Tableaux for Projection Computation and Knowledge Compilation

Christoph Wernhard
christoph.wernhard@tu-dresden.de

Technische Universität Dresden

Abstract. Projection computation is a generalization of second-order quantifier elimination, which in turn is closely related to the computation of forgetting and of uniform interpolants. On the basis of a unified view on projection computation and knowledge compilation, we develop a framework for applying tableau methods to these tasks. It takes refinements from performance oriented systems into account. Formula simplifications are incorporated at the level of tableau structure modification, and at the level of simplifying encountered subformulas that are not yet fully compiled. In particular, such simplifications can involve projection computation, where this is possible with low cost. We represent tableau construction by means of rewrite rules on formulas, extended with some auxiliary functors, which is particularly convenient for formula transformation tasks. As instantiations of the framework, we discuss approaches to propositional knowledge compilation from the literature, including adaptions of DPLL, and the hyper tableau calculus for first-order clauses.

1 Introduction

There are two established families of methods for second-order quantifier elimination, the SCAN approach, based on resolvent generation, and direct methods, based and Ackermann's lemma [9]. Over the last decade, several methods for second-order quantifier elimination have been proposed that can be classified as belonging to a third approach: tableau construction [12, 4, 14, 16, 11]. All of these are restricted to propositional logic (and thus, more specifically perform Boolean quantifier elimination) and most of them are based on DPLL which we regard as a tableau procedure, considering semantic trees as tableaux. They are combined with *knowledge compilation*, the transformation of formulas such that they meet syntactic criteria which permit to execute certain operations with low cost. They are described with different techniques, including pseudocode as common in the literature on modern DPLL methods, and with varying terminology such as *marginalization*, *computation of uniform interpolants*, *forgetting* and *projection computation* for second-order quantifier elimination or closely related tasks. In this paper, we stick with *projection computation*, specified explicitly as a generalization of second-order quantifier elimination.

Our goal in this paper is to develop a framework that realizes abstractions in two respects: First, it provides a unified view on projection computation and certain kinds of knowledge compilation. Second, it abstracts from differences in tableau construction methods that are inessential for the adaption to such tasks. Adaptions of the methods of successful automated tableau systems, such as modern SAT solvers or hyper tableau systems, can then be modeled as instantiations of the framework.

The basic idea is that the whole tableau itself is the objective of computation – in contrast to a single branch representing a model, or a proof of unsatisfiability where the tableau represents the search trace. A computed tableau then represents a transformed input formula that satisfies syntactic criteria which permit certain operations – especially projection computation – to be performed with low cost.

The paper is structured as follows: Preliminaries on projection and the considered compilation target format, linkless formulas, are given in Sect. 2, followed by the definition of a relation that generalizes the relationships between inputs and outputs of projection computation, knowledge compilation, and their combinations. In Sect. 3, a tableau variant is presented that is particularly suited for formula transformation tasks, since tableaux are represented as formulas with some additional structure indicating operators. Tableau construction rules then closely resemble formula rewriting rules. A tableau calculus is introduced and – based on the relation defined in Sect. 2 – shown to satisfy correctness properties with respect to projection computation and knowledge compilation. Refinements of the calculus that seem essential for practical application, such as and-nodes, tableau level simplification and backjumping, are discussed in Sect. 4. Methods for projection computation and knowledge compilation that are based on well known practical successful methods such as DPLL and hyper tableaux are then modeled in Sect. 5 as instances of this tableau framework.

Notation. Unless specially noted, we assume that a formula is in negation normal form, constructed from first-order literals, truth value constants $\top, \bot$, binary connectives $\wedge, \vee$ and the universal first-order quantifier $\forall$. N-ary connectives are understood as meta-level notation with respect to this syntax. We write the positive (negative) literal with atom A as $+A$ ($-A$) and the complement of literal L as $\widetilde{L}$. As usual, a *sentence* is a formula without free variables. A *universal literal* is a sentence of the form $\forall x_1 \ldots \forall x_n \; L$, where $n \geq 0$ and L is a literal. The symbol LIT denotes the set of all universal literals. We assume a fixed first-order signature Σ and refer to the ground atoms and ground literals constructable from it as *all* ground atoms and ground literals, respectively. The symbol ALL denotes the set of all ground literals, POS all positive ground literals. The *atom base* $\mathcal{A}(F)$ (*literal base* $\mathcal{L}(F)$) of a formula F is the set of all ground atoms (ground literals) which are instance of an of atom (literal) in F.

Semantic Framework. We use the notational variant of Herbrand interpretations described in [19]: An *interpretation* is a pair $\langle I, \beta \rangle$, where I is a *structure*, that is, a set of ground literals that contains for all ground atoms A exactly one of $+A$ or $-A$, and β is a *variable assignment*, that is, a mapping of the set of variables into the set of ground terms. The set of literals I of an interpretation $\langle I, \beta \rangle$ is called *"structure"*, since it represents a structure in the conventional sense used in model theory: The domain is the set of ground terms. Function symbols f with arity $n \geq 0$ are mapped to functions f' such that for all ground terms $t_1, ..., t_n$ it holds that $f'(t_1, ..., t_n) = f(t_1, ..., t_n)$. Predicate symbols p with arity $n \geq 0$ are mapped to $\{\langle t_1, ..., t_n \rangle \mid +p(t_1, ..., t_n) \in I\}$. Moreover, an interpretation $\langle I, \beta \rangle$ represents a conventional second-order interpretation [8] (if predicate

variables are considered as distinguished predicate symbols): The structure in the conventional sense corresponds to I, as described above, except that mappings of predicate variables are omitted. The assignment is β, extended such that all predicate variables p are mapped to $\{\langle t_1, ..., t_n\rangle \mid +p(t_1, ..., t_n) \in I\}$.

The satisfaction relation is defined as usual by a clause for each logic operator, for example: $\langle I, \beta\rangle \models (F_1 \wedge F_2)$ iff$_{\text{def}}$ $\langle I, \beta\rangle \models F_1$ and $\langle I, \beta\rangle \models F_2$. Entailment and equivalence of formulas are straightforwardly defined in terms of the satisfaction relation: A formula F_1 *entails* a formula F_2, in symbols $F_1 \models F_2$, if and only if for all interpretations $\langle I, \beta\rangle$ it holds that if $\langle I, \beta\rangle \models F_1$ then $\langle I, \beta\rangle \models F_2$. A formula F_1 is *equivalent* to a formula F_2, in symbols $F_1 \equiv F_2$, if and only if $F_1 \models F_2$ and $F_2 \models F_1$.

