W. BIBEL, S. BRÜNING, J. OTTEN, T. RATH AND T. SCHAUB

COMPRESSIONS AND EXTENSIONS

1. INTRODUCTION

This chapter introduces special techniques for enhancing proof systems in terms both of their efficiency as of their range of applicability. These techniques have been developed in the context of the implementation of the KoMeT theorem proving system to which we will occasionally allude in the text. Gains in efficiency are achieved through the general principle of compression which will be one of the two main leitmotivs of this chapter while the other concerns applications other than theorem proving in first-order logic made possible by extensions of the underlying methods. We begin this introduction with a general view on proof systems.

Theorem proving is achieved through searching for a derivation in a particular logical calculus of a given formula. Technically this may be seen as a search through a graph whose nodes represent partial derivations and whose arcs connect nodes whenever one partial derivation is a direct refinement (or extension) of the other (cf. Figure 2 of Chapter 2). Work in theorem proving tries to develop representations which avoid as much redundancy as possible, minimize the work to be done in performing the involved operation, and minimize the amount of nodes in the search space to be considered before a proof is found.

The connection method (Bibel, 1983) provides the basis for the most compact representation of partial derivations in terms of the amount of code attached to the formula to be proved that we know of. For this reason the work pursued in our research group uses this method as the basis for developing advanced proof systems. Some of the research results achieved in this pursuit are realized in the system KoMeT (Bibel et al., 1994a; Bibel et al., 1994b; Bibel et al., 1995).

The goal of such a system is to identify a set of connections in the formula which spans it. There are numerous procedural ways to achieve this goal. The simplest of them is called *extension* procedure (Bibel, 1993) (closely related to the well-known model elimination procedure — cf. Chapter 2) which is the basis of KoMeT. The system is programmed in Prolog in a way which tries to avoid one of the drawbacks of big dinosauriers among existing provers,

namely their inflexibility for modifications. High performance is not the topmost priority in the KoMeT project as the choice of Prolog indicates. Nevertheless its core is realized in PTTP technology (Stickel, 1988) so as to achieve reasonable run-times.

The lessons learnt in the past decades of AD research teach that a uniform basis is of great advantage for designing large and complex systems but that not a single (and simple) search strategy suffices to achieve a satisfactory performance. Rather one has to take into account a variety of different features formulas may exhibit and design special strategies for each of them. It has been the basic concept behind the development of KoMeT to integrate as many special strategies as possible in order to approximate the behavior of an *adequate* prover, a term introduced in (Bibel, 1991).

One of these features concerns the fact that formulas often exhibit redundant structures in their parts so that the given formula may be reduced in various ways before the proof search starts. These reductions realize the general principle of *compression* (Bibel, 1993; Bibel, 1991) which is used in AD research all over. Already on the level of the logical calculus there may be more or less compressed versions of calculi (as already pointed out in the introduction to this part of the book). As a rule the more compressed the calculi are, the more suitable they are for automation. Hence our preference for the connection calculi.

In the case of formulas their compression aims at getting a more compact representation of essentially the same information. Some deductive systems still fail to appreciate the importance of this part of deductive proof search. By transformation of the formula to normal form they sometimes even expand the formula rather than compress it. The least what has to be expected from an advanced system is that the technique of definitional transformation (Bibel, 1993) is used which limits the expansion in a quadratic — in propositional logic even linear — way and avoids the destruction of the formula's original structure. But the best solution would be to avoid an expansion altogether and rather compress it as far as possible.

The system KoMeT offers the definitional transformation to normal form as an option and additionally offers a sophisticated preprocessing of the resulting formula which reduces the formula in this way. Since most of the techniques used in this part have been described elsewhere (Bibel et al., 1994a; Bibel et al., 1994b) their description will not be repeated in this chapter. A special form of preprocessing is realized by database (or DB-) reductions. These are particularly effective if many facts are involved in the problem specification. Since they interact with the subsequent unification processes they require what is called DB-unification. Both are presented in detail in Section 2.

Techniques similar to DB-reduction abound in AD research. Identical ancestor pruning, its generalization in terms of regular proofs, unit and local lemmata, and failure caching (Letz et al., 1994) all share with DB-reduction the same principle which is formula compression so that proof parts have not to be repeated on similar structures over and over again. We cover in this chapter two further techniques falling into this category. One is a propositional prover based on the Davis-Putnam procedure but generalized for non-normal form formulas, presented in Section 3. Further, in Section 4.1, a technique for taking advantage of equivalences is shortly summarized. By making equivalences explicit, it is possible to take over one of the most important reduction techniques developed for equality reasoning, namely demodulation, to problems not explicitly noted in terms of equality. Additionally, equivalences allow to strengthen the important regularity refinement.

A different category of techniques such as tautology and subsumption constraints may rather be seen under the aspect of information compression in a way that the relevant information is available at the *right* point and supports the right decision in the selection of proof alternatives. In some cases it also allows the application of reduction rules dynamically as the necessary information has been collected to allow the compression of the formula. In Section 4.2 we summarize results showing that subsumption deletion can be successfully integrated in connection calculi: rather than comparing clauses like in resolution-based calculi, the basic proof objects underlying the respective connection calculus — i.e. connection graphs or connection tableaux — or only parts of them are compared via subsumption.

Apart from striving for the ultimate techniques in AD our group has also kept a busy eye on the applications of AD. In the course of these studies it turned out that different applications tend to demand extra features from our systems. In all cases these features can easily be integrated into the general system which in our case is KoMeT. We mention here three different such applications and their extensions.

One of the most urgent applications of deduction is in programming. Verification is where most people locate the role of AD in programming. But a much greater potential is in program synthesis which subsumes verification. One way to realize program synthesis is by stating the programming problem in a logical language and extract a program from a constructive proof once it is found. So the task in our context is to enable the use of a classical theorem prover or at least of its deductive techniques for the problem of finding constructive proofs. A multi-level approach for doing this has been described in detail in (Bibel et al., 1997; Bibel et al., 1996).

Another promising application of deduction is planning. It turned out that the most appropriate logic for planning is a resource-sensitive logic such as

the transition logic described in detail in (Bibel, 1997). Parts of this logic have been realized in systems for classical logic such as SETHEO.

The last application which we want to mention here is non-monotonic reasoning. As with the aforementioned resource-oriented logic, we are faced here with an extension to classical logic, namely the capability of drawing inferences under certain consistency assumptions. This is actually what makes this form of reasoning *non-monotonic*, since such assumptions may have to be withdrawn in the light of subsequent information. Nonetheless, we will see in Section 4.3 that this additional dimension of reasoning can be integrated homogeneously on the level of the logical calculus. Importantly, this carries over to the system level, as witnessed by its implementation through PTTP technology.

In contrast to the next two sections Section 4 just surveys the results presented there because all its material has been published at other places in detail specified in the text. On the other hand it demonstrates the breadth of techniques necessary for striving at an adequate and versatile prover. In Section 5 conclusions are drawn from the results presented in this chapter.

2. DB-REDUCTION AND DB-UNIFICATION

2.1. *Motivation*

All proof procedures with a rigid search control suffer from the fact that there are always theorems which are worst cases for the chosen search regime but would be easily proved with a different one. This can be illustrated with the left part of Figure 1 picturing the search tree for some given example.

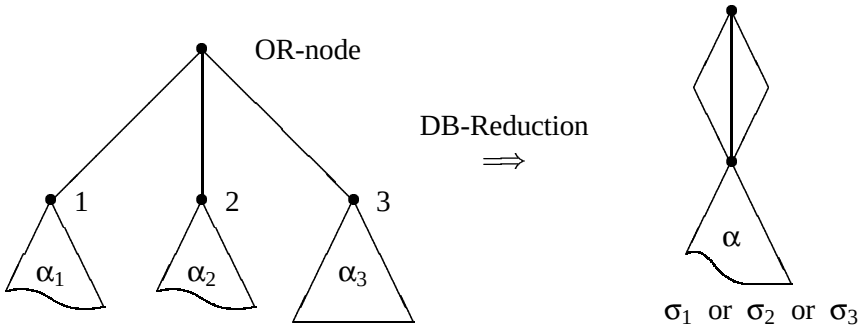

Figure 1. Illustration of a search tree with α_3 representing the successful proof

Assume our procedure follows a left-to-right control and the node labeled 3 happens to be the one leading to a successful proof. Then the proof attempts

α_1 and α_2 would unsuccessfully be carried out (thus wasting possibly enormous computational efforts) before encountering the successful proof α_3. In certain cases the proof attempts α_i are structurally identical but differ in particular instantiations only. These are the ones of interest in the present section. They allow the avoidance of the waste of efforts just noted with little overhead.

In a nutshell the solution for achieving this enhancement presented in the present section consists of separating the structural formula information from the instantiational one whereby the latter is a set of substitutions used in a constraint. This way the proof search has to be carried out only once since the instantiational part can now be handled in a unificational way. In terms of our illustration we get the right part of Figure 1, an obvious improvement.

This kind of enhancement obviously falls into the category of *compression* techniques as the original search tree has been compressed into a strictly smaller one. In this case the compression was made possible by application of unificational techniques which generally apply a *lazy-evaluation* or *by-need* technique. A concrete example for this compression technique is the following formula presented in Prolog-like notation.

$$
\begin{array}{lll}
F_1 & p(a_1,b_1). & \\
\vdots & \vdots & \\
F_k & p(a_k,b_k). & \\
F_{k+1} & q(a_k,V). & \\
R & q(f(W),Z) & :- \quad q(W,Z). \\
Q & & :- \quad p(X,Y),q(f^m(X),Y).
\end{array}
$$

The formula consists of the facts $F_1,\ldots,F_{k+1}$, $k \geq 2$, the rule R, and the query Q. The notation $f^m(X)$, $m \geq 1$, abbreviates the term with m applications of the function f. The standard Prolog regime has to attempt $k-1$ unsuccessful alternatives (corresponding to the proof attempts α_1 and α_2 in Figure 1) until the successful proof (corresponding to α_3) may be found. Each of these proof attempts requires m instances of rule R, hence altogether $k \cdot m$ instances (and $(k-1)\cdot(m+1)$ backtracking-steps). The corresponding compressed program, obtained by what is called *DB-reduction*, is the following one.

$$
\begin{array}{lll}
F_* & p(U_1,U_2) & :- \quad (U_1,U_2) \in \{(a_1,b_1),\ldots,(a_k,b_k)\}. \\
F_{k+1} & q(a_k,V). & \\
R & q(f(W),Z) & :- \quad q(W,Z). \\
Q & & :- \quad p(X,Y),q(f^m(X),Y).
\end{array}
$$

In contrast to the earlier one this program requires only m instances and no backtracking at all, because, instead of unifying the literal $p(X,Y)$ of the

query with a single fact, it is memorized that such a unification binds the variable-pair (X,Y) to one of the term-pairs of the set $\{(a_1,b_1),\ldots,(a_k,b_k)\}$. It remains to prove the goal $q(f^m(X),Y)$, which can be done by first using m instances of the rule R and then using the fact F_{k+1}. As a result X is bound to the constant a_k and Y to the variable V. Now we have to consider that the variable-pair (X,Y) must be unifiable with at least one of the term-pairs of the set $\{(a_1,b_1),\ldots,(a_k,b_k)\}$ and obviously the last member of this set can be used for this unification. So the (small) price to be paid for this improvement is the constraint $(U_1,U_2) \in \{(a_1,b_1),\ldots,(a_k,b_k)\}$ of F_*, which stores the instantiational information of the original facts, and a slightly more complex unification process (called *DB-unification*), which takes into account these substitution constraints. The price is worth paying as this example and many others from practice have demonstrated (see Subsection 2.5 below).

DB-unification was first introduced in (Bibel et al., 1987) in a preliminary form (see also (Bibel, 1988)). A brief informal description of its technically advanced form (using abstraction trees — see below) is contained in Section 4.4.3 of (Bibel, 1993). The present section offers the first comprehensive treatment which, in addition, is not restricted to DB-reduction on unit clauses.

In Subsection 2.2 we introduce definitions and notations concerning abstraction trees, the data structure we are using to store and handle substitution constraints efficiently. In Subsection 2.3 we give a general definition of DB-reduction, which merges arbitrary clauses consisting of similar literals during a preprocessing step. DB-unification, which handles the data-structures produced by the DB-reduction is defined in Subsection 2.4. In Subsection 2.5 we present some results on the performance of DB-reduction and DB-unification in practice.

2.2. *Preliminaries*

In this subsection we provide definitions and notations needed for the definition of DB-reduction and DB-unification. We use the small letters p,q to denote predicates, s,t for term lists, a,b,c,d for constants, f,g for functions, capital letters U,V,W,X,Y,Z for variables. These designators may be provided with indices. Furthermore we will denote the substitution of a variable X by a term t with $X\backslash t$. The notation $\langle X_1,\ldots,X_n\rangle\backslash\langle t_1,\ldots,t_n\rangle$ is a shorthand for the substitutions $\{X_1\backslash t_1,\ldots,X_n\backslash t_n\}$.

An abstraction tree (Ohlbach, 1990) allows a compact representation of some set of term lists (which will be used to define DB-reductions and to represent the term lists of the substitution constraints mentioned in the previous subsection).[1] For example let us consider the term lists $t_1 = \langle X, f(a,Y)\rangle$,

[1] A short discussion of abstraction trees is also contained in Chapter 1 of Part 2.

$t_2 = \langle a, f(a,g(b))\rangle$, $t_3 = \langle g(b), f(a,c)\rangle$. They may be presented as a labeled tree such that the label of a successor node is an instance of the label of its predecessor, as illustrated in Figure 2.

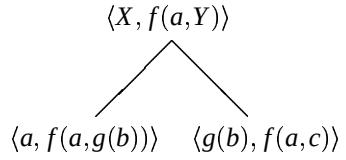

Figure 2. Tree representation of a set of term lists

The same information can be represented in an even more compact form, because it is sufficient to state the substitutions needed to reproduce the term lists t_2 and t_3 out of t_1. This yields the abstraction tree of Figure 3 (where in addition parentheses are omitted).

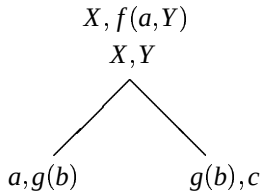

Figure 3. Simple abstraction tree

In this representation the list of variables occurring in the term list, like $\langle X,Y\rangle$ attached to the root of the present tree, is made explicit in order to emphasize that it represents the domain and the term lists of each leaf the codomain of a substitution. Application of this substitutions to the term list at the root yields the term list at the respective leaf of the original tree. For instance, the term list t_2 is obtained from this representation by application of the substitution $\langle X,Y\rangle \setminus \langle a,g(b)\rangle$ to t_1; similarly for t_3 with $\langle X,Y\rangle \setminus \langle g(b),c\rangle$ applied to t_1. To use abstraction trees as an indexing mechanism for term lists, the convention is adopted that only leaves (more precisely paths to the leaves) of an abstraction tree represent term lists. With this convention the tree in Figure 3 represents the term lists t_2 and t_3. In formal details abstraction trees are defined as follows.

