause $C_i \in CS$ we choose exactly one declaration $t_i \leqslant S_i$ with $C_i = \{t_i \leqslant S_i\} \cup C_i'$.

If CS is unsatisfiable, there exists a derivation of the empty clause using resolution and factorization. The set $\mathbf{L}_{\leqslant}$ must be updated every time a new declaration clause is derived.

Again we consider the clause set of the previous example. $CS = \{\{P(x_A)\}, \{P(x_B)\}, \{\neg P(a)\}, \{\neg P(b)\}, \{a \leqslant A, b \leqslant B\}\}$. If we choose $\mathbf{L}_{\leqslant} := \{(a \leqslant A, \{b \leqslant B\})\}$ then there is only one possible resolution step between $\{P(x_A)\}$ and $\{\neg P(a)\}$ with cws. mgu $(\{x_A \mapsto a\}, \{b \leqslant B\})$ and $Empty(\{A\}) = \{\{b \leqslant B\}\}$ resulting in $\{b \leqslant B\}$. This clause subsumes the clause $\{a \leqslant A, b \leqslant B\}$ and we obtain the new clause set $\{\{P(x_A)\}, \{P(x_B)\}, \{\neg P(a)\}, \{\neg P(b)\}, \{b \leqslant B\}\}$. We have to change $\mathbf{L}_{\leqslant}$ into $\mathbf{L}_{\leqslant} := \{(b \leqslant B, \emptyset)\}$. Again only one resolution step between $\{P(x_B)\}$ and $\{\neg P(b)\}$ with cws. mgu $(\{x_B \mapsto b\}, \emptyset)$ and $Empty(\{B\}) = \{\emptyset\}$ is possible which yields the empty clause.

3.3 Solving the Example

We solve the problem using a set of support strategy. The problem is not a hard problem, e.g. OTTER [5] can solve the problem in less than one second. But we will show that in the sorted formalization the problem can be solved without any search. Here is the compiled clause set

(1) $\{i(x_P, y_\mathsf{T}))\not\in P, y_\mathsf{T}{<}P\}$
(2) $\{i(i(x_\mathsf{T}, y_\mathsf{T}), i(i(y_\mathsf{T}, z_\mathsf{T}), i(x_\mathsf{T}, z_\mathsf{T})))<P\}$
(3) $\{i(i(n(x_\mathsf{T}), x_\mathsf{T}), x_\mathsf{T})<P\}$
(4) $\{i(x_\mathsf{T}, i(n(x_\mathsf{T}), y_\mathsf{T}))<P\}$
(5) $\{i(a, a)\not\in P\}$

The initial set of support consists of the clause $\{i(a,a)\not\in P\}$ only. Following Theorem 3.5 we choose the set of declarations $\mathbf{L}_< = \{(2)1, (3)1, (4)1\}$ (We use the format *(clause)literal* to refer to literals in clauses). As no conditioned declarations need to be considered, we have $\mathbf{T}_{\mathbf{L}_<,P} = \mathbf{T}^{\mathbf{c}}_{\mathbf{L}_<,P}$ and thus the additional notations needed to represent conditioned objects are skipped. The only applicable inference step is resolution with (1)2, because unification with one of the literals (2)1, (3)1 or (4)1 results in a function symbol clash. The corresponding unification problem is

$$\Gamma_0 \quad = \quad \{y_\mathsf{T} = i(a, a)\}$$

which is still in solved form, as $i(a,a)\in\mathbf{T}_{\mathbf{L}_<,\mathsf{T}}$. Thus we add the new clause

(6) $\{i(x_P, i(a, a))\not\in P\}$

to the set of support. The new clause cannot be resolved with clauses (2) and (4) because of a function symbol clash and unification with the literal (3)1 results in the unification problem

$$\Gamma_1 \quad = \quad \{x_P = i(n(x_\mathsf{T}), x_\mathsf{T}), x_\mathsf{T} = i(a, a)\}$$

which cannot be well sorted solved, because $i(n(i(a,a)), i(a,a))\not\in\mathbf{T}_{\mathbf{L}_<,P}$ and the term $i(n(i(a,a)), i(a,a))$ cannot be further weakened. Thus again the only possible step is resolution with clause (1) giving

(7) $\{i(x_P, i(x'_P, i(a, a)))\not\in P\}$

Trying to build a resolvent between (7) and (3) or (4) again leads to unification problems which cannot be well sorted solved. Resolution with clause (2) results in the unification problem

$$\Gamma_2 \quad = \quad \{x_P = i(x_\mathsf{T}, y_\mathsf{T}), x'_P = i(y_\mathsf{T}, z_\mathsf{T}), x_\mathsf{T} = a, z_\mathsf{T} = a\}$$

Applying the rule Application of the unification algorithm using the equations $x_\mathsf{T} = a$ and $z_\mathsf{T} = a$ twice, we obtain

$$\Gamma_3 \quad = \quad \{x_P = i(a, y_\mathsf{T}), x'_P = i(y_\mathsf{T}, a), x_\mathsf{T} = a, z_\mathsf{T} = a\}$$

We don't have $i(a, y_\mathsf{T})\in\mathbf{T}_{\mathbf{L}_<,P}$ or $i(y_\mathsf{T}, a)\in\mathbf{T}_{\mathbf{L}_<,P}$. Application of Weakening VT to the equation $x_P = i(a, y_\mathsf{T})$ using the declaration $i(x_\mathsf{T}, i(n(x_\mathsf{T}), y_\mathsf{T}))<P$ gives

$$\begin{aligned}\Gamma_4 \quad = \quad & \{x_P = i(a, y_\mathsf{T}), x'_P = i(y_\mathsf{T}, a), x_\mathsf{T} = a, z_\mathsf{T} = a,\\ & y_\mathsf{T} = i(n(x'_\mathsf{T}), y'_\mathsf{T}), x'_\mathsf{T} = a\}\end{aligned}$$

Following the rule Application we use the new equations $y_\mathsf{T} = i(n(x'_\mathsf{T}), y'_\mathsf{T})$ and $x'_\mathsf{T} = a$ as substitutions and result in

$$\begin{aligned}\Gamma_5 \;=\; \{&x_P = i(a, i(n(a), y'_\top)), x'_P = i(i(n(a), y'_\top), a),\\ &x_\top = a, z_\top = a, y_\top = i(n(a), y'_\top), x'_\top = a\}\end{aligned}$$

The first equation is now well sorted solved, i.e. $i(a, i(n(a), y'_\top)) \in \mathbf{T}_{\mathbf{L}_\leqslant, P}$. But for the second equation we still have $i(i(n(a), y'_\top), a) \notin \mathbf{T}_{\mathbf{L}_\leqslant, P}$. Applying Weakening VT using the declaration $i(i(n(x_\top), x_\top), x_\top) \leqslant P$ solves this last problem.

$$\begin{aligned}\Gamma_6 \;=\; \{&x_P = i(a, i(n(a), y'_\top)), x'_P = i(i(n(a), y'_\top), a),\\ &x_\top = a, z_\top = a, y_\top = i(n(a), y'_\top), x'_\top = a,\\ &i(i(n(a), y'_\top), a) = i(i(n(x''_\top), x''_\top), x''_\top)\}\end{aligned}$$

After application of the rules Decomposition, Orientation, and Application to the equation $i(i(n(a), y'_\top), a) = i(i(n(x''_\top), x''_\top), x''_\top)$ and it's descendants we finish in the solved problem

$$\begin{aligned}\Gamma_7 \;=\; \{&x_P = i(a, i(n(a), a)), x'_P = i(i(n(a), a), a),\\ &x_\top = a, z_\top = a, y_\top = i(n(a), a), x'_\top = a,\\ &y'_\top = a, x''_\top = a)\}\end{aligned}$$

and derive

(8) $\square$

Note that the solution of the unification problem Γ_2 is unique and computationally simple. Comparing our solution to the proof found by OTTER (Version 2.0, binary resolution), we needed only 3 steps to refute the clause set, while OTTER made 5 steps. The missing 2 steps occur in the solution of the unification problem Γ_2. The most important difference is that the sorted proof was found without any search (neither during unification nor during the application of the inference rules), i.e. we have not generated useless resolvents. OTTER produced 60 clauses to find the empty clause.

4 Discussion

We have presented a sound and complete calculus $\mathcal{CL}_\mathcal{S}$ for a new sorted first-order logic $\mathcal{L}_\mathcal{S}$. The resolution calculus established by Theorem 3.5 is the most general, most restrictive (with regard to the number of applicable inference steps), completely given calculus we know for sorted logics. Note that the indeterminism of Theorem 3.5, i.e. which declarations to choose for $\mathbf{L}_\leqslant$ is not really an indeterminism. It corresponds to a case analysis. Thus if a declaration clause is needed in a proof, all declarations will be dynamically chosen during the deduction process. Nevertheless there are a lot questions left open, e.g. how to extend this calculus by equality reasoning.

