The Model Evolution Calculus as a First-Order DPLL Method

Peter Baumgartner [a]

[a] *NICTA*
Canberra, ACT, Australia

Cesare Tinelli [b,1]

[b] *Department of Computer Science*
The University of Iowa
Iowa City, IA, USA

Abstract

The DPLL procedure is the basis of some of the most successful propositional satisfiability solvers to date. Although originally devised as a proof-procedure for first-order logic, it has been used almost exclusively for propositional logic so far because of its highly inefficient treatment of quantifiers, based on instantiation into ground formulas. The FDPLL calculus by Baumgartner was the first successful attempt to lift the procedure to the first-order level without resorting to ground instantiations. FDPLL lifts to the first-order case the core of the DPLL procedure, the splitting rule, but ignores other aspects of the procedure that, although not necessary for completeness, are crucial for its effectiveness in practice.

In this paper, we present a new calculus loosely based on FDPLL that lifts these aspects as well. In addition to being a more faithful litfing of the DPLL procedure, the new calculus contains a more systematic treatment of *universal literals*, which are crucial to achieve efficiency in practice. The new calculus has been implemented successfully in the *Darwin* system, described elsewhere. The main results of this paper are theoretical, showing the soundness and completeness of the new calculus. In addition, the paper provides a high-level description of a proof procedure for the calculus, as well as a comparison with other calculi.

Key words: DPLL procedure, first-order logic, sequent calculi, model generation

Email addresses: Peter.Baumgartner@nicta.com.au (Peter Baumgartner), tinelli@cs.uiowa.edu (Cesare Tinelli).

[1] Partially supported by Grant No. 237422 from the National Science Foundation.

1 Introduction

In propositional satisfiability the DPLL procedure, named after its authors: Davis, Putnam, Logemann, and Loveland (Davis and Putnam, 1960; Davis et al., 1962), is the dominant method for building (complete) SAT solvers. Its popularity is due to its simplicity, its polynomial space requirements, and the fact that, as a search procedure, it is amenable to powerful but also relatively inexpensive heuristics for reducing the search space. Thanks to these heuristics and to very careful engineering, the best SAT solvers today can successfully attack real-world problems with hundreds of thousands of variables and of clauses (Moskewicz et al., 2001; Goldberg and Novikov, 2002). These solvers are so powerful that many developers of automated reasoning-based tools are starting to use them as back-ends to solve *first-order* satisfiability problems, albeit often in an incomplete way, by means of ingenious domain specific translations into propositional logic (Joshi et al., 2001; Jackson, 2000; Strichman et al., 2002).

Interestingly, the DPLL procedure was actually devised in origin as a proof-procedure for first-order logic. Its treatment of quantifiers is highly inefficient, however, because it is based on enumerating all possible ground instances of an input formula's clause form, and checking the propositional satisfiability of each of these ground instances one at a time. Because of its primitive treatment of quantifiers the DPLL procedure, which predates Robinson's resolution calculus by a few years, was quickly overshadowed by resolution as the method of choice for automated first-order reasoning, and its use has been confined to propositional satisfiability ever since. [2]

Given the great success of DPLL-based SAT solvers today, two natural research questions arise. One is whether the DPLL procedure can be properly lifted to the first-order level—in the sense first-order resolution lifts propositional resolution, say. The other is whether those powerful search heuristics that make DPLL so effective at the propositional level can be successfully adapted to the first-order case. We answer the first of these two questions affirmatively in this paper, providing a complete lifting of the DPLL procedure to first-order clausal logic by means of a new sequent-style calculus, the *Model Evolution* calculus, or $\mathcal{ME}$ for short. The $\mathcal{ME}$ calculus can be used to answer the second question affirmatively as well, as we show in a companion paper (Baumgartner et al., 2006a) describing a recent implementation of the calculus.

