MAX-PLANCK-INSTITUT FÜR INFORMATIK

Killer Transformations

Hans Jürgen Ohlbach Dov Gabbay David Plaisted

MPI–I–94–226 June 94

INFORMATIK

Im Stadtwald
D 66123 Saarbrücken
Germany

Author's Address

Hans Jürgen Ohlbach
Max–Planck–Institut für Informatik
Im Stadtwald
D-66123 Saarbrücken
F. R. Germany
email: ohlbach@mpi-sb.mpg.de

Dov M. Gabbay
Imperial College of Science, Technology and Medicine
Dept. of Computing
Huxley Building, 180 Queen's Gate
London SW7 2AZ,
England
email: dg@doc.ic.ac.uk

David A. Plaisted
Department of Computer Science
CB 3175, 352 Sitterson Hall
University of North Carolina at Chapel Hill
Chapel Hill, North Carolina
email: plaisted@cs.unc.edu

Acknowledgements

This research was partially supported by the ESPRIT project 6471 MEDLAR and by the BMFT funded project LOGO (ITS 9102) as well as by the National Science Foundation under grant CCR-9108904. The second author is a SERC Senior Research Fellow.

Abstract

This paper deals with methods of faithful transformations between logical systems. Several methods for developing transformations of logical formulae are defined which eliminate unwanted properties from axiom systems without losing theorems. The elementary examples we present are permutation, transitivity, equivalence relation properties of predicates and congruence properties of functions. Various translations between logical systems are shown to be instances of K-transformations, for example the transition from relational to functional translation of modal logic into predicate logic, the transition from axiomatic specifications of logics via unary provability relations to a binary consequence relations, and the development of neighbourhood semantics for nonclassical propositional logics.

Furthermore we show how to eliminate self resolving clauses like the condensed detachment clause, resulting in dramatic improvements of the performance of automated theorem provers on extremely hard problems. As by–products we get a method for encoding some axioms in Prolog which normally would generate loops, and we get a method for parallelizing some closure computation algorithms.

Contents

1 Introduction

The development and investigation of logical and mathematical systems has various aspects. One aspect is of course the possibility to prove theorems about this system. Another even more important aspect is the possibility to understand a new system in terms of a well known old system. For example Stone's representation theorem [Sto36] for Boolean algebras correlates Boolean algebras (BA) with set theory. Every BA theorem can be proved by interpreting the BA connectives set theoretically and proving the formula in the algebra of sets.

Another example is the correlation between an axiomatic specification of a logic and its model theoretic semantics. The model theoretic semantics is usually described in terms of sets, functions and relations, and the interpretation function relates the properties of the logic with the intrinsic properties of the semantic structure. Again, the purpose is to describe a complex new system in terms of a simple, well known, standardized and therefore easy to communicate old system.

In all these cases we have two logical or mathematical systems $\mathcal{L}_1$ and $\mathcal{L}_2$ and a transformation $\Upsilon\colon \mathcal{L}_1 \to \mathcal{L}_2$. The minimal property, Υ should have is that it allows us to solve a problem of system $\mathcal{L}_1$ by solving a corresponding transformed problem in $\mathcal{L}_2$. That means if we have a problem in $\mathcal{L}_1$, and we use Υ to transform the problem into $\mathcal{L}_2$, find a solution there, and transform the solution back into $\mathcal{L}_1$, then it must be guaranteed that this is in fact a solution of the original problem. This is the soundness requirement for Υ. Completeness of Υ, on the other hand, means that we can solve *all* $\mathcal{L}_1$ problems this way. If the problem is a theorem proving problem this means that the following must be guaranteed:

$$\begin{array}{lrcl} \mathcal{L}_1\colon & \mathit{Assumption} & \Rightarrow & \mathit{Conclusion} \\ & & \text{iff} & \\ \mathcal{L}_2\colon & \Upsilon(\mathit{Assumption}) & \Rightarrow & \Upsilon(\mathit{Conclusion}). \end{array} \tag{1}$$

A further requirement for Υ should be that it simplifies matters in some sense, but in what sense? In order to get a first idea what can be simplified by a transformation, consider the well known correspondence between syntax and semantics of predicate logic. In every logic textbook you find definitions of logical connectives as for example

$$\Im \models A \wedge B \quad \text{iff} \quad \Im \models A \text{ and } \Im \models B$$

i.e the formula $A \wedge B$ with top–level connective '$\wedge$' is true in an interpretation iff both A *and* B are true. The key point is that we map the connective '$\wedge$' to the meta logical word 'and' which everybody understands. A consequence is that by this definition, '$\wedge$' inherits all the properties of 'and', in particular commutativity, idempotence and associativity of 'and'.

By mapping a component η of the system $\mathcal{L}_1$ to a component $\Upsilon(\eta)$ of $\mathcal{L}_2$, it is no longer necessary to mention the properties, η can inherit from $\Upsilon(\eta)$ any more. They are automatically true. And this illustrates the 'simplicity criterion': a transformation $\Upsilon\colon \mathcal{L}_1 \to \mathcal{L}_2$ is better if more explicit properties of a component η of the system $\mathcal{L}_1$ are true for $\Upsilon(\eta)$.

All properties of η which are not automatically true for $\Upsilon(\eta)$ require that the target system $\mathcal{L}_2$ has to be restricted such that these extra properties hold, and this usually complicates matters. A concrete example where this happens is Kripke's possible worlds semantics for modal logic. All axioms of the system K ($= \mathcal{L}_1$) are automatically true in all Kripke frames ($= \mathcal{L}_2$). Here we have an optimal transition from the syntactic system K to the semantic system. Any additional axiom, however, requires extra conditions on the frames, which makes this semantics less and less useful.

The guideline for finding good transformations from $\mathcal{L}_1$ to $\mathcal{L}_2$ is therefore the intention to 'eliminate' (or 'kill', therefore the name *Killer Transformation*) properties ϕ from $\mathcal{L}_1$ in the sense that $\Upsilon(\phi)$ is either just a tautology or in some other sense redundant or useless in $\mathcal{L}_2$.

The transformations we are going to present take place in the framework of predicate logic. Using predicate logic as a meta logic for other systems, however, the same ideas work for many other logics as well. In predicate logic, theorems can be proved by refutation. That means instead of proving that $\mathit{Assumption} \Rightarrow \mathit{Conclusion}$ is a tautology, we prove that $\mathit{Assumption} \wedge \neg\mathit{Conclusion}$ is unsatisfiable. When we transform the problem this means that we have to make sure that

$$\begin{array}{llr} \mathcal{L}_1: & \quad \mathit{Assumption} \wedge \neg \mathit{Conclusion} \quad \text{is unsatisfiable} & \\ & \qquad\qquad\qquad \text{iff} & (2) \\ \mathcal{L}_2: & \Upsilon(\mathit{Assumption} \wedge \neg \mathit{Conclusion}) \quad \text{is unsatisfiable.} & \end{array}$$

The important observation is now that it is not necessary that the transformation is equivalence preserving. It is sufficient to be *faithful*. That means if the original problem formulation has a model then so has the transformed problem (soundness of the transformation) and vice versa (completeness of the transformation).

This relatively weak condition on the transformation opens a door to a whole universe of possibilities for transformations of which only a tiny fragment has been routinely exploited so far. Skolemization of existential quantifiers during conjunctive normal form generation is an example for a routinely applied faithful transformation which is not equivalence preserving.

In this paper we present general methods, a kind of recipe, for developing faithful transformations of logical systems and provide some general soundness and completeness results. With a lot of examples we demonstrate that, following our recipes, the development of particular transformations together with the necessary proofs becomes almost trivial.

Three stages of transformation methods will be considered. In Section 2 we focus on transformations which eliminate formulae φ from formula sets Φ by using equivalences $\varphi \Leftrightarrow \psi$ and replacing every instance of φ with a corresponding instance of ψ, or, if φ also occurs in ψ by some more complex operations. While the transformation itself is trivial in most cases, the main problem we consider is where to get these equivalences from. It turns out that not the elimination of φ itself is the important part, but to find the right ψ such that, as we mentioned earlier, particular axioms η in $\Upsilon(\Phi)$ become tautologies. The elimination of φ is therefore only the vehicle for eliminating η.

A number of examples for formula transformations are listed in Section 3. The elimination of properties like symmetry, transitivity, permutation properties, and the equivalence relation property are simple cases, but with sometimes surprising applications. As we shall see, the elimination of reflexivity and transitivity of a binary relation, in particular the binary consequence relation of an axiomatic specification of a logic very naturally leads to the introduction of possible worlds semantics.

There are more examples which turn out to be instances of our transformation technique. One is the transformation of n–ary relations to binary relations, a standard trick in relational database applications. A further example is of quite different nature. It describes the transition from the relational translation of modal logic to predicate logic, to the functional translation. That means in this case we show with our technique how to transform transformations. Even more surprising are the last two examples we give. We can explain the transition from a Hilbert style axiomatization of a logic with a unary theoremhood predicate to an axiomatization in terms of a binary consequence relation. And finally in the last example we derive neighbourhood semantics for non–classical logics as the result of the elimination of certain congruence properties of logical connectives.

In Section 4 we focus on the elimination of given clauses from a clause set, in particular those clauses which usually give automated theorem provers a hard time, self resolving or recursive clauses. One of the clauses we consider as examples is the condensed detachment clause, a predicate logic encoding of the Modus Ponens rule:

$$P(i(x,y)) \wedge P(x) \Rightarrow P(y). \tag{3}$$

As we shall demonstrate, the elimination of this clause can lead to dramatic improvements of the performance of theorem provers on some of the hardest challenge problems discussed in the theorem proving literature. One effect we observed was that the elimination of clauses like condensed detachment may reveal great redundancies implicitly contained in the original formulation. The transformed clause set can be simplified before the proof search starts and this reduces the search space considerably.

A by–product of the method for elimination of clauses is a method for transforming Prolog programs in order to eliminate certain loops. As another by–product we obtained a method for parallelizing certain closure computation algorithms and proving their correctness.

Finally in the last section we investigate the elimination of particular literals. For example, given a literal like $\mathit{subset}(x,y)$, can we detect and use a definition of the *subset* predicate:

$subset(x,y) \Leftrightarrow \forall(z\ z \in x \Rightarrow z \in y)$, which is not explicitly given, but hidden in the anonymous literal set of a clause normal form? This method supports Beth's definability theorem in a constructive way.

Notions and Notation

Some notions and notations are needed in this paper which are taken from the theorem proving literature. Letters from the end of the alphabet usually denote variables. Letters from the beginning of the alphabet denote constant symbols and letters f, g, h, i denote function symbols. s and t usually denote arbitrary terms.

A *clause* is a disjunction of literals. If A_i are the negative literals and B_j are the positive literals, we can write clauses in three different ways, either as a disjunction, $\neg A_1 \vee \ldots \vee \neg A_n \vee B_1 \vee \ldots \vee B_m$ or as an implication $A_1 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee \ldots \vee B_m$ or as a set $\{\neg A_1, \ldots, \neg A_n, B_1, \ldots, B_m\}$. We shall use all three versions. The variables in clauses are always considered universally quantified. Two different clauses are always considered as variable disjoint. If they are not, take variable renamed copies. A *unit clause* or *unary clause* is a clause with one literal only.

If L is a literal then $\overline{L}$ denotes its complement, i.e. the atom is the same, but the sign is exchanged.

A *substitution* σ is an endomorphism in the free term algebra which changes only finitely many variables. We write substitutions as sets $\sigma = \{x_1 \mapsto s_1, x_2 \mapsto s_2, \ldots\}$. The variables x_i form the *domain* and the terms t_i form the *codomain* of σ. A *renaming substitution* is a substitution mapping variables to variables such that the codomain and the domain are disjoint and the codomain variables are all different. An example is $\{x \mapsto u, y \mapsto v\}$. $\{x \mapsto u, y \mapsto u\}$ is not a renaming substitution. $s\sigma$ denotes the application of the substitution σ to the term s. $\sigma\tau$ denotes the composition of the two substitutions σ and τ.

$mgu(s,t)$ is the *most general unifier* for the two terms or atoms s and t. That means if $\sigma = mgu(s,t)$ then $s\sigma = t\sigma$. Since we do not consider theory unification, there is, up to variable renaming, at most one most general unifier for two terms. Two literals are *complementary unifiable* if they have different signs and the atoms are unifiable. A substitution μ is a *matcher* for the two terms s and t iff $s\mu = t$.

A clause C *subsumes* a clause D iff $C\mu = D$ for some substitution μ and C has less or equally many literals as D. Subsumption is a weaker version of implication.

Resolution is the standard inference rule for many theorem provers [Rob65b]. The definition of the resolution rule is

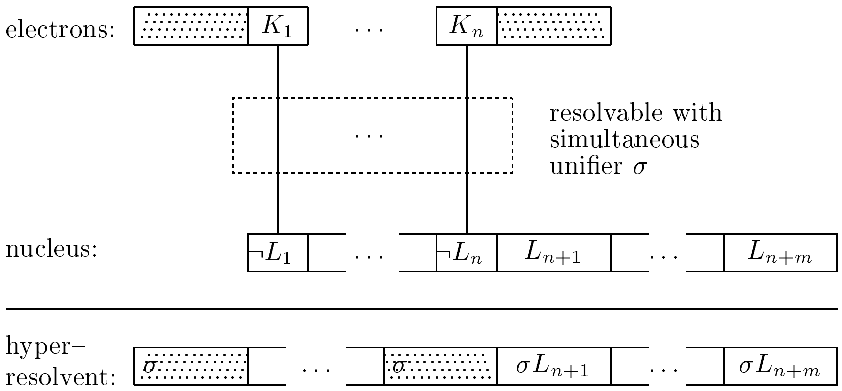
$$\begin{array}{lll} C_1: & L_1 \vee L_2 \vee \ldots \vee L_n & L_1 \text{ and } K_1 \text{ are complementary unifiable.} \\ C_2: & K_1 \vee K_2 \vee \ldots \vee K_m & \sigma \text{ is the most general unifier} \\ \hline & \sigma(L_2 \vee \ldots \vee L_n \vee K_2 \vee \ldots \vee K_m) & \end{array}$$

L_1 and K_1 are the *resolution literals.* We say that we *resolve* the clauses C_1 and C_2 *upon* the resolution literals L_1 and K_1.

Self resolution is a resolution operation between two variable renamed copies of the same clause. For example a self resolution with the transitivity clause is

$$\begin{array}{l} \neg P(x,y) \vee \neg P(y,z) \vee \quad P(x,z) \\ \qquad\qquad\qquad\qquad\qquad | \\ \qquad\qquad\qquad\quad \neg P(x',y') \vee \neg P(y',z') \vee P(x',z') \\ \hline \neg P(x,y) \vee \neg P(y,z) \vee \neg P(z,z') \vee P(x,z') \end{array}$$

Hyperresolution is the combination of several resolution steps into one step. It was developed by John Alan Robinson [Rob65a] and is described by the following schema:



Here a clause with at least one negative literal serves as 'nucleus'. For every negative literal of the nucleus, a so–called 'electron' is needed, a clause containing only positive literals. The nucleus is resolved with all electrons simultaneously, resulting in a purely positive clause, which in turn can be used as an electron for the next *positive* hyperresolution step.

2 Elimination of Formulae

In order to illustrate the basic idea of this section, suppose we have some first order predicate logic axioms Φ which, among other things, axiomatize a reflexive and transitive relation R, i.e.

$$\forall x\ R(x,x) \tag{4}$$
$$\forall x,y,z\ R(x,y) \wedge R(y,z) \Rightarrow R(x,z) \tag{5}$$

is either contained in Φ or derivable from Φ, and we want to get rid of the reflexivity and transitivity of R.

In order to show that a formula C is entailed by Φ, one usually tries to refute $\Phi \wedge \neg C$. Before the refutation is actually started, every *faithful* transformation of $\Phi \wedge \neg C$, i.e. a transformation preserving satisfiability and unsatisfiability, is allowed.

The translation we propose for faithfully eliminating reflexivity and transitivity of R exploits that these two properties together imply

$$\forall x,y\ R(x,y) \Leftrightarrow (\forall w\ R(w,x) \Rightarrow R(w,y)). \tag{6}$$

To see this, suppose $R(x,y)$ and $R(w,x)$ hold. By transitivity, $R(w,y)$ also holds, i.e. the '$\Rightarrow$'–part is shown. For the '$\Leftarrow$'–part, take $w = x$ and use the reflexivity of R to derive $R(x,y)$.

Since (6) is entailed by Φ, we could add it to $\Phi \wedge \neg C$ without losing satisfiability or unsatisfiability. However, instead of (6), we add

$$\forall x,y\ R(x,y) \Leftrightarrow (\forall w\ R'(w,x) \Rightarrow R'(w,y)) \tag{7}$$

to $\Phi \wedge \neg C$ where R' is a *new* predicate symbol. Clearly, if $\Phi \wedge \neg C$ is satisfiable then $\Phi \wedge \neg C \wedge (7)$ is also satisfiable: the interpretation of R' can be chosen to be the same as the interpretation of R. In this case (7) is equivalent to (6), which follows from Φ. Thus, (7) is also true in the extended interpretation. On the other hand, if $\Phi \wedge \neg C \wedge (7)$ is satisfiable then certainly $\Phi \wedge \neg C$ is satisfiable as well.

But now we have a definition of R in terms of R' where R' is an uninterpreted new predicate symbol. In the next step, (7) is used as a rewrite rule from left to right, replacing all occurrences of R in $\Phi \wedge \neg C$ by the formula with R'. We obtain the transformed formula $\Phi' \wedge \neg C' \wedge (7)$ with R' in place of R. This is a terminating equivalence preserving transformation.

What happens to the reflexivity and transitivity of R? $\forall x\ R(x,x)$ becomes

$$\forall x\ \forall w\ R'(w,x) \Rightarrow R'(w,x)$$

which is a tautology (by the reflexivity of '$\Rightarrow$'). The transitivity (5) becomes

$$\forall x, y, z\ (\forall w\ R'(w,x) \Rightarrow R'(w,y)) \wedge (\forall w\ R'(w,y) \Rightarrow R'(w,z)) \Rightarrow (\forall w\ R'(w,x) \Rightarrow R'(w,z))$$

which is also a tautology (by the transitivity of '$\Rightarrow$'). Thus, R' needs neither be reflexive nor transitive.

Nothing would have been gained if the definition of R, (7), could not be removed afterwards. In order to justify removing the transformer we have to show that $\Phi' \wedge \neg C' \wedge (7)$ is satisfiable if and only if $\Phi' \wedge \neg C'$ is satisfiable. Since $\Phi' \wedge \neg C'$ does not contain R any more, we can always find an interpretation for R, using (7) as definition. Therefore each model for $\Phi' \wedge \neg C'$ can be extended to a model for $\Phi' \wedge \neg C' \wedge (7)$. Thus, (7) can be eliminated. $\Phi' \wedge \neg C'$ is the final result of our transformation.

What has actually happened is that the role of the reflexivity and transitivity of R has been taken over by the reflexivity and transitivity of the implication connective. Many of the other examples we present in the paper are of a similar kind. The built–in properties of predicate logic take over the role of special properties of non–logical symbols.

The first procedure for transforming formulae Φ, typically axioms and a negated theorem, in a *faithful* way, i.e. without losing satisfiability or unsatisfiability, consists of the following sequence of steps

$$\Phi \xrightarrow{\text{extension}} \Phi\ \wedge \text{transformer} \xrightarrow{\text{transformation}} \Phi'\ \wedge \text{transformer} \xrightarrow{\text{elimination}} \Phi'$$

where 'transformer' is a formula of the kind

$$Left \Leftrightarrow Right. \tag{8}$$

The 'extension' step involves finding the transformer. In general this is still a creative step, but we shall give some heuristics and more concrete procedures for finding the transformers. The actual transformation is done in the 'transformation' step. In the simplest case the transformation strategy is just definitional replacement where (8) is used as rewrite rule from left to right. It can, however, also be a much more complex combination of rewriting, inferencing and deleting formulae. In the simplest case the transformation strategy is just definitional replacement where (8) is used as rewrite rule from left to right. It can, however, also be a much more complex combination of rewriting, inferences and deletions. In the elimination step we delete the transformer. Since removing formulae can turn unsatisfiable formula sets into satisfiable sets, this is also a non–trivial step which has to be justified.

In the sequel we call these transformations *formula K-transformations.* They are determined by the transformer $Left \Leftrightarrow Right$ and the transformation strategy. To ensure that the transformation is satisfiability preserving, which is sufficient to do theorem proving by refutation, the following lemmas have to be proved.

Definition 2.1 (Transformation Lemmas)
The *extension lemma* proves that satisfiability of Φ implies satisfiability of $\Phi \wedge$ transformer.
The *transformation lemma* proves that $\Phi \wedge$ transformer is satisfiable if and only if $\Phi' \wedge$ transformer is satisfiable, where Φ' is the transformed version of Φ.
The *elimination lemma* proves that satisfiability of Φ' implies satisfiability of $\Phi'\ \wedge$ transformer. $\lhd$

Lemma 2.2 (Faithfulness)
A K-transformation for which the extension lemma, the transformation lemma, and the elimination lemma have been proved, preserves satisfiability and unsatisfiability. We call this a *faithful* transformation. It is *sound and complete.*

Proof: The other direction of the extension lemma, satisfiability of $\Phi\ \wedge$ transformer implies satisfiability of Φ is obvious, which is also the case for the other direction of the elimination lemma, Φ' $\wedge$ transformer implies satisfiability of Φ'. Thus, the three lemmas guarantee that all three steps preserve satisfiability. $\lhd$

This lemma holds for every monotonic logical system with a suitable notion of satisfiability. There are standard cases which occur quite frequently. For these cases we can prove some of the lemmas once and for all. In an actual case, it has only to be checked whether the transformer is one of these standard types.

Definition 2.3 (Renamed Transformer)
A transformer $Left \Leftrightarrow Right$ is called a *renamed transformer* for a formula Φ iff there is a formula $Left \Leftrightarrow Right_0$ entailed by Φ and $Right$ is obtained from $Right_0$ by renaming constant, function, predicate symbols and sorts with new symbols not occurring in Φ so far. Different occurrences of the same symbol in $Right$ may be renamed differently. $Left \Leftrightarrow Right_0$ is called the *basis* of the transformer. ◁

Lemma 2.4 (Extension Lemma for Renamed Transformers)
The extension lemma (Def. 2.1) holds for renamed transformers.