Compared to the frameworks of Stickel [10] and Bürckert [2] the presented calculus is not an instance of them. Both frameworks require the a priori, static separation of a background theory, in our case the sort theory. Completeness of the calculus $\mathcal{CL}_\mathcal{S}$ relies on the dynamic change of the sort theory during the deduction process.

References

[1] C. Beierle, U. Hedstück, U. Pletat, and J. Siekmann. An order sorted predicate logic with closely coupled taxonomic information. LILOG-Report 86, IBM Germany, Scientific Center, 1989. To appear in Artificial Intelligence.

[2] H.J. Bürckert. *Constraint resolution.* Lecture Notes in Artificial Intelligence. Springer Verlag, 1991.

[3] A. Cohn. A more expressive formulation of many sorted logic. *Journal of Automated Reasoning*, 3(2):113–200, 1987.

[4] A.M. Frisch. A general framework for sorted deduction: Fundamental results on hybrid reasoning. In *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 126–136, May 1989.

[5] W. McCune. Otter 2.0. In *10th International Conference on Automated Deduction*, pages 663–664. Springer Verlag, 1990.

[6] W. McCune and L. Wos. Experiments in automated deduction with condensed detachment. In *11th International Conference on Automated Deduction.* Springer Verlag, 1992.

[7] D. Meredith. In memoriam carew arthur meredith (1904-1976). *Notre Dame Journal of Formal Logic*, 18(513-516), 1977.

[8] M. Schmidt-Schauß. *Computational aspects of an order sorted logic with term declarations.* Lecture Notes in Artificial Intelligence. Springer Verlag, 1989.

[9] J. Siekmann. Unification theory. *Journal of Symbolic Computation, Special Issue on Unification*, 7:207–274, 1989.

[10] M. Stickel. Theory resolution. *Journal of Automated Reasoning*, 1(4):333–355, 1985.

[11] C. Walther. *A Many-sorted Calculus based on Resolution and Paramodulation.* Research Notes in Artificial Intelligence. Pitman Ltd., 1987.

[12] C. Weidenbach. A sorted logic using dynamic sorts. MPI-Report MPI-I-91-218, Max-Planck-Institut für Informatik, Saarbrücken, December 1991.

[13] C. Weidenbach and H.J. Ohlbach. A resolution calculus with dynamic sort structures and partial functions. In *Proceedings of the 9th European Conference on Artificial Intelligence*, pages 688–693. Pitman Publishing, London, August 1990.

An Explanatory Framework for Human Theorem Proving

Xiaorong Huang *
Fachbereich Informatik, Universität des Saarlandes
Im Stadtwald, D-6600 Saarbrücken 11, Germany
huang@cs.uni-sb.de

Key words: Cognition, Deduction, proof plan

Abstract

Along the line of the science of reasoning proposed by Alan Bundy [8], we present in this paper a computational theory accounting for human formal deductive competence. Our goal is primarily twofold. For one thing, it is aimed to establish an explanatory framework for human theorem proving. Devised as a computational theory, for another thing, it should also set up a theoretical foundation for deductive systems which simulate the way in which human beings carry out reasoning tasks. As such, the hope is to arrive at systems which learn and plan, which share their experiences with human users in high level communications. The last requirement, we believe, makes such systems ultimately useful.

As a computational model, we cast the cognitive activities involved in theorem proving as an interleaving process of metalevel planning and object level verification. Within such a framework, emphasis is put on three kinds of tactics concerning three kinds of declarative knowledge structures. We also account for the acquisition of new tactics and methods, as well as the modifications of existing tactics and methods to suit novel problems. While the fundamental framework is sketched out formally, the mechanisms manipulating tactics and methods are only intended to be suggestive, achieved with the help of examples.

*This work was supported by the Deutsche Forschungsgemeinschaft, SFB 314 (D2, D3)

1 Introduction

In this paper, we propose a new computational theory accounting for human formal deductive competence, as opposed to theories accounting for human daily reasoning. As such, it is a response to Alan Bundy's call for a science of reasoning [8]. As our second goal, it should also provide a basis for computational systems which reveal many desirable qualities currently lacking in most of the automated reasoning systems. These include, for instance, being able to explain a proof in a more human oriented manner; being able to benefit from its successful or unsuccessful experiences, and thus displaying an evolving reasoning ability; and so on.

Statically, we cast a reasoning being as a knowledge based system. His reasoning competence is exclusively ascribed to the declarative knowledge of various sorts, and a set of special purpose reasoning procedures. In this first draft of our theory, declarative knowledge includes *rules of inference*, *assertions*, and a collection of *proof schemata*, as well as diverse *metalevel knowledge*. Since it is the combinations of chunks of declarative knowledge and reasoning procedures that are planned as unit to solve various reasoning tasks, we define these combinations as *tactics*. The total amount of tactics constitutes the *basic reasoning repertoire* at the disposal of a reasoner. The specifications of tactics, which serve as an assessment help in the planning process for a proof, are referred to as *methods*.

Dynamically, we assume the entire process from the analysis of a problem to the completion of a proof, to be an *interleaving process* of metalevel planning and object level verification. In addition, this process is centered around a representational structure called a *proof tree*, which records the current state of the development.

After setting up a general framework for our computational theory in section 2, emphasis is laid on three kinds of object-level tactics and methods in section 3, which are best understood and the concepts of metalevel tactics and methods. While the former concepts are treated formally, discussions on metalevel tactics and methods are intended to be more suggestive rather than formal. We turn to a brief description of the dynamic behavior of the interleaving process as a whole in section 4. Finally, an outlook of the future development and a discussion on possible applications in section 5 concludes this paper.

2 A Static Description

In this section, we first categorize the mental objects accommodated in our computational model, together with the corresponding procedures operating on them. They are briefly listed below.

2.1 A Proof Tree

As a theory conceiving theorem proving as an interleaving process of planning and verification, we use a structure of mental representation called a proof tree, to uniformly accommodate notions like proof sketches, proof plans and finally, proofs themselves. Formally, a proof tree is a tree where every node has four slots, usually represented as:

$$< Label, \triangle \vdash Derived - Formula, Tactic >$$

This means that the derived formula is (can or might be, respectively) derived using the tactic indicated, from the children of a node. $\triangle$ stands for the set of hypothesis the derived formula depends on. All of the last three slots of a node may have the value "*unknown*", indicating a proof sketch node. The tactic slot, in addition, may have the value *<the name of a tactic, plan>*, indicating a planned node; or just the name of a tactic, indicating a verified node.

2.2 Declarative Knowledge

The following is a listing of the different kinds of declarative knowledge accommodated in our model:

- rules of inference, including a kernel of rules innate to human being, usually referred to as the *natural logic* (NL);
- mathematical formulas collectively called *assertions*, including axioms, definitions, and theorems, interrelated in a certain conceptual structure [18];
- proof schemata, mainly evolving from proofs previously found;
- metalevel declarative knowledge, specifying possible manipulations on object-level knowledge, in particular proof schemata.

Clearly, our theory is a logic based one, built on top of the central hypothesis of the existence of a natural logic. Although there are psychological investigations which argue against this as a general hypothesis [16, 17], we believe it appropriate to assume at least, that the major mode of mathematical reasoning is a logical one.

2.3 Procedures

Procedures of diverse varieties are incorporated in our computational model:

- procedures serving as constituents of tactics and thus called by the planning procedures:
  - a small set of elementary procedures carrying out various standard *object level* reasoning tasks, via interpreting declarative knowledge;
  - an open-end set of special purpose object level reasoning procedures. The knowledge needed is interwoven in their algorithms;

  Procedures of this type usually run on a syntactic basis; neither planning nor heuristic searching is involved. These procedures are usually called as "*basic tools*" by procedures constructing proofs by planning;
- procedures not involved in the planning process per se, but partially responsible for the increase of the reasoning repertoire, examples are procedures accounting for the remembrance of proofs or rules of inference. They behave similar to perceptive procedures in a theory of general cognition [20];

- autonomous procedures responsible for more complex tasks, such as proof construction or proof checking as a whole. Usually, they are animated by relevant intentions, and handle by planning, heuristic searching as well as trial-and-error to fulfill the intentions. We have to admit that not much is known yet about these procedures, and further investigation is needed;
- other book-keeping procedures, widely ignored in this study (see [22, 20]).