2 A General View on Projection and Compilation

Projection onto Literal Scopes. Projection computation is a generalization of second-order quantifier elimination which permits, so to speak, to "quantify" upon an arbitrary set of ground literals instead of just (all ground literals with) a given predicate symbol. We pursue the following formal approach to projection computation: The syntax of formulas is extended by a projection operator $\mathsf{project}(F, S)$, where F is a formula and S specifies a set of ground literals. We call a set of ground literals in the role as argument to projection a *literal scope*. The formula $\mathsf{project}(F, S)$ is called the *literal projection* of F onto S.

Projection computation then means to compute for a given formula that contains the project operator an equivalent formula without the project operator. This is analogous to second-order quantifier elimination: computing for a given formula with second-order quantifiers an equivalent formula without second-order quantifiers. Existential second-order quantification is a special case of projection, which could be defined as $\exists p\, F \stackrel{\text{def}}{=} \mathsf{project}(F, S)$, where S is the set of all ground literals with a predicate other than p. The semantics of the project operator is specified with the following clause in the definition of the satisfaction relation, which is added to the usual clauses for the classical operators:

$$\langle I, \beta\rangle \models \mathsf{project}(F, S) \text{ iff}_{\text{def}} \text{ there exists a structure } J \text{ such that } \langle J, \beta\rangle \models F \text{ and } J \cap S \subseteq I. \qquad \text{(PROJ)}$$

Some properties of projection are displayed in Table 1. Property (viii) will be discussed in the next section. For more comprehensive material see [19, 20, 18, 13].

(i)	If $F_1 \models F_2$ then $\mathsf{project}(F_1, S) \models \mathsf{project}(F_2, S)$.
(ii)	$F \models \mathsf{project}(F, S)$.
(iii)	$\mathsf{project}(\mathsf{project}(F, S_1), S_2) \equiv \mathsf{project}(F, S_1 \cap S_2)$.
(iv)	$\mathsf{project}(F, S)$ is satisfiable iff F is satisfiable.
(v)	$\mathsf{project}(L, S) \equiv L$, if $\mathcal{L}(L) \subseteq S$.
(vi)	$\mathsf{project}(L, S) \equiv \top$, if $\mathcal{L}(L) \cap S = \emptyset$.
(vii)	$\mathsf{project}(F_1 \vee F_2, S) \equiv \mathsf{project}(F_1, S) \vee \mathsf{project}(F_2, S)$.
(viii)	If $\langle F_1, F_2\rangle$ is linkless outside S, then $\mathsf{project}(F_1 \wedge F_2, S) \equiv \mathsf{project}(F_1, S) \wedge \mathsf{project}(F_2, S)$.

Table 1. Some properties of projection that hold for formulas F, F_1, F_2, literal scopes S, S_1, S_2, and universal literals L

Linkless Formulas. Projection does not in general distribute over conjunction, as it does over disjunction, but under the precondition that the pair of the conjuncts is *linkless outside* the scope of the projection (Tab. 1. viii). This property is defined as follows, along with related properties, discussed below:

Definition 1 (Linkless). Let F, F_1, F_2 be formulas and let S be a literal scope.

(i) The pair $\langle F_1, F_2 \rangle$ is *linkless outside* S if and only if $\mathcal{L}(F_1) \cap \widetilde{\mathcal{L}(F_2)} \subseteq S \cap \widetilde{S}$.

(ii) F is *linkless outside* S if and only if each pair of conjuncts of F is linkless outside S, where the pairs of conjuncts of F are the pairs $\langle F_1, F_2 \rangle$ for each of its subformulas of the form $(F_1 \wedge F_2)$ and the pairs $\langle F_1, F_1 \rangle$ for each of its subformulas of the form $\forall x F_1$.

(iii) *Fully linkless* is a synonym for *linkless outside the empty set of literals.*

The term *linkless* stems from the concept of *link* as a pair of occurrences of complementary literals, each in a different conjunct of a conjunction, in the graph-based formula view of [15]. A pair of formulas is *linkless outside* a literal scope (Def. 1.i) if all ground atoms "involved in links" between the formulas (that is, are the atoms of complementary ground instances of two literals, one in each component of the pair) are contained in the literal scope, positively as well as negatively. For example, the pair of formulas $\langle p \vee q,\ \neg p \vee q \rangle$ is linkless outside the literal scope $\{+p, -p\}$. The relation is symmetric with respect to the pair components. With Def. 1.ii, the notion of *linkless* is transferred from pairs of formulas to formulas: A formula is linkless if for each of its conjunctive subformulas the pair of conjuncts is linkless, where "conjunctive subformulas" are introduced explicitly by the $\wedge$ operator and implicitly by universal quantification.

Linkless formulas permit to perform certain computations with low complexity. For a propositional formula that is fully linkless, satisfiability and clausal entailment can be decided in linear time [16, 20]. If F is a propositional formula that is linkless outside a literal scope S, then projection computation for $\mathsf{project}(F, S)$ can be performed in linear time by simply replacing in F all literals that are not in S with $\top$. This can be justified as follows: Since F is linkless outside S, by Tab. 1.viii and vii the project operator can be distributed inwards immediately in front of literals L. Now, if L is in S, then by Tab. 1.v it holds that $\mathsf{project}(L, S) \equiv L$; Otherwise by Tab. 1.vi it holds that $\mathsf{project}(L, S) \equiv \top$.

The LP Relation: Generalizing Projection and Compilation. The LP relation, defined in the following, covers in different instantiations the relationships between inputs and outputs of projection computation, compilation to linkless formulas, and combinations of these. The name LP suggests the combination of a syntactic requirement related to *L*inklessness with semantic conditions related to *P*rojection.