Proof: We show that every model $\Im$ of Φ can be extended to a model $\Im'$ for $\Phi \wedge (Left \Leftrightarrow Right)$ where $Right$ is a renamed variant of some $Right_0$. We interpret in $\Im'$ all the new symbols in $Right$ exactly like the original symbols in $\Im$ and find that $Right$ and $Right_0$ are equivalent under $\Im'$. Since $Left \Leftrightarrow Right_0$ is entailed by Φ, and $\Im'$ is an extension of $\Im$, $\Im'$ satisfies $Left \Leftrightarrow Right_0$, and therefore it satisfies $Left \Leftrightarrow Right$ as well. ◁

The proof of the extension lemma for this type of transformers amounts to proving that the basis $Left \Leftrightarrow Right_0$ follows from Φ. This is a standard theorem proving task which can be done with automated theorem provers.

Remark

In the new interpretation $\Im'$ the renamed symbols have been interpreted like the original ones. In this interpretation all the properties of the old symbols also hold for the renamed symbols. This means that axioms describing these properties for the new symbols could also be added without changing the satisfiability or unsatisfiability. They are not necessary for finding proofs, but they can shorten proofs and they can be helpful in successive transformations. In our standard example, reflexivity and transitivity of R, this means that we could add for example the transitivity clause for the new predicate R' to the transformed formula set. In this particular case this does not make much sense, but we shall present other examples where this is quite useful. We summerize this observation in a corollary.

Corollary 2.5 (Optional Axioms)
After a transformation with a renamed transformer, all formulae describing properties of the original symbols may be renamed in the same way as in the renamed transformer and added to the transformed set without changing satisfiability and unsatisfiability. ◁

The transformer (7) for the elimination of reflexivity and transitivity is an example for a renamed transformer. The proof of $Left \Leftrightarrow Right_0$, i.e. (6) is trivial in this case.

In the class of transformers specified in the next definition, the transformation process itself is reduced to a simple rewriting operation.

Definition 2.6 (Rewriting Transformers)
A transformer $\forall x_1, \ldots, x_n\ R(x_1, \ldots, x_n) \Leftrightarrow Right$ where R does not occur in $Right$, is called a *rewriting transformer*. The transformation strategy for rewriting transformers is just definitional replacement, i.e. all occurrences $R(s_1, \ldots, s_n)$ are replaced with the corresponding instances of $Right$. ◁

Lemma 2.7 (Transformation Lemmas for Rewriting Transformers)
The transformation lemma and the elimination lemma (Def. 2.1) hold for rewriting transformers.

Proof: The transformation lemma holds because rewriting with an equivalence is an equivalence preserving operation. The elimination lemma holds as well because the transformed formula Φ' does not contain the predicate R any more. Therefore the transformer itself can be taken as a definition for R. Each model of Φ' can be extended to a model for $\Phi' \wedge \forall x_1, \ldots, x_n\ R(x_1, \ldots, x_n) \Leftrightarrow Right$ by defining the interpretation of R according to just this equivalence. $\lhd$

Renamed rewriting transformers are the simplest transformers at all. The only thing which has to be proved for this class of transformers is that the basis $Left \Leftrightarrow Right_0$ for the transformer follows from Φ. The transformer (7) turns out to be of this simple type.

The next class of transformers is of a more general nature. It allows us to exploit completeness results for special inference strategies, as for example ordered resolution. (41) is an example for a transformer of this kind.

Definition 2.8 (Saturation Transformers)
A transformer $Left \Leftrightarrow Right$ is called a saturation transformer for the formula Φ if there is a refutation complete deduction strategy which allows one to draw inferences between the transformer and Φ only finitely many times.
Φ is transformed by drawing inferences between the transformer and Φ according to this strategy exhaustively and removing all formulae in the transformed version of Φ which are redundant, i.e. no longer necessary for finding a contradiction. $\lhd$

Lemma 2.9 (Transformation Lemmas for Saturation Transformers)
The transformation lemma and the elimination lemma (Def. 2.1) hold for saturation transformers.

Proof: Adding derived formulae preserves satisfiability and unsatisfiability. Removing redundant formulae also preserves satisfiability and unsatisfiability. Therefore the transformation lemma holds.
Since the deduction strategy which is applied during the transformation is refutation complete, and, according to the strategy, there are no further inferences with the transformer possible, the transformer is no longer necessary for finding a refutation. It becomes redundant. If the transformed formula together with the transformer is unsatisfiable then the transformed formula without the transformer is still unsatisfiable, and vice versa. Thus, the elimination lemma holds as well. $\lhd$

Notice that there are refutation complete strategies which may deactivate a formula A at a certain point during proof search and activate it later on again. That means there is no inferences possible with A, but later on after operations in other parts of the formula set, inferences between A and newly generated formulae may become possible again. In order to show that a transformer is of saturation type one has therefore to show that after the initial operations the transformer never again are activated.

The formula Φ which is transformed usually consists of the two parts, $Assumption$ and $\neg Conjecture$. At the time of the development of the K-transformation, usually only the assumption is known. Therefore the transformation lemma has to be proved for all potential conjectures. Alternatively one can specify the class of admissible conjectures for which the K-transformation works.

3 Examples for Formula Transformations

The transformation technique with formula K-transformations is illustrated with various examples. We also show that some well known transformations are actually instances of formula K-transformations.

3.1 Symmetry and Permutations

In the introduction we have shown how the reflexivity and transitivity of a relation R is absorbed by the reflexivity and transitivity or the predicate logic implication connective. Once this idea has become clear, it is quite trivial to develop K-transformations for other basic properties of relations.

The symmetry of a predicate R for example can be mapped to the commutativity of '$\wedge$' or '$\vee$' by exploiting that symmetry implies

$$\forall x, y\ R(x,y) \Leftrightarrow (R(x,y) \wedge R(y,x)) \qquad \text{and} \tag{9}$$

$$\forall x, y\ R(x,y) \Leftrightarrow (R(x,y) \vee R(y,x)). \tag{10}$$

Renaming R to R' we get the two possible admissible transformers

$$\forall x, y\ R(x,y) \Leftrightarrow (R'(x,y) \wedge R'(y,x)) \tag{11}$$

$$\forall x, y\ R(x,y) \Leftrightarrow (R'(x,y) \vee R'(y,x)). \tag{12}$$

The transformer is a renamed (Def. 2.3) rewriting (Def. 2.6) transformer and therefore faithful.

To give an intuitive example for (11), suppose $R = \approx$ and $R' = \leq$. Then (11) reads as

$$x \approx y \Leftrightarrow (x \leq y \wedge y \leq x),$$

which could for example be a part of the definition of an equivalence relation from an ordering relation.

We can even eliminate more complex permutations as symmetry. For example the permutation

$$\forall x, y, z\ R(x,y,z) \Rightarrow R(y,z,x) \tag{13}$$

implies

$$\forall x, y, z\ R(x,y,z) \Leftrightarrow (R(x,y,z) \wedge R(y,z,x) \wedge R(z,x,y)) \tag{14}$$

which yields the transformer

$$\forall x, y, z\ R(x,y,z) \Leftrightarrow (R'(x,y,z) \wedge R'(y,z,x) \wedge R'(z,x,y)). \tag{15}$$

This transformation rewrites (13) to a tautology, i.e. R' needs not have the permutation property any more. It is quite obvious how the transformation idea can be applied to other permutations. The equivalence corresponding to (14) simply lists all the permutation cases explicitly.

From a theorem proving point of view, however, this particular transformation of permutation properties is not optimal. In Section 4.5 it is shown that only positive instances of R–literals need to be transformed.

3.2 Reflexivity and Transitivity

The method for eliminating reflexivity and transitivity has been explained in the introduction. A transformer is

$$\forall x, y\ R(x,y) \Leftrightarrow (\forall w\ R'(w,x) \Rightarrow R'(w,y)). \tag{16}$$

But we can go a step further and introduce a new sort W for the variable w:

$$\forall x, y\ R(x,y) \Leftrightarrow (\forall w{:}W\ R'(w,x) \Rightarrow R'(w,y)) \tag{17}$$

which is still a renamed rewriting transformer and therefore faithful.

An example where this kind of transformer occurs very naturally is the correspondence between the subset relation $\subseteq$ for R, and the membership relation $\in$ for R':

$$\forall x, y\ x \subseteq y \Leftrightarrow (\forall w{:}W\ w \in x \Rightarrow w \in y)$$

Another example for this kind of transformer is the correspondence between a reflexive and transitive binary consequence relation $\vdash$ (for R) of a logic and a satisfiability relation $\models$ (for R'):

$$\forall x, y\ x \vdash y \Leftrightarrow (\forall w{:}W\ w \models x \Rightarrow w \models y).$$

The intended interpretation of this equivalence is: y is a consequence of x if and only if, whenever x is true in a *world* w ($w \models x$) then y is true in w ($w \models y$). This can be seen as the origin of the possible worlds framework of many non–classical logics. In [Ohl94] this idea is further developed.

Besides (17) there is a dual transformer for reflexivity and transitivity. These two properties also imply

$$\forall x, y\ R(x, y) \Leftrightarrow \forall z\ R(y, z) \Rightarrow R(x, z) \tag{18}$$

which gives rise to the transformer

$$\forall x, y\ R(x, y) \Leftrightarrow \forall w{:}W\ R'(y, w) \Rightarrow R'(x, w). \tag{19}$$

An example for this transformer is the definition of the superset relation in terms of a 'contains' relation.

The transformer (16) and (19) implicitly encode the reflexivity of R. If R is only transitive and not reflexive, we must change it a bit. The transitivity of R alone implies

$$\forall x, y\ R(x, y) \Leftrightarrow (R(x, y) \wedge \forall w\ R(w, x) \Rightarrow R(w, y)) \qquad \text{and} \tag{20}$$

$$\forall x, y\ R(x, y) \Leftrightarrow (R(x, y) \wedge \forall w\ R(y, w) \Rightarrow R(x, w)) \tag{21}$$

which gives rise to the transformers

$$\forall x, y\ R(x, y) \Leftrightarrow (R'(x, y) \wedge \forall w{:}W\ R'(w, x) \Rightarrow R'(w, y)) \tag{22}$$

$$\forall x, y\ R(x, y) \Leftrightarrow (R'(x, y) \wedge \forall w{:}W\ R'(y, w) \Rightarrow R'(x, w)). \tag{23}$$

We do not propose to use these transformations for theorem proving purposes. As we shall see in Section 4.5 there are more efficient versions for the reflexive and transitive case.

3.3 Equivalence Relations

For *equivalence relations*, i.e. reflexive, transitive and symmetric relations, we find a very simple transformer because the '$\Leftrightarrow$' connective represents an equivalence relation. From reflexivity, symmetry and transitivity of R we can prove

$$\forall x, y\ R(x, y) \Leftrightarrow (\forall w\ R(w, x) \Leftrightarrow R(w, y)) \tag{24}$$

which is turned into the transformer

$$\forall x, y\ R(x, y) \Leftrightarrow (\forall w{:}W\ R'(w, x) \Leftrightarrow R'(w, y)). \tag{25}$$

3.4 n-ary Relations

It is well known that n–ary relations can be replaced by n binary relations. This is a standard technique for example in relational database applications. We show that this transformation also fits into our framework. The transformer for eliminating ternary relations is:

$$\forall x, y, z\ R(x, y, z) \Leftrightarrow (\exists w{:}T\ R_1(w, x) \wedge R_2(w, y) \wedge R_3(w, z)). \tag{26}$$

This is a rewriting transformer, but not a renamed transformer. Therefore we have to prove the extension lemma (Def. 2.1). That means we must be able to extend every interpretation $\Im$ for R to an interpretation for the new sort T and the new predicates R_1, R_2 and R_3 such that (26) becomes true.

Suppose we have an interpretation $\Im$ for R. Let D be the domain of $\Im$. In the extended interpretation $\Im'$ we interpreted the sort symbol T as the set of triples in D^3. If in addition the new predicates are interpreted

$$\begin{aligned}
\Im'(R_1) &= \{((x, y, z), x) \mid (x, y, z) \in \Im(R)\} \\
\Im'(R_2) &= \{((x, y, z), y) \mid (x, y, z) \in \Im(R)\} \\
\Im'(R_3) &= \{((x, y, z), z) \mid (x, y, z) \in \Im(R)\},
\end{aligned}$$

it is straightforward to show that the extended interpretation satisfies (26). Thus, the extension lemma holds. The elimination of arbitrary n–ary relations for $n > 2$ is now obvious.

3.5 Functional Representation of Binary Relations

Via their possible worlds semantics, a number of non–classical logics can be translated into predicate logic. For example the semantics of the modal operators [Che80]

$$x \models \Box P \quad \text{iff} \quad \forall y\; R(x,y) \Rightarrow y \models P \tag{27}$$

$$x \models \Diamond P \quad \text{iff} \quad \exists y\; R(x,y) \wedge y \models P \tag{28}$$

gives rise to the relational translation of propositional modal logic.

$$\begin{array}{lcll} tr_r(P,w) & = & P'(w) & P \text{ a predicate symbol} \\ tr_r(\Box\varphi,w) & = & \forall v\; R(w,v) \Rightarrow tr_r(\varphi,v) & \\ tr_r(\Diamond\varphi,w) & = & \exists v\; R(w,v) \wedge tr_r(\varphi,v). & \end{array}$$

The relation R occurs only in the typical patterns

$$\begin{array}{l} \forall v\; R(w,v) \Rightarrow tr_r(\varphi,v) \\ \exists v\; R(w,v) \wedge tr_r(\varphi,v) \end{array}$$

in the translated formulae. Additionaly it may occur in some characteristic axioms axiomatizing properties of R itself. Since this is a quite typical pattern which appears not only in this application, it is worthwhile to look for a K-transformation to eliminate this relation. The transformation we need depends on whether the relation is serial (i.e. $\forall x \exists y\; R(x,y)$ holds) or not. For the serial case the transformer is

$$\forall x,y\; R(x,y) \Leftrightarrow \exists \gamma{:}AF\;\; y = apply(\gamma,x) \tag{29}$$

and for the non–serial case it is a bit more complicated

$$\forall x,y\; R(x,y) \Leftrightarrow End(x) \vee \exists \gamma{:}AF\;\; y = apply(\gamma,x). \tag{30}$$

Intuitively, the sort AF denotes the set of *accessibility functions*, i.e. functions mapping worlds to accessible worlds. *apply* is the application function and the *End* predicate marks worlds without successor literals.

Both are rewriting transformers (Def. 2.6). Therefore we need only to prove the extension lemma (Def. 2.1).

Lemma 3.1 (Extension Lemma for (29) and (30))
Every interpretation $\Im$ can be extended to a model for (29) and (30).

Proof: Suppose there is a model $\Im$ for the formula Φ to be transformed. We construct an extension $\Im'$ of $\Im$ satisfying (29) or (30) respectively as well. If D is $\Im$'s domain we define

$$\begin{array}{lll} \text{non–serial case:} & \Im'(AF) & \stackrel{\text{def}}{=} \{\gamma \in D \to D \mid \forall x\; R(x,\gamma(x))\} \\ \text{serial case:} & \Im'(AF) & \stackrel{\text{def}}{=} \{\gamma \in D \to D \mid \forall x\; R(x,\gamma(x)), \gamma \text{ is total}\} \\ \text{both cases:} & \Im'(apply) & \stackrel{\text{def}}{=} \text{apply function, i.e. } \Im'(apply(\gamma,x)) = \Im'(\gamma)(\Im'(x)) \\ & \Im'(End) & \stackrel{\text{def}}{=} \{x \in D \mid \neg\exists y\; R(x,y)\}. \end{array}$$

Obviously, $\Im'$ satisfies (29) and (30), which proves the extension lemma. $\lhd$

The interpretation of the sort AF as the set of 'accessibility functions' i.e. the set of functions mapping worlds to accessible worlds (in modal logic terminology) means that we can write $\gamma(x)$ instead of $apply(\gamma,x)$, but *this is to be understood as an abbreviation.* Notice that in the proof of the elimination lemma (Def. 2.7), instantiated for our case, the interpretation of R is reconstructed from an arbitrary interpretation of AF and *apply*. There need not be functions at all.

From (29) we can prove

$$\forall w\; \forall v\; R(w,v) \Rightarrow \varphi(v) \quad \Leftrightarrow \quad \forall \gamma{:}AF\;\; \varphi(\gamma(w)) \tag{31}$$

$$\forall w\; \exists v\; R(w,v) \wedge \varphi(v) \quad \Leftrightarrow \quad \exists \gamma{:}AF\;\; \varphi(\gamma(w)) \tag{32}$$

and from (30) we can prove

$$\begin{aligned} \forall w\ \forall v\ R(w,v) \Rightarrow \varphi(v) \quad &\Leftrightarrow \quad End(w) \vee \forall\gamma{:}AF\ \varphi(\gamma(w)) && (33) \\ \forall w\ \exists v\ R(w,v) \wedge \varphi(v) \quad &\Leftrightarrow \quad \neg End(w) \wedge \exists\gamma{:}AF\ \varphi(\gamma(w)). && (34) \end{aligned}$$

The transformation strategy is now not just simple definitional replacement with (29) and (30). Instead we use the derived equivalences (31, 32, 33, 34) wherever possible first to rewrite quantifications as a whole. Since they are equivalences, this is of course again an equivalence preserving transformation.

The original transformers (29) and (30) have to be applied to the characteristic axioms of R. We give some examples, always assuming seriality.

$\forall x\ R(x,x)$ (reflexivity)
$\to \quad \forall x\ \exists\gamma\ x = \gamma(x)$

$\forall x,y\ R(x,y) \Rightarrow R(y,x)$ (symmetry)
$\to \quad \forall x,y\ (\exists\gamma\ y = \gamma(x)) \Rightarrow (\exists\delta\ x = \delta(y))$
$\Leftrightarrow \quad \forall x,y\ \forall\gamma\ \exists\delta\ y = \gamma(x) \Rightarrow x = \delta(y)$
$\Leftrightarrow \quad \forall x\ \forall\gamma\ \exists\delta\ x = \delta(\gamma(x))$

$\forall x,y,z\ R(x,y) \wedge R(y,z) \Rightarrow R(x,z)$ (transitivity)
$\to \quad \forall x,y,z\ (\exists\gamma\ y = \gamma(x)) \wedge (\exists\delta\ z = \delta(y)) \Rightarrow (\exists\xi\ z = \xi(x))$
$\Leftrightarrow \quad \forall x,y,z\ \forall\gamma,\delta\ \exists\xi\ y = \gamma(x) \wedge z = \delta(y) \Rightarrow z = \xi(x)$
$\Leftrightarrow \quad \forall x,z\ \forall\gamma,\delta\ \exists\xi\ z = \delta(\gamma(x)) \Rightarrow z = \xi(x)$
$\Leftrightarrow \quad \forall x\ \forall\gamma,\delta\ \exists\xi\ \delta(\gamma(x)) = \xi(x)$

$\forall x,y,z\ R(x,y) \wedge R(x,z) \Rightarrow R(y,z)$ (euclideanness)
$\to \quad \forall x,y,z\ (\exists\gamma\ y = \gamma(x)) \wedge (\exists\delta\ z = \delta(x)) \Rightarrow (\exists\xi\ z = \xi(y))$
$\Leftrightarrow \quad \forall x,y,z\ \forall\gamma,\delta\ \exists\xi\ y = \gamma(x) \wedge z = \delta(x) \Rightarrow z = \xi(y)$
$\Leftrightarrow \quad \forall x,y,z\ \forall\gamma,\delta\ \exists\xi\ \delta(x) = \xi(\gamma(x))$

Applied to the modal logic case, our K-transformation turns the relational translation of modal logic into predicate logic into the functional translation (c.f. [Wal87a, Ohl88a, JR88, Her89, AE92, Ohl90, Gas92, Ohl93, Zam89]):

$$\begin{aligned} \text{serial:} \quad & tr_f(\Box\varphi, w) = \forall\gamma{:}AF\ tr_f(\varphi, \gamma(w)) \\ & tr_f(\Diamond\varphi, w) = \exists\gamma{:}AF\ tr_f(\varphi, \gamma(w)) \\ \text{non–serial:} \quad & tr_f(\Box\varphi, w) = End(w) \vee \forall\gamma{:}AF\ tr_f(\varphi, \gamma(w)) \\ & tr_f(\Diamond\varphi, w) = \neg End(w) \wedge \exists\gamma{:}AF\ tr_f(\varphi, \gamma(w)). \end{aligned}$$

The transformer (29) can be used in exactly the same way for optimizing the treatment of varying–domains in the translation of quantified modal logics. The normal translation rules for the quantifiers in the varying–domain case are

$$\begin{aligned} tr_r(\forall x\ \varphi(x), w) &= \forall x\ Exists(w,x) \Rightarrow tr_r(\varphi(x), w) \\ tr_r(\exists x\ \varphi(x), w) &= \exists x\ Exists(w,x) \wedge tr_r(\varphi(x), w) \end{aligned}$$

where $Exists(w,x)$ intuitively means that x is in the domain of the world w. Since each domain contains at least one element, $Exists$ is serial. The transformer (29) now yields an optimized translation

$$\begin{aligned} tr_f(\forall x\ \varphi(x), w) &= \forall\gamma{:}M\ tr_f(\varphi(\gamma(w)), w) \\ tr_f(\exists x\ \varphi(x), w) &= \exists\gamma{:}M\ tr_f(\varphi(\gamma(w)), w). \end{aligned}$$

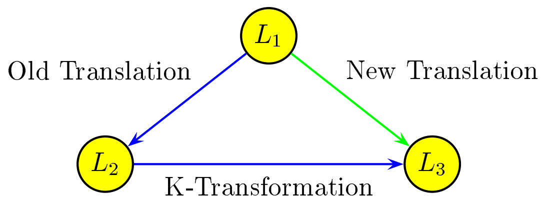
The sort M[1] denotes the set of functions mapping worlds to their domain elements, i.e. $\gamma(w) \in domain(w)$. Quantification over all γ exhausts the domain of w. In quantified modal logic we distinguish increasing domain and decreasing domain systems.

[1] We choose M to distinguish it from AF in case both transformers are applied simultaneously.

Increasing domain: $\forall x, u, v\ Exists(u, x) \wedge R(u, v) \Rightarrow Exists(v, x)$
Decreasing domain: $\forall x, u, v\ Exists(v, x) \wedge R(u, v) \Rightarrow Exists(u, x)$.