3 Tactics and Methods

3.1 General Concepts and Classifications

The task of tactics is the construction of proof trees. In [8], a tactic can be conceived as a well defined computational procedure. Since the concept of declarative knowledge plays an important role in our theory, we define tactics to be pairs represented as:

<a reasoning procedure, a collection of declarative knowledge>

In some cases, such as the special purpose tactics of Bundy [7], the collection of knowledge may be empty. In contrast, the three general levels of tactics to be introduced below are supported by elementary procedures interpreting standard knowledge structure. In the rest of this subsection, we discuss some general features along which tactics and methods may vary:

- A tactic is *cognitively primitive*, if it does not call other tactics and if its application leads to the insertion of an atomic node in the proof tree. We further assume, that a verified proof is a tree consisting only cognitively primitive tactics. Otherwise, a tactic is *cognitively compound*;
- given a goal to be proved and a collection of preconditions, *a partial* tactic only suggests a proof tree with sketch nodes.

Tactics are usually associated with one or more methods, serving as the specifications of the reasoning ability of a particular tactic [8]. In general, a method has the following slots: a *preconditions* slot specifying the the leaves of the proof trees the tactic is intended to construct; and a *postconditions* slot specifying the root of the proof trees.

Again, we distinguish between complete and partial methods: while the execution of the corresponding tactic of a *complete method* will certainly bring about the postconditions, if the preconditions are satisfied; this is only likely yet not guaranteed, when working with *partial methods.*

3.2 Three Levels of General Purpose Tactics and Methods at the Object Level

Within the framework set up so far, we are going to introduce three reasoning procedures, which are actually knowledge interpreters. Coupled with chunks of

declarative knowledge of the corresponding type, they form tactics at different levels of reasoning power. In addition, we suggest some plausible accompanying methods consulted by the planning procedures, while trying to put tactics together into a proof.

3.2.1 A Primitive Procedure Applying Rules of Inference

First and foremost, we are going to introduce a procedure which *applies* rules of inference. To simplify the situation for the current study, we advance a second minor hypothesis which assumes the order sorted predicate logic of higher-order as the working language. As the natural logic, we have adopted the *natural deduction* system first proposed by Gerhard Gentzen [12, 1]. The following are several most important rules:

$$\frac{}{\triangle, F \vdash F}\text{HYP},\quad \frac{\triangle \vdash \exists_{x:S1} Fx,\ \triangle, F_{a:S2} \vdash H,\ Subsort(S2, S1)}{\triangle \vdash H}\text{CHOICE}$$

$$\frac{\triangle, F \vdash G}{\triangle, F \Rightarrow G}\text{DED},\quad \frac{\triangle \vdash F \vee G,\ \triangle, F \vdash H,\ \triangle, G \vdash H}{\triangle \vdash H}\text{CASE}$$

Reasoning by applying rules in NL, and rules associated with them in a certain pattern [14], are referred to as reasoning at the *logic level.* While the cognitive status of the rules of inference included in the natural logic are said to be *elementary* and *innate*, a human reasoner may learn new, domain-specific rules. For example, a rule about subset might be learned: $\frac{a \in S_1, S_1 \subset S_2}{a \in S_2}$ where "a", "S_1" and "S_2" are metavariables of sort "*Element*" and "*Set*", respectively. These new rules are said to be *acquired* and *compound.* For the learning mechanism, see [14].

Now for every rule of inference, we have a tactic which applies it, defined by the usual matching process.

3.2.2 Applications of Assertions

The second kind of important declarative knowledge concerns objects such as axioms, definitions, lemmata and theorems, and even intermediate results achieved during proof searching. They are, in our theory, collectively called *assertions.* Moreover, assertions are interrelated in complex conceptual structures [18]. The notion of the application of an assertion, though never defined precisely, bears a central role in various reasoning activities. One prima facie evidence is that proofs found by mathematicians are almost exclusively documented in terms of the applications of some assertions, i.e. at the *assertion level*, which is one level above the logic level, yet still *cognitively elementary.*

Although no introspection is possible to reveal the internal structure of the procedure applying assertions, in [14], we pinned down this notion after making a crucial observation, that every application of an assertion can be expanded to a logic level proof segment, referred to as its *natural expansion* (NE). By studying the natural expansions, we came up with a precise *characterization* of the input-output relation for the primitive procedure applying assertions. Formally, the reasoning ability of a tactic applying a certain assertion equals to that of the applications of a set of *finite* compound rules of inference, this makes this concept practically useful [14].

In this paper, it suffices to understand this notion intuitively. For example, given an assertion defining subset:

$$\forall_{S_1,S_2:Set} S_1 \subseteq S_2 \Leftrightarrow \forall_{x:Element} x \in S_1 \Rightarrow x \in S_2$$

We may derive $a \in S_2'$ from $a \in S_1'$ and $S_1' \subseteq S_2'$;
$S_1' \not\subseteq S_2'$ from $a \in S_1'$ and $a \notin S_2'$;
$\forall_{x:Element} x \in S_1' \Rightarrow x \in S_2'$ from $S_1' \subseteq S_2'$
and so on; by *applying* this definition.

Although the reasoning power of the application of an assertion can be precisely defined [14], it is however not plausible to suggest that the planning decisions are made based on this time consuming information. The kind of *partial methods* we believe as one viable approximation can be defined in the following pattern (check the example above):

Method: *Application of Assertion A*

- **Preconditions:** the premises are all either a subformula of A, a specialization[1] thereof, or, thirdly, a negation of the first two cases,
- **Postconditions:** the result is either a subformula of A, a specialization thereof, or, thirdly, a negation of the first two cases.

3.2.3 A Procedure Instantiating Proof Schemata

The third level of tactics is tied to a more novel kind of knowledge structure called *proof schemata*, and a procedure *instantiating* them. Initially, they might contain a complete or partial proof found for a previous problem without metavariables. A (partial) specification of the corresponding problem can serve as the associated method. Since proof schemata are assumed to be represented mentally as proof trees using exclusively cognitively elementary tactics, their applications are usually *compound* tactics. The following is an example.

Example 1
After learning a proof showing that the power set $P(M)$ of a set M has a greater cardinality than the set itself, by proving that there is no surjective function $f : M \mapsto P(M)$, the following tactic-method pair might be added to the reasoning repertoire.

Method: Diagonalization-Power-Set-1

- **Precondition:** none
- **Post Condition:** $\neg\exists_{f:M\mapsto P(M)} Surj(f)$
- **Tactic:** Applying Schema-Diagonalization-Power-Set-1(see below)

1. $\triangle \vdash Theo; IP$

[1] P_a is defined as the specialization of both $\forall_x P_x$ and $\exists_x P_x$

1.1. $\triangle' = \triangle + \neg(Theo) \vdash \bot; ?$

1.1.1. $\triangle' \vdash \forall_{x \in P(M)} \exists_{y \in M} f(y) = x; def.Surj$

1.1.2. $\triangle' \vdash \exists_{S \in P(M)} \forall_{y \in M} f(y) \neq S; Choice$

1.1.2.1. $\triangle' \vdash \exists_{S \in P(M)} \forall_{x \in M} x \in S \Leftrightarrow x \notin f(x), ComprehensionAxiom$

1.1.2.2. $\triangle' + \forall_{x \in M} x \in S \Leftrightarrow x \in M \wedge x \notin f(x),$
$(abr.S = \{x \in M / x \notin f(x)\}$
$\vdash \exists_{S \in P(M)} \forall_{y \in M} f(y) \neq S; ?$

Notice that the numbering of label serves only to represent the tree structure. For example, node 1.1.2 has two children 1.1.2.1 and 1.1.2.2, and node 1.1.2 is also derived from its children, by applying the choice rule, as indicated in the tactic slot. The tactic slot of each node has the following possible values: the name of a rule in the natural logic, like *IP*, standing for indirect proof; the name of an assertion of various sorts, like *def.surj* or *Comprehension Axiom*, standing for the application of the definition of surjectivity and the comprehension axiom, respectively; or "?", standing for "unknown".

3.3 Metalevel Tactics and Methods

Our theory is also devised to account for the phenomena that reasoners benefit from their successful or unsuccessful experiences. In addition to those more perceptual procedures [15], this is achieved mainly through the metalevel tactics. When a reasoner is confronted with a novel problem, proof schemata evolving from previous proofs are modified to cope with the new problem. Now, we illustrate two types of metalevel tactics informally. Guided by declarative knowledge of different kinds, they are carried out by two procedures which *generalizes* or *reformulates* existing proof schemata, respectively.

Assuming the acquisition of the tactic-method pair in Example 1, we illustrate how it can be of use in handling a new problem. For space restriction, only the operations will be given, while the intermediate tactics omitted.