The FDPLL calculus by Baumgartner (Baumgartner, 2000) was the first successful attempt to lift the DPLL procedure to the first-order level without

[2] But see Section 6 for a brief overview of first-order reasoning systems that use the procedure to help them focus their search.

resorting to ground instantiations. FDPLL lifts to the first-order case the core of the DPLL procedure, the splitting rule, but ignores another major aspect, *unit propagation* (Zhang and Stickel, 1996), that although not necessary for its completeness is absolutely crucial to its effectiveness in practice. The calculus described in this paper lifts this aspect as well. While the $\mathcal{ME}$ calculus borrows many fundamental ideas from FDPLL and generalizes it, it is not an extension of FDPLL proper but of DPLL (Tinelli, 2002), a simple propositional calculus modeling the main features of the DPLL procedure. As we will see, the Model Evolution calculus is a direct lifting of DPLL in the sense that it consists of appropriate first-order versions of DPLL's rules, plus one additional rule specific to the first-order case.

A very useful feature of the DPLL procedure—and of most propositional proof procedures for that matter—is that it is able to provide a (Herbrand) model of the input formula whenever that formula is satisfiable. The procedure, and by extension the DPLL calculus, generates this model incrementally during a derivation. The Model Evolution calculus can be seen as lifting this model generation process to the first-order level. We could say that the purpose of the Model Evolution calculus is, like the DPLL calculus, to construct a Herbrand model of a given set Φ of clauses, if any such model exists.

At each step of a derivation the calculus maintains a *context* Λ, that is, a finite set of (possibly non-ground) literals. The context Λ is a finite—and compact—representation of a Herbrand interpretation I_Λ serving as a candidate model for Φ. The induced interpretation I_Λ might not be a model of Φ because it might not satisfy some clauses in Φ. The purpose of the main rules of the calculus is to detect this situation and either *repair* I_Λ, by modifying Λ, so that it becomes a model of Φ, or recognize that I_Λ is unrepairable and fail. In addition to these rules, the calculus contains a number of simplification rules whose purpose is, again like in DPLL, to simplify the clause set and, as a consequence, to speed up the computation.

We call our calculus *Model Evolution* calculus because it starts with a default candidate model, one that satisfies no positive literals, and "evolves it" as needed until it becomes an actual model of the input clause set Φ, or until it is clear that Φ has no models at all. Note that the DPLL calculus can be seen as doing exactly the same thing, but for ground formulas only. The Model Evolution calculus simply extends this behavior to non-ground formulas as well. An important by-product of this model evolution process is that every terminating derivation of a satisfiable clause set Φ produces a context whose induced interpretation is indeed a model of Φ. This makes the calculus well suited for all applications in which it is important to also provide counter-examples of invalid statements, as opposed to simply proving their invalidity.

The Model Evolution calculus is refutationally sound and complete: an input

clause set Φ is unsatisfiable if and only if the calculus (finitely) fails to find a model for Φ. The calculus is obviously non-terminating for arbitrary, satisfiable input sets. With some of these clause sets, the calculus might go on repairing their candidate model forever, without ever turning it into an actual model. The calculus is however terminating for the class of ground clauses (of course), and for the class of clauses resulting from the translation of conjunctions of Bernays-Schönfinkel formulas into clause form.[3] The termination for ground clause sets is a direct consequence of the fact that with such inputs the Model Evolution calculus reduces to the DPLL calculus, as we will show. The reasons for termination for Bernays-Schönfinkel formulas are similar to those given in (Baumgartner, 2000) for FDPLL.

As mentioned, the Model Evolution calculus is already a significant improvement over FDPLL because it is a more faithful litfing of the DPLL procedure, having additional rules for simplifying the current clause set and the current context. Another advantage over FPDLL is that it contains a more systematic and general treatment of *universal literals*, one of FDPLL's optimizations. As we will see, adding literals with *universal variables* to a context imposes stronger restrictions on future modification of that context. This has the consequence of greatly reducing the non-determinism in the calculus, and hence the potential of leading to much faster implementations.

The paper is organized as follows. After some formal preliminaries, given below, we briefly describe in Section 2 the DPLL procedure, and define the DPLL calculus, a declarative version of the procedure. We then define and discuss the Model Evolution calculus, in Section 3, showing how it extends DPLL. We prove the calculus' correctness in Section 4. In Section 5 we briefly describe a proof procedure for the calculus as implemented in the *Darwin* theorem prover (Baumgartner et al., 2006a). Then we discuss in Section 6 how the calculus compares to other calculi in related work. We conclude the paper in Section 7 with directions for further research. The more technical results needed in Section 4 are proved in detail in the appendix.