DEFINITION 2.1. *The* variable list $VL(t)$ *of a term list is the list* $(X_1,\ldots,X_n)$ *of the variables occurring in t ordered according to their first occurrences; ie. to the left of the first occurrence of* X_i *in t only the variables* $X_1,\ldots,X_{i-1}$ *occur, for* $i = 1,\ldots,n$. *An* abstraction tree *AT is a tree whose nodes are labeled with term lists whereby the following property holds. If N is any node in AT,*

t its label, N' an immediate successor node of N in AT, and t' the label of N', then $VL(t)$ is the domain and t' the codomain of a substitution. Any node N in AT represents a term list $T(N)$ obtained from the term list labeling the root R by applying to it the substitutions on the branch from R to N. $T(AT)$, the set of terms represented by AT is the set of terms represented by the leaves of AT.

The example given in Figure 3 is special insofar as it has no inner nodes, that is, nodes other than the root and leaves. By these inner nodes the actual indexing of the term lists represented by the tree is done. A simple example with an inner node is given in Figure 4, whose abstraction tree represents the term lists $t_1 = \langle a, f(a, g(b))\rangle$, $t_2 = \langle g(a), f(a, c))\rangle$ and $t_3 = \langle g(b), f(a, c)\rangle$.

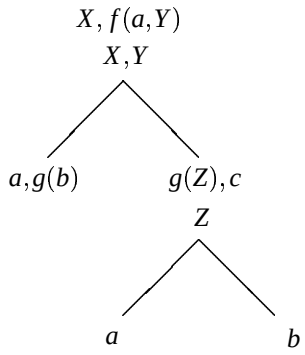

Figure 4. Abstraction tree with an inner node

Since we can represent all our examples using abstraction trees without inner nodes we will restrict the entire section to this simpler and easier-to-read form. We refer the reader interested in aspects of the more general notion to Chapter 1 in Volume 2 or to (Ohlbach, 1990; Rath, 1992).

To use abstraction trees for the representation of term lists, an operational way has to be devised how to generate a generalized term list (meant to label the root of an abstraction tree) from a given set of term lists. Such a common generalization exists only if the term lists in question have equal length. This is in fact the only restriction for using abstraction trees for indexing term lists and is a harmless one for our applications.

In order to use abstraction trees as an indexing method for DB-reductions and DB-unification, we need three operations. The first operation *build-at* $(TermLists, AT)$ builds for a given set *TermLists* of term lists of equal length an abstraction tree *AT*, whereby duplicates or instances of term lists within *TermLists* are deleted. This operation will also play a role in the definition of DB-reductions.

The second operation *get-unified-term-lists*$(AT, TermList, UTermLists)$ is needed in the algorithm for DB-unification and identifies all term lists in the abstraction tree *AT* which are unifiable with a given term list *TermList* and represents them, modified by the determined unifier, in the list of term lists *UTermLists*. If no term list represented by the abstraction tree is unifiable with the given term list, this operation creates the empty list $\langle\rangle$. Both operations are given in (Rath, 1992) (or may be constructed by use of operations described in (Ohlbach, 1990)).

The third operation also needed in the algorithm for DB-unification is a join operation, which is similar to the natural join known from databases (Ullman, 1982). The main difference between a join on abstraction trees and the natural join on database entries is that unification is not involved in the latter. For illustration let us join the abstraction trees of Figure 5, where the left abstraction tree represents the term lists $t_1 = \langle f(a), g(b)\rangle$, $t_2 = \langle f(c), d\rangle$, $t_3 = \langle f(b), W\rangle$, and the right tree represents the term lists $s_1 = \langle a, f(a, g(b))\rangle$, $s_2 = \langle g(b), f(a,c)\rangle$.

Figure 5. Abstraction trees for join operation

More precisely, the join is to be performed on the second term of (each of the nodes of) the left tree and the first term of the right one, to obtain a list of term lists of the following kind. For instance, t_1 and s_2 yield $t_4 = \langle f(a), g(b), f(a,c)\rangle$ whereby the second term derives from the respective two joined terms in t_1 and s_2. Formally, the join operation is denoted by *get-join-of-term-lists*$(AT_1,\ Pos_1,\ AT_2,\ Pos_2,\ JoinTermLists)$. Here AT_i denotes the given abstraction trees, Pos_i, $i = 1, 2$, the positions to be joined, and *JoinTermLists* the resulting term lists (the empty list if there are none). Apart from t_4 the operation yields $t_5 = \langle f(b), a, f(a, g(b))\rangle$ from t_3 and s_1, and $t_6 = \langle f(b), g(b), f(a,c)\rangle$ from t_3 and s_2.

Besides the three operations just discussed we consider a special case of abstraction trees and several notions, especially the main data structure of a DB-term list, needed for the operations of DB-reductions and DB-unification introduced thereafter in the subsequent sections.

DEFINITION 2.2. *An abstraction tree AT is a* DB-abstraction tree, *if the term list of its root node is a list of pairwise different variables. The set of*

substitutions determined by a DB-abstraction tree AT is $\mathcal{S}(AT) = \{t\backslash t_i \mid t_i \in T(AT)\}$*, where t is the term list of the root node of AT.*

DEFINITION 2.3. *Let* $\mathcal{A}$ *be a set of DB-abstraction trees. If for each of* $\mathcal{A}$*'s variables there is exactly one DB-abstraction tree in* $\mathcal{A}$ *in which the variable occurs then* $\mathcal{A}$ *is called a* DB-abstraction tree set. *The set* $\mathcal{S}(\mathcal{A})$ *of all substitutions determined by a DB-abstraction tree set* $\mathcal{A}$ *is inductively defined as follows.*

1. $\mathcal{S}(\mathcal{A}) = \{\}$ *if* $\mathcal{A} = \{\}$*; otherwise*
2. $\mathcal{S}(\mathcal{A}) = \{\sigma \cup \tau \mid \sigma \in \mathcal{S}(AT), \tau \in \mathcal{S}(\mathcal{A}_1)\}$*, with* $AT \in \mathcal{A}$ *and* $\mathcal{A}_1 = \mathcal{A} \setminus \{AT\}$.

The set of term lists which can be created out of a term list t by the application of the substitutions stored in a DB-abstraction tree set $\mathcal{A}$ *is* $I(t, \mathcal{A}) = \{t\sigma \mid \sigma \in \mathcal{S}(\mathcal{A})\}$.

DEFINITION 2.4. *If t is a term list and* Σ *is a set of substitutions then the pair* $s = (t, \Sigma)$ *is called a* DB-term list *and* $I(s) = \{t\sigma | \sigma \in \Sigma\}$ *is the set of all term lists represented by s.*

In the following we will always use DB-abstraction tree sets to represent sets of substitutions in DB-term lists. We also could have used another data structure like a simple set of substitutions or a single DB-abstraction tree instead of the DB-abstraction tree set, but as in databases it is preferable to store information (if possible) not in one large data structure but in smaller pieces (see Section 2.5 for more detailed discussions concerning this issue).

2.3. *DB-Reduction*

In this subsection we formalize the notion of DB-reduction. A detailed illustration of this operation is given in Example 2.1.

EXAMPLE 2.1. *The following formula in* Prolog*-like notation, consists of the facts* F_1, F_2, F_3*, the rule R and the query Q.*

$$
\begin{array}{lllll}
F_1 & p(g(b), a, b). & & & \\
F_2 & p(g(a), a, c). & & & \\
F_3 & p(g(W), a, d). & & & \\
R & p(f(X), Y, Z) & :- & p(X, Y, Z). \\
Q & & :- & p(f^5(g(a)), a, d).
\end{array}
$$

Using DB-reduction it is possible to merge the facts F_1, F_2, F_3*. The first operation to be applied is build-at*$(\{F_1, F_2, F_3\}, AT)$ *which yields the abstraction tree shown in Figure 6.*

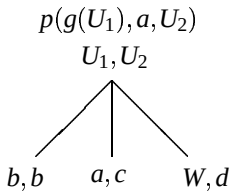

Figure 6. Abstraction tree AT for the facts F_1,F_2,F_3

Thereafter the abstraction tree AT of Figure 6 is transformed into the DB-term list $s=(t,\{AT'\})$,[2] *where t is the term list of the root of AT and* AT' *is identical to AT, except that the term list of the root is replaced by the list of variables of the root node. Because* $I(s)=\{p(g(b),a,b),\ \ p(g(a),a,c),\ p(g(W),a,d)\}$, *this DB-term list represents the facts* F_1,F_2,F_3. *Within this DB-term list the term list t represents the clause generated by the DB-reduction and the DB-abstraction tree set* $\{AT'\}$ *represents the constraints on the substitutions allowed for the variables in the literal of the clause. The* Prolog*-like program obtained from the application of DB-reduction is the following one.*

$$\begin{array}{lll} F & p(g(U_1),a,U_2) & :- \quad \langle U_1,U_2\rangle \in I(\langle U_1,U_2\rangle,\{AT'\}). \\ R & p(f(X),Y,Z) & :- \quad p(X,Y,Z). \\ Q & & :- \quad p(f^5(g(a)),a,d). \end{array}$$

After this illustration of DB-reduction its formal definition is presented which, for a beginning, is restricted to unit clauses.

DEFINITION 2.5. *Let* $\mathcal{C}$ *denote the set of all unit clauses with the same predicate symbol p and the same polarity in a given set M of clauses. If* $\mathcal{C}$ *has more than one element then we say* M' *is obtained from M by* DB-reduction *if* $M'=(M\setminus C)\cup\{c\}$ *whereby* $c=(t,\Sigma)$ *is called* DB-clause *and consists of a clause part t (which in the present case restricted to unit clauses is a unit clause) and a set of substitution constraints* Σ, *with t and* Σ *defined as follows.*

Let AT be defined by build-at$(\mathcal{C},AT)$,[3] *then t is the term list of the root of AT and* $\Sigma=\mathcal{S}(AT')$ *where* AT' *is the DB-abstraction tree created from AT by replacing the term list of the root by the variable list of the root.*

If $c=(t,\Sigma)$ *is a DB-clause, then every variable V, which occurs in t and in* Σ *is called a* DB-variable. *c* represents *the clauses of* $I(c)=\{t\sigma \mid \sigma\in\Sigma\}$.

LEMMA 2.1. *If M is a set of clauses and* M' *is created from M by the application of DB-reductions on unit clauses, then* M' *represents the same clauses as M except for unit clauses which can be deleted by subsumption.*

[2] We use the data structure DB-abstraction tree set to represent the set of substitutions in the DB-term list.

[3] Within this operation the clauses in $\mathcal{C}$ are interpreted as term lists.

Proof. According to Definition 2.5 DB-reduction affects unit clauses only and factors all unit clauses with one and the same predicate symbol and of the same polarity. According to the description of the operation *build-at* in Subsection 2.2, all subsumed clauses are deleted when building the abstraction tree AT. The rest is obvious from Definition 2.5.

Obviously DB-clauses are a special kind of DB-term lists insofar as DB-clauses are created by DB-reductions and so they represent a set of clauses instead of a set of term lists. Nevertheless a single literal of the clause part of a DB-clause together with the substitution constraints of the DB-clause can be seen as a DB-term list, therefore DB-unification (see Subsection 2.4) needed to handle the substitution constraints in DB-clauses correctly during the proof search will be defined on DB-term lists. For an efficient representation of the substitution constraints in DB-clauses we will use DB-abstraction tree sets. In Prolog-like notation such a DB-clause $c = (t, \mathcal{A})$, whereby $\mathcal{A}$ is a DB-abstraction tree set, is written as

$$t :- VL(t) \in I(VL(t), \mathcal{A}).$$

where $VL(t)$ is as in Definition 2.1.

We would like to extend the applicability of the idea underlying DB-reduction to cases other than unit clauses. This will be achieved by the generalized definition below which we first illustrate with Example 2.2.

EXAMPLE 2.2. *The following formula in* Prolog*-like notation, consists of the facts F_1 and F_2, the rules R_1, R_2, R_3 and the query Q.*

$$\begin{array}{lllll} F_1 & p(f^n(a)). & & \\ F_2 & q(a,c). & & \\ R_1 & q(f(W),a) & :- & q(W,a). \\ R_2 & q(f(W),b) & :- & q(W,b). \\ R_3 & q(f(W),c) & :- & q(W,c). \\ Q_2 & & :- & p(X), q(X,Y). \end{array}$$

Since the given formula contains no unit clauses with similar literals it is impossible to use DB-reduction in the form defined so far here. However the formula contains three rules which suggest the application of a more general form of DB-reduction which may be obtained in exactly the same way as before. We apply build-at$(\{R_1, R_2, R_3\}, AT)$ which yields the abstraction tree shown in Figure 7 from which we obtain the DB-term list $s = (t, \{AT'\})$ as in Example 2.1. Its DB-abstraction tree AT' is identical to the tree of Figure 7, except that its term list is replaced by the list of variables labeling the root.

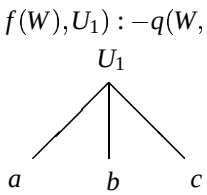

Figure 7. Abstraction tree AT of $R_1,\ldots,R_3$

Because we have $I(s)=\{(q(f(W),a):-q(W,a)),\ (q(f(W),b):-q(W,b)),\ (q(f(W),c):-q(W,c))\}$ *this DB-term list (or DB-clause) represents the rules* R_1,R_2,R_3. *The* Prolog*-like program obtained from the application of this general DB-reduction is the following one.*

$$\begin{array}{lllll} F_1 & p(f^n(a)). & & \\ F_2 & q(a,c). & & \\ R & q(f(W),U_1) & :- & q(W,U_1),\langle U_1\rangle \in I(\{\langle U_1\rangle\},\{AT'\}). \\ Q_2 & & :- & p(X),q(X,Y). \end{array}$$

After this illustration we present the formal definition of DB-reduction in the general case which subsumes Definition 2.5.

DEFINITION 2.6. *Let us call two clauses* similar *if they are of the form* $\{L_1,\ldots,L_n\}$, $\{L'_1,\ldots,L'_n\}$, $n\geq 1$, *and if for each* $i=1,\ldots,n$, L_i *has the same sign and the same predicate symbol as* L'_i.

Let $\mathcal{C}$ *denote a set of all similar clauses in a given set* M *of clauses. If* $\mathcal{C}$ *has more than one element then we say* M' *is obtained from* M *by* DB-reduction *if* $M'=(M\setminus C)\cup\{c\}$ *whereby* $c=(t,\Sigma)$ *is a DB-clause and consists of a clause part* t *and a list of substitution constraints* Σ, *with* t *and* Σ *defined as follows.*

Let AT be defined by build-at$(\mathcal{C},AT)$,[4] *then* t *is the term list of the root of AT and* $\Sigma=\mathcal{S}(AT')$ *where* AT' *is the DB-abstraction tree created from AT by replacing the term list of the root by the variable list of the root.*

LEMMA 2.2. *If* M *is a set of clauses and* M' *is created from* M *by the application of DB-reductions on clauses of arbitrary length, then* M' *represents the same clauses as* M *except for clauses which can be deleted by subsumption by a clause of equal length.*

Proof. This lemma follows directly from the definition 2.5 and 2.6 of DB-reductions, and the description of the operation *build-at*$(\ldots)$ in Subsection 2.2 in the same way as that for Lemma 2.1.