Problem 2

The interval $[0, 1]$ is not countable, that is, there is no surjective function:$f : N \longmapsto [0, 1]$.

We now show, how a proof sketch can be arrived at by successive manipulations on the tactic gained above. The first step is a *reformulation.* More exactly, the notion of a set is axiomatized now in terms of its characteristic predicate, instead of as an object. This reformulation is enabled by a truth-preserving mapping, a metalevel declarative knowledge structure coming into play at this point. Concretely in our example, the following truth-preserving mapping is used:

$$x \in S \longmapsto S(x), \qquad Subsort(Set, Object) \longmapsto Subsort(Set, Predicate)$$

where "x" and "S" are metavariables of sort *Element* and *Set.* Thus, the definition of subset used in our example

$$Subsort(Set, Object), \qquad \forall_{S1,S2:Set} S1 \subseteq S2 \Leftrightarrow \forall_{x:Element} x \in S1 \Rightarrow x \in S2$$

is replaced by

$$Subsort(Set, Predicate), \qquad \forall_{S1,S2:set} S1 \subseteq S2 \Leftrightarrow \forall_{x:Element} S1(x) \Rightarrow S2(x)$$

The second reformulation is based on the knowledge that predicates are special functions:

$$Subsort(Predicate, Function), \qquad \forall_{P:Predicate} \forall_{x:Any} P(x) \in \{t, f\}$$

This guarantees the truth-preserving property of the mapping below:

$$P(x) \longmapsto P(x) = t, \qquad \neg P(x) \longmapsto P(x) = f$$

where "P" is a metavariable of sort *Predicate*, and "x" has the sort of the argument sort of P. Formulas resulting from this mapping usually undergo some further simplifications, such as replacing $(P = t) \Leftrightarrow (Q = t)$ by $P = Q$, and $(P = t) \Leftrightarrow (Q = f)$ by $P = neg(Q)$ where the function *neg* is defined as

$$neg : \{t, f\} \longmapsto \{t, f\}, \qquad neg(x) = \begin{cases} t & : \quad x = f \\ f & : \quad x = t \end{cases}$$

After applying this mapping on the predicate $\in$, a manipulation called *generalization* is possible, which is of more heuristic nature and no more truth-preserving. In our case, term $P(M)$ may be generalized to a metavariable N of sort *Set* satisfying:

$$\forall_{x \in M, y \in N} y(x) \in \{t, f\} \tag{1}$$

which in fact defines $P(M)$. Notice that this restriction is added to the set of preconditions. As will be illustrated in the next step, a *generalization* usually adds a precondition which is a property weaker than the definition.

There is apparently still a gap between the precondition of the problem, $\forall_{x \in [0,1]} \forall_{n \in N} x(n) \in \{0, 1, 2, \cdots, 9\}$ and the precondition in (1). This can be bridged by a generalization of the two constants {0,1} and $\{0, 1, 2, \cdots, 9\}$ into a metavariable U of sort *Set*.

A new tactic-method pair is achieved based on the mapping:$\{0, 1\} \longmapsto U$

Method: $Diagonalization - N \longmapsto [0, 1]$

- **Precondition:** $\exists_{U:|U| \geq 2} \forall_{y \in N} \Leftrightarrow \forall_{x \in M} y(x) \in U$
- **Post Condition:** $\neg \exists_{f:M \mapsto N} Surj(f)$
- **Tactic:** Applying Schema-Diagonalization-$N \longmapsto [0, 1]$(see below)

1. $\triangle \vdash Theo; IP$

   1.1. $\triangle' = \triangle + \neg Theo \vdash \bot; ?$

      1.1.1. $\triangle' \vdash \forall_{x \in N} \exists_{y \in M} f(y) = x; def.Surj$

      1.1.2. $\triangle' \vdash \exists_{S \in N} \forall_{y \in M} f(y) \neq S; 2 * Choices$

         1.1.2.1. $\triangle' \vdash \exists_{g \in U \longmapsto U} \forall_{x \in U} g(x) \neq x, ?$

1.1.2.2. $\Delta'' = \Delta' + g \in U \longmapsto U \wedge \forall_{x \in U} g(x) \neq x \vdash \exists_S \forall_{x \in M} S(x) = g(f(x)(x)), Comp.Axiom$

1.1.2.3. $\Delta'' + \forall_{x \in M} S(x) = g(f(x)(x)) \vdash \exists_{S \in N} \forall_{y \in M} f(y) \neq S$;

Instantiation for $N \longmapsto [0,1]$ example: $M \longmapsto N, N \longmapsto [0,1], U \longmapsto \{0,1,\cdots,9\}$. The precondition is satisfied, since $\forall_{x \in N} \forall_{y \in [0,1]} x(y) \in \{0,1,\cdots,9\}$

While the introduction of U may be easily ascribed to the existence of heuristic knowledge of some sort, the creation of the new function g is a step demanding real human creativity. We probably have to appeal to analogical reasoning. A property of the original sets is added as a precondition: $|U| \geq 2$. We want to mention again, that we still do not know much about metalevel tactics and methods, still less how they are employed by the planner. Much investigation is needed.

4 A Dynamic Description of Theorem Proving

Since our emphasis is laid more on the static part of the theory, and since we still do not know much about the dynamic behavior, this discussion is tentative and aimed to be suggestive. For a more detailed discussion, see [15].The basic assumption is, that there is an autonomous procedure responsible for the planning process. The task of this procedure is to analyze the problem, and on the ground of the information contained in methods, to recursively break down the problem into subproblems, and call tactics sequentially to solve the subproblems. Since most methods are partial, some of the thus planned proof steps have to be verified subsequently. The current state of such a dynamic development is always reflected in an internal structure which is a proof tree. The planning mode and the verification mode may converge, when the methods employed are complete. Whenever an existing plan fails, a replanning phase usually begins.

There are also metalevel activities. In our theory, a procedure or tactic is at the metalevel if it causes changes in the knowledge base, rather than the proof tree. Metalevel tactics are usually invoked by the intention to solve a specific problem, and their application require concentration and effort, as opposed to those more perceptual procedures, like the remembrance of a proof or of a rule [15].

We can account for the acquisition of all the three kinds of declarative knowledge: in the first case it is fairly simple. If a particular subproof is carried out repeatedly, its input-output specification may be put together into a new *acquired* rule of inference and remembered. The total amount of inference rules at the disposal of a reasoner is referred to as his *natural calculus*, which contains the natural logic as a subset. Second, new axioms and definitions are constantly incorporated into the conceptual structure of a reasoner, as well as proved theorems [18]. Third, the initial proof schemata are simply proofs of some problems *learned* by the reasoner, by reading mathematical text books, for instance, or by being taught. The problem specifications can be taken over as the initial methods. Afterwards, new tactics and methods are accumulated, built by metalevel tactics in the attempts to deal with novel problems.

5 Conclusion

In this section, we examine our theory from the perspective of the two roles it is supposed to play, and sketch an outlook of further development.

As an explanatory framework, as far as we know, it is the first attempt to set up a full-fledged computational model for human formal reasoning, as opposed to daily casual reasoning studied by psychologists [6, 16, 19, 22]. For an overview, see [17]. As such, however, we must point out that much empirical studies still have to be made to substantiate the framework in general, on the one hand, and to settle the remaining unsolved problems, on the other. Among the most important are the planning process itself, and the repertoire of metalevel tactics. And as a natural consequence, they provide an excellent basis for explaining proofs found [13].

Within the community of automated deduction in AI, there are mainly two approaches pursued to build intelligent and efficient inference systems. The first paradigm uses a more human oriented framework, involving primarily operations understandable by human beings. Researches along this line can be found in [5, 9, 21, 3, 7]. More recently, Alan Bundy has called on to start a full-scaled investigation of what he calls a science of reasoning. This paper can be viewed as an initial response to this call. Compared with the way compound inference rules are generated to enhance the reasoning ability in interactive deduction systems [10], the three precisely defined levels of object level tactics are much better psychologically supported and provide a more natural way of organizing our deductive repertoire.

The second paradigm [23, 2, 4, 11] may be characterized as using the enormous computational power of the modern computers to solve problems through relatively blind searching. Despite this deep gap, some concepts developed in our framework turn out to be useful even for this type of systems, especially the notion of the applications of assertions. With its naturalness and abstractness, it provides a an adequate level of intermediate representation in the process of transforming machine generated proofs into natural language. The length of *assertion level* proofs are normally only one third of that of natural deduction proofs at the logic level [14].

References

[1] P. Andrews. Transforming Matings into Natural Deduction Proofs. *LNCS*, 87, 1980.

[2] P. Andrews. Theorem Proving via General Matings. *JACM*, 28, 1981.