The LP relation has four arguments, where the first three represent inputs: A formula and two literal scopes, "link scope" and "projection scope", where the former is a subset of or equal to the latter. The fourth argument represents the output, a formula. The relation constrains output formulas syntactically with respect to the link scope, and semantically with respect to the input formula and the projection scope: An output formula must be *linkless outside the link scope* and be equivalent to the input formula *relative to the projection scope.*

Definition 2 (The LP Relation). For formulas F, F' and literal scopes S_l, S_p such that $S_l \subseteq S_p$ the relation LP is defined as

$$\begin{aligned}&\mathsf{LP}(F, S_l, S_p, F') \stackrel{\text{def}}{=}\\ &\quad F' \text{ is linkless outside } S_l, && \text{(LP-L)}\\ &\quad \mathsf{project}(F', S_p) \equiv \mathsf{project}(F, S_p). && \text{(LP-P)}\end{aligned}$$

The LP relationship is preserved if the link scope S_l is enlarged, and also if the projection scope S_p is made smaller, that is, if $S_l \subseteq S'_l \subseteq S'_p \subseteq S_p$, then $\mathsf{LP}(F, S_l, S_p, F')$ implies $\mathsf{LP}(F, S'_l, S'_p, F')$.

Consider a program that computes for inputs F, S_l, S_p, where F is a propositional formula and S_l, S_p are representations of literal scopes such that $S_l \subseteq S_p$, an output formula F' such that $\mathsf{LP}(F, S_l, S_p, F')$ is satisfied. (The restriction to *propositional* F ensures that such an F' exists.) The program can be applied to solve the following different tasks, distinguished by the values of S_l and S_p:

- *Projection Computation.* If S_l and S_p are identical, then projection computation for $\mathsf{project}(F, S_p)$ can be performed by substituting in F' all literals that are not in S_p with $\top$, as indicated in Sect.2. Thus, the essential work for projection computation is done by computing F'.
- *Compilation to an equivalent formula that is linkless outside a given literal scope.* If S_p is the set of all literals, then Condition LP-P states that F' is equivalent to F. If S_l is the given literal scope, then Condition LP-L states that F' is linkless outside the given scope.
- *Compilation to a fully linkless equivalent formula.* This specializes the task described in the previous item by requiring S_l to be the empty set. As before, S_p is the set of all literals.
- *Deciding satisfiability.* If S_l is the empty set, then F' is fully linkless and its satisfiability can be checked in a linear postprocessing step. Since projection preserves equi-satisfiability, an arbitrary set of literals can be taken as S_p. However, the empty set as S_p is preferable, since it least constrains LP.

3 LP-Tableaux

Tableau Representation. We represent tableaux as terms, constructed like formulas from literals and logic operators, but with two additional structure indicating operators. Tableau manipulation rules can then be presented as term rewriting rules which are very similar to formula rewriting rules and thus straightforwardly exhibit semantic and structural relationships between subformulas associated with subterms of tableaux.

Definition 3 (Expression, Tableau, Leafy Formula).

(i) An *expression* is constructed from literals and logic operators, like a formula, and the two additional binary operators $/\!\!/$ and $/$. The operators $/\!\!/$ and $/$ are written in infix notation, with higher precedence than $\wedge, \vee, \bigwedge, \bigvee$, and lower precedence than the other operators.

(ii) A *tableau* is an expression which has one of the following forms:

1. *Leaf node:* $L /\!\!/ F$, where $L \in \mathsf{LIT} \cup \{\top\}$ and F is a sentence,
2. *Or-node:* $(L/F \wedge \bigvee_{i\in\{1,\ldots,n\}} T_i)$, where $n \geq 1$, $L \in \mathsf{LIT} \cup \{\top\}$, F is a sentence, and for $i \in \{1, \ldots, n\}$ it holds that T_i is a tableau.

(iii) A subexpression occurrence in a tableau is called a *subtableau* of the tableau if the subexpression is a tableau.

(iv) The *leafy formula* of an expression E, in symbols $E^{\clubsuit}$, is the NNF formula obtained from E by replacing all subexpressions of the form L/F with L and those of the form $L/\!\!/F$ with $(L \wedge F)$.

A tableau according to Def. 3.ii can be viewed as representing a tableau in the more conventional sense, an ordered tree: A *leaf node* represents a leaf of the tree. An *or-node* $(L/F \wedge \bigvee_{i\in\{1,\ldots,n\}} T_i)$ represents a non-leaf node, whose i-th child is the root of the tree represented by T_i. With a node $L/\!\!/F$ or $(L/F \wedge ...)$ two labels are associated, *literal label* L and *forward label* F. Literal labels correspond to the usual node labels in tableaux. They are restricted to literals, in contrast to complex formulas, as common for tableaux formats used by efficient automated methods. They can contain universal, but not rigid, variables. The truth value constant $\top$ is allowed as literal label to facilitate a technique discussed in Sect. 4.

If tableau node labels are restricted to literals, the – complex – input formula must be represented in some special way, external to the tableau. *Forward labels* provide a means to avoid such external structures, and, moreover, to incorporate formula simplifications and decompositions that are useful for transformation tasks but can not be straightforwardly expressed as tableau manipulations. The *forward label* of a node is a complex formula. Calculi construct it basically as a copy of the input formula on which simplifications with respect to the nodes on the branch to the node have been performed. The name *forward* should suggest that forward labels represent portions of the input that have not yet been processed (entered into the proper tableau structure), but will so at a future *"forward"* point in time. In implementations, forward labels not necessarily have to be fully materialized. A prototypical example is the DPLL procedure: The input formula, simplified by unit propagation with respect to a branch maintained by the procedure, can be modeled as forward label.

The formula view of a tableau is made explicit with the *leafy formula* operation $^{\clubsuit}$, which removes the structural operators / and $/\!\!/$. Constituents of a leafy formula are all literal labels and the forward labels of leaf nodes. Forward labels F of non-leaf nodes $(L/F \wedge ...)$ are not taken over into the leafy formula. They facilitate destructive tableau operations such as backjumping by recording forward labels of previous leafs. *Leafy formula* is defined for *expression* to be applicable also to tableau subexpressions which themselves are not tableaux.

Abstract Calculi. Our aim is to apply tableau construction methods to compute the LP relation. To this end, we consider tableaux that are constructed by rewrite rules. It should be noted that at this level of abstraction rules are characterized by means of semantic relationships, similar to the abstract formalization of DPLL in [17]. It is implicitly assumed that the rules are applied in cases where these relationships can be efficiently established, usually with well known methods.

We call sets of the considered rewrite rules *LP-Calculi* and the constructed tableaux *LP-tableaux*. Before we come to their definition (Def. 4), we need to introduce two mappings between formulas, which are involved in the rules.