The transformer rewrites these two formulae to
Increasing domain:
$\forall x, u, v\ (\exists\delta{:}M\ x = \delta(u)) \wedge R(u, v) \Rightarrow (\exists\xi{:}M\ x = \xi(v))$
$\Leftrightarrow \quad \forall u, v\ \forall\delta{:}M\ \exists\xi{:}M\ R(u, v) \Rightarrow \delta(u) = \xi(v)$
Decreasing domain:
$\forall x, u, v\ (\exists\delta{:}M\ x = \delta(v) \wedge R(u, v) \Rightarrow (\exists\xi\ x = \xi(u))$
$\Leftrightarrow \quad \forall u, v\ \forall\delta{:}M\ \exists\xi{:}M\ R(u, v) \Rightarrow \delta(v) = \xi(u)$
The transformers can be applied to the $Exists$ predicate and to the R predicate simultaneously. The final versions of the axioms are then
Increasing domain:
$\forall u, v\ \forall\delta{:}M\exists\xi{:}M\ (\exists\gamma{:}AF\ v = \gamma(u)) \Rightarrow \delta(u) = \xi(v)$
$\Leftrightarrow \quad \forall u\ \forall\gamma{:}AF\ \forall\delta{:}M\ \exists\xi{:}M\ \delta(u) = \xi(\gamma(u))$ (compare the translation of the euclideanness axiom)
Decreasing domain:
$\forall u, v\ \forall\delta{:}M\ \exists\xi{:}M\ (\exists\gamma{:}AF\ v = \gamma(u)) \Rightarrow \delta(v) = \xi(u)$
$\Leftrightarrow \quad \forall u\ \forall\gamma{:}AF\ \forall\delta{:}M\ \exists\xi{:}M\ \delta(\gamma(u)) = \xi(u)$ (compare the translation of the transitivity axiom)

In this section we have demonstrated that using K-transformations, with a minimum of effort and a few lines of proof we can reconstruct the functional translation for modal logic and extend it to an optimized translation of varying–domain systems. We composed an existing translation, namely the relational translation from modal to predicate logic with the new defined K-transformation. This is an example for a general method to modify translations. The pattern is:



Axioms for Free

In corollary 2.5 we have noticed that renamed transformers allow us in principle to transfer properties of the original symbols to the renamed symbols. Assuming extra properties is not only possible for renamed transformers, but in general for all newly introduced symbols. For example for the sort AF and the *apply*–function introduced in the transformer (29) we could add the extensionality axiom:

$$\forall\gamma, \delta{:}AF\ (\forall x\ apply(\gamma, x) = apply(\delta, x)) \Rightarrow \gamma = \delta. \tag{35}$$

The justification for adding this axiom is implicitly contained in the proof of the extension lemma 3.1 for the transformers (29) and (30). In this proof we constructed a particular interpretation for the sort AF and the function symbol *apply*. AF denotes a set of functions, and the interpretation of *apply* is really the 'apply' function. But in this particular interpretation the extensionality axiom (35) is certainly true. Therefore, together with the transformer we could add this axiom.

Of course we would not add axioms like this for theorem proving purposes, but for enabling further transformations and simplifications. An axiom which can be simplified considerably with help of (35) is the inclusion axiom for two binary relations

$$\forall x, y\ R_a(x, y) \Rightarrow R_b(x, y). \tag{36}$$

This axiom describes the correspondence property of the multi–modal logic axiom schema $[b]p \Rightarrow [a]p$. We translate (36) with two copies of the transformer (29) (with different sorts AF_a and AF_b for the two different relations, but with the same *apply*–function) and simplify the result

$$
\begin{array}{ll}
 & \forall x, y\ (\exists \gamma{:}AF_a\ y = apply(\gamma, x)) \Rightarrow (\exists \delta{:}AF_b\ y = apply(\delta, x)) \\
\Leftrightarrow & \forall x, y\ \forall \gamma{:}AF_a\ (y = apply(\gamma, x) \Rightarrow (\exists \delta{:}AF_b\ y = apply(\delta, x)) \\
\Leftrightarrow & \forall x, y\ \forall \gamma{:}AF_a\ \exists \delta{:}AF_b\ apply(\gamma, x) = apply(\delta, x) \\
\Leftrightarrow & \forall x, y\ \forall \gamma{:}AF_a\ \exists \delta{:}AF_b\ \gamma = \delta \\
\text{iff} & AF_a \sqsubseteq AF_b.
\end{array}
$$

The first two simplifications are standard predicate logic transformations. The third simplification exploits the extensionality axiom (35) and the last modification expresses that the final equation defines nothing else than the subset relationship. In a sorted logic, this can be represented as a subsort relationship between the two sorts AF_a and AF_b.

Thus, as an example for exploiting axioms, we get for free during K-transformations, we demonstrated how to use the extensionality axiom for transforming the inclusion axiom (36) for two binary relations into a simple subsort declaration. For theorem provers with sorted unification [Wal87b] this is the optimal representation of the information contained in the inclusion axiom.

3.6 From Unary to Binary Consequence Relations

Logics can be specified in various ways. Quite often, a unary consequence relation $\vdash^{1}$ is used. $\vdash^{1}\varphi$ is to be understood as 'φ is a theorem.' For example, Łukasiewicz has shown that one axiom in addition to Modus Ponens and ,

$$\vdash^{1}((p \to q) \to r) \to ((r \to p) \to (s \to p)) \tag{37}$$

$$\text{From } \vdash^{1} p \text{ and } \vdash^{1} p \to q \text{ derive } \vdash^{1} q \tag{38}$$

axiomatize the implicational fragment of propositional logic [Łuk70, p. 295]. Using predicate logic as meta logic, the axiom and the Modus Ponens rule can be written as Horn clauses:

$$\forall r, p, q, s \qquad \vdash^{1}(i(i(i(p, q), r), i(i(r, p), i(s, p)))) \tag{39}$$

$$\forall p, q \qquad \vdash^{1}(p) \wedge \vdash^{1}(i(p, q)) \Rightarrow \vdash^{1}(q) \tag{40}$$

and the theorems of the logic can be derived using deduction in ordinary predicate logic.

Alternatively one can specify a logic via a binary consequence relation $\vdash^{2}$. $\varphi \vdash^{2} \psi$ means that ψ is derivable from φ, derivable in the so specified logic. It turns out that K-transformations describe precisely the relation between unary and binary consequence relations.

In the sequel, let Φ be a predicate logic (as meta logic) axiomatization of a logic $\mathcal{L}$ in terms of a unary consequence relation $\vdash^{1}$. There must be a distinguished term $\iota(x, y)$ in Φ which is some sort of implication. The term is distinguished in the sense that all theorems of the form $\vdash^{1}\ldots$ provable from Φ are actually of the form $\vdash^{1}\iota(\ldots, \ldots)$. Moreover, the ground proof of these theorems must contain also only atoms of the form $\vdash^{1}\iota(\ldots, \ldots)$. In the Łukasiewicz example this distinguished term is just $i(x, y)$ denoting $x \to y$. Only atoms of the form $\vdash^{1}i(\ldots, \ldots)$ occur in proofs of the theorems we are interested in. In general, ι need not be a function symbol. It can for example be a term $o(n(x), y)$ encoding $\neg x \vee y$. In most cases, however it will be just an implication symbol.

The following transformer turns unary into binary consequence relations:

$$\vdash^{1}\iota(x, y) \Leftrightarrow x \vdash^{2} y. \tag{41}$$

The extension lemma (Def. 2.1) is proved by taking (41) as the definition of $\vdash^{2}$.

All atoms in Φ of the form $\vdash^{1}\iota(\ldots, \ldots)$ can be rewritten with this transformer. This is an equivalence preserving transformation. Unfortunately the transformer is not a rewriting transformer (Def. 2.6) in the general sense. For example the Modus Ponens clause cannot be rewritten directly. However, the transformer can be turned into a saturation transformer (Def. 2.8). As inference strategy, ordered resolution with redundancy elimination in the sense of Bachmair and Ganzinger is used [BG90]. The ordering is chosen such that literals with the $\vdash^{1}$predicate are bigger than the literals with the $\vdash^{2}$ predicate. Since in ordered resolution only resolutions with the maximal literals in a clause are allowed, this prevents resolutions with the right hand side of the transformer

(41). Ordered resolution now derives new clauses where instead of the unary consequence relation $\models^1$ the binary predicate $\models^2$ appears. Since we assumed that all ground proofs of theorems we are interested in contain only atoms of the form $\models^1 i(\ldots,\ldots)$, on the non–ground level ordered resolution needs to be applied only to literals $\models^1 x$ where x is a variable. All other literals are either of the form $\models^1 i(\ldots,\ldots)$ and are destructively rewritten, or they are not necessary at all for proving our theorems. For example the clauses (39) and (40) become

$$\forall r,p,q,s \qquad i(i(p,q),r) \models^2 i(i(r,p),i(s,p)) \tag{42}$$

$$\forall p,q,r,s \qquad p \models^2 q \wedge i(p,q) \models^2 i(r,s) \Rightarrow r \models^2 s. \tag{43}$$

Unfortunately ordered resolution is not destructive. That means the old clauses, in the Łukasiewicz case the Modus Ponens clause, is still contained in the formula set. Let A be an old clause and A' be the new clause which is derived from A and the transformer in a sequence of ordered resolutions until all occurrences of $\models^1$ are eliminated. All necessary instances of A are of the form $\models^1 i(\ldots,\ldots)$. But these instances are implied by A' and the transformer. Moreover, these instances of A are bigger in the corresponding multiset ordering than A' and the transformer. Therefore A is redundant and can be deleted without losing completeness. After deleting all the old clauses, there is no further ordered resolution with the transformer possible. Thus, it can also be eliminated without losing completeness. This shows that the transformer is of the saturation type. Lemma 2.9 is applicable which proves the two remaining transformation lemmas.

3.7 Congruence Properties

So far we we have considered mainly properties of relations. We can, however, also transform properties of functions. Suppose we are given a reflexive and transitive relation R and some n-ary *congruent* functions. That means for the function symbols f one can prove

$$\forall \vec{p},\vec{q} \ \bigwedge_i (R(p_i,q_i) \wedge R(q_i,p_i)) \Rightarrow R(f(\vec{p}),f(\vec{q})) \tag{44}$$

where $\vec{p}$ abbreviates $p_1,\ldots,p_n$. We show how to eliminate these congruence properties.

The transformer (17) for reflexivity and transitivity translates the congruence properties (44) for the connectives f into

$$\forall \vec{p},\vec{q} \ \bigwedge_i (\forall w{:}W \ R'(w,p_i) \Leftrightarrow R'(w,q_i)) \Rightarrow (\forall w{:}W \ R'(w,f(\vec{p})) \Rightarrow R(w,f(\vec{q}))). \tag{45}$$

A straightforward proof yields

$$\forall \vec{p}\ \forall w{:}W \ R'(w,f(\vec{p})) \quad \Leftrightarrow \quad \exists \vec{x}{:}W \ R(w,f(\vec{x})) \wedge \bigwedge_i \forall v{:}W \ R'(v,x_i) \Leftrightarrow R'(v,p_i) \tag{46}$$

as the basis for the next K-transformation, which is for all $\vec{p},w$:

$$R'(w,f(\vec{p})) \Leftrightarrow \exists \vec{x}{:}S \ N_f(w,\vec{x}) \wedge \bigwedge_i \forall v{:}W \ R_f''^{\,i}(x_i,v) \Leftrightarrow R'(v,p_i). \tag{47}$$

Here we take advantage of the fact that a predicate may well be renamed in different ways in a renamed transformer. It is easy to check that (47) in fact turns the congruence properties (45) into tautologies.

Lemma 3.2 (Faithfulness of the Transformer (47))
Let $\mathcal{A}$ be the result of the transformation with the transformer (17) for reflexivity and transitivity. If (47) operates on $\mathcal{A}$ and as transformation strategy rewriting as long as possible is chosen, then the extension, the transformation and the elimination lemmas (Def. 2.1) hold.

Proof: <u>Extension Lemma:</u> This lemma holds because the sort symbol S can be interpreted like W, $N_f(w,\vec{x})$ can be interpreted as $R'(w,f(w,\vec{x}))$, $R''^{\,i}f(x_i,v)$ can be interpreted as $R'(v,x_i)$ and then (47) is true in the extended model because (46) is.

<u>Transformation Lemma:</u> Unfortunately the transformers (47) are no longer a simple definition for a predicate. The left hand side is of the form $R'(w,f(\vec{p}))$ where w and the p_i are variables. Since,

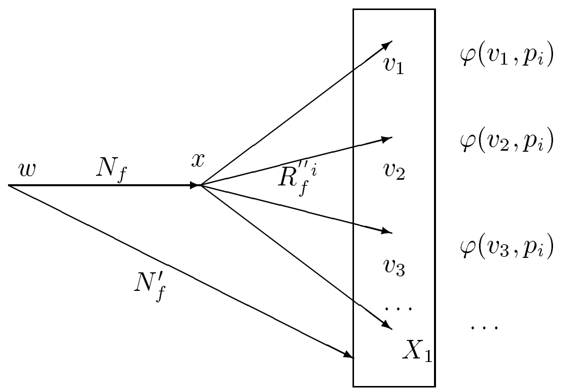
by the first transformer (17), all R'–literals in $\mathcal{A}$ are of the form $R'(v, f(\vec{s}))$ where v is a variable and the s_i are terms, the transformers can nevertheless be applied as rewrite rules. Due to the structure of the transformers' right hand sides, each rewrite step eliminates one occurrence of a function at a time. We end up with formulae $\mathcal{A}'$ containing no R'–literals which are instances of the left hand side of the transformers. So far, this is an equivalence transformation which does not affect the models at all.
Elimination Lemma: Although there are further inferences possible from the transformers and $\mathcal{A}'$, we can delete the transformers now without changing the satisfiability or unsatisfiability. The argument is proof theoretic. We show that there is a complete inference strategy which does not allow to draw any further inferences with the transformers in this particular case. The completeness of the strategy then guarantees that the transformers are useless.
The strategy we use is ordered resolution with elimination of redundant clauses [BG90]. Ordered resolution requires the definition of an ordering on literals. Only resolvents between the biggest literals in a clause need to be generated and we still have a complete procedure. The redundancy criterion, Bachmair and Ganzinger have shown to be complete is: inferences are redundant if the inferred clause follows from smaller (in the given ordering) clauses.
If we choose the ordering such that literals with the function f are bigger than the other literals, we immediately see that no resolvents with literals at the right hand side of the transformers are generated. These literals are smaller than the literal at the left hand side.
But there are still resolutions possible between the literals $R'(w, f(\vec{p}))$ from the left hand sides of the transformers and literals $R'(v, p)$ in $\mathcal{A}'$, where p is a variable. We can assume that these variables are of some basic domain sort, say D. That means there are quantifications $\forall p{:}D \ldots$. It is well known that sorts can be written as one–place predicates. Instead of $\forall p{:}D \ldots$, one writes $\forall p\ D(p) \Rightarrow \ldots$. Now we assume this syntactic form for $\mathcal{A}'$ and extend the ordering by requiring that literals $D(f(...))$ are bigger than any other literals. Rewriting with the transformers did not instantiate any variables in $\mathcal{A}'$. Therefore there are no literals $D(f(\ldots))$ in $\mathcal{A}'$, but only literals of the form $D(p)$. Each resolution between $R'(w, f(\vec{p}))$ and $R'(v, p)$ in some clause in $\mathcal{A}'$ now instantiates p to $f(\ldots)$. That means the resolvent contains literals $D(f(\ldots))$ and is therefore bigger than the parent clauses. This resolvent is redundant and needs not be generated.
Thus, we find that all resolvents with the transformers are redundant, which finally licenses their deletion. $\triangleleft$

In order to match the transformer (47) with a quite familiar structure, we change its right hand side a little bit. All R''^{i}_{f}–accessible points are collected in sets X_i and the relation N_f is replaced with a relation N'_f that takes these sets X_i as arguments. With this interpretation of the new relation N'_f one can show that the following equivalence holds.

$$\begin{array}{ll} & \exists \vec{x}{:}S\ N_f(w, \vec{x}) \wedge \bigwedge_i (\forall v{:}W\ R''^{i}_{f}(x_i, v) \Leftrightarrow \varphi(v, p_i)) \\ \Leftrightarrow & \\ & \exists \vec{X}\ N'_f(w, \vec{X}) \wedge \bigwedge_i (\forall v{:}W\ v \in X_i \Leftrightarrow \varphi(v, p_i)) \end{array} \tag{48}$$

where $\varphi(v, p_i)$ is any formula in terms of v and p_i and $\in$ is really the membership relation.

For a one-place function f a graphical illustration looks like:



Composing the transformer (49) with the transformation defined by (48), we obtain

$$R'(w, f(\vec{p})) \Leftrightarrow \exists \vec{X}\ N'_f(w, \vec{X}) \wedge \bigwedge_i (\forall v{:}W\ v \in X_i \Leftrightarrow R'(v, p_i)). \tag{49}$$

What have we got here? If as indicated in Section 3.2, R is the binary consequence relation $\vdash$, and R' is the satisfiability relation $\models$ and f is the modal $\Box$–operator satisfying the so called ME–rule:

$$(p \vdash q \wedge q \vdash p) \Rightarrow \Box p \vdash \Box q$$

which is nothing else than the congruence property then (49) represents the so called neighbourhood semantics (minimal model semantics) [Che80]:

$$w \models \Box p \Leftrightarrow \exists X\ N'_\Box(w, X) \wedge \forall v{:}W\ v \in X \Leftrightarrow v \models p.$$

or in words $\Box p$ is true in a world w iff the truth set of p, i.e. the set $\{v \mid v \models p\}$ is a neighbourhood of w. $N'_\Box$ is the *neighbourhood relation*.

Thus, the elimination of the congruence properties by means of K-transformations yields the well known neighbourhood semantics. Faithfulness of the transformer means soundness and completeness of neighbourhood semantics for arbitrary axiomatizations of logics satisfying the reflexivity and transitivity of $\vdash$ and the congruence properties. In [Ohl94] it is shown how these ideas can be carried further to generate stronger semantics for logics satisfying more properties than just the congruence properties.

3.8 A Heuristic for Finding Transformers

The first heuristic for finding transformers which eliminates a property ϕ was simply to look for connectives in the meta system, PL1 in our case, which have the same or a similar property. This way we found transformers for simple properties like transitivity or symmetry because predicate logic has connectives with these properties. If ϕ has no immediate counterpart in PL1, however, it becomes much more difficult to find a suitable transformer.

We now present a heuristic which may help in these cases. Suppose we are given a relation R with a certain property ϕ. We are looking for a definition $R(x_1, \ldots, x_n) \Leftrightarrow \psi(x_1, \ldots, x_n)$ of R, where ψ is a formula with a relation R' that has *not* the property ϕ, but which entails that R has the property ϕ. As an example, let ϕ be *density*:

$$\forall x, y\ R(x, z) \Leftrightarrow \exists y\ R(x, y) \wedge R(y, z) \tag{50}$$

The definition $R(x, z) \Leftrightarrow \psi(R')$, we are looking for, has to reconstruct density of R from a relation R' which is not necessarily dense. The heuristic for finding ψ is: *ψ must express the process of extracting the parts of R' which actually have the property ψ.* The heuristic can also formulated in a more active way: *ψ must express the process of imposing the property ψ to R'.*

For the density example this means, $\psi(x, z)$ must extract the dense parts of R', or in the active formulation, ψ must generate the dense closure of R'. That means, we must enforce density of R' between two points x and z. How can we enforce this? Well, first of all, we must generate a point between x and z. i.e. $\exists y\ R'(x, y) \wedge R'(y, z)$ must be part of ψ. But this is not enough. The process must be iterated infinitely often. That means we must ensure that for any two points between x and z, including the borderline points x and z itself, there is a third point in between. With this condition, the first point we introduced, triggers an infinite generation of new points, and we get density. With this idea in mind, the definition of ψ is now obvious.

$$\begin{aligned}
R(x, z) \quad \Leftrightarrow \quad & R'(x, z) \wedge && (51)\\
& \exists y\ R'(x, y) \wedge R'(y, z) \wedge && (52)\\
& \forall v\ (R'(x, v) \wedge R'(v, z)) \Rightarrow (\exists y\ R'(x, y) \wedge R'(y, v) \wedge \\
& \qquad\qquad\qquad\qquad \exists y\ R'(v, y) \wedge R'(y, z)) \wedge && (53)\\
& \forall v, u\ (R'(x, v) \wedge R'(v, u) \wedge R'(u, z)) \Rightarrow \exists y\ R'(u, y) \wedge R'(y, v). && (54)
\end{aligned}$$

(52) inserts the first point between x and z. (53) treats the borderline cases with x and z and (54) recursively generates a point between all points in the middle. Without (51), we would get as a side effect that the relation R is closed in the sense that it describes closed intervals. This has to be avoided. This definition works only under the assumption that R' is transitive, otherwise the recursion is not possible.

What we have got now is a renamed K-transformer for a transitive and dense relation which eliminates the density of R. The version of this equivalence where R' is replaced with R is the basis of the transformer. Density of R implies that this equivalence is true. Given the transitivity of R', the transformation of the density axiom becomes a tautology (the proof is not easy.) In Section 4.4 we use this heuristic to find a transformer for euclideanness (Ex. 4.15).

4 Elimination of Clauses

In this section we specialize to the clause based setting and investigate the problem: given a particular clause C in a clause set Φ, is it possible to find a faithful transformation $\Upsilon(\Phi)$ such that the clause C becomes superfluous?

We give an abstract characterization of this kind of transformation and then consider more concrete definitions. With some challenging examples from the theorem proving literature we show that we can obtain dramatic improvements of the performance of theorem provers. These examples are predicate logic encodings of Hilbert axiomatizations of certain logics. Improving the performance of theorem provers on this kind of examples supports the development of new logics.

4.1 Clause K-Transformations

A very first idea for a transformation that makes C superfluous could come from the following consideration. A clause $C = A_1 \wedge \ldots \wedge A_n \Rightarrow B$ can be seen as a procedure that, given instances of the premises A_i as input, computes corresponding instances of B as output. Well, if the only thing, we are interested in, is to produce B from the A_i, an alternative way to obtain this 'output' is to ensure that the transformed 'input' $\Upsilon(A_i)$ implies B, i.e. $\bigwedge_i \Upsilon(A_i) \Rightarrow B$ is a tautology. If we transform Φ such that instead or in addition to A_i itself, $\Upsilon(A_i)$ is available, we don't need C any more.