[4] Within this operation the clauses in $\mathcal{C}$ are interpreted as term lists.

The notion of DB-reduction may be even extended to clauses which are "partially similar" only. In such a case new literals may be added to the clauses until the clauses become similar. To compensate this change of the logical content unit clauses consisting of literals complementary to the added ones have to be added to M so that by unit resolution the old matrix could be restored. In other words, actually new connections are added to the matrix and these have to be chosen such that they are isolated (the new unit clauses do not resolve with literals originally in the matrix). As the example $\{\{P(a)\},\{P(b),\neg Q(x)\}\}$ demonstrates this condition of isolatedness is not always satisfiable without some proviso. It may be satisfied even in those cases, however, by extending the arity of literals such as the $Q(x)$ by 1 and inserting a new constant in all literals of the original formula and a different new constant in the added literals, so that our example would become $\{\{P(a),\neg Q(d,d)\},\{P(b),\neg Q(x,c)\},\{Q(d,d)\}\}$ which obviously is valid iff the original formula is. To this formula DB-reduction as defined in Definition 2.6 may now be applied. We refer to this kind of DB-reduction as *extended* DB-reduction.

In this most general form, DB-reduction also enables to eliminate common factors in different clauses which could alternatively be achieved by going to non-normal form matrices. Since DB-reduction keeps matrices in normal form, one can say that it provides a way to realize advantages, usually offered only by non-normal form matrices, in normal form ones. From a different perspective this extended form of DB-reduction makes formulas more regular in the sense of (Bibel and Eder, 1997) and thus easier to prove.

2.4. *DB-Unification*

The previous subsection has introduced the operation of DB-reduction. It results in a set of clauses which may contain literals (with substitution constraints) of a kind not familiar to usual theorem provers or Prolog interpreters. Therefore it is necessary to provide such provers with the capability to cope with these literals in a logically correct way. This requires a generalized form of unification, called DB-unification. As unification generates the most general unifier of two terms (or term lists), DB-unification generates a most general DB-unifier for DB-term lists, which is, roughly speaking a subset of all most general unifiers of the term lists represented by the DB-term lists ("subset" because unifiers which lead to an instance of another unifier are omitted). In detail the definition of a most general DB-unifier is as follows.

DEFINITION 2.7. *Let* $s_1 = (t_1, \Sigma_1)$ *and* $s_2 = (t_2, \Sigma_2)$ *be two DB-term lists and* $T = \{t_1'\rho \mid \rho = mgu(t_1', t_2'),\ t_1' \in I(s_1),\ t_2' \in I(s_2)\}$. *If* $T = \{\}$ *then a* most

general DB-unifier *of the DB-term lists does not exist. Otherwise the* most general DB-unifier *of the DB-term lists is* (σ,Σ)*, where* σ *is a substitution creating the common term structure of all term lists in* T *and* Σ *is the set of substitutions needed to create the parts where they differ, with* σ *and* Σ *defined as follows.*

Let AT be defined by build-at(T,AT)*; then* $\sigma = mgu(t_1,t)$*, where* t *is the term list of the root of the abstraction tree AT, and* $\Sigma = \mathcal{S}(AT')$ *where* AT' *is the DB-abstraction tree created from AT by replacing the term list of the root by the variable list of the root.*

If we use the data structure DB-abstraction tree sets to represent the set of substitutions in the DB-term lists $(l_1,\mathcal{A}_1)$ and $(l_2,\mathcal{A}_2)$, then algorithm 2.8 determines the most general DB-unifier of the two DB-term lists. The output of the algorithm is the empty set, if there exists no most general DB-unifier for the DB-term lists; otherwise the output is the most general DB-unifier $(\sigma,\mathcal{A}')$.[5] The algorithm is activated by calling *mgdbu*$(\mathcal{S},\mathcal{A})$, whereby $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$ and $\mathcal{S} = \{[s_1,t_1],\ldots,[s_n,t_n]\}$ for $l_1 = \langle s_1,\ldots,s_n\rangle$ and $l_2 = \langle t_1,\ldots,t_n\rangle$. Explanations for the functions mentioned in (7) and (8) are given thereafter.

ALGORITHM 2.8. *mgdbu*$(\mathcal{S},\mathcal{A})$

1. *If* $[f(s_1,\ldots,s_n),g(t_1,\ldots,t_n)] \in \mathcal{S}$*, whereby* $f \neq g$*, then* *mgdbu*$(\mathcal{S},\mathcal{A}) = \{\}$*;*
2. *if* $[s,t] \in \mathcal{S}$*, where* $s \neq t$ *and* s *is a variable occurring in* t*, then* *mgdbu*$(\mathcal{S},\mathcal{A}) = \{\}$*;*
3. *if* $[s,s] \in \mathcal{S}$*, then* *mgdbu*$(\mathcal{S},\mathcal{A})$:= *mgdbu*$(\mathcal{S}_1,\mathcal{A})$*, where* $\mathcal{S}_1 := \mathcal{S} \setminus \{[s,s]\}$*;*
4. *if* $[s,t] \in \mathcal{S}$*, where* $s = f(s_1,\ldots,s_n)$ *and* $t = f(t_1,\ldots,t_n)$*, then* *mgdbu*$(\mathcal{S},\mathcal{A})$:= *mgdbu*$(\mathcal{S}_1,\mathcal{A})$*, where* $\mathcal{S}_1 := \{[s_1,t_1],\ldots,[s_n,t_n]\} \cup \mathcal{S} \setminus \{[s,t]\}$*;*
5. *if* $[s,t] \in \mathcal{S}$*, where* t *is a (usual) variable and* s *is not a (usual) variable or* t *is a DB-variable and* s *is neither a (usual) variable nor a DB-variable, then* *mgdbu*$(\mathcal{S},\mathcal{A})$:= *mgdbu*$(\mathcal{S}_1,\mathcal{A})$*, where* $\mathcal{S}_1 = \{[t,s]\} \cup \mathcal{S} \setminus \{[s,t]\}$*;*
6. *if* $[s,t] \in \mathcal{S}$*, where* s *is a (usual) variable not occurring in* t*, then*

   6.1 *mgdbu*$(\mathcal{S},\mathcal{A}) := (\tau\circ\sigma,\mathcal{A}_1)$ *for* $\tau = \{s\backslash t\}$ *and mgdbu*$(\mathcal{S}\tau,\mathcal{A}) = (\sigma,\mathcal{A}_1)$*, otherwise*

   6.2 *mgdbu*$(\mathcal{S},\mathcal{A}) := \{\}$*;*

[5] A DB-abstraction tree set is used to represent the substitution constraints in Σ.

7. *if* $[s,t] \in \mathcal{S}$, *where* s *is a DB-variable not occurring in* t, *then*

   7.1 $mgdbu(\mathcal{S},\mathcal{A}) := (\tau\circ\sigma,\mathcal{A}_2)$ *for db-unifier* $(s,t,\mathcal{A}) := (\tau,\mathcal{A}_1)$ *and* $mgdbu(\mathcal{S}\tau,\mathcal{A}_1) = (\sigma,\mathcal{A}_2)$, *whenever such* σ,τ *exist; otherwise*
   7.2 $mgdbu(\mathcal{S},\mathcal{A}) := \{\}$;

8. *if* $[s,t] \in \mathcal{S}$, *where* s *and* t *are DB-variables, then*

   8.1 $mgdbu(\mathcal{S},\mathcal{A}) := (\tau\circ\sigma,\mathcal{A}_2)$ *for db_join* $(s,t,\mathcal{A}) = (\tau,\mathcal{A}_1)$ *and* $mgdbu(\mathcal{S}\tau,\mathcal{A}_1) = (\sigma,\mathcal{A}_2)$, *whenever such* σ,τ *exist; otherwise*
   8.2 $mgdbu(\mathcal{S},\mathcal{A}) = \{\}$;

9. $mgdbu(\{\},\mathcal{A}) = (\{\},\mathcal{A})$.

Algorithm 2.8 just presented invokes the algorithms *db-unifier*$(V,t,\mathcal{A})$ and *db-join*$(V_1,V_2,\mathcal{A})$ in the steps (7) and (8), resp., whose functionality is explained as follows. Algorithm *db-unifier*$(V,t,\mathcal{A})$ computes all those term lists T in the DB - abstraction tree $AT \in \mathcal{A}$ for the DB-variable V which can be unified with the term t at the position of V. If $T = \{\}$, then the result of *db-unifier*$(V,t,\mathcal{A})$ is the empty set; otherwise a tuple $(\tau,\mathcal{A}')$ is computed, whereby τ is the most general unifier of the term lists at the roots of the abstraction trees AT and AT', resulting from *build-at*(T,AT'), and $\mathcal{A}'$ is obtained as follows. If AT' has no nodes other than the root then $\mathcal{A}' = \mathcal{A} \setminus \{AT\}$, otherwise $\mathcal{A}' = \{AT''\} \cup \mathcal{A} \setminus \{AT\}$ where AT'' is the DB-abstraction tree created by replacing the term list of the root of AT' by its attached list of variables.

In the algorithm *db-join*$(V_1,V_2,\mathcal{A})$ we have to distinguish two cases. If the DB-variables V_1 and V_2 belong to the same DB-abstraction tree $AT \in \mathcal{A}$ we compute all term lists T in AT which can be unified at the positions of V_1 and V_2 (in the term lists of AT) and then proceed in the same way as in the algorithm of *db-unifier*$(\ldots)$. If the DB-variables V_1 and V_2 belong to different DB-abstraction trees $AT_1 \in \mathcal{A}$ and $AT_2 \in \mathcal{A}$ and Pos_1 is the position of V_1 in AT_1 and Pos_2 is the position of V_2 in AT_2 we use *get-join-of-term-lists* $(AT_1, Pos_1, AT_2, Pos_2, T)$ to compute the term lists of the join-operation. If $T = \{\}$, then the result of *db-join*$(V_1,V_2,\mathcal{A})$ is the empty set; otherwise a tuple $(\tau,\mathcal{A}')$ is computed, where τ is the most general unifier of the join of the root term lists of AT_1 and AT_2 at Pos_1 and Pos_2 on the one hand and the term list of the root of the abstraction tree AT' from *build-at*(T,AT') on the other, and $\mathcal{A}'$ is obtained as follows. If AT' has no nodes other than the root then $\mathcal{A}' = \mathcal{A} \setminus \{AT_1,AT_2\}$, otherwise $\mathcal{A}' = \{AT''\} \cup \mathcal{A} \setminus \{AT_1,AT_2\}$ where AT'' is the DB-abstraction tree created by replacing the term list of the root of AT' by its attached list of variables.

If a theorem prover together with DB-unification for the term lists of complementary literals is used on a formula simplified by DB-reductions the following theorem holds.[6]

THEOREM 2.3. *If F' is derived from a formula F by DB-reduction as defined in Definition 2.6 then every proof of F' by a theorem prover with DB-unification can be simulated by several instances of a theorem prover with standard unification applied to F in parallel, and every proof of F by a theorem prover with standard unification can be simulated by a theorem prover with DB-unification applied to F'.*

Proof. Any proof method works on connections in the given formula. So we can restrict ourselves to the analysis of what happens if a single connection is treated. Any connection between two clauses c and c' in F' selected by the DB-prover may correspond to more than one connections in F. This is because c (and c') may be the result of factoring the clauses $c_1,\ldots,c_m$ (and $c'_1,\ldots,c'_n$, resp.) by DB-reduction and between each pair c_i, c_j, $i = 1,\ldots,m$, $j = 1,\ldots,n$, there is such a corresponding connection. We thus need $n \cdot m$ standard provers who explore all of these different possibilities independently, some of which may find out that the connection is not unifiable. The proof of the theorem follows by iteration of this analysis. Conversely, to a connection in F there exists exactly one connection in F' to which it corresponds from which the proof follows immediately.

Note that the stepwise simulation in this theorem does not hold for extended DB-reductions (introduced at the end of Section 2.3) because by extended DB-reduction new literals are imported into the formula which possibly lead to differing proof structures.

2.5. *DB-Reductions and DB-Unification in Practice*

To test the usefulness of the compression and lazy-evaluation principles realized by DB-reduction and DB-unification, these were integrated into the proof system KoMeT. The tests were performed on the 2755 problems of the TPTP-problem library (Sutcliffe et al., 1994) using three different option settings for the prover with a maximum time limit of 300 seconds on a Sun SPARCstation 20. A small selected subset of the results is shown in Table I. All option settings included iterative deepening on the depth of the proof and identical ancestor pruning. The only difference was that in the first setting DB-reduction was not used (noted as "–" in the first column of Table I), in the second setting DB-reduction in its simplest form given in Definition 2.5

[6] For a more detailed proof we refer to (Rath, 1992).

with DB-unification was used (noted "db" in the second column) and in the third the more general DB-reduction of Definition 2.6 with DB-unification was used (noted "db*"). For each example and option run-time and number of clauses in the corresponding DB-matrix are given.

Table I. Test results of KoMeT without and with DB-reduction

problem	–	db	db*
BOO003-1.p	>300 (37)	>300 (29)	**29.6** (16)
BOO005-1.p	118.5 (37)	**22.5** (29)	>300 (16)
BOO012-3.p	>300 (49)	**94.6** (35)	>300 (20)
CAT002-4.p	>300 (27)	>300 (22)	**21.0** (12)
COL052-2.p	**150.3** (16)	>300 (14)	>300 (8)
COM004-1.p	5.2 (25)	**2.4** (21)	4.7 (11)
FLD002-3.p	>300 (29)	**75.2** (26)	>300 (22)
FLD016-3.p	>300 (32)	**64.4** (26)	269.9 (22)
FLD069-1.p	>300 (32)	>300 (28)	**142.6** (19)
GEO003-3.p	>300 (81)	**19.5** (68)	30.9 (23)
GEO026-3.p	>300 (79)	>300 (65)	**58.9** (20)
GEO059-3.p	>300 (79)	264.2 (69)	**66.9** (24)
GRP013-1.p	>300 (22)	**205.5** (15)	>300 (9)
GRP125-2.003.p	>300 (21)	**151.4** (11)	293.1 (9)
GRP131-2.002.p	265.2 (13)	>300 (11)	**120.5** (8)
KRS015-1.p	27.9 (26)	27.9 (26)	**25.5** (24)
LCL076-3.p	**35.9** (6)	41.3 (4)	>300 (3)
LCL143-1.p	>300 (22)	>300 (16)	**59.9** (8)
LCL182-1.p	**42.4** (9)	71.8 (5)	>300 (4)
MSC007-2.002.p	**166.5** (16)	>300 (15)	>300 (13)
NUM003-1.p	**5.6** (13)	6.6 (7)	55.5 (3)
PLA010-1.p	>300 (31)	156.0 (14)	**38.3** (9)
PLA015-1.p	>300 (31)	>300 (14)	**48.9** (9)
PRV006-1.p	91.6 (26)	52.7 (18)	**45.2** (11)
PUZ025-1.p	88.1 (24)	72.4 (21)	**21.9** (13)
SET118-7.p	>300 (237)	**286.1** (196)	>300 (66)
SET563-6.p	>300 (190)	>300 (156)	**140.2** (61)
SYN178-1.p	>300 (369)	**94.1** (339)	124.6 (313)
SYN298-1.p	>300 (369)	**198.5** (339)	207.8 (313)
TOP001-2.p	6.1 (13)	6.1 (13)	**5.4** (12)

Since Table I can present only a small fragment of the results of these tests, we note that overall it was possible to prove (out of the 3060 problems) 734 problems without DB-reductions, 778 problems with the simplest form of DB-reduction of Definition 2.5, 774 problems with the general DB-reduction of Definition 2.6, and 824 problems altogether (with any of the three modes).