1.1 Formal Preliminaries

In this paper, we use two disjoint, infinite sets of variables: a set X of *universal* variables, which we will refer to just as variables, and another set V, which we will always refer to as *parameters*. The reason for having two types of variables will be explained later. We will use, possibly with subscripts, u, v to denote elements of V, x, y to denote elements of X, and w to denote elements of $V \cup X$. We fix a signature Σ throughout the paper. We denote by Σ^{sko} the expansion of Σ obtained by adding to Σ an infinite number of (Skolem)

[3] Such clauses contain no function symbols, but no other restrictions apply.

constants not already in Σ. By Σ-term (Σ^{sko}-term) we mean a term of signature Σ (Σ^{sko}) over $V \cup X$. In the following, we will simply say "term" to mean a Σ^{sko}-term. If t is a term we denote by $\mathcal{V}ar(t)$ the set of t's variables and by $\mathcal{P}ar(t)$ the set of t's parameters. A term t is *ground* iff $\mathcal{V}ar(t) = \mathcal{P}ar(t) = \emptyset$. Two terms are *variable-disjoint (parameter-disjoint)* iff they have no variables (parameters) in common. They are *disjoin*t iff they are both variable- and parameter-disjoint. We extend the above notation and terminology to literals and clauses in the obvious way.

We adopt the usual notion of substitution over Σ^{sko}-expressions or sets thereof. We also use the standard notion of unifier and of most general unifier. We will denote by $\{w_1 \mapsto t_1, \ \ldots, \ w_n \mapsto t_n\}$ the substitution σ such that $w_i\sigma = t_i$ for all $i = 1, \ldots, n$ and $w\sigma = w$ for all $w \in X \cup V \setminus \{w_1, \ldots, w_n\}$. Also, we will denote by $\mathcal{D}om(\sigma)$ the set $\{w_1, \ldots, w_n\}$ and by $\mathcal{R}an(\sigma)$ the set $\{w_1\sigma, \ldots, w_n\sigma\}$.

If σ is a substitution and W a subset of $X \cup V$, the restriction of σ to W, denoted by $\sigma_{|W}$ is the substitution that maps every $w \in W$ to $w\sigma$ and every $w \in (V \cup X) \setminus W$ to itself. A substitution ρ is a *renaming on* $W \subseteq (V \cup X)$ iff $\rho_{|W}$ is a bijection of W onto W. For instance $\rho := \{x \mapsto u, v \mapsto u, u \mapsto v\}$ is a renaming on V. Note however that ρ is not a renaming on $V \cup X$ as it maps both x and v to u. We call a substitution simply a *renaming* if it is a renaming on $V \cup X$. We call a substitution σ *parameter-preserving*, or *p-preserving* for short, if it is a renaming on V. Similarly, we call σ *variable-preserving* if it is a renaming on X. Note that a renaming is parameter-preserving iff it is variable-preserving. For example, the renaming $\{x \mapsto y, y \mapsto x, u \mapsto v, v \mapsto u\}$ is both variable- and parameter-preserving, wheres the renaming $\{x \mapsto v, v \mapsto x\}$ is neither variable-preserving nor parameter-preserving.

If s and t are two terms, we say that s *is more general than* t, and write $s \gtrsim t$, iff there is a substitution σ such that $s\sigma = t$[4]. We say that s is *a variant of* t, and write $s \approx t$, iff $s \gtrsim t$ and $t \gtrsim s$ or, equivalently, iff there is a renaming ρ such that $s\rho = t$. We write $s \gtrnsim t$ if $s \gtrsim t$ but $s \not\approx t$. We say that s *is parameter-preserving more general than* t, and write $s \geq t$, iff there is a parameter-preserving substitution σ such that $s\sigma = t$. When $s \geq t$ we will also say that t is *a p-instance of* s. Since the empty substitution is parameter-preserving and the composition of two parameter-preserving substitutions is also parameter preserving, it is immediate that the relation $\geq$ is, like $\gtrsim$, both reflexive and transitive. We say that s is *a parameter-preserving variant, or p-variant, of* t, and write $s \simeq t$, iff $s \geq t$ and $t \geq s$; equivalently, iff there is a parameter-preserving renaming ρ such that $s\rho = t$. We write $s \gtrneq t$ if $s \geq t$ but $s \not\simeq t$. Note that both $\simeq$ and $\approx$ are equivalence relations.