[3] J. Bates. The architecture of prl: A mathematical medium. Technical Report CMU-CS-90-149, School of Computer Science, CMU, July 1990.

[4] W. Bibel. *Automated Theorem Proving.* Vieweg, Braunschweig, 1983.

[5] R. S. Boyer and J. S. Moore. *A Computational Logic.* Academic Press, 1979.

[6] M. D. Braine. On the Relation Between the Natural Logic of Reasoning and Standard Logic. *Psychological Review*, 85(1), Jan 1978.

[7] A. Bundy. The Use of Explicit Plans to Guide Inductive Proofs. In *Proc. of 9th International Conference on Automated Deduction*, 1988.

[8] A. Bundy. A Science of Reasoning: Extended Abstract. In *Proc. of 10th International Conference on Automated Deduction.* Springer, 1990.

[9] R. L. Constable. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Inc., 1986.

[10] P. T. David Basin, Fausto Giunchiglia. Automating meta-theory creation and system extension. Technical Report DAI No. 543, Univ. of Edinburgh, 1991.

[11] N. Eisinger and H. J. Ohlbach. The markgraf karl refutation procedure. In *Lecture Notes in Comp. Sci.*, 230 (CADE 86), 1986.

[12] G. Gentzen. Untersuchungen über das logische Schließen I. *Math. Zeitschrift*, 39, 1935.

[13] X. Huang. Reference Choices in Mathematical Proofs. In *Proc. of ECAI-90, L. C. Aiello (Ed).* Pitman Publishing, 1990.

[14] X. Huang. An Extensible Natural Calculus for Argument Presentation. Technical Report SEKI SR-91-3, Uni. Kaiserslautern, 1991.

[15] X. Huang, M. Kerber, and M. Kohlhase. Theorem proving as a planning and verification process. Technical Report to appear as SEKI Report, Uni. des Saarlandes, 1992.

[16] P. Johnson-Laird. *Mental Models.* Harvard Univ. Press, Cambridge, Massachusetts, 1983.

[17] P. Johnson-Laird and R. Byrne. *Deduction.* Ablex Publishing Corporation, 1990.

[18] M. Kerber. On the representation of mathematical knowledge in frames and its consistency. In *WOCFAI '91*, 1991.

[19] G. Lakoff. Linguistics and natural logic. *Syntheses*, 22, 1970.

[20] A. Newell. *Unified Theories in Cognition.* Harvard University Press, Cambridge, MA, 1990.

[21] L. C. Paulson. *Logic and Computation.* Cambridge university Press, 1987.

[22] L. J. Rips. Cognitive Processes in Propositional Reasoning. *Psychological Review*, 90, 1983.

[23] J. Robinson. A machine-oriented logic based on the resolution principle. *J. of ACM*, 12, 1965.

Towards First-order Deduction Based on Shannon Graphs

Joachim Posegga & Bertram Ludäscher

Universität Karlsruhe
Institut für Logik, Komplexität und Deduktionssysteme
Am Fasanengarten 5, 7500 Karlsruhe, Germany
Email: {posegga,ludaesch}@ira.uka.de

March 19, 1992

Abstract

We present a new approach to Automated Deduction based on the concept of Shannon graphs, which are also known as Binary Decision Diagrams (BDDs). A Skolemized formula is first transformed into a Shannon graph, then the latter is compiled into a set of Horn clauses. These can finally be run as a Prolog program trying to refute the initial formula. It is also possible to precompile axiomatizations into Prolog and load these theories as required.

Keywords: Automated Deduction, Shannon Graphs, Binary Decision Diagrams

1 Introduction

Logical formulae are usually defined using some or all of the connectives $\{\neg, \vee, \wedge, \rightarrow, \leftrightarrow\}$. Alternatively, one can use a ternary *if-then-else* operator "*sh*", which can be viewed as an abbreviation: $sh(A, S_0, S_1) \equiv (\neg A \wedge S_0) \vee (A \wedge S_1)$. Thus, $sh(A, S_0, S_1)$ corresponds to an expression of the form "**if** A **then** S_1 **else** S_0". Formulae using only this operator are called *Shannon graphs* (introduced by Shannon in [Shannon, 1938]) or *B*inary *D*ecision *D*iagrams [Lee, 1959]. BDDs have been successfully used for Boolean function manipulation [Bryant, 1986, Brace *et al.*, 1990], for example in the context of hardware verification [Kropf & Wunderlich, 1991] or for processing database queries [Kemper *et al.*, 1992].

Up to now, this method has hardly influenced research in Automated Deduction, probably because it has not yet been extended to full first-order logic[1]. We will show how this can be done and argue that the concept of Shannon graphs is indeed a useful framework for implementing a first-order deduction system.

The underlying idea for the proposed proof procedure is to transform a formula into a Shannon graph and compile this graph into Horn clauses. When run as a

[1] Only Orłowska [Orłowska, 1969] has described an extension to a decidable subset of first-order logic.

Prolog program the clauses simulate traversing the Shannon graph. This process tries to show properties of the graph which are equivalent to the fact that the formula cannot have a model.

An experimental Prolog-implementation of a propositional prover based on these principles proved to be quite efficient (see Section 3.1); so, the extension of the principle to first-order logic, which is currently being implemented seems promising.

The paper is organized as follows: in Section 2, Shannon graphs and their semantics are given together with a transformation from formulae to Shannon graphs. Properties relating to the soundness and completeness of our proof method are stated. Section 3 describes how one can implement the proof procedure by generation of Horn clauses and gives some test results for propositional logic.

2 First-order Shannon Graphs

Let $\mathcal{L}$ be the language of first-order calculus defined in the usual way; $\mathcal{L}_{At}$ are the atomic formulae of $\mathcal{L}$. "sh" is a new ternary connective, which can be regarded as an abbreviation: $sh(A, S_0, S_1) \equiv (\neg A \wedge S_0) \vee (A \wedge S_1)$.

Definition 1 *The set of Shannon graphs is denoted by $\mathcal{SH}$ and defined as the smallest set such that*

(1) $\mathbf{0}, \mathbf{1} \in \mathcal{SH}$

(2) if $\mathcal{S}_0, \mathcal{S}_1 \in \mathcal{SH}$ and $A \in \mathcal{L}_{At}$ then $sh(A, \mathcal{S}_0, \mathcal{S}_1)$ is in $\mathcal{SH}$.

The **truth-value** $\mathrm{val}_{\mathcal{D},\beta}$ *of a Shannon graph $\mathcal{S}$ in a given structure $\mathcal{D}$ with a fixed variable assignment β is defined as follows:*

$$\mathrm{val}_{\mathcal{D},\beta}(\mathcal{S}) = \begin{cases} \mathsf{false} & \text{if } \mathcal{S} = \mathbf{0} \\ \mathsf{true} & \text{if } \mathcal{S} = \mathbf{1} \\ \mathrm{val}_{\mathcal{D},\beta}(sh_-) & \text{if } \mathcal{S} = sh(A, sh_-, sh_+) \text{ and } \mathrm{val}_{\mathcal{D},\beta}(A) = \mathsf{false} \\ \mathrm{val}_{\mathcal{D},\beta}(sh_+) & \text{if } \mathcal{S} = sh(A, sh_-, sh_+) \text{ and } \mathrm{val}_{\mathcal{D},\beta}(A) = \mathsf{true} \end{cases}$$

■

We can visualize the formulae of $\mathcal{SH}$ as binary trees with leaves **0** and **1**. Each nonterminal node is labeled with the atomic formula occurring in the first argument of a "sh"-term. Nodes have a negative and a positive edge leading to subtrees representing the second or third argument of the corresponding "sh"-term, respectively. Consider the formula $a \wedge b$: its Shannon graph $\mathcal{S}_1 = sh(a, \mathbf{0}, sh(b, \mathbf{0}, \mathbf{1}))$ is shown in Figure 1.