Unfortunately it may be the case (and necessary) that the 'output' B of the clause C needs to be used as 'input' for C again. This is typical for self resolving clauses like transitivity. But after eliminating C, B is not transformed. Thus, the 'output' B is not available any more as input. Therefore our first guess for a condition on Υ has to be strengthened: We must ensure that the transformed 'input' implies the *transformed* 'output' $\Upsilon(B)$. That means $\bigwedge_i \Upsilon(A_i) \Rightarrow \Upsilon(B)$ must be a tautology.

On the technical level this condition must be refined because for non–ground clauses where variables may become instantiated you cannot just transform clauses by transforming their literals separately. The next definition gives the precise conditions.

Definition 4.1 (Clause K-Transformation)
A function Υ mapping clauses to clause sets (possibly infinitely large) is called a *K-transformation for a clause* $C = A_1 \wedge \ldots \wedge A_n \Rightarrow B_1 \vee \ldots \vee B_m$ iff

1. $D \in \Upsilon(D)$ for all clauses D,
2. $C \wedge D \Rightarrow \Upsilon(D)$ for all clauses D.
3. $\Upsilon(D_1 \vee D_2) \Leftrightarrow \Upsilon(D_1) \vee \Upsilon(D_2)$ for all *ground clauses* D_1 and D_2.
4. For every clause D: if D' is a ground instance of a transformed ground instance of D then D' is also a ground instance of a clause in $\Upsilon(D)$.
5. For all ground instances $C\rho$ of C: $\bigwedge_{i=1}^{n} \vec{\forall}\Upsilon(A_i\rho) \Rightarrow \bigvee_{j=1}^{m} \vec{\forall}\Upsilon(B_j\rho)$.
   $\vec{\forall}\Upsilon(\ldots)$ means that all variables introduced by Υ are universally quantified.

For a clause set Φ let $\Upsilon(\Phi) \stackrel{\text{def}}{=} \{\Upsilon(D) \mid D \in \Phi\}$. $\lhd$

The first condition deals with the fact that the transformation Υ is intended to eliminate the need for inferences with C. However, D need not be involved directly in inferences with C, but in inferences with other clauses in Φ. Therefore the original clauses D might still be necessary.

On the other hand, if it can be guaranteed that the original clauses D can be derived from $\Upsilon(\Phi)\setminus\{D\}$ then D can be deleted. This can be seen as a further optimization of $\Upsilon(\Phi)$: all inferred clauses can be deleted without changing satisfiability and unsatisfiability. Therefore we do not treat this kind of optimization in the clause K-transformation formalism.

The second condition guarantees soundness of the transformation: the transformed clauses are consequences of the original ones. Therefore satisfiability is preserved.

The next two conditions are structural conditions. Number 3 relates the transformation of single literals or parts of clauses with the transformation of the whole clause. If Υ computes more than one clause, $\Upsilon(D_1) \vee \Upsilon(D_2)$ is to be understood as the conjunctive normal form, i.e. the list of clauses $\{E_1 \vee E_2 \mid E_i \in \Upsilon(D_i)\}$. On the ground level this condition means that one can transform literals individually and combine the results to get the transformation of the clause. This is usually not the case for the non–ground level because transformation of single literals with variables occurring also in the other literals of the clause may involve instantiations which affect the other literals as well.

The fourth condition is a kind of lifting property. It relates transformations on the ground level with transformations on the non–ground level.

The last condition is responsible for making C superfluous. It is the precise formulation of the condition we have mentioned above, the transformed premises of C must imply the transformed conclusion. Unfortunately it requires a test for *all* ground instances of C, and these are usually infinitely many. In the second part of this section we therefore reduce this condition to a more feasible and automatizable test.

Before we come to the soundness and completeness proof for clause K-transformations, let us illustrate the technique with two examples. The actual class of clause K-transformations we have in mind transforms clause sets by adding resolvents with C. Υ is represented by a set S_Υ of clauses, usually C itself together with some of its consequences, for example self resolvents. For each of these clauses $S \in S_\Upsilon$, there is a selected literal $L_S \in S$ to be resolved upon.

For a unary clause L we define

$$\Upsilon(L) = \{E \mid E = Res(L, S, L_S), S \in S_\Upsilon\} \cup \{L\}$$

where $Res(L, S, L_S)$ denotes the resolvent between the clauses L and S upon the selected literal $L_S \in S$. For an arbitrary clause D we define $\Upsilon(D)$ as the set of all simultaneous resolvents between subsets of literals in D and clauses in S_Υ with the selected literals L_S (together with D itself of course). The examples show how this is to be understood. See also def. 4.11 for the exact definition.

The conditions 1-4 of definition 4.1 are fulfilled by this class of clause K-transformations, for example condition 4 is a consequence of the lifting lemma for the resolution rule.

Example 4.2 (Transitivity)
$C = P(x,y) \wedge P(y,z) \Rightarrow P(x,z)$
$S_\Upsilon = \{C\}$, $L_C = P(x,y)$.
That means

$$\begin{array}{rcl} \Upsilon(P(s,t)) & = & \{P(s,t), \\ & & \forall z\ P(t,z) \Rightarrow P(s,z)\} \end{array}$$

$$\begin{array}{ll} \Upsilon(P(s,t) \vee P(q,r)) & \\ \quad = \ \{P(s,t) \vee P(q,r), & \\ \qquad (\forall z\ P(t,z) \Rightarrow P(s,z)) \vee P(q,r), & (\Leftrightarrow (\neg P(t,z) \vee P(s,z) \vee P(q,r))) \\ \qquad P(s,t) \vee (\forall z\ P(r,z) \Rightarrow P(q,z)), & (\Leftrightarrow (P(s,t) \vee \neg P(r,z) \vee P(q,z))) \\ \qquad (\forall z\ P(t,z) \Rightarrow P(s,z)) \vee (P(r,z) \Rightarrow P(q,z))\} & (\Leftrightarrow (\neg P(t,z) \vee P(s,z) \vee \neg P(r,z') \vee P(q,z'))). \end{array}$$

Notice that in the presence of the reflexivity axiom $P(x,x)$, only the last clause needs to be generated because all other clause can be derived from the last clause and the reflexivity axiom. This is one of the post optimizations possible in particular cases. For reflexivity and transitivity this results in precisely the transformation Brand used in his modification method to eliminate the transitivity of the equality predicate [Bra75].
In order to check the main condition, condition 5 of 4.1, suppose $P(a,b) \wedge P(b,c) \Rightarrow P(a,c)$ is a ground instance of C.

$$\vec{\forall}\Upsilon(P(a,b)) \wedge \vec{\forall}\Upsilon(P(b,c)) \Rightarrow \vec{\forall}\Upsilon(P(a,c))$$

becomes

$$\begin{aligned}&(P(a,b) \wedge \forall z\ P(b,z) \Rightarrow P(a,z))\\ &\wedge (P(b,c) \wedge \forall z\ P(c,z) \Rightarrow P(b,z))\\ &\Rightarrow (P(a,c) \wedge \forall z\ P(c,z) \Rightarrow P(a,z))\end{aligned}$$

which is in fact a tautology.
The transformation is independent of the structure of a, b and c. Therefore the condition holds for all ground instances of C. This is one of the rare cases where this condition is easy to check. The reason is that resolution with transitivity requires only matching, no unification. This makes the structure of a, b and c irrelevant. $\triangleleft$

Example 4.3 (Condensed Detachment)
The condensed detachment clause is a predicate logic encoding of the Modus Ponens rule. It is the key clause for many predicate logic formulations of Hilbert axiomatizations of logics. Many of the really hard challenging problems found in the Automated Reasoning literature use this clause.

$$C = P(i(x,y)) \wedge P(x) \Rightarrow P(y)$$

$$\begin{aligned}S_\Upsilon = \quad &\{P(i(x,y)) \wedge P(x) \Rightarrow P(y),\\ &P(i(x,i(z_1,z_2))) \wedge P(x) \wedge P(z_1) \Rightarrow P(z_2),\\ &...,\\ &P(i(x,i(z_1,i(\ldots z_i))) \wedge P(x) \wedge P(z_1) \wedge \ldots \wedge P(z_{i-1}) \Rightarrow P(z_i),\\ &...\ \}\end{aligned}$$

S_Υ consists of resolvents between the first and last literal of C. The first literal $P(i(x,i(z_1,i(\ldots z_i)))$ is the selected literal in all clauses.
An example for the translation of a (unary) clause:

$$\begin{aligned}&\Upsilon(P(i(i(i(x,y),z),i(i(z,x),i(u,x)))))\\ &\quad = \quad \{P(i(i(i(x,y),z),i(i(z,x),i(u,x)))),\\ &\qquad\quad P(i(i(x,y),z) \Rightarrow P(i(i(z,x),i(u,x))),\\ &\qquad\quad P(i(i(x,y),z) \wedge P(i(z,x)) \Rightarrow P(i(u,x)),\\ &\qquad\quad P(i(i(x,y),z) \wedge P(i(z,x)) \wedge P(u) \Rightarrow P(x),\\ &\qquad\quad P(i(i(i(u_1,u_2),y),z) \wedge P(i(z,i(u_1,u_2))) \wedge P(u) \wedge P(u_1) \Rightarrow P(u_2),\\ &\qquad\quad ...\ \}\end{aligned}$$

Notice that from the fifth clause onwards we have to instantiate into the clause to be transformed. This causes the transformation to become infinite. (Predicate logic would be decidable if we did not encounter such an effect somewhere.)
Checking condition 5 of def. 4.1 is also no longer finite. But we shall see below that not all ground instances, but only those of the structure $P(i(a,i(b_1,i(\ldots b_i)))$ need to be tested.

$$\Upsilon(P(i(a,b_1))) \wedge \Upsilon(P(a)) \Rightarrow \Upsilon(P(b_1))$$

becomes $\quad (P(i(a,b_1)) \wedge (P(a) \Rightarrow P(b_1)) \wedge P(a)) \Rightarrow P(b_1)$

which is a tautology.

$$\Upsilon(P(i(a,i(b_1,b_2)))) \wedge \Upsilon(P(a)) \Rightarrow \Upsilon(P(i(b_1,b_2)))$$

becomes

$$\begin{aligned}&(P(i(a,i(b_1,b_2)))\\ &\wedge (P(a) \Rightarrow P(i(b_1,b_2))) \wedge (P(a) \wedge P(b_1) \Rightarrow P(b_2))\\ &\wedge P(a))\\ &\Rightarrow P(i(b_1,b_2)) \wedge (P(b_1) \Rightarrow P(b_2))\end{aligned}$$

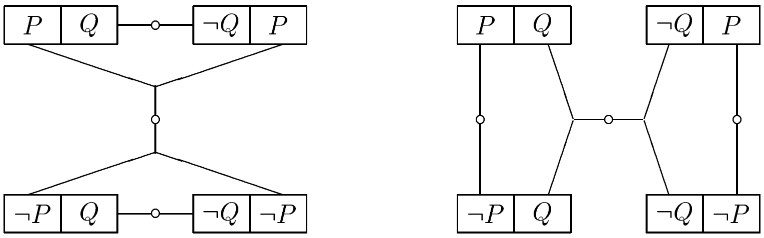
which is also a tautology. In all cases the structure of a is irrelevant. Therefore other instantiations for x need not be tested.

The schema should be clear. It can be proved for all $i \geq 1$. $\lhd$

Soundness and completeness of a transformation can be proved either on the semantic level by transforming models, or on the syntactic level by transforming proofs or *representations of proofs*. The actual definition of clause K-transformations is such that a semantic completeness proof is, if not impossible, extremely complicated. Therefore we choose to prove it on the syntactic level. The key part of the proof is to show that a refutation with the original clause set can be transformed into a refutation of the transformed clause set. To this end we need a represetnation of a refutation. The most abstract and for our purposes the most suitable representation of refutations of clause sets are *refutation graphs* as developed by Shostak [Sho76] and further investigated by Norbert Eisinger [Eis91].

A refutation graph for a clause set consists of *clause nodes* and *polylinks*. Clause nodes represent ground instances of the clauses. They consist of *literal nodes* labelled with the corresponding ground instances of the literals. Polylinks consist of two *shores*, the postive and the negative shore. The members of the positive shore are connected to literal nodes labelled with a positive literal L and the members of the negative shores are connected to the literal nodes labelled with the negative literal $\neg L$. Each literal node is connected to *only one* shore. The shores of polylinkls represent potential factoring operations and the link itself represents a potential resolution operation.

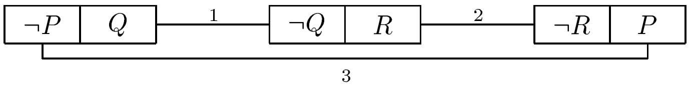
For example the clause set $\{\{P, Q\}, \{\neg P, Q\}, \{\neg Q, P\}, \{\neg Q, \neg P\}\}$ is unsatisfiable and has the follwing two refutation graphs.



Each box is a literal node and sequences of boxes are clause nodes. (Sometimes we also align literal nodes *vertically* to clause nodes.) In the left graph the upper and lower link are mono–links. Both shores consist of one element only. The link in the middle, however, is a polylink. Its positive shore consists of the two upper parts connected to the $\boxed{P}$ literal nodes and its negative part consists of the two lower parts connected to the $\boxed{\neg P}$ literal nodes.

The left graph represents a resolution refutation where at first the two resolutions between the Q–literals are performed. The results are P and $\neg P$ which resolve to the empty clause. The right graph represents a resolution refutation where at first the P–literals are resolved and then the Q–literals. Notice that in both cases the order in which the first two resolvents are generated is irrelevant. This is reflected in the symmetry of the graph. Thus, refutation graphs abstract from irrelevant orderings of resolution steps.

A very important notion in the refutation graph theory is the notion of a *path* through the graph. A path is a sequence of literal nodes and polylinks such that no polylink occurs twice and polylinks 'entered' at one side must be 'left' at the opposite side. That means entering a link at one shore and leaving it at the same shore is forbidden. A *cycle* is a path starting and ending at the same literal node. A very important property of refutation graphs is: *they are cycle free*. The following graph for example contains a cycle. It is therefore not a refutation graph.



Theorem 4.4 (Shostak)
A clause set Φ is unsatisfiable if and only if there is a (cycle free) refutation graph whose clause nodes represent ground instances of the clauses in Φ. $\lhd$

Theorem 4.5 (Soundness and Completeness of Υ)
For every clause set Φ and every clause K-transformation Υ for a clause $C \in \Phi$:
Φ is satisfiable if and only if $\Upsilon(\Phi \setminus \{C\})$ is satisfiable.

Proof: Since by condition (Def. 4.1,2), $\Phi' \stackrel{\text{def}}{=} \Upsilon(\Phi \setminus \{C\})$ is a consequence of Φ, it is certainly satisfiable if Φ itself is satisfiable.
Now suppose Φ is unsatisfiable.
In the sequel we assume that without loss of generality $C \stackrel{\text{def}}{=} \neg A_1 \vee \ldots \vee \neg A_n \vee B_1 \vee \ldots \vee B_m$ has at least one positive literal. If this is not the case, we can just exchange the polarity of all predicate symbols. This does not make any difference.
Since Φ is unsatisfiable, there exists a refutation graph G_0 for Φ. If G_0 does not contain any instance of C we are done. If it contains instances of C we transform the refutation graph for Φ into a refutation graph for Φ' which shows that Φ' is also unsatisfiable.
First of all, however, we need some auxiliary notions.

1. If D is a clause or a set of clauses then $[D]_{gr}$ denotes a set of ground instances of D.

2. If $C' \stackrel{\text{def}}{=} \neg A'_1 \vee \ldots \vee \neg A'_n \vee B'_1 \vee \ldots \vee B'_m$ is a ground instance of C then $C'' \stackrel{\text{def}}{=} \neg A'_1 \vee \ldots \vee \neg A'_n \vee E_1 \vee \ldots \vee E_m$ where $E_j \in [\Upsilon(B'_j)]_{gr}, j = 1, \ldots, m$ is called a Υ_C–instance. (Think of Υ_C–instances as instances of C itself or its self resolvents of C.)

   C'' is a *proper* Υ_C–instance if $C'' \neq C'$. The E'_j–parts are called the *originally positive* literal sets in C'' (because although they may be positive or negative, they originate from the positive literals in C and they usually consist of more than one literal). Correspondingly the $\neg A_i$–parts are called the originally negative literals in C'' (they never are transformed).

3. If for a literal L in a clause $D = L \vee S$ in Φ we have $cnf(L \vee \Upsilon(S)) \subseteq \Phi'$ then L is called an *original literal* in Φ'. (cnf denotes the conjunctive normal form.)

4. A Υ_C–instance C'' is called *right marginal* in a refutation graph iff C'''s originally positive literal sets are not connected to any other Υ_C–instance.

5. For a Υ_C–instance C'' in a refutation graph G there is a *distance* $|C''|_G$ defined as follows:

   If all originally negative literals of C'' are not connected to any other Υ_C–instance then $|C''|_G \stackrel{\text{def}}{=} 1$. If $D''_1, \ldots, D''_k$ are Υ_C–instances connected to the originally negative literals of C'' then $|C''|_G \stackrel{\text{def}}{=} max_i |D''_i|_G$.

   For clauses D which are no Υ_C–instances let $|D|_G = 0$.

   $|...|_G$ is well defined because refutation graphs are cycle free.

6. There is a measure $|G| = (a, b)$ for refutation graphs defined as follows: If there are no Υ_C–instances in G at all, then $|G| \stackrel{\text{def}}{=} (0, 0)$, otherwise

   $a \stackrel{\text{def}}{=} max\{|C''|_G \mid C''$ is right marginal Υ_C–instance in $G\}$ and
   $b \stackrel{\text{def}}{=} |\{C'' \mid |C''|_G = a\}|$.

   $|G|$ measures the largest chains of connected Υ_C–instances. Its second component counts the number of Υ_C–instances sitting at the positive end of chains with the largest length. This is well defined, again because refutation graphs are cycle free.

   The lexicographic ordering $<$ on $|G|$ is obviously *well founded*.

   C'' is called a *maximal* Υ_C–instance in G iff $|C''|_G = a$.

   (maximal Υ_C–instance are right marginal, but not vice versa.)

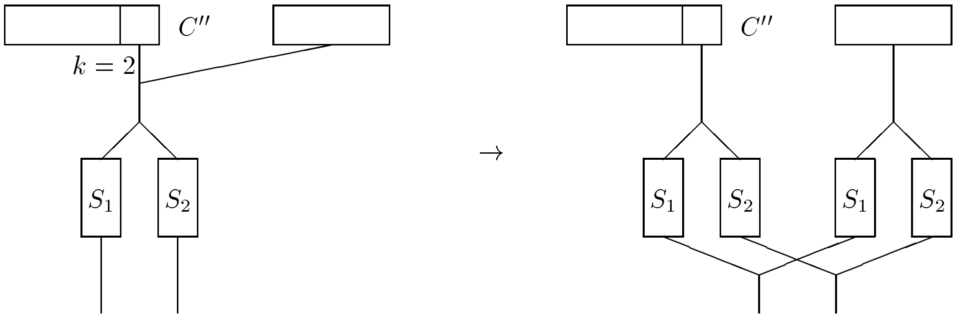
Starting with G_0 we define a transformation $G_r \to G_{r+1}$ and show the following *invariants*

1. G_{r+1} has all the standard properties of a refutation graph.

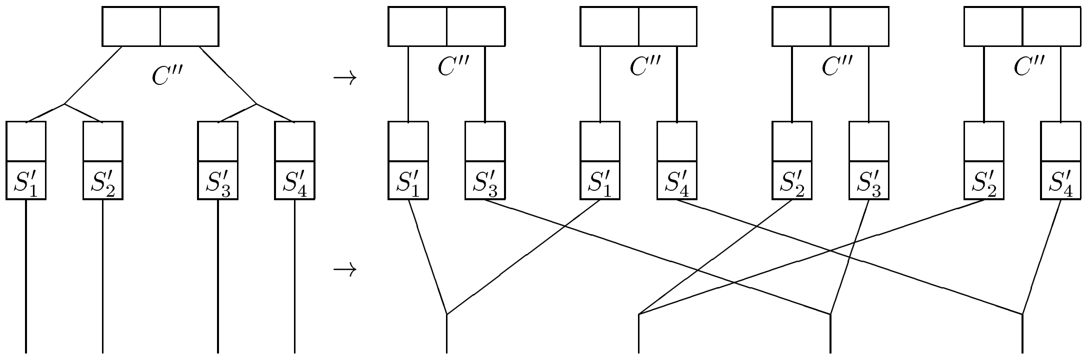
2. The clauses in G_{r+1} are either instances of clauses in Φ' or they are Υ_C–instances.

3. The literals connected to the originally negative literals of Υ_C–instances are original literals.

If we can further show $|G_{r+1}| < |G_r|$ then the sequence of transformations eventually terminates with a refutation graph containing no Υ_C–instances at all any more. This is a refutation graph for Φ', which shows that Φ' is unsatisfiable.
For the transformation of G_r to G_{r+1} we first choose a maximal Υ_C–instance C''. By a routine transformation on refutation graphs we firstly replace all poly–links connected to C'' by mono–links. This gives us a kind of normal form in the C''–part of the graph. This transformation consists of two parts. In the first part all multiple shores at the C''–side are replaced by singleton shores and in the second part all multiple shores at the opposite side of C'' are replaced by singleton shores.
As the following picture illustrates

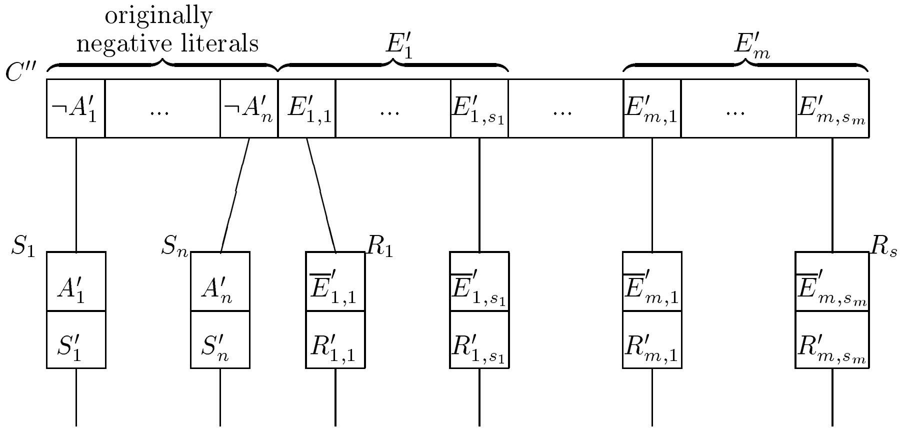


the clauses S_i connected to C'' with links having k shores ($k = 2$ in the example depicted above) at the C'' side are copied k times. The other links connected to the S_i split into k shores, one for each copy. The result G_r' is again a refutation graph and it has the invariance properties 1 – 3. Furthermore $|G_r'| = |G_r|$ because all the S_i have smaller distance than the C''. We can copy them as we like without changing the size $|G_r|$. We do this transformation for all of C'''s links with multiple shores at the C''–side.
In the second part of the transformation we need to copy C'' itself together with the clauses connected to C'' in order to get rid of the multiple shores at the opposite side of C''. If for a link l, $|l|$ denotes the number of l's shores at the opposite side of C'', and $l_1, \ldots, l_r$ are all the links connected to C'' then this represents $k \stackrel{\text{def}}{=} \Pi_{i=1}^{r} |l_i|$ mono–links. That means we need k copies if C'' and a corresponding number of copies of the adjacent clauses. For the case of $r = 2$ and $|l_1| = |l_2| = 2$ the transformation is depicted in the next figure.