All of the above about substitutions is extended from terms to *literals*, that is,

[4] The unification literature would write $s \lesssim t$ in the case above.

atomic formulas or negated atomic formulas, in the obvious way. We denote literals in general by the letters K, L. We denote by $\overline{L}$ the complement of a literal L. As usual, a *clause* is a disjunction $L_1 \vee \cdots \vee L_n$ of zero or more literals. We denote clauses by the letters C and D and the empty clause by $\Box$. We write $L \vee C$ to denote a clause obtained as the disjunction of a (possibly empty) clause C and a literal L. When convenient, we will treat a clauses as the set of its literals.

A *Skolemizing substitution* is a substitution θ with $\mathcal{D}om(\theta) \subseteq X$ that replaces each variable in $\mathcal{D}om(\theta)$ by a fresh Skolem constant and every remaining element of $X \cup V$ by itself. A *Skolemizing substitution for a literal L (clause C)* is a Skolemizing substitution θ with $\mathcal{D}om(\theta) = \mathcal{V}ar(L)$ ($\mathcal{D}om(\theta) = \mathcal{V}ar(C)$). We write L^{sko} (C^{sko}) to denote the result of applying to L (C) some Skolemizing substitution for L (C).

A *(Herbrand) interpretation* I over some signature Ω is a set of ground Ω-literals that contains either L or $\overline{L}$, but not both, for every ground Ω-literal L. Satisfiability of Ω-literals and Ω-clauses in I is defined as follows. The interpretation I satisfies (or is a model of) a ground literal L, written $I \models L$, iff $L \in I$; I satisfies a ground clause C, iff $I \models L$ for some L in C; I satisfies a clause C, iff $I \models C'$ for all ground instances C' of C; I satisfies a clause set Φ, iff $I \models C$ for all $C \in \Phi$ in C. The interpretation I *falsifies* a literal L (a clause C) if it does not satisfy L (C). Sometimes we will also say that a clause C is valid in I to mean that $I \models C$.

2 The DPLL Calculus

The DPLL procedure can be used to decide the satisfiability of ground (or propositional) formulas in conjunctive normal form or, more precisely, the satisfiability of finite sets of ground clauses. The three essential operations of the procedure are (i) unit resolution with backward subsumption, (ii) unit subsumption, and (iii) recursive reduction to smaller problems. The procedure can be roughly described as follows. [5]

> Given an input clause set Φ, whose satisfiability is to be checked, apply *unit propagation* to it, that is, close Φ under unit resolution with backward subsumption, and eliminate in the process (a) all non-unit clauses subsumed by a unit clause in the set and (b) all unit clauses whose atom occurs only once in the set. If the closure Φ^* of Φ contains the empty clause, then fail. If Φ^* is the empty set, then succeed. Otherwise, choose an arbitrary

[5] See the original papers (Davis and Putnam, 1960; Davis et al., 1962), among others, for a more complete description.

$$\textsf{Split}\quad \frac{\Lambda \vdash \Phi,\ L \vee C}{\Lambda, L \vdash \Phi,\ L \vee C \qquad \Lambda, \overline{L} \vdash \Phi,\ L \vee C}\quad \text{if}\ \begin{cases} C \neq \Box, \\ L \notin \Lambda, \\ \overline{L} \notin \Lambda \end{cases}$$

$$\textsf{Assert}\quad \frac{\Lambda \vdash \Phi,\ L}{\Lambda, L \vdash \Phi,\ L}\quad \text{if}\ \begin{cases} L \notin \Lambda, \\ \overline{L} \notin \Lambda \end{cases} \qquad \textsf{Subsume}\quad \frac{\Lambda,\ L \vdash \Phi,\ L \vee C}{\Lambda,\ L \vdash \Phi}$$

$$\textsf{Empty}\quad \frac{\Lambda \vdash \Phi, \Box}{\Lambda \vdash \Box}\quad \text{if}\ \ \Phi \neq \emptyset \qquad \textsf{Resolve}\quad \frac{\Lambda,\ \overline{L} \vdash \Phi,\ L \vee C}{\Lambda,\ \overline{L} \vdash \Phi,\ C}$$