Semantically, a Shannon graph can be regarded as a case-analysis over the truth-values of atoms occurring in a formula. Assume there is a sequence of nodes and edges from the root to an arbitrary leaf of a Shannon graph. We prefix each atomic

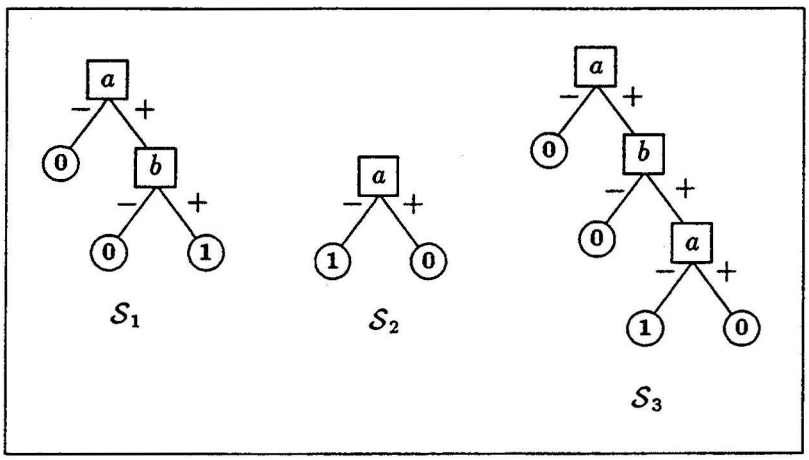


Figure 1: Shannon Graphs for $a \wedge b$, $\neg a$, and $a \wedge b \wedge \neg a$

formula with the sign of the corresponding node's outgoing edge: for example in $\mathcal{S}_1$ in Figure 1, $[+a, +b]$ is such a sequence leading to a **1**-leaf. We will call such a sequence a *path*.

If the atoms of some path can be consistently interpreted with the truth-value that their prefix suggests (ie **true** for a positive and **false** for a negative edge), then the path represents an interpretation. Under this interpretation the whole formula will have the truth-value with which the leaf is labeled.

Conversely, if an atomic formula and its complement occur on a path as in $[+a, +b, -a]$ in $\mathcal{S}_3$, then the path is called *inconsistent* or *closed* and there is no need to investigate it any further.

Let us assume henceforth that a Skolemized formula $F(X_1, \ldots, X_n)$ (or $F(\bar{X})$ for short) with the free variables $X_1, \ldots, X_n$ is given[2].

In order to prove the unsatisfiability of the universal closure $Cl_\forall F(\bar{X})$ of $F(\bar{X})$, we proceed by constructing a logically equivalent Shannon graph $\mathcal{F}(\bar{X})$ and try to show that it contains no consistent path to a **1**-leaf.

The construction is done in the following way: obviously, an atomic formula A is logically equivalent to $sh(A, \mathbf{0}, \mathbf{1})$. If we already have two Shannon graphs $\mathcal{A}$ and $\mathcal{B}$ representing the formulae A and B, we construct the Shannon graph for $A \wedge B$ by replacing all **1**-leaves of $\mathcal{A}$ with $\mathcal{B}$. This replacement is denoted $\mathcal{A}[\frac{1}{\mathcal{B}}]$. In Figure 1, $\mathcal{S}_3$ is the result of replacing the **1**-leaf in $\mathcal{S}_1$with $\mathcal{S}_2$. Analogously, disjunctions are handled by replacing **0**-leaves.

Now the transformation $conv : F(\bar{X}) \mapsto \mathcal{F}(\bar{X})$ which maps a formula to its Shannon graph is given by

[2]Actually Skolemization can easily be integrated into the function *conv*, below.

Definition 2

$$conv(F) = \begin{cases} sh(F, \mathbf{0}, \mathbf{1}) & \text{if } F \in \mathcal{L}_{At} \\ conv(A)[\frac{\mathbf{1}}{conv(B)}] & \text{if } F = A \wedge B \\ conv(A)[\frac{\mathbf{0}}{conv(B)}] & \text{if } F = A \vee B \\ conv(A)[\frac{\mathbf{0}}{\mathbf{1}}, \frac{\mathbf{1}}{conv(B)}] & \text{if } F = A \rightarrow B \\ conv(A)[\frac{\mathbf{0}}{\mathbf{1}}, \frac{\mathbf{1}}{\mathbf{0}}] & \text{if } F = \neg A \end{cases}$$

■

One easily verifies that the time and space complexity of the above transformation *conv* as well as the size of the resulting Shannon graph are proportional to the size of the input formula if *structure sharing* is used for the replacement of leaves. Ie, if several leaves are replaced by the same graph, we do not copy this graph but introduce edges to a single instance of it. Therefore, *conv* constructs a directed, acyclic graph, and not a tree.

Remark 1 *Note, that there is a cheap way to optimize the generated graph: assume we are to construct a graph for $A \wedge B$; we can either construct $conv(A)[\frac{\mathbf{1}}{conv(B)}]$, or the logically equivalent graph $conv(B)[\frac{\mathbf{1}}{conv(A)}]$. Although both graphs do not differ in size, the trees they represent generally do. As those trees are the potential search space for a proof, it is of course desirable to construct the graph that represents the smallest of those trees. Fortunately, this size can easily be determined in advance:*

Assume that $\#_{\mathbf{1}}, \#_{\mathbf{0}}$, and $\#_{at}$ denote the number of $\mathbf{1}$-leaves, $\mathbf{0}$-leaves, and (non-terminal) nodes of the tree represented by a Shannon graph. If $\mathcal{A}$ and $\mathcal{B}$ are Shannon graphs, then the size of the tree for $\mathcal{G} = \mathcal{A}[\frac{\mathbf{1}}{\mathcal{B}}]$ will be:

$$\#_{at}(\mathcal{A}) + \#_{\mathbf{0}}(\mathcal{A}) + \#_{\mathbf{1}}(\mathcal{A}) \cdot (\#_{at}(\mathcal{B}) + \#_{\mathbf{0}}(\mathcal{B}) + \#_{\mathbf{1}}(\mathcal{B}))$$

or, equivalently,

$$\underbrace{\#_{\mathbf{0}}(\mathcal{A}) + \#_{\mathbf{1}}(\mathcal{A}) \cdot \#_{\mathbf{0}}(\mathcal{B})}_{\#_{\mathbf{0}}(\mathcal{G})} + \underbrace{\#_{at}(\mathcal{A}) + \#_{\mathbf{1}}(\mathcal{A}) \cdot \#_{at}(\mathcal{B})}_{\#_{at}(\mathcal{G})} + \underbrace{\#_{\mathbf{1}}(\mathcal{A}) \cdot \#_{\mathbf{1}}(\mathcal{B})}_{\#_{\mathbf{1}}(\mathcal{G})}$$

The size of the tree in case of a disjunction can be determined analogously. Although this heuristic works only locally and does not guarantee to generate the best global result, it has proven to be very useful in practice. ■

Let us call $\mathcal{F}(\bar{X})\sigma$ *closed* if all paths to $\mathbf{1}$-leaves in $\mathcal{F}(\bar{X})\sigma$ are inconsistent, where σ substitutes the free variables $\bar{X}$ of $\mathcal{F}(\bar{X})$. Now a basic property of Shannon graphs can be stated:

Proposition 1 *Let $\mathcal{F}(\bar{X}) = conv(F(\bar{X}))$. If there is a grounding substitution σ such that $\mathcal{F}(\bar{X})\sigma$ is closed then $F(\bar{X})\sigma$ is unsatisfiable.* ■

How do we find such a σ? We traverse a Shannon graph building a path. If a path contains two unifiable atoms $A_1\sigma = A_2\sigma$ with complementary signs then σ will close that path.

Consider the following example:

Let $P_0(X) := p(a) \wedge \neg p(f(f(a))) \wedge (p(X) \rightarrow p(f(X)))$. We want to show that $Cl_\forall P_0(X)$ is unsatisfiable. The equivalent Shannon graph $\mathcal{P}_0(X) = conv(P_0(X))$ is shown on the left in Figure 2:

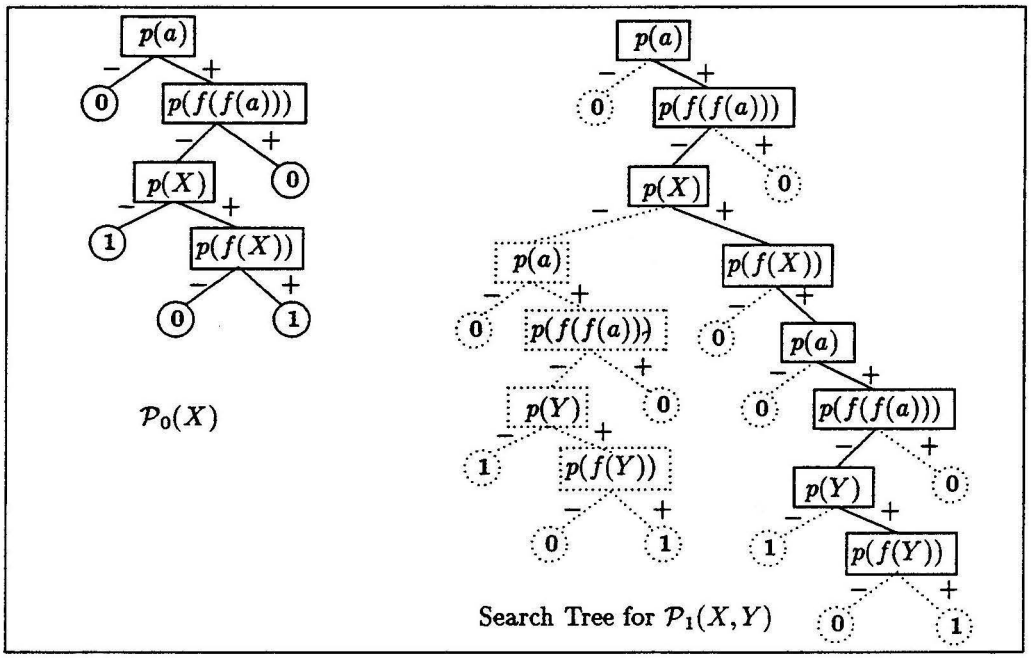


Figure 2: Proving inconsistency of $Cl_\forall P_0(X) := p(a) \wedge \neg p(f(f(a))) \wedge \forall X(p(X) \rightarrow p(f(X)))$