The first is *restriction* of a sentence F by a literal sentence L, in symbols $F|_L$. Of this operation, we require that it satisfies the properties given in Tab. 2. For propositional sentences, restriction can be realized simply by replacing in F all occurrences of L with $\top$ and of $\widetilde{L}$ with $\bot$. A variant for clausal first-order sentences and universal literals is hinted in Sect. 5.

The second auxiliary operation is *scope preserving simplification* of a sentence F with respect to a literal scope S. In symbols it is written as F^*, assuming that the scope parameter is specified in the context. It is assumed to be a simplification, that is, intuitively, an operation that can be performed fast and be applied only a number of times that is polynomial in the size of the formula. The properties required of a scope preserving simplification are also listed in Tab. 2. An example that applies to clausal propositional sentences is the replacement of clauses containing an atom A, where $+A$ and $-A$ are not in S, by their resolvents upon A, in cases where the total number of clauses is reduced.

(R1) $L \wedge F \equiv L \wedge F\|_L$.	(S1) $\mathsf{project}(F^*, S) \equiv \mathsf{project}(F, S)$.
(R2) $\mathcal{L}(F\|_L) \subseteq \mathcal{L}(F)$.	(S2) $\mathcal{L}(F^*) \subseteq \mathcal{L}(F)$.
(R3) $\mathcal{A}(F\|_L) \cap \mathcal{A}(L) = \emptyset$.	

Table 2. Required properties of restriction $F|_L$ and simplification F^*

Definition 4 (LP-Calculus, LP-Tableau). Let F_0 be a sentence, S_l and S_p where $S_l \subseteq S_p$ be literal scopes, $|$ be a restriction function, and * be a simplification function preserving the scope S_p. An *LP-calculus* is a set of tableau rewrite rules which can be parameterized with these items. A tableau constructed by applying a finite number (including zero) of rewrite steps with rules of an LP-calculus to an initial tableau as specified with Init (Tab. 3) is called an *LP-tableau for F_0 obtained with* the LP-calculus. The parameter S_l is referred as *link scope* and S_p as *projection scope*.

Initial State and the Extend Rule. Consider Init and Extend defined in Tab. 3. Init specifies the initial state referenced in the definition of *LP-tableau*. Extend is a tableau construction rule that can be considered as an "abstract rule" since it specifies semantic and structural conditions which, as shown in Sect. 5, are matched by various more specific rules, including well known tableau rules. Extend models the attachment of n successor nodes to a leaf. The *if*-preconditions state that the forward label F of the leaf entails a disjunction of n universal literals, whose literal bases are contained in that of F. For each disjunct L_i, a new node with literal label L_i is attached. Its forward label is the forward label F of the former leaf, restricted by L_i and then simplified, where the projection scope S_p (an implicit parameter) is preserved.

LP-Calculi Compute the LP Relation. That LP-calculi can be used to compute the LP relation is formally stated as Theorem 1. For clarity, we just consider instances of Extend as tableau rules, but the proofs in essence extend also to other rules, as indicated in Sect. 4. Theorem 1 refers to two properties which are defined before the theorem statement:

Init:	$\top /\!\!/ F_0^*$	
Extend:	$L /\!\!/ F \;\longrightarrow\; L/F \wedge \bigvee_{i\in\{1,\ldots,n\}} L_i /\!\!/ F|^*_{L_i}$	if $\begin{cases} F \models \bigvee_{i\in\{1,\ldots,n\}} L_i, \text{ where } n \geq 1 \\ \text{For } i \in \{1,\ldots,n\}: \\ \quad L_i \in \mathsf{LIT} \text{ and } \mathcal{L}(L_i) \subseteq \mathcal{L}(F) \end{cases}$
And-Separate:	$L /\!\!/ \bigwedge_{i\in\{1,\ldots,n\}} F_i \;\longrightarrow\; L/ \bigwedge_{i\in\{1,\ldots,n\}} F_i \wedge \bigwedge_{i\in\{1,\ldots,n\}} \top /\!\!/ F_i^*$	*See Sect. 4* if $\begin{cases} n \geq 2 \\ \text{For } i,j \in \{1,\ldots,n\} \text{ s.th. } i \neq j: \\ \quad \langle F_i, F_j \rangle \text{ is linkless outside } S_l \end{cases}$
True-Up:	$L/F \wedge (K /\!\!/ \top \vee \bigvee_{i\in\{1,\ldots,n\}} T_i) \;\longrightarrow\; L /\!\!/ \top$	*See Sect. 4* if $\begin{cases} n \geq 0 \\ \mathcal{L}(K) \cap S_p = \emptyset \end{cases}$
And-True-Up:	$L/F \wedge \bigwedge_{i\in\{1,\ldots,n\}} \top /\!\!/ \top \;\longrightarrow\; L /\!\!/ \top$	if $n \geq 2$
True-Below-Cut-Up:	$L/F \wedge (K /\!\!/ \top \vee \widetilde{K} /\!\!/ \top) \;\longrightarrow\; L /\!\!/ \top$	if K is ground

Table 3. Initial state and [abstract] rules of LP-calculi

Definition 5 (Unlinking LP-Calculus, Terminal LP-Tableau).
(i) An LP-calculus is called *unlinking* if and only if for all LP-tableaux obtained by the calculus which have a leaf node whose forward label is not linkless outside the link scope S_l it holds that some subtableau can be rewritten with a rule of the LP-calculus.
(ii) An LP-tableau is called *terminal* with respect to an LP-calculus if and only if none of its subtableaux can be rewritten with a rule of the LP-calculus.

Theorem 1 (LP-Calculi Compute the LP Relation). *Let F_0 be a sentence, S_l, S_p be literal scopes such that $S_l \subseteq S_p$, and let CALC be an LP-calculus which is unlinking and whose rules are instances of* Extend. *If T is a terminal LP-tableau for F_0 that is obtained by CALC for link scope S_l and projection scope S_p, then* $\mathsf{LP}(F_0, S_l, S_p, T^{\clubsuit})$.

The theorem holds since the two conditions of LP are satisfied with respect to the given parameters, which is shown with two lemmas below. We first consider Condition LP-L, verified by Lemma 1 which is preceded by an auxiliary definition and proposition.

Definition 6 (Subformula Within a Leaf). Let T be an LP-tableau. A subformula occurrence in $T^{\clubsuit}$ is called *within a leaf with respect to T* if and only if it occurs within a subformula F that has been obtained in the mapping of T to $T^{\clubsuit}$ by replacing a leaf node $L /\!\!/ F$ with F.