Fig. 1. The DPLL Calculus.

literal L from Φ^* and check recursively, and separately, the satisfiability of $\Phi^* \cup \{L\}$ and of $\Phi^* \cup \{\overline{L}\}$, succeeding if and only if one of the two subsets is satisfiable.

The essence of this procedure can be captured by a sequent-style calculus, the DPLL calculus, first described in (Tinelli, 2002), consisting of the derivation rules in Figure 1.

The calculus manipulates sequents of the form $\Lambda \vdash \Phi$, where Λ, the *context* of the sequent, is a finite set of ground literals and Φ is a finite (multi)set of ground clauses. [6]

The intended goal of the calculus is to derive a sequent of the form $\Lambda \vdash \emptyset$ from an initial sequent $\emptyset \vdash \Phi_0$, where Φ_0 is a clause set to be checked for satisfiability. If that is possible, then Φ_0 is satisfiable; otherwise, Φ_0 is unsatisfiable. Informally, the purpose of the context Λ is to store incrementally a set of *asserted literals*, i.e., a set of literals in Φ_0 that must or can be true for Φ_0 to be satisfiable. When $\Lambda \vdash \emptyset$ is derivable from $\emptyset \vdash \Phi_0$, the context Λ is indeed a witness of Φ_0's satisfiability as it describes a (Herbrand) model of Φ_0: one that satisfies an atom p in Φ_0 iff p occurs positively in Λ.

The context is grown by the Assert and the Split rules. The Assert rule models the fact that every literal occurring as a unit clause in the the current clause set must be satisfied for the whole clause set to be satisfied. The Split rule corresponds to the decomposition in smaller subproblems of the DPLL procedure. This rule is the only *don't-know* non-deterministic rule of the calculus. It is used to guess the truth value of an *undetermined* literal L in the clause

[6] As customary, we write $\Lambda, L \vdash \Phi, C$, say, to denote the sequent $\Lambda \cup \{L\} \vdash \Phi \cup \{C\}$.

set Φ of the current sequent $\Lambda \vdash \Phi$, where by undetermined we mean such that neither L nor $\overline{L}$ is in the context Λ. The guess allows the continuation of the derivation with either the sequent $\Lambda, L \vdash \Phi$ or with the sequent $\Lambda, \overline{L} \vdash \Phi$.

The other two main operations of the DPLL procedure, unit resolution with backward subsumption and unit subsumption, are modeled respectively by the Resolve and the Subsume rule. The Resolve rule removes from a clause all literals whose complement has been asserted—which corresponds to generating the simplified clause by unit resolution and then discarding the old clause by backward subsumption. The Subsume rule removes all clauses that contain an asserted literal—because all of these clauses will be satisfied in any model in which the asserted literal is true.

The DPLL calculus is easily proven sound, complete and terminating. It can be shown (Tinelli, 2002) that the calculus maintains its completeness even if one constrains the Split rule to split only on positive literals. [7] In other words, there is no loss of completeness if Split is replaced by the rule:

$$\textsf{Split'} \quad \frac{\Lambda \vdash \Phi,\ L \vee C}{\Lambda, L \vdash \Phi,\ L \vee C \qquad \Lambda, \overline{L} \vdash \Phi,\ L \vee C} \quad \text{if} \begin{cases} L \text{ is positive,} \\ C \neq \Box, \\ L \notin \Lambda, \\ \overline{L} \notin \Lambda \end{cases}$$