Notice that there can't be paths between the left group and the right group of the clauses S_i' in the left picture above, because otherwise there would be a cycle. Therefore there are no paths in the transformed graph between the clauses adjacent to C''. Thus, we have not introduced cycles. There can, however, be links connecting S_1' and S_2' and links connecting S_3' and S_4'. The shores of these links are copied as well in order to connect the copies of the S_i' clauses.
The resulting graph G_r'' still preserves the invariance properties 1 – 3 and all copies of C'' are still right marginal and their distance has not changed. But the size of the graph has changed. If $|G_r| = (a, b)$ then $|G_r''| = (a, b + k - 1)$. In order to get a smaller graph G_{r+1} we must therefore eliminate all k copies of C''. We show how to do it for one copy and repeat this operation k times.

Now take any of these copies of $C'' \stackrel{\text{def}}{=} \neg A'_1 \vee \ldots \vee \neg A'_n \vee E'_1 \vee \ldots \vee E'_m$. The $\neg A_i$ are the originally negative literals. By the third invariance property they are connected to original literals A'_i of some clauses S_i which may or may not be Υ_C–instances. The $E'_j \stackrel{\text{def}}{=} (E'_{j,1} \vee \ldots \vee E'_{j,s_j}) \in [\Upsilon(B'_j)]_{gr}$ are the originally positive literal sets. Since C'' is maximal, the $E'_{j,l}$ are not connected to any other Υ_C–instance. A typical situation looks like this:

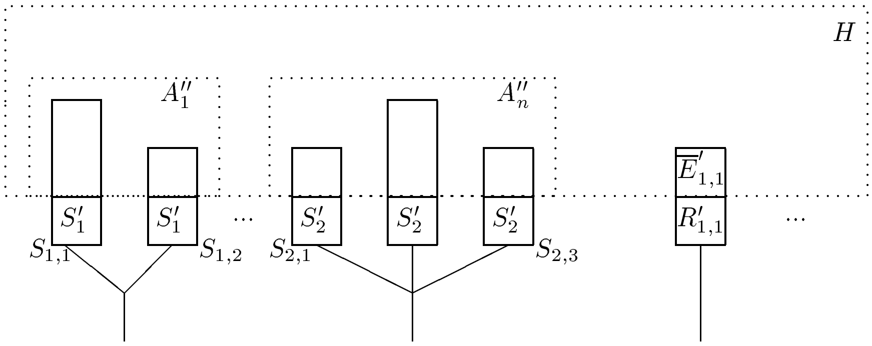


The literals $\overline{E}'_{j,l}$ are the complement of the literals $E'_{j,l}$. There are no paths from any S'_i to any S'_j or $\overline{R}'_{j,l}$ because otherwise we would have a cycle going through this path and C''. We show that we can transform the A'_i, and together with the $\overline{E}'_{j,l}$–literals we get an unsatisfiable clause set whose refutation graph can replace C''.

The condition (4.1,5): $\bigwedge_{i=1}^{n} \vec{\forall}\Upsilon(A'_i) \Rightarrow \bigvee_{j=1}^{m} \vec{\forall}\Upsilon(B'_j)$ implies
$\vec{\forall}\Upsilon(A'_1) \wedge \ldots \wedge \vec{\forall}\Upsilon(A'_n) \Rightarrow \vec{\forall}E_1 \vee \ldots \vee \vec{\forall}E_m$ where $E'_j \in [E_j]_{gr}, E_j \in \vec{\forall}\Upsilon(B'_j)$.
Thus, $\vec{\forall}\Upsilon(A'_1) \wedge \ldots \wedge \vec{\forall}\Upsilon(A'_n) \wedge \vec{\exists}\neg E_1 \vee \ldots \vee \vec{\exists}\neg E_m$ is unsatisfiable. We can instantiate the universally quantified variables in $\vec{\forall}\Upsilon(A'_1)$ as we like and choose the Skolem constants for the existentially quantified variables such that the Skolemized version of $\vec{\exists}\neg E_j$ is just $\overline{E}'_{j,1} \wedge \ldots \wedge \overline{E}'_{j,s_j}$.
Thus,

$$A''_1 \wedge A''_n \wedge \overline{E}'_{1,1} \wedge \ldots \wedge \overline{E}'_{m,s_m} \tag{55}$$

is unsatisfiable where A''_i is a finite subset of a ground instance of $\vec{\forall}\Upsilon(A'_1)$ (here we use the compactness of predicate logic). Let H be a refutation graph for (55). The C''–part of the refutation graph G_r now are replaced as follows: If $A''_i \stackrel{\text{def}}{=} A''_{i,1} \wedge \ldots \wedge A''_{i,o_i}$ then each clause $S_i = S'_i \vee A'_i$ is replaced with the clauses $S'_i \vee A''_{i,1}, \ldots, S'_i \vee A''_{i,o_i}$. The links connected to the S'_i get multiple shores, one for each copy. The new literal sets $A''_{i,l}$ and the literals $\overline{E}'_{j,u}$ are connected by the refutation graph H. The structure of this part of the transformed graph is now



Repeating this for all copies of C'' we obtain the new graph G_{r+1}. We have to show the invariance properties 1 – 3 and $|G_{r+1}| < |G_r|$.

1. First of all, G_{r+1} has still the structural properties of a refutation graph because H is a cycle free refutation graph and there are no path from any S'_i to any S'_j or $\overline{E'_{j,l}}$. Therefore no path within H can be completed with paths outside H to a cycle.

2. In order to show that the new clauses S_{i,o_i} are either Υ_C–instances or instances of clauses in Φ' we exploit that the literals A'_i are original literals and the homomorphy condition (4.1, 3): $\Upsilon(D_1 \vee D_2) \Leftrightarrow \Upsilon(D_1) \vee \Upsilon(D_2)$.

   If the S_i themselves are Υ_C–instances then the S_{i,o_i} are Υ_C–instances as well, by definition of Υ_C–instances and because the A'_i are original literals (this is important because Υ does not transform transformed literals again).

   Now suppose $S_i = S'_i \vee A'_i$ is an instance of a clause in Φ'. A'_i as an original literal has never been transformed. That means $S_i \in [\Upsilon(F_i)]_{gr} \vee A'_i$ where $\Phi_i \vee A'_i$ is a ground instance of a clause in Φ. Because of condition 3 in def. 4.1, $\Upsilon(F_i) \vee \Upsilon(A'_i) \Leftrightarrow \Upsilon(F_i \vee A'_i)$, which, by the lifting condition 4 means that each S_{i,o_i} is a ground instance of a clause in Φ'.

3. The literals connected to the originally negative literals of Υ_C–instances in G''_r are original literals. Let $S_{i,o_i} = S'_i \vee A''_{i,o_i}$ be a new Υ_C–instance. The originally negative literals in S_{i,o_i} are the literals in the S'_i part. But the other sides of the links connected to these literals have not been changed. Therefore they are still connected to original literals. The clauses R_j are no Υ_C–instances. Thus, they are not concerned here.

We had $|G_r| = (a, b)$ and $|G''_r| = (a, b + k - 1)$.
Case 1: There is another right marginal Υ_C–instance in G_{r+1} with distance a. Since we have removed all k copies of C'', we get $|G_{r+1}| = (a, b + k - 1 - k) = (a, b - 1) < |G_r|$.
Case 2: There is no right marginal Υ_C–instance in G_{r+1} with distance a.
Then $|G_{r+1}| = (a - 1, b') < (a, b) = |G_r|$.
This finishes the proof. $\triangleleft$

Notice that in the proof of the theorem the refutation graph for the transformed clauses is constructed without transforming negative literals. Since we can exchange the polarity of the predicates, this means that in general only either the positive or the negative literals need to be transformed by Υ. Transforming positive as well as negative literals usually complicates things rather than simplifying them.

4.2 Literal Triggered Clause K-Transformations

The definition of clause K-transformations we have given is more general than the class of transformations we have actually in mind. With this general definition, however, we worked out precisely those properties which are essential for the soundness and completeness. A more restricted definition would have forced us to deal with technicalities hiding more of the important aspects than clarifying things.

Nevertheless in order to give a more 'operational' definition, that means a definition of clause K-transformations where the infinite condition 5 can be tested with finite effort, we need to say more about the way Υ operates. To this end we define a class of so called *literal triggered clause K-transformations*. They are characterized by a set L_Υ of literals and each literal in L_Υ separately triggers a transformation. For example in Example 4.2 and 4.3, L_Υ consists of the selected literals. For this class of transformers, the condition 5 need not be tested for all ground instances, but only for certain instances derived from the characteristic literals. In simple cases as the transitivity and condensed detachment examples, it is just a single ground instance per characteristic literal which determines a case to be tested. In more complicated cases also ground instances of sets of unified literals in L_Υ are necessary.

Definition 4.6 (Literal Triggered Clause K-Transformations)
A function Υ mapping clauses to clause sets is called a *literal triggered K-transformation for a clause* $C = A_1 \wedge \ldots \wedge A_n \Rightarrow A_{n+1} \vee \ldots \vee A_m$ iff the first four conditions of def. 4.1 hold and the fifth condition is replaced by two new conditions:

5'. There is a set L_Υ of literals and a function Υ' computing sets of clauses such that for all ground literals L: $\Upsilon(L) = \{L\} \cup \bigcup_{K \in L_\Upsilon, L = K\mu} \Upsilon'(K)\mu$

6. In order to formulate the last condition we define $\Upsilon_0(L) \stackrel{\text{def}}{=} \Upsilon(L\psi)\psi^{-1}$ where ψ is a ground substitution replacing variables x by constants a_x and ψ^{-1} reverses this substitution, i.e. all constants a_x are replaced by the variables x. That means Υ_0 works like Υ, but does not instantiate variables in L.

   In the sequel we define for a literal L and a *set* $\mathcal{K}$ of literals, $mgu(L, \mathcal{K})$ as the simultaneous unifier for L with *all* literals in $\mathcal{K}$.

   Starting with $\Sigma_0 \stackrel{\text{def}}{=} \{\epsilon\}$ where ϵ is the identity substitution, a set Σ of *test substitutions* is computed as follows:

   For $i = 1, \ldots, m$: $\Sigma_i \stackrel{\text{def}}{=} \{\sigma \circ (\{\rho_{A_1}\} \cup \{mgu(A_i\sigma, \mathcal{K})_{|A_i} \mid \mathcal{K} \subseteq L_\Upsilon\}) \mid \sigma \in \Sigma_{i-1}\}$

   where ρ_{A_i} is a renaming substitution which renames those variables occurring in A_i which do not occur in $A_1, \ldots, A_{i-1}$, and for a *set* Θ of substitutions, $\sigma \circ \Theta_{|A_i}$ denotes the set $\{(\sigma\theta)_{|A_i} \mid \theta \in \Theta\}$.

   $\Sigma \stackrel{\text{def}}{=} \Sigma_m$.

   The *test substitution set condition* to be satisfied is that for all $\sigma \in \Sigma$:

$$\vec{\forall}(\vec{\forall}\Upsilon_0(A_1\sigma) \wedge \ldots \wedge \vec{\forall}\Upsilon_0(A_n\sigma) \Rightarrow \vec{\forall}\Upsilon_0(A_{n+1}\sigma) \vee \ldots \vee \vec{\forall}\Upsilon_0(A_m\sigma)) \qquad (56)$$

   is a tautology, where the inner quantifier $\vec{\forall}\Upsilon_0(A_i\sigma)$ does not quantify over the variables in $A_i\sigma$. These variables are bound in the outer quantifier.

$\triangleleft$

The literals in L_Υ are assumed to be available as infinitely many variable renamed copies, and each time such a literal is used for something, a new copy is taken.

We illustrate the test with the two previous examples, transitivity and condensed detachment.

Example 4.7 (Transitivity)
$C = P(x, y) \wedge P(y, z) \Rightarrow P(x, z)$
$L_\Upsilon = \{P(u, v)\}$.
$\Sigma_1 = \{\{x \mapsto x_1, y \mapsto y_1\}\} \cup \{mgu(P(x, y), \{P(u, v)\})_{|A_1}\} = \{\{x \mapsto x_1, y \mapsto y_1\}\}$
$\Sigma_2 = \{\{x \mapsto x_1, y \mapsto y_1\} \circ (\{\{z \mapsto z_1\}, mgu(P(y_1, z), \{P(u_1, v_1)\})_{|A_2}\}\})$
$\quad = \{\{x \mapsto x_1, y \mapsto y_1, z \mapsto z_1\}\}$
$\Sigma_3 = \Sigma_2$
The test substitution set condition now simply requires that the variables in C are to be considered as constants and then
$\vec{\forall}\Upsilon_0(P(a_{x_1}, a_{y_1})) \wedge \vec{\forall}\Upsilon_0(P(a_{y_1}, a_{z_1})) \Rightarrow \vec{\forall}\Upsilon_0(P(a_{x_1}, a_{z_1}))$
is to be proved. As we have shown in example 4.2 with the K-transformation defined for transitivity, this formula is in fact a tautology. $\triangleleft$

Example 4.8 (Condensed Detachment)
$C = P(i(x, y)) \wedge P(x) \Rightarrow P(y)$
$L_\Upsilon = \{P(i(u, v)), P(i(u, i(v_1, v_2))), P(i(u, i(v_1, i(v_2, v_3)))), \ldots\}$
The characteristic literals L_Υ form an infinite chain where each literal is an instance of all previous literals. Therefore the simultaneous unification of arbitrary sets of these literals with an external literal can be done by just unifying the most specific one in this set with the external literal. This observation (which is not un-typical) simplifies the computation of the sets Σ_i considerably.

$$
\begin{array}{lll}
\Sigma_1 & = & \{\{x \mapsto x_1, y \mapsto y_1\}, mgu(P(i(x,y)), P(i(u,v)))_{|A_1}, \\
& & mgu(P(i(x,y)), P(i(u, i(v_1,v_2))))_{|A_1}, \ldots\} \\
& = & \{\{x \mapsto x_1, y \mapsto y_1\}, \{x \mapsto u, y \mapsto v\}, \{x \mapsto u, y \mapsto i(v_1,v_2)\}, \ldots,\} \\
& \sim & \{\{x \mapsto x_1, y \mapsto y_1\}, \{x \mapsto u, y \mapsto i(v_1,v_2)\}, \ldots,\} \qquad \text{(an obvious simplification)} \\
\Sigma_2 & = & \{x \mapsto x_1, y \mapsto y_1\} \circ (\{\epsilon, mgu(P(x_1), P(i(u,v))), \ldots\})_{|A_2} \\
& & \cup \{x \mapsto u, y \mapsto i(v_1,v_2)\} \circ (\{\epsilon, mgu(P(u), P(i(u_2,v_2))), \ldots\})_{|A_2} \cup \ldots \\
& = & \{\{x \mapsto x_1, y \mapsto y_1\}, \{x \mapsto i(u,v), y \mapsto y_1\}, \{x \mapsto x_1, y \mapsto i(v_1,v_2)\}, \ldots\} \\
& & \cup \{\{x \mapsto u, y \mapsto i(v_1,v_2)\}, \{x \mapsto i(v_1,v_2), y \mapsto i(u_2,v_2)\}, \ldots\} \cup \ldots \\
\Sigma_3 & = & \Sigma_2
\end{array}
$$

$\sim$ means 'equal up to variable renaming'. It is instructive to see what happens if we rearrange the literals in the clause C. Nobody stops us from writing C as $C = P(x) \wedge P(i(x,y)) \Rightarrow P(y)$ and the test still should work.

$$
\begin{array}{lll}
\Sigma_1 & = & \{\{x \mapsto x_1\}, mgu(P(x), P(i(u,v)))_{|A_1}, mgu(P(x), P(i(u, i(v_1,v_2))))_{|A_1}, \ldots\} \\
& & = \{\{x \mapsto x_1\}, \{x \mapsto i(u,v)\}, \{x \mapsto i(u, i(v_1,v_2))\}, \ldots\} \\
\Sigma_2 & = & \{x \mapsto x_1\} \circ (\{\{y \mapsto y_1\}, mgu(P(i(x_1,y)), P(i(u,v))), \ldots\})_{|A_2} \\
& & \cup \{x \mapsto i(u,v)\} \circ (\{\{y \mapsto y_1\}, mgu(P(i(i(u,v),y)), P(i(u_1,v_1))), \ldots\})_{|A_2}, \cup \ldots \\
& \sim & \{\{x \mapsto x_1, y \mapsto y_1\}, \{x \mapsto x_1, y \mapsto i(u,v)\}, \ldots\} \\
& & \cup \{\{x \mapsto i(u,v), y \mapsto y_1\}, \{x \mapsto i(u,v), y \mapsto v_1\}, \{x \mapsto i(u,v), y \mapsto i(v_1,v_2)\}, \ldots\} \cup \ldots \\
\Sigma_3 & = & \Sigma_2
\end{array}
$$

In both cases we obtain all combinations of instantiations of x and y with terms $i(s,t)$ where s is a variable and t is either also a variable or again a term of this structure. $\lhd$

In the transitivity example 4.7 we saw that the test substitution set is a singleton. Therefore only a single ground instance of the clause C needs to be tested in order verify the test substitution set condition. A quick examination of the definition of the test substitution set Σ shows that this is always the case when the characteristic literals have only variables as arguments.

Corollary 4.9 (Characteristic Literals with Variables)
If the characteristic literals L_Υ are all of the form $P(x_1, \ldots, x_n)$ with different variables x_i then the test substitution set Σ is always a singleton and only a single ground instance of the clause C needed to be tested in order verify the test substitution set condition.

The following theorem guarantees soundness and completeness of literal triggered clause K-transformations.

Theorem 4.10 (Faithfulness of Literal Triggered Clause K-Transformations)
For every clause set Φ and every literal triggered clause K-transformation Υ for a clause $C \in \Phi$: Φ is satisfiable if and only if $\Upsilon(\Phi \setminus \{C\})$ is satisfiable.

Proof: We show condition 5 of definition 4.1:

$$
\vec{\forall}\Upsilon(A_1\rho) \wedge \ldots \wedge \vec{\forall}\Upsilon(A_n\rho) \Rightarrow \vec{\forall}\Upsilon(A_{n+1}\rho) \vee \ldots \vee \vec{\forall}\Upsilon(A_m\rho) \tag{57}
$$

for all ground substitutions ρ and then apply Theorem 4.5.
Before we can start the main part of the proof, a property that correlates Υ with Υ_0 must be proved. Suppose for a literal L and two substitutions σ and ϕ, $L\sigma\phi$ is ground and $L\sigma\phi$ is an instance of all literals in L_Υ unifiable with σ, i.e.

$$
\{K \in L_\Upsilon \mid L\sigma\phi \leq K\} = \{K \in L_\Upsilon \mid L\sigma \leq K\}. \tag{58}
$$

This means if $L\sigma\phi = K\mu$ then $L\sigma = K\mu'$ where $\mu = \mu'\phi$. Furthermore

$$
\begin{array}{lll}
\Upsilon(L\sigma\phi) & = & \{L\sigma\phi\} \cup \bigcup_{K \in L_\Upsilon, L\sigma\phi = K\mu} \Upsilon'(K)\mu \\
& = & \{L\sigma\phi\} \cup \bigcup_{K \in L_\Upsilon, L\sigma = K\mu'} \Upsilon'(K)\mu'\phi \\
& = & (\{L\sigma\} \cup \bigcup_{K \in L_\Upsilon, L\sigma = K\mu'} \Upsilon'(K)\mu')\phi \\
& = & \Upsilon_0(L\sigma)\phi.
\end{array} \tag{59}
$$

For the main part of the proof let ρ be an arbitrary ground substitution. ρ can be decomposed into $\rho = \rho_1 \dots \rho_m$ where $\rho_i = \rho_{|var(A_1,\dots,A_i)}$. ρ_i instantiates the variables up to A_i.
Let $\sigma_0 = \sigma'_0 = \epsilon$. For $i > 0$:
Let $\mathcal{K}_i \stackrel{\text{def}}{=} \{K \in L_\Upsilon \mid A_i\rho_i = K\mu \text{ for some } \mu\}$.
If $K_i = \emptyset$, let $\sigma_i = \sigma'_i = \sigma_{i-1}\rho_{A_i}$ $(\in \Sigma_i)$,
otherwise let $\sigma'_i = \sigma'_{i-1} mgu(A_i\sigma_{i-1}, \mathcal{K}_i)$ and $\sigma_i \stackrel{\text{def}}{=} \sigma'_{i|A_i} \in \Sigma_i$.
Let ϕ_i such that $\sigma_i\phi_i = \rho_i$
$\Upsilon(A_i\rho) = \Upsilon(A_i\rho_i)$
$= \{A_i\rho_i\} \cup \bigcup_{K \in L_\Upsilon, A_i\rho_i = K\mu} \Upsilon'(K)\mu$ (Def. 4.6, 4)
$= \{A_i\rho_i\} \cup \bigcup_{K \in \mathcal{K}_i} \Upsilon'(K)\sigma'_i\phi_i$ (since $A_i\sigma'_i = K\sigma'_i \succ A_i\sigma'_i\phi_i = K\sigma'_i\phi_i \succ A_i\rho_i = K\sigma'_i\phi_i$
$\succ \mu_{|K} = (\sigma'_i\phi_i)_{|K}$)
$= (\{A_i\sigma_i\} \cup \bigcup_{K \in \mathcal{K}_i} \Upsilon'(K)\sigma'_i)\phi_i$
$= \Upsilon_0(A_i\sigma_i)\phi_i$
Notice that $A_i\rho_i = A_i\sigma_i\phi_i$ is an instance of all literals in L_Υ unifiable with $A_i\sigma_i$. Therefore the condition (59) holds.
We end up with $\sigma \stackrel{\text{def}}{=} \sigma_m$ $(\in \Sigma)$ and $\phi \stackrel{\text{def}}{=} \phi_m$ such that $\rho = \sigma\phi$. Applying condition (56) and instantiating the still free variables with ϕ we find

$$\vec{\forall}\Upsilon_0(A_1\sigma)\phi \wedge \dots \wedge \vec{\forall}\Upsilon_0(A_n\sigma)\phi \Rightarrow \vec{\forall}\Upsilon_0(A_{n+1}\sigma)\phi \vee \dots \vee \vec{\forall}\Upsilon_0(A_m\sigma)\phi$$

The construction of σ guarantees (58) for all $A_i\sigma\phi$. Therefore we can apply (59) and this proves condition (57). $\triangleleft$

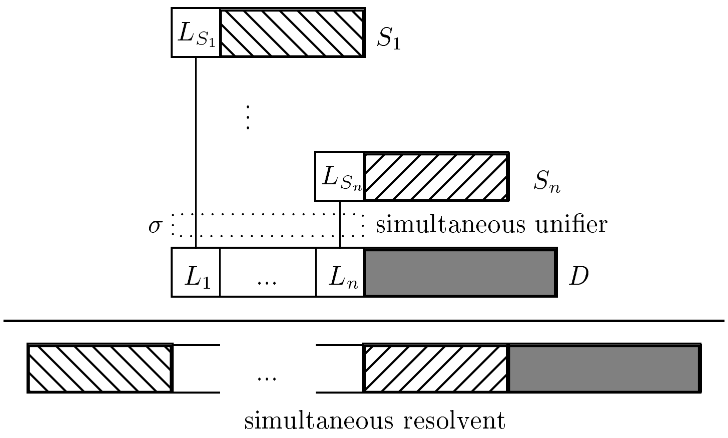
4.3 Resolution Clause K-Transformations

We now define a concrete class of clause K-transformations. These *resolution clause K-transformations* work by adding resolvents of consequences of C to the clause set Φ.