These data demonstrate that the simplest form of DB-reduction gives the best results for the PTTP collection. It is relatively simple and yet has an enormous effect in a substantial number of cases. In particular there are 90 examples in PTTP among those provable by KoMeT which could not be proved without the technique described in the present section. On the other hand a look into the details of the table reveals that each mode has its advantages. For instance, BOO012-3.p is proved by none except the simplest form of DB-reduction, BOO003-1.p by none except general DB-reduction, and COL052-2.p by none except the DB-reduction-free version.

The lessons to be drawn from these experiences is that in cases where the respective reduction technique actually simplifies the problem the prover is clearly enhanced in its performance. But there are other examples where the efforts required for eg. the time-consuming join operation on DB-abstraction trees slows down the prover to an extent that it fails to provide a proof at all within the given time limit. Especially DB-reduction on general clauses gives rise to expensive join operations on big DB-abstraction trees. In such cases it is very useful to replace (if possible) one big DB-abstraction tree in every DB-clause by a lot of small DB-abstraction trees. An implementation of this splitting of DB-abstraction trees is in progress.

This splitting of DB-abstraction trees might also have a positive performance effect if used carefully for the creation of DB-abstraction trees within DB-unification. We optimized the DB-unification in another way by delaying the join operations until the end of the unification process on two terms and then calculating all join operations to be effected on two DB-abstraction trees in a single step. Another optimization of the operations *get-unified-term-lists* $(\ldots)$ and *get-join-of-term-lists*$(\ldots)$ is already implemented as well. The description given so far suggests to first determine all term lists which are the results of these operations and then build a new DB-abstraction tree from scratch. This is too time consuming. Rather, in our implementation, we take advantage of the information in the old DB-abstraction trees during the computation of *get-unified-term-lists*$(\ldots)$ and *get-join-of-term-lists*$(\ldots)$ to create the new DB-abstraction trees.

In comparing the performance of KoMeT in absolute terms as demonstrated by these examples an important caveat has to be taken into account. First, by activating KoMeT's TPTP-technology-based compiler along with a similar setting as in the first column of Table I KoMeT is able to prove a total of 904 examples out of the PTTP list. As we mentioned earlier though DB-reduction has been realized in a prototypical implementation in Prolog only which necessarily is much slower (and thus less successful in total) than a high performance theorem prover like SETHEO (Moser et al., 1997). Nevertheless the prototypical experiments demonstrate clearly that DB-reduction is

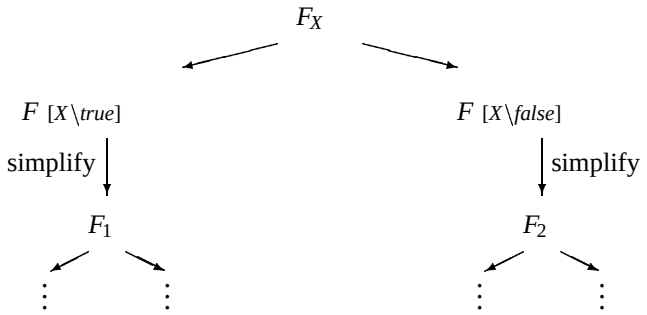

Figure 8. The splitting step of the Davis-Putnam procedure

a necessary feature for *any* successful prover.

3. A NON-CLAUSAL DAVIS-PUTNAM PROCEDURE

3.1. *Motivation*

Decision procedures for classical propositional logic, i.e. procedures determining whether or not a given propositional formula is valid, play an important role in intellectics, computer science and, of course, mathematical logic. Automated theorem proving, solving hard combinatorial optimization problems, and verifying circuits in hardware design are some of the applications.

The Davis-Putnam procedure (Davis and Putnam, 1960; Davis et al., 1962) is one of the best known and most successful decision procedures for classical propositional logic. The essential idea of this procedure consists of the following step: assign the truth values *true* and *false* to a selected propositional variable X of the investigated formula F and simplify the resulting formulas yielding the formulas F_1 and F_2, respectively (see Figure 3.1). The original formula F is valid if and only if both resulting formulas F_1 and F_2 are valid. Applying this step recursively to the resulting formulas F_1 and F_2 yields a search tree whose leaves are marked with *true* or *false*. If *all* leaves are marked with *true* the formula F is valid, otherwise F is not valid.

There are many implementations of the Davis-Putnam procedure, e.g. C-SAT, LDPP (Uribe and Stickel, 1994), POSIT (Freeman, 1995), SATO (Zhang, 1993), SATX (Li, 1996), TABLEAU (Crawford and Auton, 1993). All these implementations have in common, that they require the formulas to be proven be given in clausal form. Since the usual transformation into clausal form is based on the application of distributivity laws, this leads to an exponential increase of the resulting clausal form in the worst-case. Using a definitional transformation (Eder, 1992; Plaisted and Greenbaum, 1986) yields (at most)

a linear increase of the resulting formula's size at the expense of introducing new (propositional) variables.

The aim of this section is *not* to present yet another implementation of the Davis-Putnam procedure, but to introduce a *non-clausal* version of this method. By generalizing this procedure to arbitrary formulas we avoid any transformation steps. Thus we avoid any increase of the size of the formula and shorten the search tree considerably. Furthermore, the application of an additional split rule becomes possible which further reduces the search space.

During the proof search formulas are represented as matrices. A matrix is not only a compact representation of a formula but also one of the corresponding search space. In the following we will show how the matrix representation of arbitrary (non-clausal) formula may serve as a basis for a generalized non-clausal form decision procedure which results in a substantial *compression* of the search space.

3.2. *A Non-Clausal Davis-Putnam Procedure*

For completeness we begin by defining the syntax of (arbitrary) formulas and their matrices in propositional logic.

DEFINITION 3.1. Formulas *are defined inductively as follows. Any propositional variable A is a formula. If F and G are formulas then* $(\neg F)$, $(F \wedge G)$, $(F \vee G)$ *and* $(F \rightarrow G)$ *are also formulas, with the logical connectives* $\neg$ *(negation),* $\wedge$ *(conjunction),* $\vee$ *(disjunction) and* $\rightarrow$ *(implication).*

A literal *is either a variable or its negation. The* negation $\overline{L}$ *of a literal L is defined as* $\overline{L} = A$*, if* $L = \neg A$ *(for some variable A), otherwise* $\overline{L} = \neg L$.

A signed formula *is a pair* (F, p) *consisting of a formula F and a* polarity *p, where* $p \in \{0, 1\}$.

The matrix *of a signed formula* (F, p) *is inductively defined by way of Table II.* $\{M_G\}$, $\{M_H\}$ *and* $\{M_G, M_H\}$ *therein are called* clauses. *The* matrix *of a formula F is the matrix of the signed formula* $(F, 0)$. *A matrix is* valid *iff the corresponding formula is valid.*

REMARK 3.1. *Matrices of the form* $M = \{\ldots, \{\{C_1, \ldots, C_n\}\}, \ldots\}$ *can be simplified to* $M' = \{\ldots, C_1, \ldots, C_n, \ldots\}$ *where* $C_1, \ldots, C_n$ *are clauses. Clauses of the form* $C = \{\ldots, \{\{M_1, \ldots, M_m\}\}, \ldots\}$ *can be simplified to* $C' = \{\ldots, M_1, \ldots, M_m, \ldots\}$ *where* $M_1, \ldots, M_m$ *are matrices.*

In the (usual) clause-based Davis-Putnam procedure a matrix consists of a set of clauses, where each clause is a set of literals. In our non-clausal approach a clause is a set of matrices, where each matrix is either a literal or a set of clauses. As in the normal form case we can visualize a matrix M as a

Table II. Definition of the matrix of a signed formula

(F,p)	matrix of (F,p)	where M_G / M_H is the matrix of
$(A,0)$, A a variable	$\{\{A\}\}$	–/–
$(A,1)$, A a variable	$\{\{\neg A\}\}$	–/–
$((\neg G),p)$	M_G	$(G,1-p)$ / –
$((G\wedge H),1)$	$\{\{M_G\},\{M_H\}\}$	$(G,1)$ / $(H,1)$
$((G\vee H),0)$	$\{\{M_G\},\{M_H\}\}$	$(G,0)$ / $(H,0)$
$((G\to H),0)$	$\{\{M_G\},\{M_H\}\}$	$(G,1)$ / $(H,0)$
$((G\wedge H),0)$	$\{\{M_G,M_H\}\}$	$(G,0)$ / $(H,0)$
$((G\vee H),1)$	$\{\{M_G,M_H\}\}$	$(G,1)$ / $(H,1)$
$((G\to H),1)$	$\{\{M_G,M_H\}\}$	$(G,0)$ / $(H,1)$

$$\left[\left[\left[\left[\begin{array}{c}A\\B\end{array}\right]\quad\left[\ \neg C\ \right]\quad\left[\begin{array}{c}C\\\left[\ \left[\ \neg A\ \right]\quad\left[\ \neg B\ \right]\ \right]\end{array}\right]\right]\right]\right]$$
$$\left[\left[\left[\ \left[\ D\ \right]\quad\left[\ \neg D\ \right]\ \right]\right]\right]$$

Figure 9. The matrix of $((\neg A\vee\neg B)\wedge C\to\neg(C\to A\wedge B))\wedge(D\vee\neg D)$

two-dimensional matrix by placing its clauses horizontally and the matrices of each clause vertically.

EXAMPLE 3.1. *Consider the formula* $F=((\neg A\vee\neg B)\wedge C\to\neg(C\to A\wedge B))\wedge(D\vee\neg D)$. *The (simplified) matrix of* F *is* $\{\{\{\{A,B\},\{\neg C\},\{C,\{\{\neg A\},\{\neg B\}\}\}\},\{\{D,\neg D\}\}\}\}$. *The two-dimensional visualization of this matrix is given in Figure 9.*

Within the *negation normal form*, a clause is interpreted as the *conjunction* of its matrices and a non-variable matrix is interpreted as the *disjunction* of its clauses. Therefore a clause is *true* iff *all* its elements are *true*. A matrix is *true* iff *at least one* of its clauses is *true*. A clause/matrix which is not *true* is *false*.

OBSERVATION 3.1. *The empty matrix* $\{\}$ *is* false, *the empty clause* $\{\}$ *is* true. *A matrix* $\{\ldots,\{\},\ldots\}$ *containing the empty clause is* true. *A clause* $\{\ldots,\{\},\ldots\}$ *containing the empty matrix is* false.

The non-clausal Davis-Putnam procedure is defined in Figure 10. L denotes a literal and lit_M is the set containing all literals of M. $\#\mathrm{diff}(M',M'')$

input: matrix M representing an arbitrary formula F
output: *true*, if F is valid; *false* otherwise

begin NCDP(M)
  if $M = \{\}$ then return *false*;
  if $\{\} \in M$ then return *true*;
  if $\{L\} \in M$ then return NCDP(MREDUCE$_{\bar{L}}(M)$); /* UNIT */
  for all $L \in \text{lit}_M$ with $\bar{L} \notin \text{lit}_M$ do $M :=$MREDUCE$_{\bar{L}}(M)$; /* PURE */
  if $M = \{C_1, \ldots, C_{j-1}, \{M_1, M_2\}, C_{j+1}, \ldots, C_n\}$ and
    #diff$(M', M'') > 2$ for some j
  then if NCDP$(\{C_1, \ldots, C_{j-1}, \{M_1\}, C_{j+1}, \ldots, C_n\}) =$ *true* and /* beta- */
    NCDP$(\{C_1, \ldots, C_{j-1}, \{M_2\}, C_{j+1}, \ldots, C_n\}) =$ *true* /* splitting */
    then return *true* else return *false*;
  select $L \in \text{lit}_M$;
  if NCDP(MREDUCE$_L(M)$)=*true* and /* splitting */
    NCDP(MREDUCE$_{\bar{L}}(M)$)=*true*
    then return *true* else return *false*;
end NCDP.

Figure 10. The non-clausal Davis-Putnam procedure

is the number of different variables which occur only in M' or only in M''. Calling NCDP(M) returns *true*, if the matrix M is valid, otherwise it returns *false*. Some explanations of the algorithm follow.

If the matrix M is an empty set (or contains an empty clause), *false* (or *true*, respectively) are returned. Otherwise a literal L occuring in M is selected and *true*, respectively *false*, is assigned to it (*splitting*). Only if both resulting matrices are valid, the matrix M is valid.

Assigning *true* to all occurrences of L (and *false* to $\bar{L}$) in the matrix M and returning the resulting simplified matrix is performed by the procedures MREDUCE$_L(M)$ and CREDUCE$_L(C)$ (see Figure 11). MREDUCE$_L(M)$ performs this assignment for the matrix M. If the matrix is the literal L (or $\bar{L}$), the truth values *true* (i.e. $\{\{\}\}$) (or *false*, i.e. $\{\}$, resp.) are returned. Otherwise the assignments of its clauses are evaluated. If there is a *true* clause (i.e. $\{\}$), the whole matrix is deleted and *true* (i.e. $\{\{\}\}$) is returned. We call this step *matrix elimination*. It is illustrated in the first part of Figure 12. Notice that the *literal deletion* step in the clause-based approach is a special case of the matrix elimination step (where the matrix M is a literal).

Consider, for example, the matrix in Figure 13 (where P, Q, R are literals and M is a matrix) along with its partial clausal form. In the clause-based procedure the assignment of *true* to the literal R will *only* delete the literal R.

```
begin MREDUCE_L(M)
    if M=L then return {{}};                          /* assign true */
    if M=L̄ then return {};                            /* assign false */
    if M is a literal then return M;
    M_1 := {C' | C' = CREDUCE_L(C) and C ∈ M};
    if {} ∈ M_1 then return {{}};                     /* matrix elimination */
    return {C | C ∈ M_1 and C ≠ {{}} };                /* simplify */
end MREDUCE.

begin CREDUCE_L(C)
    C_1 := {M' | M' = MREDUCE_L(M) and M ∈ C};
    if {} ∈ C_1 then return {{}};                     /* clause elimination */
    return {M | M ∈ C_1 and M ≠ {{}} };                /* simplify */
end CREDUCE.
```

Figure 11. The procedures MREDUCE and CREDUCE

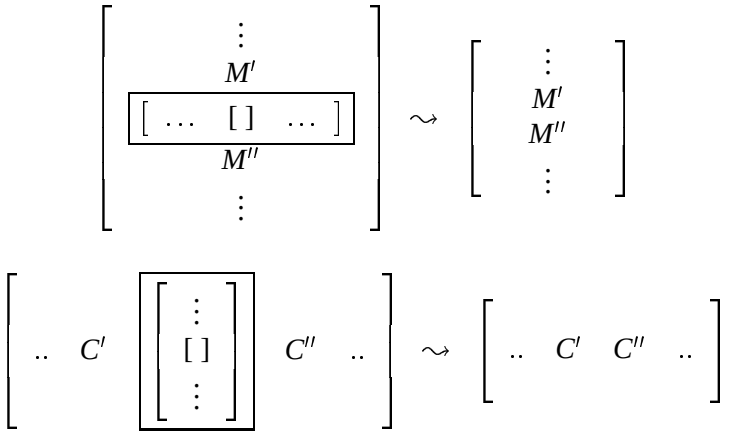

Figure 12. Matrix elimination and clause elimination

Figure 13. An example of a matrix for matrix elimination

In the non-clausal procedure the entire matrix [[P] [Q] [R]] will be deleted resulting in M as the remaining problem. This means that additional proof steps have to be performed in the clause-based procedure which are not necessary in the non-clausal procedure. On the other hand each literal deletion step in the clause-based procedure can be simulated by a corresponding matrix elimination step in the non-clausal approach.