The path $[+p(a), -p(f(f(a))), -p(X)]$ to the leftmost 1-leaf in $\mathcal{P}_0(X)$ can be closed with the substitution $\sigma = [X/a]$. Unfortunately there is no substitution such that *both* paths to 1-leaves become inconsistent, so we cannot prove the unsatisfiability of $Cl_\forall P_0(X)$ with $\mathcal{P}_0(X)$. To do this we need to consider a consequence of the compactness of first-order logic:

Proposition 2 *$Cl_\forall F(\bar{X})$ is unsatisfiable iff $\exists k \in I\!N, \exists \sigma : (F(\bar{X}_0) \wedge \ldots \wedge F(\bar{X}_k))\sigma$ is unsatisfiable.*

σ maps variables to ground terms, ie to terms from the Herbrand-universe U_F of F and each $\bar{X}_i$ is a new *variable vector $X_{i,1}, \ldots, X_{i,n}$ distinct from all $\bar{X}_j$ with $j < i$.*

■

Since there is no σ such that $\mathcal{P}_0(X)\sigma$ can be closed (cf. Figure 2), we **extend** $\mathcal{P}_0(X)$ by replacing its 1-leaves with an instance of $\mathcal{P}_0$ with renamed variables. The result $\mathcal{P}_1(X,Y) := \mathcal{P}_0(X)[\frac{1}{\mathcal{P}_0(Y)}]$ is by Definitions 1 and 2 logically equivalent to $P_1(X,Y) := P_0(X) \wedge P_0(Y)$. Now the substitution $\sigma = [X/a, Y/f(a)]$ yields

a closed Shannon graph $\mathcal{P}_1(X,Y)\sigma$. Hence $P_1(X,Y)\sigma$ is unsatisfiable and by Proposition 2 so is $Cl_\forall P_0(X)$.

Note that the dotted subtree rooted at the negative edge of $p(X)$ can be pruned from the search space: all paths going through this edge are closed with the substitution $[X/a]$. Additionally, all edges to **0**-leaves are actually never considered during the proof search.

For the general case, consider the sequence of Shannon graphs

$$\begin{aligned} \mathcal{F}_0(\bar{X}_0) &= conv(F(\bar{X}_0)) \\ \mathcal{F}_{i+1}(\bar{X}_0,\ldots,\bar{X}_{i+1}) &= \mathcal{F}_i(\bar{X}_0,\ldots,\bar{X}_i)[\tfrac{\mathbf{1}}{\mathcal{F}_0(\bar{X}_{i+1})}] \end{aligned}$$

Starting from the initial Shannon graph $\mathcal{F}_0(\bar{X}_0)$ of a Skolemized formula $F(\bar{X}_0)$ we construct an *extension* $\mathcal{F}_{i+1}$ of $\mathcal{F}_i$ by replacing its **1**-leaves with an instance of the initial graph with new variables. This corresponds to constructing the conjunction of Proposition 2.

From the Definitions 1 and 2 it follows that $val_{\mathcal{D},\beta}(\mathcal{F}_k(\bar{X}_0,\ldots,\bar{X}_k)) = val_{\mathcal{D},\beta}(F(\bar{X}_0)\wedge\ldots\wedge F(\bar{X}_k))$, so if $Cl_\forall F(\bar{X}_0)$ is unsatisfiable (cf. Proposition 2) we will eventually arrive at a $\mathcal{F}_k$ which can be closed for some σ.

3 Generating Horn clauses

To show the unsatisfiability of a formula $Cl_\forall F(\bar{X})$, we have to traverse its Shannon graph $\mathcal{F}_0(\bar{X})$ trying to find a substitution that closes all paths from the root to **1**-leaves; if there is no such substitution, we extend $\mathcal{F}_0(\bar{X})$. For implementing the above method, we propose to compile $\mathcal{F}_0(\bar{X})$ into a set of Horn clauses, which simulate the Shannon graph traversal.

Remember that $\mathcal{F}_0(\bar{X})$ has a representation which is proportional to the size of the input formula. We proceed as follows:

For leaves no clauses are generated; for every *nonterminal* node in $\mathcal{F}_0$ we generate exactly one Prolog-clause, which must accomplish one or more of the following tasks :

- find a substitution that closes the current path
- extend a **1**-leaf
- transfer control to the clauses for child nodes

Therefore we supply the clause with the current variable binding, the path constructed so far, and a designator of the current extension (level). A clause succeeds if at each **1**-leaf reachable from the corresponding node the path can be made inconsistent. To get a more definite idea, we will briefly discuss a simplified clause for the node $p(X)$ of $\mathcal{P}_0$, cf. Figure 2:

```
node_3(Binding,Path,Level):-                                          (1)
    nth(Level,Binding,[X]),                                           (2)
    (close([-p(X)|Path])                                              (3)
    ; (NLevel is Level+1,node_1(Binding,[-p(X)|Path],NLevel))),       (4)
    node_4(Binding,[+p(X)|Path],Level).                               (5)
```

In line (2), all free variables occurring in the atomic formula with which the current node is labeled, are bound (in this case only `X`). Technically, this can be done in the following way: since each level has its own set of variables, the variable bindings are level-dependent. It is assumed that `Binding` is a list of such variable bindings where the *n-th* element stores the binding for the *n-th* level. Each binding itself is also a list, which holds the value of each variable at a certain position; during compile time, a list is constructed that matches these position(s) in order to extract the required bindings. As our example contains only one variable, the matching list is just `[X]`.

The predicate `close` in line (3) tries to make `Path` together with `-p(X)` inconsistent as we reach a **1**-leaf at the negative edge; if this fails, line (4) extends the graph by increasing the level counter and calling the clause for the root-node again. So extending the graph is modeled without asserting new Prolog clauses. This is correct since each level establishes its own variable bindings.

Calling the clause for the root-node again will succeed if all paths to **1**-leaves in one of the next extensions can be made inconsistent.

Finally, we have to show that all paths to **1**-leaves reachable from the positive edge of $p(X)$ are also inconsistent therefore we call the child `node_4` at the positive edge.

Our proposed proof procedure will be complete if we use bounded depth-first search with iterative deepening and force the predicate `close` to enumerate all possible solutions (ie ways to close paths) on backtracking. Another, more efficient approach is to implement a *fair* selection scheme for `close`.

It should be noted that the example clauses are not a particular efficient way to code the problem in Prolog; clearness and readability has been preferred over efficiency.

3.1 Propositional Logic

The method described above has been implemented for propositional logic; a first-order version is currently under development. In the propositional case, the structure of the generated Prolog clauses can be kept much simpler, since no extensions take place and no variable bindings need to be considered. The maximal length of the generated paths is also known in advance, so much more efficient data-structures can be used for representing paths. Performance figures for pigeon hole formulae with a propositional version of the prover are shown in Table 1:

		Search Space			CPU Time [msec]		
Problem	Graph Nodes	Potential	Explored	Closed Paths	Preproc	Compile	Proof
pig2	4	9	3	2	17	83	< 17
pig3	18	2,045	60	37	100	300	< 17
pig4	48	$6.4 \cdot 10^7$	882	644	233	683	< 17
pig5	100	$3.0 \cdot 10^{15}$	9,677	7,351	466	1,417	83
pig6	180	$1.5 \cdot 10^{27}$	156,756	124,265	816	2,533	1,717
pig7	294	$5.7 \cdot 10^{43}$	2,961,695	2,414,873	1,383	4,050	28,733

The CPU times are measured on a Sun-4 with Quintus Prolog 3.0.; it was not possible to measure times less than 17 msec.

Table 1: Some Test Results.