Definition 4.11 (Resolution Clause K-Transformations)
A *resolution clause K-transformations* for a clause C is characterized by a set S_Υ of consequences of C (usually C itself together with some of its self resolvents and subsumed clauses), and for each clause $S \in S_\Upsilon$ a literal $L_S \in S$ is selected.
The definition of Υ is:

$$\Upsilon(D) = \{D\} \cup \{Res(D, L_1, \dots, L_n, S_1, \dots, S_n) \mid L_1, \dots, L_n \in D, S_1, \dots, S_n \in S\_\Upsilon\} \quad (60)$$

where $Res(D, L_1, \dots, L_n, S_1, \dots, S_n)$ means simultaneous resolution between D and the clauses $S_1, \dots, S_n$ with resolution literals L_i and the selected literals $L_{S_i} \in S_i$ (very similar to hyperresolution). This is a graphical illustration of the operation



Example 4.2 showed how this works. $\triangleleft$

We verify that the structural conditions 1-5' of literal triggered clause K-transformations are satisfied by resolution clause K-transformations. Only the test substitutions set must be checked individually.

Theorem 4.12 (Conditions 1-5')
Resolution clause K-transformations for a clause C satisfy the conditions 1-5' of literal triggered clause K-transformations.

Proof:

1. $D \in \Upsilon(D)$ for all clauses D:
   satisfied by the definition of $\Upsilon(D)$.

2. $C \wedge D \Rightarrow \Upsilon(D)$ for all clauses D:

   The characteristic clauses S_Υ are implied by C, and $\Upsilon(D)$ as a set of resolvents between D and clauses in S_Υ is therefore also implied by C and D.

3. $\Upsilon(D_1 \vee D_2) \Leftrightarrow \Upsilon(D_1) \vee \Upsilon(D_2)$ for all *ground clauses* D_1 and D_2:

   On the ground level, resolvents are formed by concatenating parts of the parent clauses. Therefore this property is a consequence of the associativity of the concatenation operation.

4. For every clause D: if D' is a ground instance of a transformed ground instance of D then D' is also a ground instance of a clause in $\Upsilon(D)$:

   Let D'' be a ground instance of D and D' an instance of a clause in $\Upsilon(D'')$, i.e. D' is an instance of a simultaneous resolvent with D''. By the lifting lemma for resolution, D'' is also a ground instance of a corresponding resolvent with D.

5'. There is a set L_Υ of literals and a function Υ' computing sets of clauses such that for all ground literals L: $\Upsilon(L) = \{L\} \cup \bigcup_{K \in L_\Upsilon, L = K\mu} \Upsilon'(K)\mu$:

   The characteristic literals L_Υ are just the *complements* $\overline{L}_S$ of the selected literals in S_Υ.

$$\begin{aligned}
\Upsilon(L) &= \{L\} \cup \{Res(L, L, S) \mid S = L_S \vee S_1 \in S_\Upsilon, L_S \text{ resolvable with } L\} \\
&= \{L\} \cup \{S_1\mu \mid S = L_S \vee S_1 \in S_\Upsilon, L = \overline{L}_S\mu\} \\
&= \{L\} \cup \textstyle\bigcup_{K \in L_\Upsilon, K = K\mu} \Upsilon'(K)\mu
\end{aligned}$$

   where $\Upsilon'(L_S) \stackrel{\text{def}}{=} S_1$ for $S = (L_S \vee S_1) \in S_\Upsilon$

$\lhd$

The structure of resolution clause K-transformations is now clear. In order to find a transformation for a given clause C we need to define the characteristic clauses and the selected literals. To this end we start with C itself and choose one or more literals in C as selected literals. Then we systemically generate self resolvents and select again one or more literals. Each time we have chosen a literal, we check the test substitution set condition until it eventually comes true.

Furthermore the search for relevant self resolvents can be improved by testing whether the actual version of the transformer is a K-transformation for the self resolvent. (It may well be the case that a transformer is not a clause K-transformation for C itself, but for a self resolvent of C.) In this case this self resolvent and all further self resolvents derived from this one are unnecessary. For example the self resolvent on the second literal of the condensed detachment clause

$$\begin{array}{c}
\neg P(i(x,y)),\quad \underline{\neg P(x)}, P(y) \\
\neg P(i(x',y')), \neg P(x'),\quad \overline{P(y')} \\
\hline
\neg P(i(x,y)), \neg P(i(x',x)), \neg P(x'), P(y)
\end{array}$$

is useless already for a transformer Υ_0 where S_{Υ_0} consists only of the condensed detachment clause itself with the first literal selected.

In order to check this we apply the test:

$$(\Upsilon_0(P(i(a,b))) \wedge \Upsilon_0(P(i(c,a))) \wedge \Upsilon_0(P(c))) \Rightarrow \Upsilon_0(P(b))$$

becomes

$$(P(i(a,b)) \wedge (P(a) \Rightarrow P(b)) \wedge P(i(c,a)) \wedge (P(c) \Rightarrow P(a)) \wedge P(c)) \Rightarrow P(b)$$

which is a tautology. It is easy to check that for all further ground instances with $i(\ldots,\ldots)$ terms of the structure specified in the example 4.8 this test also succeeds. Therefore this self resolvent is redundant. Unfortunately the self resolvents with the first literal are not redundant, and we need the infinitely many selected clauses listed in example 4.3. In the next example we show how to deal with this infinite transfomer and we demonstrate the power of this transformation technique.

Example 4.13 (Implicational Calculus)
Łukasiewics has propsed an axiomatisation of the implicational fragment of propositional logic with Modus Ponens (61) and one single axiom (62) [Łuk70, p. 295]

$$\vdash x \to y \text{ and } \vdash x \text{ implies} \vdash y \tag{61}$$
$$\vdash ((x \to y) \to z) \to ((z \to x) \to (u \to x)). \tag{62}$$

The predicate logic encoding of this Hilbert system consists of the condensed detachment axiom (63) and one unit clause (64).

$$P(i(x,y)) \wedge P(x) \Rightarrow P(y) \tag{63}$$
$$P(i(i(i(x,y),z),i(i(z,x),i(u,x)))). \tag{64}$$

It has become a famous challenge problem in the automated reasoning literature because some of the theorems provable from these two axioms are extremely hard to prove. Only the Otter system [McC90, McC89] was able to prove them all. The proofs took hours and the system had to generate millions of clauses.
In example 4.3 we have shown how the transformer for the condensed detachment clause transforms the clause (64). We got the infinitely many clauses

$$P(i(i(i(x,y),z),i(i(z,x),i(u,x)))) \tag{65}$$
$$P(i(i(x,y),z) \Rightarrow P(i(i(z,x),i(u,x))) \tag{66}$$
$$P(i(i(x,y),z) \wedge P(i(z,x)) \Rightarrow P(i(u,x)) \tag{67}$$
$$P(i(i(x,y),z) \wedge P(i(z,x)) \wedge P(u) \Rightarrow P(x) \tag{68}$$
$$P(i(i(i(u_1,u_2),y),z) \wedge P(i(z,i(u_1,u_2))) \wedge P(u) \wedge P(u_1) \Rightarrow P(u_2) \tag{69}$$
$$P(i(i(i(u_1,i(u_2,u_3)),y),z) \wedge P(i(z,i(u_1,i(u_2,u_3)))) \wedge P(u) \wedge P(u_1) \wedge P(u_2) \Rightarrow P(u_3) \tag{70}$$

...

This is not yet suitable as input to an automated theorem prover. But with some standard and well known tricks we can convert it into a clause set suitable as input to any resolution based theorem prover.
First of all we notice that the literals $P(u)$ in the clauses from (68) onwards are superfluous. They can be resolved away with the first clause (65) without instantiating the other variables. Therefore they are simply deleted. This eliminates a redundancy which is implicitly present in the original formulation, but which cannot be detected there. This is one reason why the original formulation causes millions of subsumed clauses to be generated.
The next observation is that copies of the same literals occur several times. The usual trick to avoid the computational overhead caused by multiple copies of the same formulae is to introduce abbreviations, new predicates which stand for more complex formulae. We introduce a predicate q_1 that abbreviates $P(i(i(x,y),z))$. Notice that the variable y does not occur outside this literal. Therefore q_1 needs only depend on x and z. We obtain the following clause set.

$$P(i(i(i(x,y),z),i(i(z,x),i(u,x)))) \tag{65}$$
$$P(i(i(x,y),z) \Rightarrow q_1(x,z) \tag{71}$$
$$q_1(x,z) \Rightarrow P(i(i(z,x),i(u,x))) \tag{72}$$
$$q_1(x,z) \wedge P(i(z,x)) \Rightarrow P(i(u,x)) \tag{73}$$
$$q_1(x,z) \wedge P(i(z,x)) \Rightarrow P(x) \tag{74}$$
$$q_1(i(u_1,u_2),z) \wedge P(i(z,i(u_1,u_2))) \wedge P(u_1) \Rightarrow P(u_2) \tag{75}$$

$$q_1(i(u_1, i(u_2, u_3)), z) \wedge P(i(z, i(u_1, i(u_2, u_3)))) \wedge P(u_1) \wedge P(u_2) \Rightarrow P(u_3) \tag{76}$$
...

The fact that q_1 does not depend on y means that all instances of $P(i(i(x,y),z)$ which differ only in the binding for the variable y are mapped to the same q_1–literal. This also eliminates a source for redundancy. The literals $q_1(x,z) \wedge P(i(z,x))$ also occur several times. They are abbreviated by a new predicate $q_2(x)$. We get

$$P(i(i(i(x,y),z), i(i(z,x), i(u,x)))) \tag{65}$$
$$P(i(i(x,y),z) \Rightarrow q_1(x,z) \tag{71}$$
$$q_1(x,z) \Rightarrow P(i(i(z,x), i(u,x))) \tag{72}$$
$$q_1(x,z) \wedge P(i(z,x)) \Rightarrow q_2(x) \tag{77}$$
$$q_2(x) \Rightarrow P(i(u,x)) \tag{78}$$
$$q_2(x) \Rightarrow P(x) \tag{79}$$
$$q_2(i(u_1,u_2)) \wedge P(u_1) \Rightarrow P(u_2) \tag{80}$$
$$q_2(i(u_1, i(u_2,u_3))) \wedge P(u_1) \wedge P(u_2) \Rightarrow P(u_3) \tag{81}$$
...

In the next step we use the regularities of the infinite clauses (80) onwards for a finite encoding. The key observation is that the second argument of f successively is instantiated with new terms $i(u_i, u_{i+1})$. The effect of the clause (80) is to check whether a literal $q_2(i(u_1,u_2))$ is available where $P(u_1)$ is also provable and then to generate $P(u_2)$. If in fact the binding to u_2 is again a term of the form $i(u_2,u_3)$ and $P(u_2)$ is provable then the next clause (81) generates $P(u_3)$, etc. This iterative schema can be encoded recursively with two further auxiliary predicates q_3 and q_4.

$$P(i(i(i(x,y),z), i(i(z,x), i(u,x)))) \tag{65}$$
$$P(i(i(x,y),z) \Rightarrow q_1(x,z) \tag{71}$$
$$q_1(x,z) \Rightarrow P(i(i(z,x), i(u,x))) \tag{72}$$
$$q_1(x,z) \wedge P(i(z,x)) \Rightarrow q_2(x) \tag{77}$$
$$q_2(x) \Rightarrow P(i(u,x)) \tag{78}$$
$$q_2(x) \Rightarrow P(x) \tag{79}$$
$$q_2(x) \Rightarrow q_3(x) \tag{82}$$
$$q_3(i(u,v)) \wedge P(u) \Rightarrow q_4(v) \tag{83}$$
$$q_4(v) \Rightarrow q_3(v) \tag{84}$$
$$q_4(v) \Rightarrow P(v) \tag{85}$$

The clauses (83) and (84) realize a recursive loop where the second argument of $i(u,v)$ is subsequently instantiated in the way described above and the binding for the first argument u is checked with the $P(u)$ literal. The clause (82) is the entry point into this loop and the clause (85) is the exit from this loop. Together these four clauses have the same effect as the infinitely many clauses we had before.
The clause (78) is a potential source for an infinite explosion of the size of terms. Fortunately this clause is not independent of the other clauses. It can be derived from the remaining clauses. Therefore we can delete it. This does not mean that this potential infinite explosion is banned, but it is reduced considerably. The final result of our transformation is now:

$$P(i(i(i(x,y),z), i(i(z,x), i(u,x)))) \tag{65}$$
$$P(i(i(x,y),z) \Rightarrow q_1(x,z) \tag{71}$$
$$q_1(x,z) \Rightarrow P(i(i(z,x), i(u,x))) \tag{72}$$
$$q_1(x,z) \wedge P(i(z,x)) \Rightarrow q_2(x) \tag{77}$$
$$q_2(x) \Rightarrow P(x) \tag{79}$$
$$q_2(x) \Rightarrow q_3(x) \tag{82}$$
$$q_3(i(u,v)) \wedge P(u) \Rightarrow q_4(v) \tag{83}$$
$$q_4(v) \Rightarrow q_3(v) \tag{84}$$
$$q_4(v) \Rightarrow P(v) \tag{85}$$

On first glance these axioms look more complicated than the original two axioms (61) and (64), in particular since the clauses (79), (83) and (84) seem to represent a disguised version of the condensed detachment clause again. Therefore we have to demonstrate that theorem provers in fact can do better with these clauses as with the original ones. From our experiments with the Otter 3.0 theorem prover we got the following results.

	theorem	original	transformed	improvement
1	$P(i(x,x))$	1.91	0.11	17.3
2	$P(i(x,i(y,x)))$	1.94	0.13	14.9
3	$P(i(i(i(x,y),x),x))$	4.77	0.52	9.2
4	$P(i(i(x,y),i(i(y,z),i(x,z))))$	2520.77	49.51	50.9
5	$P(i(x,i(i(x,y),y)))$	35.07	13.75	2.5

The numbers in the third and fourth column give the total CPU time in seconds, Otter 3.0 needed to prove the theorem (on a Solburn machine with Super Sparc processors), once from the original two axioms, and then from the transformed axioms, both times with the hyperresolution inference rule. The first three examples were proved with Otter's default options. The theorems 4 and 5 are among the most complicated theorems, current day automated theorem provers can prove. But they can't prove them without additional guidance. The guidance we gave to the system is a redundancy criterion: If a clause $P(s)$ has been derived then all other derived clauses containing instances of s *as subterms* are deleted[2]. (It is an open question whether this deletion strategy preserves completeness). This heuristic was applied to both versions, the proof search with the original clauses as well as with the transformed clauses. Therefore the comparison is fair. (McCune used a weaker heuristic and reported much longer CPU times, 7933 sec. for example 4 and 2172 sec. for example 5 on a Sparc 1+ machine [MW92]).
We also tried problem 4 without any restriction. The system had to be stopped after 16 hours of CPU time and the generation of 3.2 million clauses. Setting the limit on the size of terms to 20, a proof was found after 3333 seconds (5.7 million clauses generated). With this limit, and in addition the above mentioned deletion rule activated, 943 seconds were needed to find a proof (0.5 million clauses generated). With the same settings, the transformed clause set was refuted in 40.4 seconds (0.03 million clauses generated) $\triangleleft$

Improving the performance of known solutions is nice, but the real test for a new technique are examples beyond the current theorem provers capabilities. William McCune listed in his CADE article [MW92] a number of theorem proving problems which could not be solved so far. We tried some of them.

Example 4.14 (Implicational Calculus with Falsehood)
According to Meredith [Mer53], Modus Ponens together with the following single axiom

$$\vdash ((((x \to y) \to (z \to F)) \to u) \to v) \to ((v \to x) \to (z \to x)) \tag{86}$$

axiomatizes the implicational fragment of propositional logic with falsehood (F).
The predicate logic encoding of this Hilbert system consists of the condensed detachment axiom, again together with one unit clause (87)

$$\begin{aligned} &P(i(x,y)) \wedge P(x) \Rightarrow P(y) \\ &P(i(i(i(i(i(x,y),i(z,F)),u),v),i(i(v,x),i(z,x)))). \end{aligned} \tag{87}$$

Again we use the transformer for condensed detachment and simplify the resulting clauses with the same methods as explained in the previous example 4.13. The result is

$P(i(i(i(i(i(x,y),i(z,F)),u),v),i(i(v,x),i(z,x))))$
$P(i(i(i(i(x,y),i(z,F)),u),v)) \Rightarrow q_1(v,x,z)$
$q_1(v,x,z) \Rightarrow P(i(i(v,x),i(z,x)))$
$q_1(v,x,z) \wedge P(i(v,x)) \Rightarrow q_2(z,x)$

[2]The dynamic demodulation mechanism in Otter can be used to delete these clauses. Clauses `P(s) -> (i(x,s) = junk).` and `P(s) -> (i(s,x) = junk).` are added and the weight of `junk` is set bigger than the weight limit.

$q_2(z,x) \Rightarrow P(i(z,x))$
$q_2(z,x) \wedge P(z) \Rightarrow q_3(x)$
$q_3(x) \Rightarrow P(x)$
$q_3(i(u,v)) \wedge P(u) \Rightarrow q_4(v)$
$q_4(v) \Rightarrow q_3(v)$
$q_4(v) \Rightarrow P(v)$

Unfortunately the transformed clause set of this example did not contain any of the redundancies we discovered in the previous example. Therefore we did not expect such significant improvements due to the transformation and subsequent simplification.
From these axioms we proved with Otter 3.0 six theorems. The inference rule was hyperresolution.

	theorem	original	transformed	factor
1	$P(i(i(x,y),i(i(y,z),i(x,z))))$	9049	3000	3
2	$P(i(x,i(y,x)))$	0.11	0.11	1
3	$P(i(i(i(x,y),x),x))$	6.45	15.61	1/2.4
4	$P(i(F,x))$	0.06	0.06	1
5	$P(i(i(i(x,F),F,x)))$	2.08	4.63	1/2.2
6	$P(i(i(x,i(y,z)),i(i(x,y),i(x,z))))$	21685	16089	1.3

Again the numbers in the third and fourth column are the total CPU time in seconds needed to find the proof. Theorems 2-5 are quite simple. And in fact, our transformation brought no improvement at all, at best we got the same CPU times as for the original version. Theorem 1 and theorem 6, however, were the really hard ones. No machine generated proof was known so far. We first found proofs for both theorems from the transformed axioms.
We did two experiments with the first theorem and the transformed clauses. In the first run, we limited the size of the derived literals to 20. This was the only restriction on the search space. The proof took 13590 seconds (3 hrs 46 min). The system generated 10250093 clauses and kept 180341 clauses. The proof consisted of 575 hyperresolution steps. The proof depth (depth in the search space) was 192.
In the second run we restricted the size of the terms to 18, switched off backward subsumption (backward subsumption turned out to be very expensive and completely useless) and applied a similar deletion rule as in the problems 4 and 5 of the implicational calculus example above: if a clause $P(s)$ is derived and s is not of the form $i(F,t)$ then all clauses containing instances of s as subterms are deleted. The proof now took only 3000 seconds (50 min). Only 2000588 clauses were generated and 32540 clauses were kept. The proof was a bit shorter than the previous one, 483 hyperresolution steps, with proof depth 164.
Once we had found a proof in the transformed system we also succeeded to find a proof with the original axioms. We used the same control heuristics as before. After 2 hrs, 30 min of CPU time and after having generated 3833104 clauses the system came up with a proof. The proof was much shorter, only 69 hyperresolution steps.
Similar things happened with the sixth theorem. This example turned out to be even harder than the previous one. We restricted the size of the terms to 18, switched off backward subsumption and applied the same deletion rule as before. The proof took 4 hrs, 28 min. The length of the proof is 513 hyperresolution steps. The proof depths is 186. The system generated 7550566 clauses of which 51489 were kept.
With the same control options we then found a proof from the original axioms. The proof took 6 hrs, 1 min. 8744542 clauses were generated and 25647 kept. The proof was again shorter, only 74 hyperresolution steps.
Although the transformation was not useful for the simple examples, and the factors don't look impressive for the complex examples, it saved more than 3 hours of CPU time for both examples together, and this is the important number. ◁

4.4 An Operational View of Clause K-Transformations

In order to understand the effect of clause K-transformations, a certain operational view on clauses is very helpful. A clause like the transitivity clause can serve as a nucleus for a hyperresolution step. Hyperresolution with transitivity takes two other clauses, the 'electrons' as input partners and generates the hyperresolvent as output. For example, the transitivity clause accepts the two clauses $R(a, b)$ and $R(b, c)$ and generates $R(a, c)$. Repeated hyperresolution with the transitivity clause successively generates the transitive closure of a basic relation. For example from $R(a, b)$, $R(b, c)$, $R(c, d)$ and $R(d, e)$ three hyperresolution steps generate $R(a, c)$, $R(b, d)$ and $R(a, d)$ and this describes the full transitive closure of the basic relation. Thus, the transitivity clause itself can be identified with a transitive closure computation process. The transitivity clause is the active component and the other clauses are the data part.