CREDUCE$_L(C)$ performs the assignment of *true* to the literal L for a clause C, i.e. the assignments of its matrices are evaluated. If there is a *false* matrix (i.e. $\{\}$), the whole clause is deleted and *false* (i.e. $\{\{\}\}$) is returned. Like in the clause-based procedure we call this step *clause elimination* which is illustrated in the second part of Figure 12.

Before splitting the matrix, NCDP performs the unit clause reduction rule (UNIT) and the pure literal reduction rule (PURE) as in the standard Davis-Putnam procedure. It employs again the procedures MREDUCE and CREDUCE for that purpose too. In addition a second form of splitting is employed which is called *beta-splitting rule* and is illustrated in Figure 14. It is justified by the following lemma.

LEMMA 3.1. *A matrix* $M = \{C_1,\ldots,C_{j-1},\{M_1,M_2\},C_{j+1},\ldots,C_n\}$ *is valid iff* $M' = \{C_1,\ldots,C_{j-1},\{M_1\},C_{j+1},\ldots,C_n\}$ *and* $M'' = \{C_1,\ldots,C_{j-1},\{M_2\}, C_{j+1},\ldots,C_n\}$ *are valid.*

Proof. According to the main theorem of the connection method (Bibel, 1987a) a matrix is valid iff each path through it is complementary (i.e. contains a connection). The lemma does nothing but partitioning the set of paths through the entire matrix in 2 disjoint subsets.

Whereas matrix elimination and clause elimination are the basic steps of the non-clausal proof procedure and necessary to guarantee both correctness and completeness, the beta-splitting rule is additionally included only to reduce the search space even more. The search space (i.e. the complexity) of our basic procedure is essentially determined by the number of different variables in the matrix M. Since every elimination of a variable splits the matrix into two smaller ones, the worst-case complexity is $t(M) = 2^{\#(M)}$ where $\#(M)$ is the number of different variables in M. The aim of the beta-splitting rule is to reduce this number of different variables by splitting M into two matrices M' and M'' as shown in Figure 14.

Recall that $\#\mathrm{diff}(M',M'')$ is the number of different variables which occur only in M' or only in M''. Then $t(M) = 2^{\#(M)} = 2^{\frac{1}{2}(\#(M')+\#(M'')+\#\mathrm{diff}(M',M''))}$. Suppose $\#(M') = \#(M'') = m$, then we obtain $t(M) = 2^{m+\frac{1}{2}\#\mathrm{diff}(M',M'')}$. The complexity to prove both resulting matrices is $t(M')+t(M'') = 2^{\#(M')}+2^{\#(M'')} = 2^{m+1}$. This means, if their are more than 2 different variables occurring

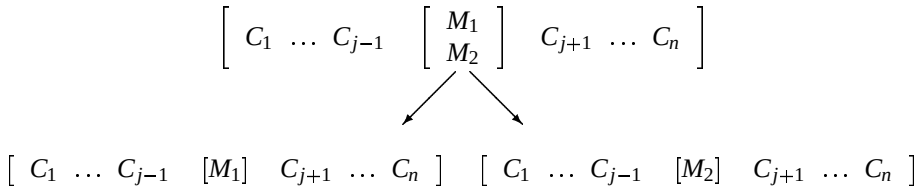

Figure 14. The beta-splitting rule

only in M' or only in M'' (i.e. $\#\mathrm{diff}(M', M'') > 2$), the application of the beta-splitting rule will reduce the complexity of our problem. A recursive application of the beta-splitting rule can even shorten exponential proofs to linear ones (see example in the next subsection).

EXAMPLE 3.2. *Consider the formula* $F = ((\neg A \vee \neg B) \wedge C \rightarrow \neg(C \rightarrow A \wedge B)) \wedge (D \vee \neg D)$. *A* non-*clausal proof of this formula is given in Figure 15. We mention that the* clausal *proof of this example consists of 76 literals whereas the* non-clausal *proof only needs 26 literals (as the reader may verify).*

3.3. *Some Experimental Results*

In the following we will compare the performance of our non-clausal proof procedure to a clause-based Davis-Putnam prover. For this purpose we use a Prolog implementation of our non-clausal proof procedure and compare its performance with that of the Davis-Putnam prover of the KoMeT system (Bibel et al., 1994a).[7] As mentioned in the introduction KoMeT is also implemented in Prolog so that we get a fair comparison. Moreover it is one of the few theorem provers providing not only the standard transformation into clausal form, but also various definitional transformations (Eder, 1992; Plaisted and Greenbaum, 1986).

In Table III the first column contains the name of the problem, the next two columns contain the times used by the Davis-Putnam prover of KoMeT with the standard transformation ("DPstandard") as well as the definitional transformation ("DPdefini"), and the last column contains the time used by our prover ncDP. Times are measured on a Sun SPARCstation 10 with ECLiPSe Prolog and are given in seconds, where "$>$600" means that no proof was found within 600 seconds.

[7] The Prolog code of the non-clausal implementation (which is less than 3 kBytes) can be found in (Otten, 1997) or on the WWW http://www.intellektik.informatik.tu-darmstadt.de /∼ jeotten/ncDP.

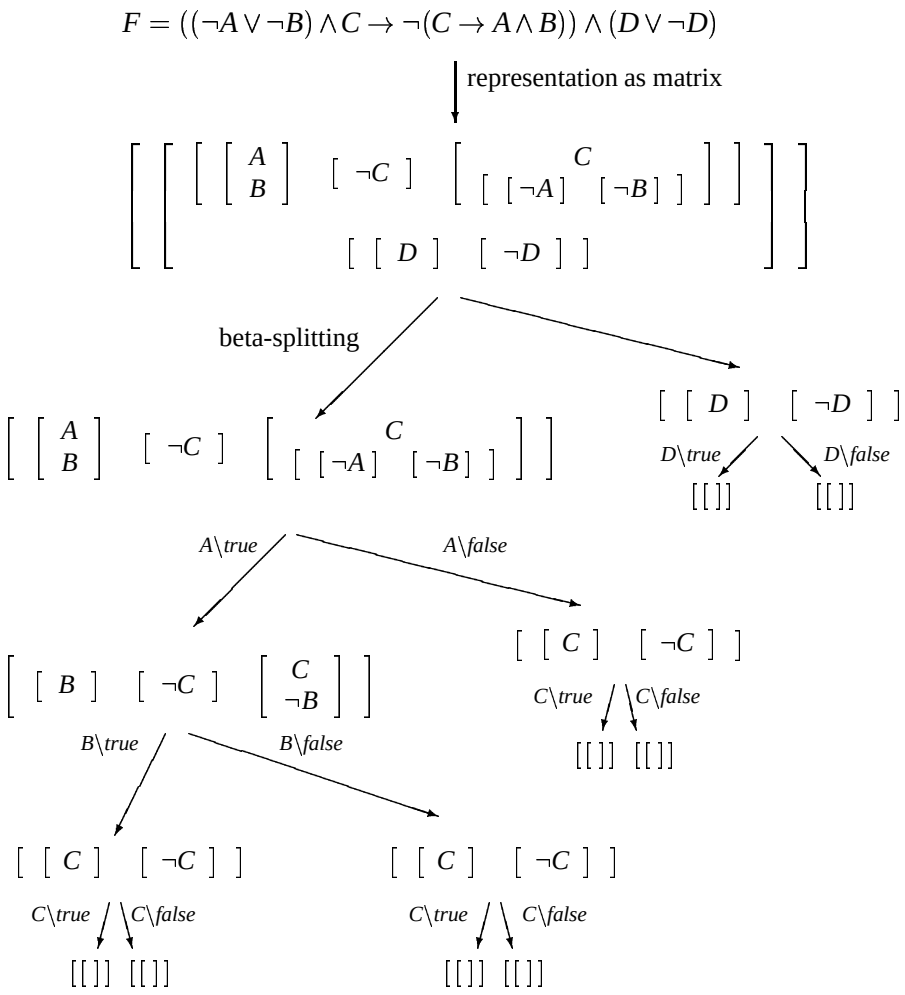

Figure 15. Non-clausal proof of the formula $((\neg A \vee \neg B) \wedge C \rightarrow \neg (C \rightarrow A \wedge B)) \wedge (D \vee \neg D)$

We start with formulas which are in clausal form, namely the "complete" formulas com$n$ (containing n distinct variables) and the "pigeonhole" formulas pigeon$n$ ($n+1$ pigeons into n holes). As we see ncDP is implemented roughly as well as, actually slightly better than, DPstandard. DPstandard applies PURE reduction after each splitting step while ncDP dispenses with application of PURE (in a sense subsumed by beta-splitting) except for a sin-

Table III. Test results

name	DPstandard	DPdefi ni	ncDP
com8	3.30	59.45	1.38
com9	7.81	244.67	4.05
com10	19.10	>600	11.73
pigeon4	0.75	1.68	0.53
pigeon5	5.80	5.58	4.00
pigeon6	80.10	26.28	33.21
ft6	31.69	0.72	0.43
ft8	>600	2.72	2.46
ft10	>600	13.60	13.63
samp10	64.36	2.34	<0.01
samp12	451.89	9.85	0.01
samp14	>600	46.52	0.02
ipell1(f)	0.50	0.42	0.01
ipell4(f)	0.60	0.40	0.01
ipell10(t)	0.28	0.53	0.05
ipell12(f)	>600	81.18	2.29
ipell14(f)	1.88	0.60	<0.01
ipell17(f)	10.92	1.51	<0.01
ipell71a(t)	0.21	0.18	<0.01
ipell71b(f)	5.90	0.72	0.05
ipell72a(t)	0.28	0.77	0.11
ipell72b(t)	0.83	2.05	0.56
dan1(f)	>600	>600	0.86
dan2(t)	>600	>600	0.76
dan3(f)	>600	80.18	2.31

gle one to the initial matrix. This may lead to some overhead in DPstandard and thus explains why even in the case of normal form matrices ncDP may have an advantage. Conversely, ncDP can simulate any DPstandard proof for normal form problems without any overhead in general. The generally poor performance of DPdefini is explained by the overhead of introducing definitional variables and their definitions which mostly becomes useless for formulas in clausal form. Exceptions, like pigeon6, are explained by the fact that DPdefini replaces equal subformulae by a single variable which may lead to shorter proofs.

The formulas ft$n$ in Table III are of the form $\neg\neg(p_1 \leftrightarrow (p_2 \leftrightarrow \ldots (p_{n-1} \leftrightarrow p_n)\ldots) \leftrightarrow (\ldots(p_1 \leftrightarrow p_2) \leftrightarrow p_3)\ldots \leftrightarrow p_n)$, the formulas samp$n$ are of the form $((p_1 \rightarrow p_1)\wedge(p_2 \rightarrow p_2)\wedge\ldots\wedge(p_n \rightarrow p_n))$. As we see both, DPdefini

and ncDP, outperform DPstandard in these examples in a significant way, providing proofs also in cases where DPstandard fails. The interesting fact, however, is that even DPdefini is outperformed by orders of magnitude.

In (Korn and Kreitz, 1997) the intuitionistic validity of a propositional formula F is decided by translating it into a formula F' which is classically valid iff F is intuitionistically valid. The resulting formulas are strongly in non-clausal form. We have applied this translation to the (propositional) formulas in (Pelletier, 1986) to decide, whether they are intuitionistically valid.[8] The resulting formulas are ipell$n$ where n is the problem's number according to (Pelletier, 1986). The formulas dan1, dan2 and dan3 are the corresponding transformations of the formulas $((a \to b) \to c) \wedge ((d \to e) \to b) \wedge ((g \to h) \to e) \to c$, $((a \to b) \to c) \wedge ((d \to e) \to b) \wedge ((g \to a) \to e) \to c$ and $((p \leftrightarrow q) \leftrightarrow r) \leftrightarrow (p \leftrightarrow (q \leftrightarrow r))$, respectively. The additional character after the name of each formula indicates if it is valid ("t") or not ("f"). Table III again demonstrates the excellent performance of ncDP in comparison with both competitors. Altogether these experimental results show that the transformation to clausal form may yield formulas which are almost impossible to prove, especially if the standard transformation to clausal form is used, while ncDP is still able to prove them easily.

In conclusion we summarize the three main advantages of the non-clausal proof procedure as follows.

1. *Avoidance of the transformation into any clausal form*. This transformation is sometimes not feasible or the resulting formula is too large for the prover to find a proof. Even if we use the definitional transformation additional variables are introduced which still increase the complexity of the problem in a noticeable way.
2. *Application of matrix elimination steps*. This leads to formulas which are smaller in comparison with those obtained by the corresponding literal deletion performed on the corresponding formulas in clausal form (as demonstrated by the example further above).
3. *Application of the beta-splitting rule*. Especially in the case of formulas which represent independent problems, this is an essential technique (see samp$n$ examples above). *Beta-splitting* can shorten proofs from exponential (in the length of the input formula) to linear ones.

[8] In Table III only the more diffi cult formulas are presented. For all other formulas the proof took less than 0.5 seconds for each prover.

4. Equivalences, Subsumption, Default Reasoning

In the present section three different topics are surveyed in an illustrative way only since the technical details are already published elsewhere (as specified within the text). The first two are further instances of the general leitmotiv of compression of this chapter while the last one illustrates how classical proof techniques can be exported to extended logics.

4.1. *Equivalences*

An inference mechanism working on a specific problem representation usually has no access to information which is only implicitly included in the representation of a formula. For instance, proof systems which apply a transformation of problems into clausal form, cannot exploit the information about equivalences which may be included implicitly in the resulting clause set. An interesting aspect of equivalences, however, is their use for simplification based on the fact that equivalence of literals is closely related to equality of terms. Given a literal-equivalence $A \leftrightarrow B$, a literal B might be replaced (or demodulated) by A if $A < B$ holds for some Noetherian ordering $<$ (cf. (Bachmair and Ganzinger, 1992)). Hence, such a use of equivalences would allow to transfer reduction techniques developed for equality reasoning to problems not noted in terms of equality.