Another change that does not alter the calculus in any fundamental way is the replacement of the Assert and Empty rules by the following more powerful versions:

$$\textsf{Assert'} \quad \frac{\Lambda \vdash \Phi,\ L_1 \vee \cdots \vee L_n \vee L}{\Lambda, L \vdash \Phi,\ L_1 \vee \cdots \vee L_n \vee L} \quad \text{if} \begin{cases} n \geq 0, \\ \overline{L_1}, \ldots, \overline{L_n} \in \Lambda, \\ L \notin \Lambda, \\ \overline{L} \notin \Lambda \end{cases}$$

$$\textsf{Close} \quad \frac{\Lambda \vdash \Phi,\ L_1 \vee \cdots \vee L_n}{\Lambda \vdash \Box} \quad \text{if} \begin{cases} \Phi \neq \emptyset \text{ or } n > 0, \\ \overline{L_1}, \ldots, \overline{L_n} \in \Lambda \end{cases}$$

Note that Assert' and Close reduce respectively to Assert and Empty given earlier when $L_1 \vee \cdots \vee L_n$ has no literals (i.e., if $n = 0$). The reason Assert'

[7] This fact is known in the SAT literature and is used as an optimization some DPLL-based SAT solvers.

and Close do not really change the calculus is that each application of Assert', respectively Close, can be simulated by n applications of Resolve followed by one application of Assert, repectively Empty. We point out that with Close the Resolve rule becomes superfluous for completeness.

We mention the Split', Assert' and Close rules here because they will facilitate our comparison between the Model Evoution calculus and DPLL.

3 The Model Evolution Calculus

The Model Evolution calculus is a direct lifting of the DPLL calculus to the first-order level. The lifting is achieved with a suitable first-order version of the rules Split', Assert', Subsume, Resolve and Close of DPLL, with the addition of an extra rule, Compact, which is a simplification rule specific to the first-order case.

Similarly to DPLL, the derivation rules of the Model Evolution calculus apply to and produce sequents of the form $\Lambda \vdash \Phi$. This time, however, Λ is a finite set of literals possibly with variables and parameters, called again a context, and Φ is a set of clauses possibly with variables.

As mentioned in the introduction, the context Λ in a sequent $\Lambda \vdash \Phi$ determines an interpretation I_Λ meant to be a model of Φ. The purpose of the main rules of the calculus is to recognize when I_Λ is not a model of Φ, and *repair* it so that it can become one. The repairs are both localized and incremental, and based on the computation of most general unifiers. The progressive repair process or *evolution* of the candidate model starts with a default interpretation and continues until an actual model is found or no further repairs are possible. The calculus is non-deterministic because in some cases the current interpretation can be repaired in two alternative ways, neither of which can be ruled out a priori. With an initial sequent $\Lambda_0 \vdash \Phi_0$ then, this gives rise to a search space of possible evolution sequences for I_{Λ_0}, the initial candidate model for Φ_0.

To ease the technical presentation it comes handy to work with a pseudo-literal $\neg v$, where v is a parameter that ranges over *atoms*. The intention is to have $\neg v$ stand by default for all negative literals.

We will show that when Φ_0 is unsatisfiable and Λ_0 is just $\{\neg v\}$ all possible evolution sequences are finitely failed—making the calculus complete. We will also show that, conversely, if all evolution sequences for $I_{\{\neg v\}}$ are finitely failed, then Φ_0 is guaranteed to be unsatisfiable—making the calculus sound as well. In the process, we will also show that non-failed finite sequences that cannot be grown further end with a context whose candidate model is indeed a model

of Φ_0.

3.1 Contexts and Interpretations

The defining aspect of the calculus, modeled after FDPLL, is the way contexts are extended to the first-order case, and the rôle they play in driving the derivation and the model generation process. Therefore, we start our description of the calculus with them.

Definition 3.1 (Context) *A* context *is a set of the form* $\{\neg v\} \cup S$ *where* $v \in V$ *and* S *is a finite set of literals.*

Where L is a literal and Λ a context, we will write $L \in_{\approx} \Lambda$ if L is a variant of a literal in Λ, $L \in_{\simeq} \Lambda$ if L is a p-variant of a literal in Λ, and $L \in_{\leq} \Lambda$ if L is a p-instance of a literal in Λ.