Proposition 1. *Let S_l, S_p be literal scopes such that $S_l \subseteq S_p$ and assume that T is an LP-tableau obtained with an LP-calculus whose rules are instances of* Extend, *for link scope S_l and projection scope S_p. If there is an occurrence of a subformula of the form $(G \wedge H)$ in $T^{\clubsuit}$ that is not within a leaf with respect to T, then $\langle G, H \rangle$ is linkless outside S_l.*

By Prop. 1, a subformula $(F_1 \wedge F_2)$ where $\langle F_1, F_2 \rangle$ is not linkless outside S_l can in $T^{\clubsuit}$ only occur *within a leaf.* If T is terminal with respect to an unlinking calculus, this possibility is also excluded, which implies the lemma for LP-L:

Lemma 1 (Terminal LP-Tableaux are Linkless Outside the Link Scope). *Let S_l be a literal scope and let CALC be an LP-calculus which is unlinking and whose rules are instances of* Extend*, for link scope S_l. If T is a terminal LP-tableau that is obtained by CALC, then $T^{\clubsuit}$ is linkless outside S_l.*

We now turn to Condition LP-P, which is verified by Lemma 3. Its proof is based on the following Lemma 2, which states a precondition under which the result of rewriting a subformula G of F with a formula G' such that $\mathsf{project}(G', S) \equiv \mathsf{project}(G, S)$ yields a formula F' such that $\mathsf{project}(F', S) \equiv \mathsf{project}(F, S)$. Following [7], we use the following notation: If T is a term with a subterm occurrence replaced by a *hole*, and U is a term, then $T[U]$ is T with the hole replaced by U. At the same time $T[U]$ indicates that the term T contains an occurrence of the subterm U.

Lemma 2 (Scope Preservation by Unlinked Replacement). *If S is a literal scope and $G, G', F[G]$ are sentences such that (1.)* $\mathsf{project}(G', S) \equiv \mathsf{project}(G, S)$*, (2.) for all subformulas of F of the form $(F_1[G] \wedge F_2)$ or $(F_2 \wedge F_1[G])$ it holds that $\langle G, F_2 \rangle$ is linkless outside S, (3.) for all subformulas of F of the form $(F_1[G] \vee F_2)$ or $(F_2 \vee F_1[G])$ it holds that F_1 and F_2 do not have free variables in common, and (4.) conditions (2.–3.) also hold for G' in place of G, then* $\mathsf{project}(F[G'], S) \equiv \mathsf{project}(F[G], S)$.

Proof (Sketch). The sentence $F[G]$ is equivalent to a sentence $((G \wedge F_1) \vee F_2)$, where F_1 and F_2 are sentences and $\langle G, F_1 \rangle$ is linkless. This sentence can be obtained from $F[G]$ by rewriting subformulas of the form $((H[G] \vee H_1) \wedge H_2)$ – where H, H_1 and H_2 are sentences – with the equivalent $((H[G] \wedge H_2) \vee (H_1 \wedge H_2))$, and in addition some standard equivalences. Similarly, $F[G']$ is equivalent to $((G' \wedge F_1) \vee F_2)$, where $\langle G', F_1 \rangle$ is linkless too. From precondition (1.) follows $\mathsf{project}((G' \wedge F_1) \vee F_2, S) \equiv \mathsf{project}((G \wedge F_1) \vee F_2, S)$ which then implies the conclusion of the lemma. □

It can be shown that a rewriting step with Extend matches the preconditions of Lemma 2 with respect to leafy formulas: Precondition (1.) can be verified from the definition of Extend, (2.) follows as a corollary from Prop. 1 since $S_l \subseteq S_p$, and (3.) follows from the definition of tableau. By induction on the tableau structure, as constructed from Init by rewriting with Extend, the lemma for LP-P can be proven:

Lemma 3 (LP-Calculi Preserve the Projection Scope). *Let F_0 be a sentence, S_p be a literal scope, and let CALC be an LP-calculus whose rules are instances of* Extend*. If T is an LP-tableau for F_0 that is obtained by CALC for projection scope S_p, then* $\mathsf{project}(T^{\clubsuit}, S_p) \equiv \mathsf{project}(F_0, S_p)$.

4 LP-Tableau Refinements

And-Nodes. The definition of *tableau* (Def. 3.ii) can be extended by the following clause:

3. *And-node:* $(L/F \wedge \bigwedge_{i\in\{1,\ldots,n\}} T_i)$, where $n \geq 2$, $L \in \mathsf{LIT} \cup \{\top\}$, F is a sentence, and for $i \in \{1,\ldots,n\}$ it holds that T_i is a tableau.

And-nodes are like or-nodes, but with $\bigwedge$ in place of $\bigvee$ and n restricted by $n \geq 2$. By the latter restriction, tableaux with a single child are unambiguously classified as or-nodes. And-nodes are constructed with the And-Separate rule (Tab. 3). In this rule, the $\bigwedge$ operator on the left side is understood modulo associativity and commutativity. The rationale for Theorem 1 given in Sect. 3 straightforwardly carries over to LP-tableaux with and-nodes and the And-Separate rule. And-Separate constructs nodes with literal label $\top$ to preserve the structural uniformity of the tableau.

And-nodes allow to model a decomposition technique that has been introduced in DPLL adaptions for model counting [2] and also applied to knowledge compilation into DNNF[1] [11]. There, the component formulas $F_1,\ldots,F_n$ are required to have pairwise disjoint atom bases. After processing them independently, the numbers of their models are multiplied, or their compiled equivalents are conjoined, respectively. For the purpose of compilation to formulas that are linkless outside link scope S_l, the linklessness condition on $F_1,\ldots,F_n$ in And-Separate, which is weaker than requiring disjoint atom bases, is sufficient. Forward labels are quite convenient to formally handle the integration of such decomposition techniques into tableau methods.

Tableau Level Simplifications. Simplification operations * map forward labels, or subformulas of them, to further forward labels, but they do not modify the tableau structure. Here we consider rules which modify the tableau structure, but whose effect can also be viewed as a formula simplification – of the tableau's leafy formula. Following [10], we call them *at the tableau level*.