Now we transform the data part by adding for each positive R–literal the resolvent with the first literal (we could also chose the second one) of the transitivity clause. In our example this means $R(a, b)$, $R(b, c)$ and $R(c, d)$ are transformed into

$$\begin{array}{ll} C_1 & R(a,b) \\ C_2 & R(b,c) \\ C_3 & R(c,d) \\ R_1 & R(b,z) \Rightarrow R(a,z) \\ R_2 & R(c,z) \Rightarrow R(b,z) \\ R_3 & R(d,z) \Rightarrow R(c,z) \end{array}$$

The new clauses can now also serve as nuclei in hyperresolution steps. For example hyperresolution with R_1 takes clauses with literals $R(b, s)$ for an arbitrary s as input and produces $R(a, s)$ as output, i.e. it computes a part of the transitive closure of the basic relation. Thus, we have turned the passive data C_1, C_2, C_3 into active processes. Instead of the transitivity clause as a single process, we have now three different processes represented by the clauses R_1, R_2 and R_3. Moreover these processes can work in parallel. And in fact we have *parallelized the closure computation process* for the particular basic relation consisting of C_1,C_2 and C_3.

The question is now: are the new processes sufficient to compute the full transitive closure? In our simple example this is obviously the case. R_2 computes from C_3 the new fact $R(b, d)$ and R_1 computes from this new fact the second part $R(a, d)$ and from C_2 the remaining part $R(b, d)$ of the transitive closure. Our general completeness result and the test described in example 4.2 confirm that this is really sufficient in general.

Notice that our transformation turns only the initial clauses into active processes, not the derived clauses. They remain passive data. In logic terms this means that resolution of the derived clauses with the transitivity clause is not performed and this in turn means that no self resolvent is used.

The development of a resolution clause K-transformation for a given clause is a search process which can in principle be fully automated – at least for the finite cases. As always with complicated search processes, some heuristics may improve the efficiency and the result. We illustrate with another example that the operational view of the transformations can help finding simple clause K-transformations.

Example 4.15 (Euclideanness)
Let us try to find a resolution clause K-transformation for the euclideanness clause

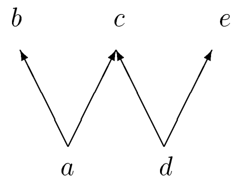
$$R(x, y) \wedge R(x, z) \Rightarrow R(z, y). \tag{88}$$

We have to go through a sequence of clauses consisting of the euclideanness clause itself and its consequences, select for each clause a literal and then check the test substitution set condition.
We start with Υ_1 characterized by the clause (88) itself and select the first literal. Since the selected literal has only variables as arguments, we need to test only a single ground instance.
$\Upsilon_1(R(a, b)) \wedge \Upsilon_1(R(a, c)) \Rightarrow \Upsilon_1(R(c, b))$
becomes

$(R(a,b) \wedge (\forall z\ R(a,z) \Rightarrow R(z,b)) \wedge R(a,c) \wedge (\forall z\ R(a,z) \Rightarrow R(z,c)))$
$\Rightarrow (R(c,b) \wedge (\forall z\ R(c,z) \Rightarrow R(z,b)))$.
This is *not* a tautology. Now we have a choice. Either we test the first self resolvents, or we take a copy of the euclideanness clause and select the second literal. (A fully automated search algorithm has a branching point here). A semantic consideration, however, and the heuristic formulated in Section 3.8 gives us a much better result.
Given the two clauses $C_1 \stackrel{\text{def}}{=} R(a,b)$ and $C_2 \stackrel{\text{def}}{=} R(a,c)$ for example, Υ_1 adds the two resolvents

$$\begin{array}{ll} R_1: & R(a,z) \Rightarrow R(z,b) \\ R_2: & R(a,z) \Rightarrow R(z,c). \end{array}$$

Obviously R_1 and R_2 as 'active processes' are able to compute the euclidean closure $R(c,d)$ and $R(b,c)$ of C_1 and C_2. But a slightly more complicated example shows that this is not sufficient in general. Consider the basic relation which is depicted by the following graph

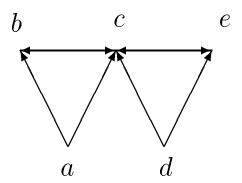


(89)

Υ_1 adds

$$\begin{array}{ll} R_1: & R(a,z) \Rightarrow R(z,b) \\ R_2: & R(a,z) \Rightarrow R(z,c) \\ R_3: & R(d,z) \Rightarrow R(z,c) \\ R_4: & R(d,z) \Rightarrow R(z,e) \end{array}$$

With these clauses the four additional arrows:



can be derived, but this is not the full euclidean closure. The remaining two arrows $R(b,e)$ and $R(e,b)$ cannot be generated. What we would need for example is a clause

$$R(a,c) \wedge R(c,z) \wedge R(c,v) \Rightarrow R(v,z)$$

to compute at least one of the missing arrows from the initial clause $R(a,c)$ and the derived clauses $R(c,b)$ and $R(c,e)$.
A candidate for an extension of Υ_1 is therefore

$$R(x,y) \wedge R(y,z) \wedge R(y,v) \Rightarrow R(v,z) \tag{90}$$

where the first literal is selected. This clause is subsumed by the euclideanness clause and therefore of course implied by it. Now we define Υ_2 as the resolution clause K-transformation such that the characteristic clauses are the two clauses (88) and (90), and both times the first literal is selected.
We have to test
$\Upsilon_2(R(a,b)) \wedge \Upsilon_2(R(a,c)) \Rightarrow \Upsilon_2(R(c,b))$
which becomes
$(R(a,b) \wedge (\forall z\ R(a,z) \Rightarrow R(z,b)) \wedge (\forall z,v\ R(b,z) \wedge R(b,v) \Rightarrow R(v,z))$
$\wedge R(a,c) \wedge (\forall z\ R(a,z) \Rightarrow R(z,c)) \wedge (\forall z,v\ R(c,z) \wedge R(c,v) \Rightarrow R(v,z)))$
$\Rightarrow (R(c,b) \wedge (\forall z\ R(c,z) \Rightarrow R(z,b))) \wedge (\forall z,v\ R(b,z) \wedge R(b,v) \Rightarrow R(v,z))$.
This is in fact a tautology. Therefore Υ_2 is a sound and complete transformer for the euclideanness clause. $\triangleleft$

We have developed a framework for clause K-transformations and we have provided criteria guaranteeing the faithfulness of the transformers, which can be tested by computer programs. The condensed detachment example and the euclideanness example, however show that the development of a good clause K-transformation for non–trivial examples still requires intuition and a certain understanding of the operational behaviour of these clauses. Automating this part of the process therefore is really non–trivial. It requires an expert system with suitable heuristics and means for dealing with infinitely many clauses.

4.5 Further Applications of Clause K-Transformations

We comprise the examples of clause K-transformations we have either developed already or which are now trivial to obtain. We leave the verification of these examples as an exercise to the reader. The transformations are written as a list S_Υ of characteristic clauses

$$L \Rightarrow \begin{cases} rest_1 \\ rest_2 \\ \ldots \end{cases}$$

where L is a common selected literal of all the clauses $L \Rightarrow rest_i$.

Symmetry:

$$R(x,y) \Rightarrow R(y,x) \tag{91}$$

Permutation: $(R(x,y,z) \Rightarrow R(y,z,x))$

$$R(x,y,z) \Rightarrow \begin{cases} R(y,z,x) \\ R(z,x,y) \end{cases} \tag{92}$$

Transtivity:

$$R(x,y) \Rightarrow (R(y,z) \Rightarrow R(x,z)) \tag{93}$$

alternatively (the second literal is selected)

$$R(x,y) \Rightarrow (R(z,x) \Rightarrow R(z,y)) \tag{94}$$

Euclideanness:

$$R(x,y) \Rightarrow \begin{cases} R(x,z) \Rightarrow R(z,y) \\ R(y,z) \wedge R(y,v) \Rightarrow R(v,z) \end{cases} \tag{95}$$

Condensed Detachment:

$$\begin{array}{l} P(i(x,y)) \Rightarrow P(x) \Rightarrow P(y) \\ P(i(x,i(z_1,z_1))) \Rightarrow P(x) \wedge P(z_1) \Rightarrow P(z_1) \\ \vdots \\ P(i(x,i(z_1,i(\ldots z_i))) \wedge P(x) \wedge P(z_1) \wedge \ldots \wedge P(z_{i-1}) \Rightarrow P(z_i) \\ \ldots \end{array} \tag{96}$$

4.5.1 Prolog Coding Tricks

All the clauses we have mentioned above cannot be used in Prolog programs because they immediatly cause Prolog to loop. For example the symmetry clause in a Prolog program r(X,Y) :- r(Y,X). immediately generates an inifinite loop. The same happens with more complex permutation clauses. Our clause K-transformations provide a simple solution to this problem.

Symmetry:
In the symmetry case (r is symmetric), every Prolog clause
r(s,t) :- body.
is replaced with the two clauses
r(s,t) :- body.
r(t,s) :- body.
and the symmetry clause is not needed any more.

Permutation:
For the permutation example ('r(Y,Z,X) :- r(X,Y,Z)' holds), we replace every Prolog clause
r(s,t,u) :- body.
with the three clauses
r(s,t,u) :- body.
r(t,u,s) :- body.
r(u,s,t) :- body.

Transitivity:
If the transitive relation is the closure of some basic relation, there is no problem in Prolog. For example the ancestor relation as transitive closure of the parent relation can be programmed as
ancestor(X,Z) :- parent(X,Z).
ancestor(X,Z) :-parent(X,Y), ancestor(Y,Z).
If, however, the relation stands on its own, the transitivity has to be programmed explicitly in Prolog. The transitivity clause
r(X,Z) :- r(X,Y),r(X,Z).
however, works only for successful queries r(s,t) with ground terms s and t. In all other cases Prolog loops. With clause K-transformations we can either use the transformer (93) or the transformer (94) to solve this problem.

For all clauses
r(s,t) :- body.
we either add
r(s,Z) :- r(t,Z),body. (first transformer)
or alternatively
r(Z,t) :- r(Z,s),body. (second transformer)
If the relation is transitive *and reflexive*, we keep the reflexivity clause
r(X,X).
but remove all other original clauses r(s,t) :- body.

Whereas these examples were straightforward applications of the K-transformation idea, the other two examples, euclideanness and condensed detachment, cannot be implemented in Prolog without further tricks.

4.5.2 Indirect Transformations

We gave some examples for formula K-transformations where new symbols where introduced from nowhere. The transformation (29) for binary relations is an example of this kind. Similar manipulations are possible with clause K-transformations. The trick is to add certain clauses first and then to use them for the transformation. Of course arbitrary clauses can not be added to a clause set without extra provisions. It must always be proved that every model for the original clause set can be extended to a model for the extended clause set (the other direction is trivial.).

We give an example for this trick. For every *serial* binary predicate R we can add to every clause set the formulae

$$\forall x, y\ R(x,y) \Rightarrow \exists\gamma{:}AF\ y = apply(\gamma, x) \tag{97}$$
$$\forall x, \gamma{:}AF\ R(x, apply(\gamma, x)). \tag{98}$$

As shown in Section (3.5), any model for the original clause set Φ with the predicate R can be extended to a model for Φ and the two new formulae: The sort AF is interpreted as the set of accessibility functions mapping points to accessible points. *apply* is interpreted as the 'apply'–functions. This interpretation satisfies both (97) and (98).

These two formulae are now used to generate a clause K-transformation. The formula (97) is the characteristic clause and the first literal is the selected literal. In order to get a clause normal form, we can Skolemize the existential quantifier in the formula (97) directly or after applying the transformation. This makes no difference. The test substitution set check is trivial in this case. Therefore the transformer is faithful.

Applied to the modal logic case in the same way as in Section (3.5), we can turn this transformation into the so called semi functional translation for modal logic formulae *in negation normal form* into predicate logic [Non93].

$$tr_s(\Box\varphi, w) = \forall v\ R(w, v) \Rightarrow tr_s(\varphi, v) \tag{99}$$
$$tr_s(\Diamond\varphi, w) = \exists\gamma{:}AF\ tr_s(\varphi, \gamma(w)). \tag{100}$$

Converted to clauses the result of this transformation does not contain any positive R–literals. The only positive R–literals are (98) which has to be added always and positive literals specifying properties of the accessibility relation. This can be exploited to transform these properties also. For example modal logic S4 is characterized by the reflexivity and transitivity of the accessibility relation. But we have just developed clause K-transformations for transitivity (93) and (94). The only formula with a positive literal (98) becomes either

$$\begin{array}{ll} \forall x, \gamma{:}AF\ \forall z\ R(apply(\gamma, x), z) \Rightarrow R(x, z) & \text{or} \\ \forall x, \gamma{:}AF\ \forall z\ R(z, x) \Rightarrow R(z, apply(\gamma, x)). & \end{array}$$

Each of these formulae together with the reflexivity clause now specifies S4 modal logic in the semi functional translation. Compared to the three literal transitivity clause, this clause is much easier to handle. The semi functional translation is not as compact as the functional translation for modal logic, but it allows for an optimized treatement of the theory clauses without extra theory unification algorithms.

5 Elimination of Literals

An important question is whether it is possible to find K-transformations *automatically*. Unfortunately there is not much hope to find K-transformations like the decomposition of n–ary relations into binary relations, see (26), or the functional representation of binary relations, see (29) automatically. These transformations exploit elemenary properties of relations which are not represented syntactically in any way. But in many other cases the essential information for formula K-transformations is in fact syntactically accessible. A formula K-transformation is represented by an equivalence $L \Leftrightarrow C$. That means it represents a definition of L in terms of something else. Definitions like for example

$$\forall x, y\ subset(x, y) \Leftrightarrow (\forall z\ in(z, x) \Rightarrow in(z, y)) \tag{101}$$

where L is $subset(x, y)$ are easy to spot automatically in a formula set. Converted to clause form, however, such definitions look much more complicated. The clause form representation of (101) for example, is

$$\begin{array}{l} \neg subset(x, y), \neg in(z, x), in(z, y) \\ subset(x, y), in(f(x, y), x) \\ subset(x, y), \neg in(f(x, y), y) \end{array}$$

where f is a Skolem function. How to reconstruct the original definition from these three clauses is not obvious. Nevertheless, the information is there and one should be able to dig it out. Moreover, according to Beth's definability theorem, if a literal is implictly defined[3] by some Φ, it is also explictly defined, i.e. $\Phi \Rightarrow (L \Leftrightarrow \phi)$ is provable for some ϕ. Thus, the information about the definition of L is syntactically available.

Suppose $L \Leftrightarrow \phi$ is in fact provable from some clause set Φ (our method works best, but is not restricted to, clauses). This means $\Phi \Rightarrow \vec{\forall}\ (L \Rightarrow \phi)$ and $\Phi \Rightarrow \vec{\forall}\ (\neg L \Rightarrow \neg\phi)$. Therefore $\Phi \wedge \vec{\exists}\ (L \wedge \neg\phi)$

[3] The intuitive definition of 'implicitly defined' is: a literal L is implicity defined by Φ iff whenever an interpretation fixes the semantics of all other symbols in Φ then there is no choice for the interpretation of L in order to satisfy Φ. For example if $\Phi = P \Leftrightarrow Q$ then there is no choice for P as soon as the interpretation of Q is fixed. If, however $\Phi = P \vee Q$ then one can satisfy Φ by making Q true and there is still a choice for P.

as well as $\Phi \wedge \vec{\exists}\,(\neg L \wedge \phi)$ are unsatisfiable. Let L' and ϕ' be the Skolemized version of $\exists\,(L \wedge \phi)$. L' is a ground literal, ϕ' not necessarily. Summerizing, both

$$\Phi \wedge L' \wedge \neg\phi' \qquad \text{and} \qquad \Phi \wedge \neg L' \wedge \phi' \tag{102}$$

are unsatisfiable.

Let C' be all resolvents between L' and clauses in Φ and let D' be all resolvents between $\neg L'$ and clauses in Φ. Since L' is ground we have (by the definition of the resolution rule)

$$L' \Rightarrow C' \quad \text{and} \quad \neg L' \Rightarrow D'. \tag{103}$$

Furthermore, using the purity deletion principle which says that a clause becomes superfluous as soon as all resolvents with one literal are generated, we can replace the L' in (102) by the C' or D' respectively. That means

$$\Phi \wedge C' \wedge \neg\phi' \qquad \text{and} \qquad \Phi \wedge D' \wedge \phi' \tag{104}$$

are also unsatisfiable. Thus, $\Phi \Rightarrow C' \Rightarrow \neg\phi'$ and $\Phi \Rightarrow D' \Rightarrow \phi'$ are tautologies, which entails $\Phi \Rightarrow \neg(C' \wedge D')$. Simple propositional consequences of this together with (103) and (104) are now $\Phi \Rightarrow L' \Leftrightarrow C'$ and $\Phi \Rightarrow \neg L' \Leftrightarrow D'$. Since the Skolem constants introduced at the beginning are arbitrary, we finally get $\Phi \Rightarrow \vec{\forall} L \Leftrightarrow C$ and $\Phi \Rightarrow \vec{\forall}\neg L \Leftrightarrow D$, where the Skolem constants in C' and D' are replaced by variables.

This means the algorithm for finding a definition for L is quite simple: Generate resolvents with L' and $\neg L'$ and check whether C' and D' together with Φ is unsatisfiable. The above considerations are essentially the soundness and completeness proof of this method.

The algorithm given below is a slight modification of this idea with particular emphasis on using the definition as K-transformation afterwards.

Definition 5.1 (Finding Definitions)
Let Φ be a set of clauses. Let L be the literal we want to find the definition for. Let us indicate the variables in L by $L[x_1...x_p]$. Suppose $(\{C_1, ..., C_m\} \cup \{D_1...D_n\} \cup G) \subseteq \Phi$, where Φ' is a specified subset of Φ. One of the C_i or D_j may be the tautology $L \vee \neg L$. Let L' be the ground instance $L[c_1...c_p]$ of L obtained by replacing all variables by distinct new constant symbols $c_1...c_p$. Let $C'_1...C'_m$ be obtained by resolving L' with $C_1...C_m$, respectively, and let $D'_1...D'_n$ be obtained by resolving $\neg L'$ with $D_1...D_n$, respectively. We need not include all possible resolvents here, but only a subset of the possible resolvents of L' and $\neg L'$ with the clauses C_i and D_j. Then we test if $\Phi' \wedge C'_1 \wedge ... \wedge C'_m \wedge D'_1 \wedge ... \wedge D'_n$ is unsatisfiable, possibly by searching for a resolution proof of the empty clause. This unsatisfiability test is often easy, since many variables in the C'_i and D'_j have been replaced by new constant symbols. Let C' be $C'_1 \wedge ... \wedge C'_m$. We sometimes indicate the occurrences of the new constant symbols in C' by $C'[c_1...c_p]$. Similarly, let D' be $D'_1 \wedge ... \wedge D'_n$. We sometimes indicate the constant symbols in D' by $D'[c_1...c_p]$. Let C be C' with the occurrences of $c_1...c_p$ replaced by new variables $x_1...x_p$; thus C is $C'[x_1...x_p]$. Similarly, let D be $D'[x_1...x_p]$. If we find that $\Phi' \wedge C' \wedge D'$ is unsatisfiable, then we have a definition of L: $L \Leftrightarrow C$ or equivalently $\neg L \Leftrightarrow D$ $\lhd$

In many cases it is clear which are the clauses C_i and D_j. But it is usually not clear which are the right resolvents with L'. Therefore this part may involve a search process. But from the examples we have checked so far there are only a few choices and the search is really simple.

Lemma 5.2 (Soundness of the Method for Finding Transformers)
If C and D are found as described in definition 5.1 then $\Phi \Rightarrow (L \Leftrightarrow C)$ and $\Phi \Rightarrow (\neg L \Leftrightarrow D)$.

Proof: First, $\Phi \Rightarrow (L' \Rightarrow C')$ because C' is derived from $\Phi \cup \{L'\}$ by resolution, and therefore $\Phi \wedge L' \Rightarrow C'$. Since the c_i are new constant symbols that occur nowhere else in Φ, $\Phi \wedge L \Rightarrow C$. Therefore $\Phi \Rightarrow (L \Rightarrow C)$. Similarly, $\Phi \Rightarrow (\neg L \Rightarrow D)$. We also have that Φ implies $(C \wedge D \Rightarrow false)$ since $\Phi' \wedge C' \wedge D'$ is unsatisfiable and $\Phi \Rightarrow \Phi'$. Since $\Phi \Rightarrow (C \wedge D \Rightarrow false)$, $\Phi \;\Rightarrow\; (C \Rightarrow \neg D)$ and $\Phi \Rightarrow (D \Rightarrow \neg C)$. Thus $\Phi \;\Rightarrow\; \wedge\,(L \Rightarrow C)(C \Rightarrow \neg D)$ so $\Phi \;\Rightarrow\; (L \;\Rightarrow\; \neg D)$, i.e. $\Phi \Rightarrow (D \Rightarrow \neg L)$.

Similarly, $\Phi \Rightarrow (\neg L \Rightarrow \neg C)$. Since we already have $\Phi \Rightarrow (\neg L \Rightarrow D)$ and $\Phi \Rightarrow (L \Rightarrow C)$, we obtain $\Phi \Rightarrow (L \Leftrightarrow C)$ and $\Phi \Rightarrow (\neg L \Leftrightarrow D)$. $\triangleleft$

The two equivalences $L \Leftrightarrow C$ and $\neg L \Leftrightarrow D$ can be used to replace all occurrences of L with C and all occurrences of $\neg L$ with D. Moreover, it can be shown that the resolution partners C_i and D_j used to find C and D can be removed. We therefore define an *eager version* of the transformation as follows:

Definition 5.3 (Eager Transformation)
Let Φ be a clause set and let Φ', $\{C_1, ..., C_m\}, \{D_1...D_n\}$ be the clauses in definition 5.1 used to find the transformation $L \Leftrightarrow A$ and $\neg L \Leftrightarrow B$. Φ' is disjoint from $(\{C_1, ..., C_m\} \cup \{D_1...D_n\})$.
Let Ψ' be $\Phi \setminus (\{C_1, ..., C_m\} \cup \{D_1...D_n\} \cup \Phi')$. Let Ψ'' be Ψ' with all instances of L and $\neg L$ in Ψ' replaced by corresponding instances of A and B. We assume that all literals in Ψ'' unifiable with L or $\neg L$ are in the instances of A and B introduced by the replacement. Now $\Psi \stackrel{\text{def}}{=} \Psi'' \cup \Phi'$ is the result of the transformation. $\triangleleft$

Theorem 5.4 (Soundness and Completeness of the Eager Transformation)
If Ψ is the result of an eager transformation then Ψ is unsatisfiable iff Φ is unsatisfiable.