In this vein, a calculus with logical equivalence was proposed in (Socher-Ambrosius, 1989; Socher-Ambrosius, 1990). It combines resolution with the possibility to derive literal-equivalences and to use them as rewrite rules. In particular, it was shown that the reduction part of a corresponding calculus can be considerably improved by literal demodulation. In the case of top-down backward-chaining connection calculi (instead of saturation calculi based on resolution), the situation is more complicated since a straightforward application of demodulation techniques would result in an incomplete calculus. However, as we have shown in (Brüning, 1995) a careful exploitation of equivalences is also feasible in such calculi and can result in enormous reductions of the search space.[9]

The basic idea of (Brüning, 1995) is to treat equivalences and the remaining clauses of a problem specification separately. That is, given a clause set $\mathcal{S} \cup Cl(\mathcal{E})$ (where $\mathcal{E}$ denotes a set of literal-equivalences and $Cl(\mathcal{E})$ denotes the clausal representation of $\mathcal{E}$), only clauses from $\mathcal{S}$ are used for the construction of connection tableaux, whereas the elements of $\mathcal{E}$ are used for sim-

[9] In this and the following subsection we use *connection tableaux* instead of matrices as basic proof objects which according to Chapter 2 amounts to a negligible representational difference.

plifying the elements from $\mathcal{S}$. Clearly this simple separation is not sufficient to obtain a complete calculus. To retain completeness one has to introduce two new derivation steps. The first one is the so-called *equivalence step* which allows to replace an open goal L by a literal K in case L is equivalent to K (wrt $\mathcal{E}$).[10] The second one is the so-called *L-paramodulation step* which allows to test (via ordered literal-demodulation) whether a set of literal equivalences is unsatisfiable.

One might object that the introduction of these new inference steps makes a refined handling of equivalences useless. Fortunately this is not the case. On the one hand, the simplification of the input clauses by demodulation can reduce the search space considerably. On the other hand, equivalences can be used to strengthen the important regularity restriction: Instead of requiring that no branch contains two identical literals, one can demand that, given a set of equivalences $\mathcal{E}$, no branch contains two literals which are *equivalent* wrt $\mathcal{E}$. A connection tableau satisfying this condition is called *E-regular wrt* $\mathcal{E}$, or $\mathcal{E}$-regular for short.

The following example (taken from (Brüning, 1995)) illustrates the usefulness of our approach.

EXAMPLE 4.1. *Consider the clause set* $\mathcal{S}$*:*

(1)	$\{q(f^n(a))\}$	(4)	$\{p(Y), \neg p(f(Y))\}$
(2)	$\{q(V), \neg p(V)\}$	(5)	$\{p(f(Z)), \neg p(Z)\}$
(3)	$\{p(U), \neg q(f(U))\}$	(6)	$\{\neg p(a)\}$

Clauses (4) and (5) encode the literal equivalence $E = p(f(Y)) \leftrightarrow p(Y)$*. Rewriting* $\mathcal{S}$ *with* E*, clauses (4) and (5) become tautological and are removed from* $\mathcal{S}$*. Now, consider a derivation with top-clause* $\{\neg p(a)\}$*. It is easy to verify that any derivation applying extension steps using clauses (2) and (3) to build up the term* $f^n(a)$ *is pruned (because the resulting connection tableau would not be E-regular). Therefore, the only possibility to derive the open goal* $\neg p(f^{n-1}(a))$ *(for which a subproof consists of two extension steps with clauses (1) and (3)) is to apply an equivalence step. Considering the original clause set, this means that only clauses (4) and (5) are used for this purpose. This reduces a search space of exponential size (in n) to one of linear size.*[11]

An important aspect which has not been mentioned yet is the *generation* of equivalences. In case literal-equivalences are not available from the beginning, the ability to generate equivalences in the course of a deduction is an

[10] As shown in (Brüning, 1995), the use of equivalence steps corresponds to a restricted use of extension steps with clauses from $Cl(\mathcal{E})$.

[11] Note that in case only the "ordinary" regularity restriction is employed, the size of the pruned search space remains exponential.

important requirement. In the case of top-down backward-chaining calculi, it is shown in (Brüning, 1995) that this can be achieved by analyzing connection tableaux. Basically, the idea is to make use of the fact that a set of implications $\{L_1 \to L_2, \ldots, L_{n-1} \to L_n, L_n \to L_1\}$ on the one hand expresses that the literals L_i are pairwise equivalent but, on the other hand, if used successively in a derivation, result in a non-regular connection tableau (because in order to solve L_1, the same goal is generated as subgoal). Hence, new equivalences can be extracted from connection tableaux which are *not* E-regular.

As an example, consider the clause set $\{\{p(X), p(f(X))\}, \{\neg p(X), \neg p(f^n(X))\}\}$. It can be shown that this set is unsatisfiable if n is even. However, the proof is not trivial at all — at least for top-down backward-chaining calculi. However, after few inference steps, it is possible to generate the equivalence $p(X) \leftrightarrow p(f(X))$ from a non-regular connection tableau (with depth 5). Using this equivalence, the above clause set can be rewritten to the set $\{\{p(X)\}, \{\neg p(X)\}\}$, which is obviously unsatisfiable.

The mechanisms sketched in this section have been implemented in a prototypical proof system using Prolog as programming language. Some promising results were achieved. For example, the clause set used in the previous paragraph can be solved for $n = 20$ in 2 seconds on a Sparc Station 1, which is quite impressive for a top-down backward-chaining calculus. For more examples illustrating the usefulness of our approach we refer to (Brüning, 1995).

There are two directions to generalize the achieved results. The first one is to consider arbitrary equivalences rather than literal-equivalences. Such an approach was presented in (Lee and Plaisted, 1990) with CLIN (Lee and Plaisted, 1992) as the basic calculus (which is not a backward-chaining calculus). There, the use of definitions as rewrite rules was proposed and a remarkable improvement of CLIN was achieved. However, completeness issues and the possibility to derive new equivalences were not considered. The second direction for generalization is the use of conditional equivalences. This would be similar to the use of conditional rewrite rules in the terminology of rewriting.

4.2. *Subsumption*

Subsumption deletion (eg. see (Chang and Lee, 1973; Loveland, 1978)) is the most effective mechanism to avoid logical redundancies in resolution-based calculi. One distinguishes two basic kinds of subsumption: *forward-subsumption* discards newly generated clauses which are subsumed by already existing clauses whereas *backward-subsumption* removes old clauses which are subsumed by new ones. Although subsumption tests can be rather

expensive, these mechanisms are in many cases inevitable in order to find a refutation.

Such reduction mechanisms, however, are not applicable in a straightforward manner in backward-chaining connection calculi.[12] This is because such calculi do not enumerate derivable clauses but try to find a proof by enumerating derivations. Needed are therefore mechanisms which aim at avoiding a derivation if the possibility to extend it to a proof would imply the same possibility for a different and (hopefully) smaller derivation. Prominent examples of such mechanisms are the so-called *identical ancestor check* and its generalization, *regularity*, which are successfully used in several theorem provers (eg. see (Stickel, 1988; Bayerl et al., 1992)).

In (Baumgartner and Brüning, 1997) we studied a further possibility to avoid redundant derivations, which is mostly neglected in connection with top-down backward-chaining calculi: the use of adapted forms of subsumption which compare two connection tableaux (or parts of them) in order to check whether the derivation generating the first connection tableau is redundant w.r.t. the derivation generating the second one. In case clause sets are restricted to Horn clauses, this approach has been extensively studied in (Bol et al., 1991) (in the context of SLD-resolution). There it is shown that whenever the set of open goals after applying a sequence of inference steps $d_1, \ldots, d_n$ is subsumed by some former set of open goals (that is the set of open goals occurring in the connection tableau generated after some d_j with $j < n$), the last inference step, d_n, can be withdrawn. In case of arbitrary (i.e. non-Horn) clause sets, however, it is no longer sufficient only to compare sets of open goals. This is due to the fact that ancestor goals (i.e. A-literals in Loveland's terminology (Loveland, 1978)), which may become important to perform reduction steps, are ignored. Hence, a first step to retain completeness would be to define a subsumption relation on tableaux which compares open goals as well as all possible ancestors. However, it becomes clear quite immediately that without further refinements such a proceeding would be (rather) useless since a connection tableau T_2 generated from a connection tableau T_1 usually contains more (and different) ancestor goals than T_1.

In (Baumgartner and Brüning, 1997), we proposed two approaches to overcome this problem. Both of them provide criteria which allow to identify ancestor goals that can be safely ignored as potential candidates for the application of reduction steps. The first and more important one is based on the *disjunctive positive refinement*. This refinement (which is a generalized version of Plaisted's positive refinement (Plaisted, 1990)) restricts reduction steps to those that use a positive ancestor goal from clauses containing at least

[12] See Footnote 9.

two positive literals (such literals are called *disjunctive positive*) and therefore allows to ignore all negative ancestors. The second approach is to predetermine which ancestors cannot be used for reduction steps in order to solve a goal G. This is achieved by the concept of *reachability* originally proposed in (Neugebauer, 1992) (in a different context).

With these two approaches at hand, (Baumgartner and Brüning, 1997) provides several pruning techniques based on subsumption of connection tableaux. Besides a generalized version of the aforementioned technique for Horn-clause sets (which can be seen as a kind of forward subsumption), a kind of backward subsumption is introduced, which prunes a derivation D_1 if its connection tableau is subsumed (in the above sense) by a connection tableau of a derivation D_2 which is generated *after* D_1 in the course of the deduction process. This requires a new organization of the proof process: instead of constructing *one* connection tableau by guessing inference steps and backtracking on failure, all connection tableaux that are built are stored explicitly. The resulting calculus is a saturation procedure which allows to delete all (forwardly and backwardly) subsumed connection tableaux.

Apart from subsumption techniques comparing entire connection tableaux, (Baumgartner and Brüning, 1997) also introduces the so-called *T-context check* which only takes a single tableau branch into account.[13] It prevents derivations where a goal has to be solved which is equally, or more, "complicated" than some of its ancestor goals. Obviously, the complexity of open goals cannot simply be compared via subsumption. In addition, one has to take care of variable dependencies to other open goals and, as pointed out above, ancestor goals which might become important for the application of reduction steps. An interesting aspect of the T-context check is its independence of the computation rule. This is roughly speaking due to the fact that its definition is based on the connection tableau generated by any derivation rather than a particular one.[14]

To illustrate the potential of the proposed techniques, comprehensive results achieved with two prototypical proof systems are presented in (Baumgartner and Brüning, 1997). The first proof system (named "The_Mission") implements the aforementioned variants of forward and backward subsumption whereas the second proof system implements the T-context check. It turned out, that the number of connection tableaux to be considered (and therefore the required inference steps and time to find a proof) can in many

[13] In the context of SLD-resolution a related technique has been proposed in (Besnard, 1989b) and further developed in (Bol et al., 1991).

[14] Note that there may be many different derivations that generate the same tableau.

cases be considerably reduced (with both systems). As expected, the pruning power of subsumption as used in The_Mission is in fact stronger than the one of the T-context check. However, the T-context check can be implemented very easily without resulting in high computational overheads (what is sometimes the case for the mechanisms used in The_Mission). Therefore it might not only be interesting to improve the implementation of The_Mission; additionally it might be worthwhile to think about an extension of the T-context check which retains its simplicity (in view of an efficient implementation) and covers more of the cases were the mechanisms used in The_Mission can be applied successfully.

A number of further results reported in (Baumgartner and Brüning, 1997) deal with the relationships between subsumption-based and other refinements. In particular, the compatibility with restricted variants of regularity has been studied.[15] It turned out that the presented techniques can be safely combined with *blockwise regularity*, a restricted form of regularity, which "only" requires that two literals on a branch must be different unless there is a disjunctive positive literal between them or both are themselves disjunctive positive.

4.3. *A case-study on nonstandard inferences: Query answering in default logics*

Classical logic provides already by itself a powerful system for knowledge representation and reasoning, offering at once a language, a semantics and highly efficient reasoning methods. Nonetheless one often needs extended logical formalisms for modeling the manifold applications to be dealt with by intelligent systems. Once such an extension is conceived, one can then draw on the large experience gathered on implementations for classical logic for obtaining a corresponding efficient reasoning system. Among numerous such approaches, serving various purposes, we find those dealing with reasoning from incomplete knowledge exemplarily described in this section.

To be more precise, we detail an extension of clausal connection calculi for handling incomplete world descriptions by means of default information. For this, we have chosen Reiter's *default logic* (Reiter, 1980), one of the most prominent approaches to default reasoning, as a point of departure. Since its introduction, it has proven to be extremely valuable for formalizing default reasoning in various domains. Among others, it has been applied to diagnosis (Reiter, 1987), natural language (Mercer, 1988), inheritance networks

[15] Note that due to the incompatibility of the positive refi nement with *unrestricted* regularity, the techniques presented in (Baumgartner and Brüning, 1997) cannot be compatible with this restriction, too.

(Etherington and Reiter, 1983), terminological logics (Baader and Hollunder, 1992), and databases (Cadoli et al., 1994).

Default logic augments classical logic by *default rules* that differ from standard inference rules in sanctioning inferences that rely upon given as well as absent information. Knowledge is represented in default logics by *default theories* (D,W) consisting of a consistent set of formulas W and a set of default rules D. A default rule $\frac{A:B}{C}$ has two types of antecedents: A *prerequisite* A which is established if A is derivable and a *justification* B which is established if B is consistent. If both conditions hold, the *consequent* C is concluded by default. A set of such conclusions (sanctioned by default rules and classical logic) is called an *extension* of an initial set of facts: given a set of formulas W and a set of default rules D, any such extension E is a deductively closed set of formulas containing W such that, for any $\frac{A:B}{C} \in D$, if $A \in E$ and $\neg B \notin E$ then $C \in E$.[16]

In what follows, we are actually interested in implementing the basic approach to query-answering in default logic that allows for determining whether a formula is in *some* extension of a given default theory.[17] For this endeavor, we follow an approach to default query-answering proposed in (Schaub, 1995). This approach furnishes a mating-based characterization of default proofs inside the framework of the connection method (Bibel, 1987b). As shown in (Schaub, 1995; Schaub et al., 1996), this approach is especially qualified for implementations by means of existing automated theorem provers, since it integrates the notion of a default proof into a calculus for classical logic; existing classical theorem provers are then more easily adaptable for implementing default reasoning.

In order to keep our exposition simple, we restrict our attention to propositional default theories, consisting of so-called normal atomic default rules only, that is, default rules, whose atomic consequents are equivalent to their respective atomic justifications; also, the prerequisite is assumed to be atomic. The interested reader may consult (Schaub, 1995) for a treatment of full-fledged default rules.

Let us start with a brief introduction to a default proof theory, abstracting from underlying logical inferences: given a normal default theory (D,W) and a formula F, a *default proof* for F from (D,W) is a finite sequence of default rules $\langle \delta_i \rangle_{i \in I}$ with $\delta_i \in D$ for all $i \in I$ for some ordered index set I such that

$$(1.1) \qquad W \cup \mathit{Conseq}(\{\delta_i \mid i \in I\}) \quad \vdash \quad F \ ,$$

[16] A formal introduction to default logic can be found in (Reiter, 1980; Besnard, 1989a).