True-Up (Tab. 3) is such a rule. The disjunction operators on its left side are understood modulo associativity and commutativity. It does in general not preserve equivalence of the leafy formula, but equivalence with respect to the projection scope S_p. As can be easily verified, if T, T' are LP-tableaux matching the two sides of an instance of True-Up, then $\mathsf{project}(T'^{\clubsuit}, S_p) \equiv \mathsf{project}(T^{\clubsuit}, S_p)$. And-True-Up (Tab. 3) supplements True-Up by applying to tableaux with a particular form that can arise when And-Separate is involved. The given rationale of Theorem 1 straightforwardly carries over to LP-tableaux whose rules include True-Up and And-True-Up.

In the case that L is – like K – not in the projection scope S_p, a rewriting step with True-Up can effect that True-Up becomes applicable again, with the old L in the role of the new K. If S_p is empty, in this way, when a model has been found – indicated by a leaf node with forward label $\top$ – the True-Up rule can be

[1] *Decomposable negation normal form (DNNF)* [4] is a sublanguage of fully linkless propositional formulas: An atom is not allowed to occur in both conjuncts of conjunctive subformulas, no matter whether in complementary or identical literals.

repeatedly applied until the whole tableau consists of just a single node $\top /\!\!/ \top$. The method then terminates "inherently" after finding the first model instead of having to be stopped extraneously from computing alternative models.

Tableau level simplifications have also been considered in [10], but only simplifications that preserve equivalence. Rule [10, Fig 1., rightmost] seems useful for LP-calculi, especially since it can effect that True-Up becomes applicable. Its adaption is shown in Tab. 3 as True-Below-Cut-Up.

Backjumping. Dependency directed backtracking belongs to the important techniques of efficient automated tableau systems. Backjumping, as abstractly described for DPLL in [17], is a variant of it, which can be coarsely modeled for LP-tableaux as a two-step process: An application of Extend, which is preceded by an application of the following auxiliary rule Truncate-for-Backjump:

$$L/F \wedge \bigvee_{i\in\{1,\ldots,n\}} T_i \;\longrightarrow\; L/\!\!/F \qquad\qquad \text{if } n \geq 2$$

Backjumping is applicable to a subtableau matching the left side of Truncate-for-Backjump when it has been established (by means of data structures not represented in LP-tableaux) that $F \models K$ for a ground literal $K \in \mathcal{L}(F)$. The subtableau $L/\!\!/F$ resulting from Truncate-for-Backjump is then rewritten by Extend to $(L/F \wedge K/\!\!/F|^*_K)$. The given rationale of Theorem 1 can be extended to apply also for LP-tableaux with Truncate-for Backjump.

For theorem proving or model computation tasks, backjumping is commonly applied with respect to a branch in which siblings to the right represent possibilities to explore in future computation and siblings to the left are either not present or can be ignored, since they correspond to parts of the problem that already have been solved. For transformation tasks this is different. Although siblings to the left might correspond to already transformed parts of the input formula, a backjumping step can effect that these are deleted and an improved transformation, in which more unit lemmas are available below some nodes, is computed. Examples for this can be found in [20], along with a more fine grained modeling of backjumping.

The termination arguments for DPLL with backjumping (e.g. [17]) can be generalized for propositional transformation tasks. An ordering relation $>_t$ on tableaux can be defined such that $T >_t T'$ whenever T' is obtained from T by a rewrite step with one of the rules in Tab. 3 or by backjumping, as a Truncate-for-Backjump step immediately followed by Extend. The relation $>_t$ can be defined as follows: With a tableau T a sequence of pairs $\langle l_0, r_0\rangle \ldots \langle l_{n-1}, r_{n-1}\rangle$ is associated, where n is the size of its longest branch and for $i \in \{0,\ldots,n-1\}$ the left component l_i is defined as the number of leaf nodes in T at depth i which do not have $\top$ as forward label, and the right component r_i is defined as the number of nodes in T at depth $i+1$. (The root is understood as having depth 0.) Now, $T_1 >_t T_2$ if and only if the sequence associated in this way with T_1 is lexicographically greater than that associated with T_2, where the elements of the sequences are compared lexicographically, and the components of these elements by numerical value.

5 Instantiations of the LP-Tableau Framework

DPLL Adaptions. Rules Split and Unit (Tab. 4) are two instances of Extend, familiar from the DPLL procedure for propositional sentences. Split ensures that an LP-calculus has the *unlinking* property, and thus can be used to compute LP according to Theorem 1. In order to just achieve the *unlinking* property, the legitimacy of K in the precondition of Split can be further restricted by requiring $\{K, \widetilde{K}\} \not\subseteq S_l$ and that F has a subformula $(F_1[K] \wedge F_2[\widetilde{K}])$. On the other hand, performing Split on literals K that do not meet these criteria might be heuristically beneficial in cases where it enables effective simplifications by the * function. The Unit rule is an option. LP-calculi allow unit propagation also to be performed just within computation of forward labels by the * function.

In [5, 11] an adaption of DPLL for knowledge compilation of propositional CNF into DNNF is presented, by means of pseudocode, where the compilation result is understood as trace of a procedure run. Apart from caching techniques, this method can in essence be modeled more abstractly by an LP-calculus with Split, Unit, And-Separate (for the DNNF target format, the precondition of And-Separate has to be strengthened: The F_i must have pairwise disjoint atom bases) and backjumping, supplemented by a rule application strategy that corresponds to depth-first tree construction.

DPLL and other practically successful tableau procedures operate space efficiently by keeping in memory at any point of time just a piece of the tableau under construction. While this is clearly beneficial for theorem proving tasks where a yes/no answer is searched, it can also be utilized for formula transformation tasks by methods that output pieces of their overall output as soon as they are computed. Such a method has been described for an LP-calculus (in a different notational framework) without And-Separate, but with backjumping [20].