"pig n" stands for "n pigeons do not fit into $n-1$ holes"; "Graph Nodes" shows the size of the initial graph. The columns labeled "Search Space" give the potential search space, ie, the size of the tree represented by the initial graph, and the number of nodes that have actually been visited. "Closed Paths" is the number of paths that have been constructed and closed, which corresponds to the number of partial models that have been generated and ruled out.

"Preproc" is the time for computing the initial graph and generating the Prolog clauses; "Compile" gives the time required by the Prolog System to compile the latter. "Proof" is the time used for executing the generated Prolog clauses, ie, the search for a model.

4 Conclusions

We have described a method to compile first-order formulae via Shannon graphs into Horn clauses. The method differs from other approaches to deduction by Horn clause generation (e.g. [Stickel, 1988]), in that the generated clauses have no logical relation to the formula that is to be proven (ie are not a logically equivalent variant of the formula), but that they are *procedurally* equivalent to the search for a model. This is reflected by the fact that Prolog's SLD-resolution can be used and no meta-level inference is required.

It is straightforward to compile Shannon graphs into other target languages. This has already been done for C and i386-Assembler in the case of propositional logic. The performance of the version generating C code is roughly comparable to the Prolog version. This is basically due to the fact that the generated Prolog clauses have a very simple structure and can be efficiently handled by Prolog. For a first-order version, however, C can be expected to perform better, since some optimizations are hard to encode in Prolog.

The version of the prover compiling to Assembler is clearly the fastest one, yielding results about 20–30 times better than those presented in Table 1 for Prolog. If an application requires very short run-times, it would even be possible to by-pass the use of an Assembler and generate machine language, directly.

Note that the compilation process can also be done in advance for a set of axioms. Such precompiled theories can then be loaded into a deduction system and used for inference. In this context it is also reasonable to *reduce* a Shannon graph before compiling it into Prolog. Reduction is a well-known operation on Shannon graphs, which often decreases its size considerably. Though reduction may add to the cost of preprocessing, the resulting search space at run-time will be smaller.

References

[Brace *et al.*, 1990] Karl S. Brace, Richard L. Rudell, & Randal E. Bryant. Efficient implementation of a BDD package. In *Proc. 27th ACM/IEEE Design Automation Conference*, pages 40 – 45. IEEE Press, 1990.

[Bryant, 1986] Randal Y. Bryant. Graph–based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C–35:677 – 691, 1986.

[Kemper *et al.*, 1992] A. Kemper, G. Moerkotte, & M. Steinbrunn. Optimization of boolean expressions in object bases. In *Proc. Intern. Conf. on Very Large Databases*, 1992.

[Kropf & Wunderlich, 1991] T. Kropf & H.-J. Wunderlich. A common approach to test generation and hardware verification based on temporal logic. In *Proc. Intern. Test Conf.*, pages 57–66, Nashville, TN, October 1991.

[Lee, 1959] C. Lee. Representation of switching circuits by binary decision diagrams. *Bell System Technical Journal*, 38:985–999, 1959.

[Orłowska, 1969] Ewa Orłowska. Automatic theorem proving in a certain class of formulae of predicate calculus. *Bull. de L'Acad. Pol. des Sci., Série des sci. math., astr. et phys.*, XVII(3):117 – 119, 1969.

[Posegga, 1992] Joachim Posegga. First-order shannon graphs. In *Proc. Workshop on Automated Deduction / Intern. Conf. on Fifth Generation Computer Systems*, ICOT TM–1184, Tokyo, Japan, June 1992.

[Shannon, 1938] C. E. Shannon. A symbolic analysis of relay and switching circuits. *AIEE Transactions*, 67:713 – 723, 1938.

[Stickel, 1988] Mark E. Stickel. A Prolog Technology Theorem Prover. In E. Lusk & R. Overbeek, editors, *9th International Conference on Automated Deduction*, Argonne, Ill., May 1988. Springer-Verlag.

SUCCESS AND FAILURE OF EXPERT SYSTEMS IN DIFFERENT FIELDS OF INDUSTRIAL APPLICATION[1]

Reinhard Bachmann, Thomas Malsch and Susanne Ziegler
Lehrstuhl Technik und Gesellschaft, Fachbereich 11, Universität Dortmund, Postfach 500 500, 4600 Dortmund 50

Abstract

The euphoric period in the history of expert systems has definitely come to an end. It is presently time to review the efforts which have been made in this field. Oriented to our conceptual framework of knowledge transformation, some trip wires which can seriously jeopardize the success of expert-system-developing projects are described in the following.

1 Introduction

Our research interests focus on the following questions: (a) Which are the application fields where expert system technology can be regarded as successful and where does it fail? (b) What are the dimensions we have to refer to in order to explain success and failure of expert-system-developing projects?

Of course we are not the first to ask these questions (Mertens 1987; Coy, Bonsiepen 1989; Daniel, Striebel, Clemens-Schwartz 1989; Lutz, Moldaschl 1989; Hillenkamp 1989; Bullinger, Kornwachs 1990; Christaller 1991). But there are four reasons why we find expert system technology

---

[1] This contribution is a sketch of intermediate results of a current empirical research project which is financed by the German Ministry for Research and Technology and conducted by the University of Dortmund/Dept. Technology and Society. Our findings are predominantly based on first-hand information we have gathered in 56 interviews on 22 expert systems.

worth to be reexamined thoroughly. (a) The unrealistic euphoria of the 1980s has vanished. (b) There is still a lack of explanation of the discouragingly high rate of failed systems. (c) The greatest part of research in this field focuses on merely technical or economic issues and does not reflect social and organizational preconditions. (d) And if the studies do explicitly deal with these contexts (Hillenkamp 1989; Lutz, Moldaschl 1989) they do not go beyond the application side and only reflect the social impacts of technology without examining development processes. In our view it is necessary to avoid limitations of this kind.

The application fields we have investigated are all industrial ones and are listed as follows:

- Customer-specific product configuration
- Machinery fault-diagnosis
- Synthesis and analysis in the chemical laboratory
- Engineering of production processes
- Determination of manufacture planning

We have concentrated on these fields because they are considered to be major industrial application areas in Germany as well as worldwide (Mertens, Borkowski, Geis 1990). 15 of the 22 expert systems we studied in total were apt for detailed analysis. Although we have focused on systems described as "running systems" in German expert system literatureit became obvious that many of these systems were anything but successful

Table 1: The distribution of success and failure over the application fields

AREAS	SUCCESS-FUL	FAILED	IN DEVELOP-MENT	TOTALS
CONFIGURATION	4	-	-	4
MACHINERY FAULT-DIAGNOSIS	-	5	-	5
CHEMICAL LABORATORY	1	2	-	3
PROCESS ENGINEERING	1	1	-	2
MANUFACTURE PLANNING	-	-	1	1
TOTALS	6	8	1	15

running systems. As the following table shows, 8 expert systems failed either in the stage of development or later in practice. Only 6 systems can be considered successful in that they were technically functionable. (Only 4 of them, in our view, have definitely proved to be successful also in the commercial sense).

This table shows a second remarkable finding: Success and failure are distributed unevenly over the studied fields. In the configuration area, all systems which we have examined were successful whereas in the area of diagnosis (which is according to common opinion a "classic" area of application for knowledge-based systems) no system in our sample of 5 was successfully implemented. In chemical laboratory and process control we evaluated one successful and one failed system in each area. The manufacture planning system we studied was at the testing stage of application. At present we do not know of its effectiveness.

As we do not only want to classify systems in regard to their success, we shall give some explanations for the distribution of success and failure within as well as across the application fields. In other words we want to identify trip wires on the way to success. For this purpose we will adopt the theoretical concept of knowledge transformation (Malsch 1987), which serves us as a conceptual framework for the reconstruction of the process of the development and application of expert systems. The process of knowledge transformation can be segmented in three stages: (1) knowledge acquisition, (2) the "objectivation" of knowledge in the form of a technical artefact and (3) the reintegration of the knowledge represented in the expert systems into the context of application. Each of these stages contains specific risks of failure. In the following we will refer to the crucial points in the process of knowledge transformation, seemingly appropriate for revealing some important aspects in the question of success and failure of expert system projects.

2 The process of knowledge transformation

(1) The phase of knowledge acquisition involves the risk of losing or distorting knowledge. We have located several neuralgic points on the level

Author's Address

Hans Jürgen Ohlbach (Editor)
Max–Planck–Institut für Informatik
Im Stadtwald
D-6600 Saarbrücken 11
F. R. Germany
ohlbach@mpi-sb.mpg.de

Publication Notes

This is a preprint of the proceedings of the German Workshop on Artificial Intelligence (GWAI) 1992. The final version will appear in the Lecture Notes in Artificial Intelligence.