[17] Membership in *all* extensions is actually computable by appeal to a procedure testing membership in *some* extension (Thielscher and Schaub, 1995).

$$(1.2) \quad W \cup \mathit{Conseq}(\{\delta_0, \ldots, \delta_{i-1}\}) \quad \vdash \quad \mathit{Prereq}(\delta_i)\ ,$$
$$(1.3) \quad W \cup \mathit{Conseq}(\{\delta_0, \ldots, \delta_{i-1}\}) \quad \nvdash \quad \neg\mathit{Conseq}(\delta_i)\ ,$$

where $\mathit{Conseq}(\cdot)$ and $\mathit{Prereq}(\cdot)$ provide us with the consequents and prerequisites of a given set of default rules.

Condition (1.2) spells out that D' has to be grounded in W; this property reflects the character of an inference rule. In general, a set of default rules D is *grounded* in a set of facts W iff there exists an enumeration $\langle\delta_i\rangle_{i\in I}$ of D that satisfies Condition (1.2). Condition (1.3) expresses the notion of *incremental consistency*. Here, the "consistent" application of a default rule is checked whenever one is applied.[18] So, for verifying whether a query F is derivable from a default theory (D, W) it is, in these terms, enough to determine a grounded and consistent set of default rules $D_F \subseteq D$ that allows for proving F from the facts in W and the consequents of all default rules in D_F.

As an example, consider the following set of statements about a child predisposed to an allergy against milk products: "children normally eat ice-cream", "ice-cream usually contains milk", "ice-cream usually contains sugar", and "milk is an allergen in case of a predisposition". The corresponding default theory along with facts $\mathsf{child} \wedge \mathsf{predispo}$ (expressing that the considered child has the aforementioned predisposition) is the following one:

$$\left(\left\{\ \frac{\mathsf{child}:\mathsf{icecream}}{\mathsf{icecream}},\ \frac{\mathsf{icecream}:\mathsf{milk}}{\mathsf{milk}},\ \frac{\mathsf{icecream}:\mathsf{sugar}}{\mathsf{sugar}}\ \right\},\right.$$
$$(1.4) \qquad \left.\{\mathsf{child}, \mathsf{predispo}, \mathsf{milk} \wedge \mathsf{predispo} \rightarrow \mathsf{allergen}\}\right)$$

For instance, we can explain the presence of an allergen in the above situation by proving $\mathsf{allergen}$ from $\mathsf{child} \wedge \mathsf{predispo}$ by means of the following default proof (i.e. sequence of rule applications).

$$(1.5) \quad \left\langle\ \frac{\mathsf{child}:\mathsf{icecream}}{\mathsf{icecream}},\ \frac{\mathsf{icecream}:\mathsf{milk}}{\mathsf{milk}}\ \right\rangle$$

Importantly, this proof can be found by a top-down backward-chaining procedure which would start from the query chaining down up to the facts in a completely local fashion without any consideration of the irrelevant default rule $\frac{\mathsf{icecream}:\mathsf{sugar}}{\mathsf{sugar}}$. Such procedures are actually used by most automated theorem provers, which illustrates the need for local (default) proof procedures. Here localness is guaranteed by the incrementality of the conditions (1.2) and (1.3); it is not obtainable in general such as, for instance, in Reiter's full-fledged default logic.

Let us now turn to the aforementioned mating-based characterization of default proofs: the approach of (Schaub, 1995) relies on the idea that a default

[18] In the case of non-normal default theories this latter consistency check requires considerably more efforts than in the normal case described here.

rule $\frac{A;B}{C}$ can be decomposed into a *classical implication* $A \to C$ along with two proof-theoretic conditions on the usage of the resulting clause $\{\neg A, C\}$; these conditions are referred to as *admissibility* and *compatibility*. Intuitively, both of them rely on a sequence of clauses, stemming from default rules only, which is induced by the underlying mating (Schaub, 1995). Such a sequence amounts to an enumeration of default rules $\langle \delta_i \rangle_{i \in I}$, as given in default proofs (cf. conditions (1.1)–(1.3)). In fact, while admissibility provides the proof-theoretic counterpart of Condition (1.2), that is groundedness, compatibility enforces the notion of consistency described in Condition (1.3).

In order to find out whether a formula F is in some extension of a default theory (D, W), one proceeds as follows. First, we transform the default rules in D into a set of indexed implications W_D. In our example, this encoding yields the set

$$\begin{aligned} W_D \quad = \quad & \{\mathsf{child}_{\delta_1} \to \mathsf{icecream}_{\delta_1}, \\ & \mathsf{icecream}_{\delta_2} \to \mathsf{milk}_{\delta_2},\ \mathsf{icecream}_{\delta_3} \to \mathsf{sugar}_{\delta_3}\}. \end{aligned} \tag{1.6}$$

The indexes denote the respective default rules in (1.4) from left to right.

Second, we transform both W and W_D into their clausal forms, C_W and C_D. The clauses in C_D are called δ-*clauses*; they are of the form[19] $\{\neg\alpha_\delta, \gamma_\delta\}$; all other clauses are referred to as ω-*clauses*. In our example, we obtain the following matrix for $C_W \cup C_D$:

$$\begin{aligned} & \{\{\mathsf{predispo}\}, \{\mathsf{child}\}, \{\neg\mathsf{predispo}, \neg\mathsf{milk}, \mathsf{allergen}\}\} \\ \cup \quad & \{\{\neg\mathsf{child}_{\delta_1}, \mathsf{icecream}_{\delta_1}\}, \{\neg\mathsf{icecream}_{\delta_2}, \mathsf{milk}_{\delta_2}\}, \\ & \qquad\qquad \{\neg\mathsf{icecream}_{\delta_3}, \mathsf{sugar}_{\delta_3}\}\} \end{aligned} \tag{1.7}$$

Finally, a query F is derivable from (D, W) if there is a spanning mating for the matrix $C_W \cup C_D \cup \{\neg F\}$ agreeing with the concepts of admissibility and compatibility.

As an illustration, let us explain the presence of an allergic reaction. For this, we have to add the clause containing the negated query $\{\neg\mathsf{allergen}\}$ to Matrix (1.7). The resulting matrix can be given (after some clause reordering) two-dimensionally in the way shown in Figure 16. As indicated by means of the arcs linking the respective literals, this matrix has a spanning mating. As it stands, this is a classical proof of allergen from $W \cup W_D$. In order to verify that this proof is also a default proof we have to confirm admissibility and compatibility. For this, we consider the enumeration:

$$\langle \{\neg\mathsf{child}_{\delta_1}, \mathsf{icecream}_{\delta_1}\}, \{\neg\mathsf{icecream}_{\delta_2}, \mathsf{milk}_{\delta_2}\} \rangle\ . \tag{1.8}$$

[19] Observe that the atomic format allows us to deal with binary δ-clauses only.

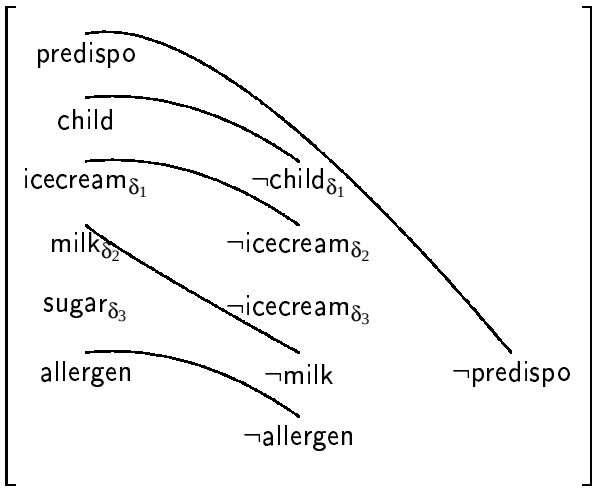

Figure 16. The matrix with its proof for the icecream example

This sequence corresponds to default proof (1.5); it is (roughly) obtained by reversing the sequence obtained by collecting all δ-clauses when following the connections, starting from the query-clause.

Recall that the groundedness condition (1.2) stipulates that each prerequisite α_{δ_i} of a default rule in a default proof $\langle\delta_i\rangle_{i\in I}$ has a classical proof from W and consequences of rules in sequence $\langle\delta_j\rangle_{j<i}$ only. This criterion is mapped on that of admissibility:[20] we must verify that *all* those matrices possess a spanning mating which consist of, in turn: (A1) the negated prerequisite of one δ-clause in the sequence, viz. $\{\neg\alpha_{\delta_i}\}$, (A2) all previous δ-clauses in the same sequence, viz. $\{\{\neg\alpha_{\delta_j},\gamma_{\delta_j}\} \mid j < i\}$, and (A3) all clauses in C_W. Each instance of this setting amounts to a classical proof of α_{δ_i} from the premises given in (1.2).

In our example, we thus have to consider the following submatrices[21] of the matrix in Figure 16:

$$C_W \cup \{\{\neg\mathsf{child}_{\delta_1}\}\} \qquad \text{and}$$

$$C_W \cup \{\{\neg\mathsf{child}_{\delta_1}, \mathsf{icecream}_{\delta_1}\}, \{\neg\mathsf{icecream}_{\delta_2}\}\}\ ;$$

both of which are spanned by the mating in Figure 16 (even though some connections in Figure 16 are obsolete). We thus obtain two proofs for prereq-

[20] Note that admissibility relies on δ-clauses rather than default consequences.

[21] A matrix M is a *submatrix* of another matrix M' if M emerges from M' by removing literals and/or entire clauses.

uisites that are independent of the δ-clauses in the upper part of Sequence (1.8). Hence, the matrix in Figure 16 and its mating constitute an admissible proof.

Remember that the consistency condition (1.3) requires that each consequence γ_{δ_i} of a (normal) default rule in a default proof $\langle \delta_i \rangle_{i \in I}$ is consistent with W and all consequences of rules in sequence $\langle \delta_j \rangle_{j<i}$. In presence of an admissibility check, this criterion can be mapped on the following notion of compatibility.[22] For this, we must verify that *all* matrices which consist of, in turn: (C1) one δ-clause in the sequence, viz. $\{\neg\alpha_{\delta_i}, \gamma_{\delta_i}\}$, (C2) all previous δ-clauses in the same sequence, viz. $\{\{\neg\alpha_{\delta_j}, \gamma_{\delta_j}\} \mid j < i\}$, and (C3) all clauses in C_W, possess *no* spanning mating. Together these conditions amount to the proof-theoretic counterpart of consistency as given in Condition (1.3), verifying consistency of each δ-clause in turn.

In our example, we thus have to consider the following submatrices of the matrix in Figure 16:

$$C_W \cup \{\{\neg \mathsf{child}_{\delta_1}, \mathsf{icecream}_{\delta_1}\}\} \quad \text{and}$$

$$C_W \cup \{\{\neg \mathsf{child}_{\delta_1}, \mathsf{icecream}_{\delta_1}\}, \{\neg \mathsf{icecream}_{\delta_2}, \mathsf{milk}_{\delta_2}\}\} \; .$$

Both matrices possess no spanning mating, which establishes compatibility in our example. To see this, observe that the latter matrix contains the path

$$\{\mathsf{predispo}, \mathsf{child}, \mathsf{allergen}, \mathsf{icecream}_{\delta_1}, \mathsf{milk}_{\delta_2}\} \; ,$$

which contains no complementary literals. Clearly, the existence of such a path for the latter matrix implies the same for the former matrix.[23] Notably any such path represents a (partial) *model* of the considered formula. In this way, the actual task of consistency checking can be mapped onto the generation of propositional models (Brüning and Schaub, 1996).

To summarize, we have shown that the classical proof of allergen from $W \cup W_D$ given by the Matrix in Figure 16 and its spanning mating enjoys admissibility and compatibility, which renders it a default proof of allergen. The overall method is shown to be sound and complete in (Schaub, 1995). It is important to note that the verification of admissibility and compatibility relies on substructures of the corresponding classical proof (established by complementarity); this leaves much room for structure sharing in order to ease the additional burden added by the treatment of default rules.

[22] Note that compatibility relies on δ-clauses rather than default consequences.

[23] Considering both matrices may seem redundant from our declarative point of view. This is, however, different from a procedural point of view involving incremental consistency checks.

Corresponding proof procedures have already been developed in different settings and implemented in a system (Schaub, 1995; Schaub and Brüning, 1996). Most of them are carried out by means of inference operations known from connection calculi (Bibel, 1987b) or model elimination (Loveland, 1978), namely *extension* and *reduction* steps. For incorporating default reasoning into such a calculus, (roughly) the extension step has to be adapted: whenever a δ-clause $\{\neg A_\delta, C_\delta\}$ is used as input clause, one has to guarantee (i) that only C_δ is resolved upon, and (ii) that after such an "extension step" the ancestor goals of the resulting subgoal $\neg A_\delta$ must not be used for later reduction steps. Moreover, (iii) each such "extension step" must guarantee the compatibility of the obtained proof segment.

A corresponding proof system, called XRay, is described in (Schaub et al., 1996; Schaub and Nicolas, 1997).

5. Conclusions

Within this volume, the part on tableaux, of which this chapter is the last one, has embedded novel results within the classical line of research pursued within tableaux-like calculi. In a sense it has been the task of the present chapter to point out the great potential yet contained in the employment of specialized techniques which for special classes of problems are indispensable for their successful treatment. Here "specialization" is not meant as an independent approach outside of the general tableaux techniques. Rather it is meant as a special variant within the general approach focusing on particular syntactic features of formulas. Most if not all such variants may be regarded as instances of the general principle of compression.

In fact, it is for this principle of compression that we have grounded our work in the connection method rather than in standard tableau methods, since the connection method is already a compressed version of tableaux. Given this basis we have presented two specialized techniques in detail. One applies to the class of problems which involve many facts and are in this respect similar to standard databases. Consequently we have been talking of DB-reduction and DB-unification in the realization of this particular technique. Its application to problems heavily loaded with facts drastically reduces the number of backtracking steps during proof search, because the technique does not commit to a particular solution until it is forced to. Because of this, the technique may also be considered as a lazy evaluation technique which, as shown in Section 2, in fact amounts to the compression of the search space of possible tableaux. As an additional benefit, not only one but several solutions, encoded in the substitution represented by the DB-abstraction trees, are found

with this technique. This is a clear advantage in case the prover has to search for more or even all solutions of a problem.

Although a large amount of work was done to examine the possibilities of compression and lazy evaluation in the area of automated theorem proving, there are still four main aspects of DB-unification which look very promising but were not examined in detail up to now. The first is the effect of splitting the DB-abstraction trees during the creation of the DB-clauses and within DB-unification (as shortly outlined in Section 2.5). The second aspect is the combination of regularity, subsumption, tautology, and failure caching constraints (see Chapter 2) with the data structures used in DB-unification. It will require the implementation of additional operations on DB-abstraction trees. Furthermore, the combination of DB-unification with the creation of unit lemmata (Astrachan and Stickel, 1992) looks very promising because, with lazy evaluation, all unit lemmata may be handled in one proof attempt so that the search space does not have to be examined again and again for every single lemma. And, finally, it will be very interesting to study the effect of melting more (possibly even all) non-unit clauses of a formula into a single clause by using extended DB-reduction. This is comparable to an extensive factorization of the formula amounting to a non-normal form of the formula, but done here on the basis of a normal form theorem prover. As noted at the end of Section 2.3 this amounts to making the given formulas more regular and thus easier to prove.