Propositional Clausal Tableaux. In [16] it has been observed that a fully developed regular clausal tableau for a given propositional CNF formula represents an equivalent to the formula that is in DNNF; thus any method for computing such a tableau can be considered as a DNNF compiler. Such a method can be modeled by an LP-calculus with an instance of Extend, the rule ClausalExpansion (Tab. 4), and a simplification * that just propagates truth value constants

Split:		
$L/\!\!/F$	$\longrightarrow$ $L/F \wedge (K/\!\!/F\|^*_K \vee \widetilde{K}/\!\!/F\|^*_{\widetilde{K}})$	if $K, \widetilde{K} \in \mathcal{L}(F)$
Unit:		
$L/\!\!/F$	$\longrightarrow$ $L/F \wedge K/\!\!/F\|_{K}*$	if $\begin{cases} \mathcal{L}(K) \subseteq \mathcal{L}(F) \\ F \models K \end{cases}$
ClausalExpansion *(for propositional CNF)*:		
$L/\!\!/F$	$\longrightarrow$ $L/F \wedge \bigvee_{i\in\{1,\ldots,n\}} L_i/\!\!/F\|^*_{L_i}$	if $\bigvee_{i\in\{1,\ldots,n\}} L_i$ is a clause in F, where $n \geq 1$
HyperTableauExtension *(for CNF)*:		
$L/\!\!/F$	$\longrightarrow$ $L/F \wedge \bigvee_{i\in\{1,\ldots,n\}} +A_i\sigma/\!\!/F\|^*_{+A_i\sigma}$	if $\begin{cases} \bigvee_{i\in\{1,\ldots,n\}} +A_i \text{ is a clause in } F, \text{ where } n \geq 1 \\ \bigvee_{i\in\{1,\ldots,n\}} +A_i\sigma \text{ is pure [1]} \end{cases}$

Table 4. Instances of the Extend abstract rule

(removing clauses containing $\top$, removing $\bot$ from clauses, returning $\top$ for the empty clause set, and $\bot$ for a clause set containing the empty clause). In the LP-calculus framework, a clause attached by ClausalExpansion stems not directly from the input formula, but from the simplified forward label of the node to which it is attached. In this way, the violation of regularity and the construction of branches with complementary literals is automatically prevented. In a terminal tableau, each leaf has a truth value constant as forward label.

In [16] also a second method for compilation of propositional NNF formulas into DNNF has been proposed, but not related to the clausal tableau construction: Rewriting of arbitrary subformulas with the Shannon expansion. As shown in [20], this method can be simulated by LP-calculi with Split and And-Separate.

Incorporating Projection Computation into Knowledge Compilation. Projection to an application relevant subvocabulary is a means to compensate somewhat for the size blow-up inherent in knowledge compilation. In the literature, so far projection computation has mainly been considered as an a-posteriori operation that can be performed with low cost on results of compilation [4, 6, 16] (An exception is the DPLL-based clausal compiler described in [14] which incorporates Boolean quantifier elimination). But projection computation can in part be incorporated into knowledge compilation, effecting potentially drastic efficiency improvements of the compilation process, since superpolynomial size reductions by projection become effective already in intermediate compilation stages [20, Theorem 7]. With LP-calculi this is realized by the projection scope parameter, which is passed into the interior of the calculus: The simplification function * can involve operations that preserve equivalence just with respect to the projection scope, and is applied to intermediate subformulas. The True-Up rule performs special cases of projection computation in intermediate stages as simplifications at the tableau level.

Hyper Tableaux. The hyper tableau calculus [1] is typically used in applications for first-order model computation where the intended semantics of a formula is the set of its minimal Herbrand models. Models computed by the hyper tableau calculus per se are not minimal. Applications accept non-minimal outputs as harmless redundancies, or use extra means to enforce minimality [3]. Hyper tableaux add two new aspects to the previously considered LP-calculi instantiations: first-order clauses and minimization.

The rule HyperTableauExtension (Tab. 4) is an instance of Extend that models hyper tableau construction. Its second *if*-precondition states that σ is a substitution under which the literals $+A_i$ have pairwise disjoint sets of variables. In the rule presentation quantifiers have been omitted: Free variables are assumed to be universally quantified, where the variable scope does not extend a clause. The literals $+A_i\sigma$ are understood as universal literals.

A restriction function $F|_L$ for a first-order CNF F and universal literal L (without multiple occurrences of the same variable) can be realized for example by applying diff-expansion [18] to compute a CNF which has the same Herbrand expansion as F (with respect to the function symbols in the fixed signature Σ) and does not contain a literal whose atom is unifiable with – but no instance of – the atom of L. In the formula resulting from diff-expansion, instances of L are

then replaced by $\top$ and instances of $\widetilde{L}$ by $\bot$. For example, if the function symbols in Σ are the constant a and the unary function symbol f, then $(\forall x(-\mathsf{p}(x) \vee +\mathsf{q}(x)) \wedge \forall x(+\mathsf{p}(x) \vee +\mathsf{r}(x)))|_{+\mathsf{p}(\mathsf{a})} = (\mathsf{q}(\mathsf{a}) \wedge \forall x(-\mathsf{p}(\mathsf{f}(x)) \vee +\mathsf{q}(\mathsf{f}(x))) \wedge \forall x(+\mathsf{p}(\mathsf{f}(x)) \vee +\mathsf{r}(\mathsf{f}(x))))$. As a first approximation, the simplification * can be just equivalence preserving truth value propagation, as described above for clausal tableaux.

We now consider hyper tableau construction in combination with projection. Define min as logic operator such that the models of $\mathsf{min}(F)$ are exactly the minimal models of F:

$$\langle I, \beta\rangle \models \mathsf{min}(F) \text{ iff}_{\text{def}} \ \langle I, \beta\rangle \models F \text{ and there does not exist a structure } J \text{ such that } \langle J, \beta\rangle \models F \text{ and } J \cap \mathsf{POS} \subset I \cap \mathsf{POS}. \quad \text{(MIN)}$$

For expressions T, define $T^{\spadesuit}$ as the formula which is defined like $T^{\clubsuit}$, except that leaf nodes $L /\!\!/ F$ are replaced just by L instead of $(L \wedge F)$. It then holds in general that $T^{\clubsuit} \models T^{\spadesuit}$. For an LP-tableau T obtained with an LP-calculus with $\mathsf{HyperTableauExtension}$ as the only rule, $T^{\spadesuit}$ contains only positive literals and if T is terminal then $\mathsf{min}(T^{\spadesuit}) \equiv \mathsf{min}(T^{\clubsuit})$. Since $T^{\spadesuit}$ contains only positive literals it is fully linkless and $\mathsf{project}(T^{\spadesuit}, S_p)$ can be computed linearly by substituting literals of which no instance is in S_p with $\top$, as shown in Sect. 2 (assuming further that for each literal in $T^{\spadesuit}$ either all or no instances are in S_p).

Theorem 2, which follows, then justifies a method to compute $\mathsf{project}(F_0, S_p)$ where S_p contains only positive literals: Compute a terminal tableau T with $\mathsf{HyperTableauExtension}$ for F_0 and projection scope S_p and apply the linear projection computation by substitution with $\top$ to $T^{\spadesuit}$.