Proof: Note that the 'clauses' of Ψ may have a more complicated structure than usual, that is, some of their literals may be instances of A or B. We nevertheless can consider these formulae as literals. Suppose Ψ is unsatisfiable. Now, all clauses of Ψ are logical consequences of Φ. For example, if $C \cup \{L[t_1...t_p]\}$ is in Φ then $C \cup \{A[t_1...t_p]\}$ is a logical consequence of Φ since $\Phi' \wedge L[x_1...x_p] \Rightarrow A[x_1...x_p]$ for all x_i. Similarly, if $C \cup \{\neg L[t_1...t_p]\}$ is in Φ then $C \cup \{B[t_1...t_p]\}$ is a logical consequence of Φ. Therefore, Ψ is a logical consequence of Φ, so if Ψ is unsatisfiable, so is Φ.

Now, suppose Ψ is satisfiable. Then there is a model $\Im$ of Ψ. For simplicity we can assume it is a Herbrand model. Let $\Im'$ be a model constructed in the following way: If $\Im$ satisfies $A[t_1...t_p]$ then $\Im'$ satisfies $L[t_1...t_p]$, for all ground terms $t_1...t_p$. If $\Im$ satisfies $B[t_1...t_p]$ then $\Im'$ satisfies $\neg L[t_1...t_p]$. If $\Im$ satisfies neither $A[t_1...t_p]$ nor $B[t_1...t_p]$, then we can choose $\Im'$ arbitrarily on $L[t_1...t_p]$ and $\neg L[t_1...t_p]$. We first show that $\Im$ cannot satisfy both $A[t_1...t_p]$ and $B[t_1...t_p]$. This is because Φ' implies $(A[x_1...x_p] \wedge B[x_1...x_p] \Rightarrow \mathit{false})$ and $\Phi' \subseteq \Psi$. Therefore such a structure $\Im'$ exists.
Now we show that $\Im'$ satisfies the C_i and D_j.
If there are some occurrences of instances of L or $\neg L$ remaining in the unmodified part of Ψ, then it may be that $\Im$ and $\Im'$ disagree on some ground instances of L or $\neg L$, since the clauses C_i and D_j implying $L \Leftrightarrow A$ and $\neg L \Leftrightarrow B$ may be removed from Φ by this transformation. That is, we may have $\Im \models N$ and $\Im' \not\models N$ for some instance N of L or $\neg L$. However, if Ψ has no such literals unifiable with L $\neg L$, this will not matter. Also, since $\Im$ is a model of Ψ, $\Im'$ is a model of Φ. Since $\Im$ satisfies at least one literal of each clause in Ψ, $\Im'$ satisfies at least one literal of each clause in Φ. The only time there can be a difference between the way $\Im$ and $\Im'$ treat clauses in Ψ and Φ, respectively, is when $\Im$ fails to satisfy either $A[t_1...t_p]$ or $B[t_1...t_p]$. In this case, $\Im'$ will satisfy one of $L[t_1...t_p]$ and $\neg L[t_1...t_p]$; this will only have the effect of possibly making more literals of Φ true. However, the remaining construction of $\Im'$ guarantees that at least one literal of each clause in Φ is satisfied by $\Im'$, so this does not matter. $\triangleleft$

We can also define a *lazy version* of the transformation in a slightly different way:

Definition 5.5 (Lazy Transformation)
Let Ψ be Φ with some or all occurrences of clauses $C \cup \{L[t_1...t_p]\}$ replaced by $C \cup \{A[t_1...t_p]\}$ and some or all occurrences of clauses $C \cup \{\neg L[t_1...t_p]\}$ replaced by $C \cup \{B[t_1...t_p]\}$, for all terms t_i. Thus we replace some or all instances of L and $\neg L$ by corresponding instances of A and B. We allow this transformation to be applied to more than one literal of a clause of Φ. However, the clauses $C_1...C_m$ and $D_1...D_n$ are retained in Ψ and not transformed, nor are the clauses in Φ' transformed. The clauses in Φ' may contain literals that unify with L or $\neg L$. $\triangleleft$

Theorem 5.6 (Faithfulness of the Lazy Transformation)
Ψ is unsatisfiable iff Φ is unsatisfiable.

Proof: The proof is much as before in the eager case. We can show as before that if Ψ is unsatisfiable then Φ is unsatisfiable. Suppose Ψ is satisfiable; then it has a model $\Im$ as before. From this we construct a model $\Im'$ as above. The difference is that we still have the clauses C_i and D_j in Φ, so that $\Psi \Rightarrow (L \Leftrightarrow A) \wedge (\neg L \Leftrightarrow B)$. This means that for any ground instance N of L or $\neg L$, $\Im \models N$ iff $\Im' \models N$. Reasoning as above, $\Im'$ is a model of Φ. $\lhd$

Note that if the eager version of the transformation is used, Ψ will not contain any instances of L or $\neg L$. Thus we have simplified the set Φ, in some sense; this is especially true if L or $\neg L$ is of the form $P(x_1...x_p)$ where P is a predicate symbol; in this case, all occurrences of P are eliminated. If the clauses C_i and D_j contain other literals unifiable with L and $\neg L$, then the lazy version must be used and we cannot necessarily eliminate all instances of L or $\neg L$. However, in this case, we can sometimes use complete refinements of resolution to show that the clauses C_i and D_j are unnecessary.

If the clauses C_i and D_j are not present in Φ, or are too difficult to detect, we can try to create them. One way to do this is by resolving clauses of Φ. There may be other ways too.

Note that the process of finding and applying such a transformation is completely automatic – it could be added as a preprocessing routine to a theorem prover, to search for such literals L and systematically replace them as indicated above. This can be made semi-interactive by having the prover search for such transformations and query the user about whether they should be done. No general argument about semantics is necessary for this, just a check whether the C_i' and D_j' are unsatisfiable, which can be done by a limited search for a refutation of some kind.

A slightly similar transformation is used with good results in the paper [PP91]. The value of replacing predicates in set theory by their definitions in improving the efficiency of a theorem prover is also shown in a paper by Plaisted and Greenbaum, [PG86].

We now give some general conditions guaranteeing the existence of the clauses C_i and D_j as required by this method. Suppose we have a definition in the form $L \Leftrightarrow W$ where L is a literal and W is some first-order formula. This could be something like $subset(X,Y) \Leftrightarrow (\forall Z)(Z \in X \Rightarrow Z \in Y)$. We consider this as universally quantified, i.e., $(\forall x_1...x_n)(L = W)$. If we convert this to clause form in the usual way we get clauses for $(L \Rightarrow W)$ and $(W \Rightarrow L)$. The clauses for $(L \Rightarrow W)$ will be of the form $\neg L \vee C_i''$ and those for $(W \Rightarrow L)$ will be of the form $D_j'' \vee L$. Also, the clause form for W is just $\vee_i C_i''$ and that for $\neg W$ is $\vee_j D_j''$. Now, if we apply the above idea for automatically detecting killer transformations to eliminate L, we replace the variables of L and $\neg L$ by constants, obtaining L' and $\neg L'$, respectively, and do resolutions with L' and $\neg L'$. This gives us the clauses D_j'' (from L') and C_i'' (from $\neg L'$), with variables in L and $\neg L$ replaced by constants; we indicate these by D_j' and C_i', respectively. Then $\wedge_j D_j' \wedge \wedge_i C_i'$ is the clause form for $(\exists x_1...x_n)(W \wedge \neg W)$. This formula is unsatisfiable. Since the conversion to clause form is satisfiability preserving, $\wedge_j D_j' \wedge \wedge_i C_i'$ is also unsatisfiable. Thus we could automatically detect these clauses C_i and D_j as indicated above.

The point of this is that if we actually include definitions of something in the input clauses, and translate them into clause form in the usual way, then this method is capable of automatically extracting them again and using them for definitional replacement. So if we give a collection of set theory axioms and theorems of set theory, then a prover could automatically (without being told that the input is set theory) rediscover these definitions and replace set operations by their definitions and thus become a much better prover for set theory. This means the user would not have to indicate definitions in any special way. Or if the input contained definitions 'by accident' that the user didn't know about, they could be detected.

There is another characteristic that the clauses C_i and D_j often have. Suppose that C_i is (up to renaming of variables) $C_i'' \cup \{\neg L\}$ and D_j is $D_j'' \cup \{L\}$. Suppose that these occurrences of $\neg L$ and L are resolved away to obtain C_i' and D_j'. Then A is $C_1'' \wedge ... \wedge C_m''$ and B is $D_1'' \wedge ... \wedge D_n''$. We do not transform the clauses C_i and D_j in the computation of Ψ. However, if we were to transform the clauses C_i and D_j, we would obtain the clauses $C_i'' \vee B$ and the clauses $D_j'' \vee A$. Since A is $C_1'' \wedge ... \wedge C_m''$ and B is $D_1'' \wedge ... \wedge D_n''$, we would obtain essentially the clauses $C_i'' \vee D_j''$ for all i and j. The conjunction of all these clauses $C_i'' \vee D_j''$ is equivalent to $A \vee B$. If $A \vee B$ is a tautology, then the clauses $C_i'' \vee D_j''$ are tautologies.

We can give some general conditions under which $A \vee B$ will be a tautology. For example, suppose that S was obtained by converting first-order formulae into clause form, and one of these formulae was of the form $L \Leftrightarrow W$. Then this is equivalent to $L \Rightarrow W$ and $W \Rightarrow L$. When this is converted to clause form, we will obtain clauses of the form $\neg L \vee C_i''$ and of the form $L \vee D_j''$ where $C_1'' \wedge ... \wedge C_m''$ is the clause form of W and $D_1'' \wedge ... \wedge D_n''$ is the clause form of $\neg W$. In this case, we have that $C_i'' \vee D_j''$ may be a tautology for all i and j, because these clauses arise when converting $W \vee \neg W$ to clause form. If W contains no quantifiers, then the transformation to clause form is equivalence preserving and then $C_i'' \vee D_j''$ will be a tautology for all i and j. However, if W contains quantifiers, then the transformation to clause form is only satisfiability preserving, not validity preserving. Therefore, even though $W \vee \neg W$ is a tautology, it is not necessary that $A \vee B$ be a tautology.

When we are looking for the clauses C_i and D_j, we can prefer those such that $A \vee B$ is a tautology. This doesn't necessarily make the method more powerful, but it does help to find clauses that correspond in a natural way to definitions. Note that this test for tautology and logical consequence can be done automatically by a theorem prover, in the sense of partial decidability.

For example, if we have the following definition: $x \in (y \cap z) \Leftrightarrow (x \in y \wedge x \in z)$. Here L is the formula $x \in (y \cap z)$. When we convert this to clause form, we obtain the following clauses:

$x \in (y \cap z) \Rightarrow x \in y$

$x \in (y \cap z) \Rightarrow x \in z$

$(x \in y) \wedge (x \in z) \Rightarrow x \in (y \cap z)$

Now, we have L is $x \in (y \cap z)$ and $\neg L$ is $\neg(x \in (y \cap z))$. Then A is $(x \in y \wedge x \in z)$ and B is $(\neg x \in y) \vee (\neg x \in z)$. Note that $A \vee B$ is a tautology and that $A \wedge B$ is unsatisfiable. Also, we have C_1'' is $x \in y$ and C_2'' is $x \in z$ and D_1'' is $(\neg x \in y) \vee (\neg x \in z)$. We can verify that $C_i'' \vee D_j''$ is a tautology for all i and j.

Unfortunately, the C_i'' and D_j'' contain literals that unify with L and $\neg L$, so we cannot use the eager version of the transformation here. This can be helped by introducing a new version of the '$\in$' predicate, but when this is used it will not create tautologies and so the transformed versions of the C_i and D_j must still be retained.

Another example comes from the conversion of the definition of subset to clause form, as follows:

$subset(x, y) \Leftrightarrow (\forall z)(z \in x \Rightarrow z \in y)$

For this definition, there are no occurrences of *subset* in the right-hand side, so the eager version of the transformation can be used. In this way, the set Ψ will not contain any occurrences of the *subset* predicate. We note that a quantifier is needed to express the definition of subset. However, we obtain the effect of replacing *subset* by its definition even in a Skolemized setting in which the quantifier has been removed. Since the right-hand side contains a quantifier, $A \vee B$ is not a tautology.

A further example of the value of this transformation approach is if the Boolean connectives are defined by statements like $true(X) \wedge true(Y) \Rightarrow true(and(X, Y))$, $true(and(X, Y)) \Rightarrow true(X)$, $true(and(X, Y)) \Rightarrow true(Y)$, and similarly for other Boolean connectives. Then using the transformation technique, we can detect that predicates of the form $true(and(X, Y))$ are defined and can be eliminated in favor of $true(X) \wedge true(Y)$. Now, since the definitions contain other literals that unify with L and $\neg L$, we cannot use the eager versions of the transformations. This means that it is necessary to retain the definition of 'and' in Ψ. However, if we use ordered resolution, then the literals containing 'and' will be the largest literals in their clauses. It is possible that in Ψ they will not resolve with anything else except other literals in the definition of 'and.' This will produce only tautologies, which can be deleted. In this way, we eliminate all occurrences of 'and.' The same procedure can be applied to the other Boolean connectives. Note that if Φ contains no other function symbols than 'and', 'or', and 'not', and their definitions can be eliminated in this way, then the transformed Φ will contain no function symbols at all. In this way we transform a problem into a simpler one that is in a decidable sublanguage of first-order logic. We note that the definition of $x \in (y \cap z)$ may sometimes be eliminated in the same way, if ordered resolution is used after the transformation to Ψ. This is possible because $A \vee B$ is a tautology, showing an advantage of choosing transformations with this tautology property.

6 Summary

We have presented methods for developing transformations of logical systems which guarantee that formulae are theorems in the original system if and only if the transformed formulae are theorems in the transformed system. With examples from very different areas of classical and non–classical logics we have demonstrated that the transformations are very easy to apply. The proof obligations are extremely simple in most cases. Finding the transformers also seems to be quite straightforward. This makes K-transformations a very powerful tool for generating and investigating transformations of logical systems and in particular for improving the performance of automated theorem provers.

References

[AE92] Yves Auffray and Patrice Enjalbert. Modal theorem proving: An equational viewpoint. *Journal of Logic and Computation*, 2(3):247–297, 1992.

[BG90] Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplification. In *CADE-10: 10th International Conference on Automated Deduction*, Lecture Notes in Artificial Intelligence, pages 427–441, Kaiserslautern, FRG, 1990. Springer-Verlag. Copy filed.

[Bra75] Daniel Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4(4):412–430, 1975.

[Che80] B. F. Chellas. *Modal Logic: An Introduction.* Cambridge University Press, Cambridge, 1980.

[Eis91] Norbert Eisinger. *Completeness, Confluence, and Related Properties of Clause Graph Resolution.* Research Notes in Artificial Intelligence. Pitman Ltd., London, 1991.

[Gas92] Olivier Gasquet. Deduction for multimodal logics. In *Proc. of Applied Logic Conference (Logic at Work). Amsterdam*, December 1992.

[Her89] Andreas Herzig. *Raisonnement automatique en logique modale et algorithmes d'unification.* PhD thesis, Université Paul-Sabatier, Toulouse, 1989.

[JR88] Peter Jackson and Han Reichgelt. A general proof method for modal predicate logic without the Barcan Formula or its converse. DAI Research Report 370, Department of Artificial Intelligence, University of Edinburgh, 1988.

[Łuk70] J. Łukasiewicz. *Selected Works.* North Holland, 1970. Edited by L. Borkowski.

[McC89] William W. McCune. *OTTER User's Guide.* Mathematical and Computer Science Devision, Argonne National Laboratory, april 1989.

[McC90] William McCune. OTTER 2.0. In Mark Stickel, editor, *Proc. of* 10^{th} *Internation Conference on Automated Deduction, LNAI 449*, pages 663–664. Springer Verlag, 1990.

[Mer53] C. A. Meredith. Single axioms for the systems (C,N), (C,0), and (A,N) of the two-valued propositional calculus. *Journal of Computing Systems*, 1:155–164, 1953.

[MW92] Williman McCune and Larry Wos. Experiments in automated deduction with condensed detachment. In Deepak Kapur, editor, *Autmated Deduction – CADE 11, Lecture Notes in AI, vol. 607*, pages 209–223. Springer Verlag, 1992.

[Non93] Andreas Nonnengart. First-order modal logic theorem proving and functional simulation. In Ruzena Bajcsy, editor, *Proceedings of the 13th IJCAI*, volume 1, pages 80 – 85. Morgan Kaufmann Publishers, 1993.

[Ohl88a] Hans Jürgen Ohlbach. A resolution calculus for modal logics. In Ewing Lusk and Ross Overbeek, editors, *Proc. of* 9^{th} *International Conference on Automated Deduction, CADE-88 Argonne, IL*, volume 310 of *Lecture Notes in Computer Science*, pages 500–516, Berlin, Heidelberg, New York, 1988. Springer-Verlag. Extended version appeared in [Ohl88b].

[Ohl88b] Hans Jürgen Ohlbach. A resolution calculus for modal logics. SEKI Report SR-88-08, FB Informatik, Universität Kaiserslautern, Germany, 1988. PhD Thesis, short version appeared in [Ohl88a].

[Ohl90] Hans Jürgen Ohlbach. Semantics based translation methods for modal logics. SEKI Report SR-90-11, FB. Informatik, Univ. of Kaiserslautern, 1990. Finally published in [Ohl91].

[Ohl91] Hans Jürgen Ohlbach. Semantics based translation methods for modal logics. *Journal of Logic and Computation*, 1(5):691–746, 1991.

[Ohl93] Hans Jürgen Ohlbach. Optimized translation of multi modal logic into predicate logic. In Andrei Voronkov, editor, *Proc. of Logic Programming and Automated Reasoning (LPAR)*, volume 698 of *Lecture Notes in Artificial Intelligence*, pages 253–264. Springer Verlag, 1993.

[Ohl94] Hans Jürgen Ohlbach. Synthesizing semantics for extensions of propositional logic. Technical Report MPI-I-94-225, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1994.

[PG86] David Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, 1986.

[PP91] David Plaisted and Richard Potter. Term rewriting: some experimental results. *Journal of Symbolic Computation*, 11:149–180, 1991.

[Rob65a] John Alan Robinson. Automated deduction with hyper-resolution. *International Journal of Computer Mathematics*, 1(3):227–234, 1965.

[Rob65b] John Alan Robinson. A machine-oriented logic based on the resolution principle. *Journal of the Association for Computing Machinery (JACM)*, 12(1):23–41, 1965.

[Sho76] Robert E. Shostak. Refutation graphs. *Artificial Intelligence*, 7(51-64), 1976.

[Sto36] M. H. Stone. The theory of representations for boolean algebras. *Transactions of American Mathematical Society*, 40:37–111, 1936.

[Wal87a] Lincoln A. Wallen. Matrix proof methods for modal logics. In *Proc. of* 10^{th} *IJCAI*, pages 917–923. IJCAI, Morgan Kaufmann Publishers, 1987.

[Wal87b] Christoph Walther. *A Many-Sorted Calculus Based on Resolution and Paramodulation.* Research Notes in Artificial Intelligence. Pitman Ltd., London, 1987.

[Zam89] N.K. Zamov. Modal resolutions. *Izvestiya VUZ. Mathematika*, 33(9):22–29, 1989. Also published in Soviet Mathematics, Allerton Press.

Below you find a list of the most recent technical reports of the research group *Logic of Programming* at the Max-Planck-Institut für Informatik. They are available by anonymous ftp from our ftp server `ftp.mpi-sb.mpg.de` under the directory pub/papers/reports. If you have any questions concerning ftp access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Regina Kraemer
Im Stadtwald
D-66123 Saarbrücken
GERMANY
e-mail: `kraemer@mpi-sb.mpg.de`

MPI-I-94-216	P. Barth	Linear 0-1 Inequalities and Extended Clauses
MPI-I-94-209	D. A. Basin, T. Walsh	Termination Orderings for Rippling
MPI-I-94-208	M. Jäger	A probabilistic extension of terminological logics
MPI-I-94-207	A. Bockmayr	Cutting planes in constraint logic programming
MPI-I-94-201	M. Hanus	The Integration of Functions into Logic Programming: A Survey
MPI-I-93-267	L. Bachmair, H. Ganzinger	Associative-Commutative Superposition
MPI-I-93-265	W. Charatonik, L. Pacholski	Negativ set constraints: an easy proof of decidability
MPI-I-93-264	Y. Dimopoulos, A. Torres	Graph theoretical structures in logic programs and default theories
MPI-I-93-260	D. Cvetković	The logic of preference and decision supporting systems
MPI-I-93-257	J. Stuber	Computing Stable Models by Program Transformation
MPI-I-93-256	P. Johann, R. Socher	Solving simplifications ordering constraints
MPI-I-93-250	L. Bachmair, H. Ganzinger	Ordered Chaining for Total Orderings
MPI-I-93-249	L. Bachmair, H. Ganzinger	Rewrite Techniques for Transitive Relations
MPI-I-93-243	S. Antoy, R. Echahed, M. Hanus	A needed narrowing strategy
MPI-I-93-237	R. Socher-Ambrosius	A Refined Version of General E-Unification
MPI-I-93-236	L. Bachmair, H. Ganzinger, C. Lynch, W. Snyder	Basic Paramodulation
MPI-I-93-235	D. Basin, S. Matthews	A Conservative Extension of First-order Logic and its Application to Theorem Proving
MPI-I-93-234	A. Bockmayr, F. J. Radermacher	Künstliche Intelligenz und Operations Research
MPI-I-93-233	A. Bockmayr, S. Krischer, A. Werner	Narrowing Strategies for Arbitrary Canonical Rewrite Systems
MPI-I-93-231	D. Basin, A. Bundy, I. Kraan, S. Matthews	A Framework for Program Development Based on Schematic Proof