The second specialized technique presented in this chapter is a non-clausal decision procedure for propositional logic in the form of a generalization of the standard Davis-Putnam procedure. While theoretically this generalization is straightforward, we obtain by this representation of formulas by non-normal form matrices (suggested earlier for connection methods) a compression of the formula as well as of the corresponding search space.

In experiments the new procedure has been compared to a clause-based Davis-Putnam procedure. The results show that the standard transformation into clausal form and even the definitional transformation can blow up the proof process. Whereas the standard transformation may considerably increase the size of the formulas, the definitional transformation introduces extra variables. Working directly on the compact representation of the non-clausal matrices does not suffer from either problem and therefore allows shorter proofs. The shortening is due to the application of the matrix elimination steps and the beta-splitting rule.

It does of course not make much sense to apply our procedure to problems which are already formulated in clausal form. For such problems specialized (clause-based) proof procedures are generally more efficient. For problems which are formulated in a non-clausal form (like those deriving from formulas

in intuitionistic propositional logic) the non-clausal approach, however, often fares much better.

Of course, there is still plenty of room for improvements and further research on this co-$\mathcal{NP}$ complete problem. To begin with a "simple" one, the performance of the procedure could be greatly enhanced by implementing it in a more machine-oriented programming language (like C). Furthermore, there are many optimization techniques for clausal provers which could be transferred to the non-clausal case (for example, the selection of the *splitting literal*). Another interesting issue would be a comparison to other (complete) proof methods such as BDDs (Uribe and Stickel, 1994).

The research reported in this chapter is carried out in the context of the development of the theorem prover KoMeT. Many techniques other than those reported here have been studied and partially integrated in it, all published elsewhere as mentioned in the introduction. In order to give a feel for the flavor of some of these additional techniques, the chapter summarizes two further ones of a rather different nature, namely the treatment of equivalences in an equation-oriented way and the incorporation of subsumption. Both are again good examples of specialization through compression in view of special syntactic features. With all these different techniques we envisage a future computationally adequate theorem prover which will behave optimally also for problems of a very specific nature without compromising generality.

On the surface intellectics appears to be much more than reasoning in classical logic. Therefore intellecticians have occasionally doubted whether theorem proving might be useful at all for their applications. A second line of our research centered around KoMeT has therefore been to extend the area of competence for theorem provers in general and for techniques used in KoMeT in particular. Apart from the extensions mentioned in the introduction the chapter ends with a technique which makes KoMeT-type provers applicable to nonmonotonic reasoning.

Acknowledgements. We want to thank Ulrich Furbach and David Plaisted for their numerous suggestions for improvement of the text and presentation.

W. Bibel

REFERENCES

Astrachan, O. L. and M. E. Stickel: 1992, 'Caching and Lemmaizing in Model Elimination Theorem Provers'. In: D. Kapur (ed.): *Automated Deduction — Cade-11*. pp. 224–238.

Baader, F. and B. Hollunder: 1992, 'Embedding Defaults into Terminological Knowledge Representation Formalisms'. In: B. Nebel, C. Rich, and W. Swartout (eds.): *Proceedings of the Third International Conference on the Principles of Knowledge Representation and Reasoning*. Cambridge, MA, pp. 306–317.

Bachmair, L. and H. Ganzinger: 1992, 'Non-Clausal Resolution and Superposition with Selection and Redundancy Criteria'. In: A. Voronkov (ed.): *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, Vol. 624 of *Lecture Notes in Artificial Intelligence*. pp. 273–284.

Baumgartner, P. and S. Brüning: 1997, 'A Disjunctive Positive Refinement of Model Elimination and its Application to Subsumption Deletion'. *Journal of Automated Reasoning* **19**, 205–262.

Bayerl, S., R. Letz, J. Schumann, and W. Bibel: 1992, 'SETHEO: A High-Performance Theorem Prover'. *Journal of Automated Reasoning* **8**, 183–212.

Besnard, P.: 1989a, *An Introduction to Default Logic*, Symbolic Computation — Artificial Intelligence. Berlin: Springer.

Besnard, P.: 1989b, 'On infinite loops in logic programming'. Technical Report 488, IRISA, Rennes, France.

Bibel, W.: 1983, 'Matings in Matrices'. *Comm. ACM* **26**, 844–852.

Bibel, W.: 1987a, *Automated Theorem Proving*. Braunschweig: Vieweg Verlag, 2. edition.

Bibel, W.: 1987b, *Automated Theorem Proving*. Braunschweig: Vieweg Verlag, 2. edition.

Bibel, W.: 1988, 'Advanced Topics in Automated Deduction'. In: R. Nossum (ed.): *Fundamentals of Artificial Intelligence II*. Berlin, pp. 41–59.

Bibel, W.: 1991, 'Perspectives on Automated Deduction'. In: R. S. Boyer (ed.): *Automated Reasoning: Essays in Honor of Woody Bledsoe*. Utrecht: Kluwer Academic, pp. 77–104.

Bibel, W.: 1993, *Deduction: Automated Logic*. London: Academic Press.

Bibel, W.: 1997, 'Let's plan it deductively!'. In: M. Pollack (ed.): *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-97)*. Morgan Kaufmann, San Mateo, CA, pp. 1549–1562. A revised and extended version is submitted to the AIJ.

Bibel, W., S. Brüning, U. Egly, D. Korn, and T. Rath: 1995, 'Issues in Theorem Proving Based on the Connection Method'. In: P. Baumgartner, R. Hähnle, and J. Posegga (eds.): *Theorem Proving with Analytical Tableaux and Related Methods, Proceedings of the Fourth International Workshop, TABLEAUX'95*. Berlin, pp. 1–16.

Bibel, W., S. Brüning, U. Egly, and T. Rath: 1994a, 'KoMeT'. In: A. Bundy (ed.): *Proceedings of the International Conference on Automated Deduction, CADE-94*. Berlin, pp. 783–787. 1994.

Bibel, W., S. Brüning, U. Egly, and T. Rath: 1994b, 'Towards an Adequate Theorem Prover Based on the Connection Method'. In: *Proceedings of the Sixth International Conference on Artificial Intelligence and Information-Control of Robots*. pp. 137–148.

Bibel, W. and E. Eder: 1997, 'Decomposition of tautologies into regular formulas and strong completeness of connection-graph resolution'. *Journal of the ACM* **43**(1).

Bibel, W., D. Korn, C. Kreitz, F. Kurucz, J. Otten, S. Schmitt, and G. Stolpmann: 1997, 'A multi-level approach to program synthesis'. In: N. E. Fuchs (ed.): *Proceedings of the 7th Workshop on Logic Program Synthesis and Transformation (LOPSTR-97)*. Springer, Berlin.

Bibel, W., D. Korn, C. Kreitz, and S. Schmitt: 1996, 'Problem-Oriented Applications of Automated Theorem Proving'. In: L. C. Aiello (ed.): *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO'96)*. Springer, Berlin, pp. 1–21.

Bibel, W., R. Letz, and J. Schumann: 1987, 'Bottom–up Enhancements of Deductive Systems'. In: I. Plander (ed.): *Proceedings of 4th International Conference on Artifi cial Intelligence and Information–Control Systems of Robots*. Smolenice, CSSR, pp. 1–10.

Bol, R. N., K. R. Apt, and J. W. Klop: 1991, 'An analysis of loop checking mechanisms for logic programming'. *Theoretical Computer Science* **86**, 35–79.

Brüning, S.: 1995, 'Exploiting Equivalences in Connection Calculi'. *Journal of the IGPL* **3**(6), 857–886.

Brüning, S. and T. Schaub: 1996, 'A model-based approach to consistency-checking'. In: Z. Ras and M. Michalewicz (eds.): *Proceedings of the Ninth International Symposium on Methodologies for Intelligent Systems*, Vol. 1079 of *Lecture Notes in Artifi cial Intelligence*. pp. 315–324.

Cadoli, M., T. Eiter, and G. Gottlob: 1994, 'Default Logic as a Query Language'. In: J. Doyle, P. Torasso, and E. Sandewall (eds.): *Proceedings of the Fourth International Conference on the Principles of Knowledge Representation and Reasoning*. pp. 99–108.

Chang, C. L. and R. C.-T. Lee: 1973, *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York.

Crawford, J. and L. Auton: 1993, 'Experimental Results on the Crossover Point in Satisfi ability Problems'. In: *Proceedings 11th National Conference on AI*. pp. 21–27.

Davis, M., G. Logemann, and D. Loveland: 1962, 'A Machine Program for Theorem-Proving'. *Communications of the ACM* **5**, 394–397.

Davis, M. and H. Putnam: 1960, 'A Computing Procedure for Quantifi cation Theory'. *Journal of ACM* **7**, 201–215.

Eder, E.: 1992, *Relative Complexities of First Order Calculi*. Vieweg Verlag.

Etherington, D. and R. Reiter: 1983, 'On Inheritance Hierarchies with Exceptions'. In: *Proceedings of the AAAI National Conference on Artifi cial Intelligence*. pp. 104–108.

Freeman, J.: 1995, 'Improvements to Propositional Satisfi ability Search Algorithms'. Ph.D. thesis, University of Pennsylvania.

Korn, D. and C. Kreitz: 1997, 'Effi ciently Deciding Intuitionistic Propositional Logic via Translation into Classical Logic'. In: *Proceedings of the 14th Conference on Automated Deduction*.

Lee, S.-J. and D. A. Plaisted: 1990, 'Reasoning with Predicate Replacement'. In: *Proceedings of ISMIS*.

Lee, S.-J. and D. A. Plaisted: 1992, 'Eliminating Duplication with the Hyper-Linking Strategy'. *Journal of Automated Reasoning* **9**, 25–42.

Letz, R., , K. Mayr, and C. Goller: 1994, 'Controlled Integration of the Cut Rule into Connection Tableau Calculi'. *Journal of Automated Reasoning* **13**, 297–337.

Li, C.: 1996, 'Exploiting Yet More the Power of Unit Clause Propagation to Solve the 3-SAT Problem'. In: *Proceedings ECAI'96 Workshop on Advances in Propositional Deduction*. pp. 11–16.

Loveland, D.: 1978, *Automated Theorem Proving: A Logical Basis*. New York: North-Holland.

Mercer, R.: 1988, 'Using Default Logic to Derive Natural Language Suppositions'. In: *Proceedings of Canadian Society for Computational Studies of Intelligence Conference*. pp. 14–21.

Moser, M., O. Ibens, R. Letz, J. Steinbach, C. Goller, J. Schumann, and K. Mayr: 1997, 'SETHEO and E-SETHEO. The CADE-13 Systems.'. *Journal of Automated Reasoning* **18**(2), 237–246.

Neugebauer, G.: 1992, 'From Horn Clauses to First Order Logic: A Graceful Ascent'. Technical Report AIDA–92–21, FG Intellektik, FB Informatik, TH Darmstadt.

Ohlbach, H. J.: 1990, 'Abstraction Tree Indexing for Terms'. In: *ECAI 90. Proceedings of the 9th European Conference on Artifi cial Intelligence*. pp. 479–484.

Otten, J.: 1997, 'On the Advantage of a Non-Clausal Davis-Putnam Procedure'. Technical Report AIDA–97–01, Technische Universität Darmstadt, FG Intellektik.

Pelletier, F.: 1986, 'Seventy-fi ve Problems for Testing Automatic Theorem Proving'. *Journal of Automated Reasoning* **2**, 191–216.

Plaisted, D. A.: 1990, 'A Sequent-Style Model Elimination Strategy and a Positive Refi nement'. *Journal of Automated Reasoning* **4**(6), 389–402.

Plaisted, D. A. and S. Greenbaum: 1986, 'A Structure-Preserving Clause Form Translation'. *Journal of Symbolic Computation* **2**, 293–304.

Rath, T.: 1992, 'Datenbankunifi kation'. Technical Report AIDA–92–09, FG Intellektik, FB Informatik, TH Darmstadt.

Reiter, R.: 1980, 'A logic for default reasoning'. *Artifi cial Intelligence Journal* **13**, 81–132.

Reiter, R.: 1987, 'A Theory of Diagnosis from First Principles'. *Artifi cial Intelligence Journal* **32**(1), 57–95.

Schaub, T.: 1995, 'A New Methodology for Query Answering in Default Logics via Structure-Oriented Theorem Proving'. *Journal of Automated Reasoning* **15**(1), 95–165.

Schaub, T. and S. Brüning: 1996, 'Prolog technology for default reasoning'. In: W. Wahlster (ed.): *Proceedings of the European Conference on Artifi cial Intelligence*. pp. 105–109.

Schaub, T., S. Brüning, and P. Nicolas: 1996, 'XRay: A Prolog Technology Theorem Prover for Default Reasoning: A System Description.'. In: M. McRobbie and J. Slaney (eds.): *Proceedings of the Conference on Automated Deduction*, Vol. 1104 of *Lecture Notes in Artifi cial Intelligence*. pp. 293–297.

Schaub, T. and P. Nicolas: 1997, 'An implementation platform for query-answering in default logics: The XRay system, its implementation and evaluation'. In: J. Dix, U. Furbach, and A. Nerode (eds.): *Proceedings of the Fourth International Conference on Logic Programming and Non-Monotonic Reasoning*, Vol. 1265 of *Lecture Notes in Artifi cial Intelligence*. pp. 442–453.

Socher-Ambrosius, R.: 1989, 'A Resolution Calculus Extended by Equivalence'. In: D. Metzing (ed.): *Proceedings of the German Workshop on Artifi cial Intelligence*. pp. 102–106.

Socher-Ambrosius, R.: 1990, 'Simplifi cation and Reduction for Automated Theorem Proving'. Technical Report SR-90-10, FB Informatik, Universität Kaiserslautern. Dissertation.

Stickel, M. E.: 1988, 'A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler'. *Journal of Automated Reasoning* **4**, 353–380.

Sutcliffe, G., C. Suttner, and T. Yemenis: 1994, 'The TPTP Problem Library'. In: A. Bundy (ed.): *Proceedings of the International Conference on Automated Deduction, CADE-94*. Berlin, pp. 252–266. 1994.

Thielscher, M. and T. Schaub: 1995, 'Default Reasoning by Deductive Planning'. *Journal of Automated Reasoning* **15**(1), 1–40.

Ullman, J. D.: 1982, *Principles of Database Systems*. Rockville MD: Computer Science Press.

Uribe, T. and M. Stickel: 1994, 'Ordered Binary Decision Diagrams and the Davis-Putnam Procedure'. In: J.-P. Jouannaud (ed.): *Proceedings 1st International Conference on Constraints in Computational Logics*, Vol. 845 of *Lecture Notes in Computer Science*. pp. 34–49.

Zhang, H.: 1993, 'A Decision Procedure for Propositional Logic'. *Association for Automated Reasoning Newsletter* **22**, 1–3.