Theorem 2 also justifies an extension to standard hypertableau methods which is useful for projection computation and potentially also for model computation in cases where only model fragments from a subvocabulary are relevant to the application: The use of simplifications * which just preserve equivalence with respect to the projection scope. As a simple example consider the CNF sentence $F_0 = (+\mathsf{p} \wedge (-\mathsf{p} \vee +\mathsf{q} \vee +\mathsf{r}) \wedge (-\mathsf{q} \vee +\mathsf{s}) \wedge (-\mathsf{r} \vee +\mathsf{s}))$ and projection scope $S_p = \{+\mathsf{p}, +\mathsf{s}\}$. A value for F_0^* could then be $(+\mathsf{p} \wedge (-\mathsf{p} \vee +\mathsf{s}))$, which leads with one $\mathsf{HyperTableauExtension}$ step to the terminal tableau $(+\mathsf{p}/F_0^* \wedge +\mathsf{s} /\!\!/ \top)$, without the need to branch for $+\mathsf{q}$ and $+\mathsf{r}$.

Theorem 2 (Projection and Hyper Tableaux). *Let F_0 be a sentence in CNF, $S_p \subseteq \mathsf{POS}$ be a literal scope and let CALC be an LP-calculus with* $\mathsf{HyperTableauExtension}$ *as its only rule. If T is a terminal LP-tableau for F_0 that is obtained by CALC for projection scope S_p, then $\mathsf{project}(T^{\spadesuit}, S_p) \equiv \mathsf{project}(F_0, S_p)$.*

Proof (Sketch). By Lemma 3 it holds that $\mathsf{project}(F_0, S_p) \equiv \mathsf{project}(T^{\clubsuit}, S_p)$. The theorem is then implied by $\mathsf{project}(T^{\clubsuit}, S_p) \equiv \mathsf{project}(T^{\spadesuit}, S_p)$, which can be shown as follows: The left to right direction is implied by $T^{\clubsuit} \models T^{\spadesuit}$ which holds in general. It remains to show the right-to-left direction. Since $T^{\spadesuit}$ contains no existential quantifiers, it satisfies the following condition: Each model of $T^{\spadesuit}$ is an extension (i.e. superset w.r.t. positive literals) of a minimal model of $T^{\spadesuit}$. From this condition and the fact that $S_p \subseteq \mathsf{POS}$ it can be derived that $\mathsf{project}(T^{\spadesuit}, S_p) \models \mathsf{project}(\mathsf{min}(T^{\spadesuit}), S_p)$. As mentioned above, since T is terminal it holds that $\mathsf{min}(T^{\spadesuit}) \equiv \mathsf{min}(T^{\clubsuit})$. It then follows that $\mathsf{project}(T^{\spadesuit}, S_p) \models \mathsf{project}(\mathsf{min}(T^{\clubsuit}), S_p)$, which implies $\mathsf{project}(T^{\spadesuit}, S_p) \models \mathsf{project}(T^{\clubsuit}, S_p)$. □

6 Conclusion

We presented an approach for applying tableau methods to projection computation, a generalization of second-order quantifier elimination. We developed a formal framework that extends, subsumes and relates a variety of methods and observations that so far have been formulated dispersed and in more ad-hoc ways. We have discussed some subtle issues exposed by the framework, such as the integration of formula simplifications that perform projection computation into knowledge compilation, and projection in combination with minimal model computation. Since the framework can be used to model techniques of efficient automated systems, it provides a basis for the specification of implementations.

References

1. P. Baumgartner, U. Furbach, and I. Niemelä. Hyper tableaux. In *JELIA'96*, volume 1126 of *LNAI*, pages 1–17. Springer, 1996.
2. R. J. Bayardo and J. D. Pehoushek. Counting models using connected components. In *AAAI-2000*, pages 157–162, 2000.
3. F. Bry and A. H. Yahya. Positive unit hyperresolution tableaux and their application to minimal model generation. *J. Autom. Reason.*, 25(1):35–82, 2000.
4. A. Darwiche. Decomposable negation normal form. *JACM*, 48(4):608–647, 2001.
5. A. Darwiche. New advances in compiling CNF to decomposable negation normal form. In *ECAI 2004*, pages 328–332, 2004.
6. A. Darwiche and P. Marquis. A knowledge compilation map. *JAIR*, 17:229–264, 2002.
7. N. Dershowitz and D. A. Plaisted. Rewriting. In *Handbook of Automated Reasoning*, volume I, chapter 1, pages 537–610. Elsevier Science, 2001.
8. H.-D. Ebbinghaus, J. Flum, and W. Thomas. *Einführung in die mathematische Logik*. Spektrum Akademischer Verlag, Heidelberg, 4th edition, 1996.
9. D. M. Gabbay, R. A. Schmidt, and A. Szałas. *Second-Order Quantifier Elimination: Foundations, Computational Aspects and Applications*. CollegePublications, 2008.
10. R. Hähnle, N. V. Murray, and E. Rosenthal. Normal forms for knowledge compilation. In *ISMIS 2005*, pages 304–313, 2005.
11. J. Huang and A. Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *IJCAI-05*, pages 156–162, 2005.
12. J. Kohlas, R. Haenni, and S. Moral. Propositional information systems. *J. Logic and Comp.*, 9(5):651–681, 1999.
13. J. Lang, P. Liberatore, and P. Marquis. Propositional independence – formula-variable independence and forgetting. *JAIR*, 18:391–443, 2003.
14. K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *CAV 2002*, pages 250–264, 2002.
15. N. V. Murray and E. Rosenthal. Dissolution: Making paths vanish. *JACM*, 40(3):504–535, 1993.
16. N. V. Murray and E. Rosenthal. Tableaux, path dissolution and decomposable negation normal form for knowledge compilation. In *TABLEAUX 2003*, pages 165–180, 2003.
17. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: from an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *JACM*, 53(6):937–977, Nov. 2006.
18. C. Wernhard. Semantic knowledge partitioning. In *JELIA 04*, pages 552–564, 2004.
19. C. Wernhard. Literal projection for first-order logic. In *JELIA 08*, pages 389–402, 2008.
20. C. Wernhard. *Automated Deduction for Projection Elimination*. Number 324 in Dissertationen zur Künstlichen Intelligenz (DISKI). AKA/IOS Press, 2009.