The calculus works only with *non-contradictory* contexts.

Definition 3.2 (Contradictory) *A literal* L *is* contradictory with *a context* Λ *iff* $L\sigma = \overline{K}\sigma$ *for some* $K \in_{\simeq} \Lambda$ *and some p-preserving substitution* σ*. A context* Λ *is* contradictory *iff it contains a literal that is contradictory with* Λ*.*

Example 3.3 *Let* $\Lambda := \{\neg v,\ p(x_1, y_1),\ \neg q(v_1)\}$*. Then* $\neg p(h(x), u)$*,* $\neg p(v, u)$*, and* $q(y)$ *are all contradictory with* Λ*. However,* $q(f(v))$ *and* $r(x)$*, say, are not. (Recall that* x, x_1, y_1 *are variables while* v, v_1, u *are parameters.)*

A non-contradictory context induces a unique Herbrand interpretation by means of the next three notions.

Definition 3.4 (Shields) *Let* K*,* L *be literals with* $K \gtrsim L$*. A literal* K' strongly shields L from K *iff* $K \gtrsim \overline{K'} \gtrsim L$*, and* K' shields L from K *iff* $K \gtrsim \overline{K''} \gtrsim L$ *for some literal* K'' *with* $K' \geq K''$*. A context* Λ (strongly) shields L from K *iff it contains a literal that (strongly) shields* L *from* K*.*

Equivalently, Λ shields L from K iff $K \gtrsim \overline{K''} \gtrsim L$, for some $K'' \in_{\leq} \Lambda$.

Definition 3.5 (Covers) *Let* L *be a literal and* Λ *a context. A literal* K strongly covers L in Λ *iff* $K \gtrsim L$ *and* Λ *does not shield* L *from* K*, and* K covers L in Λ *iff* $K \gtrsim L$ *and* Λ *does not strongly shield* L *from* K*.*

Definition 3.6 (Productivity) *Let* L *be a literal,* C *a clause, and* Λ *a context. A literal* K produces L in Λ *iff*

(1) K *covers* L *in* Λ*, and*
(2) there is no $K' \in \Lambda$ *that*

(a) strongly covers $\overline{L}$ in Λ and
(b) shields L from K.

The context Λ produces L *iff it contains a literal* K *that produces* L *in* Λ. *The context* Λ produces C *iff it produces one of* C*'s literals.*

From this definition it follows that any *non-contradictory* context containing a parameter-free literal K produces all instances of K and does not produce any instance of $\overline{K}$. This is a special case of a more general result, saying that any literal K produces all its p-instances in that context and does not produce any p-instance of $\overline{K}$ (cf. Lemma 8.6 below).

To help clarify their relationship, the concepts of shielding, covering and producing are illustrated in Figure 2.

Example 3.7 Consider the context $\Lambda = \{\neg v, \neg p(x,u,a), p(v,a,x)\}$. Now, $p(u,u,w)$ produces $p(a,a,a)$ in Λ, because $p(u,u,w)$ covers $p(a,a,a)$ in Λ and there is no $K' \in \Lambda$ that strongly covers $\neg p(a,a,a)$ in Λ and that shields $p(a,a,a)$ from $p(u,u,w)$. In fact, the only candidate literal for K' is $\neg p(x,u,a)$. That literal does shield $p(a,a,a)$ from $p(u,u,w)$—which means that $p(u,u,w)$ does not strongly cover $p(a,a,a)$ in Λ; however, it does not strongly cover $\overline{p(a,a,a)} = \neg p(a,a,a)$ in Λ because $p(v,a,x) \in \Lambda$ shields $\neg p(a,a,a)$ from $\neg p(x,u,a)$.

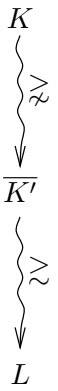
The context Λ produces $p(a,a,b)$ because $p(v,a,x) \gtrsim p(a,a,b)$ and Λ does not shield $p(a,a,a)$ from $p(v,a,x)$. But Λ does not produce $\neg p(v,a,a)$. Although $\neg p(x,u,a) \gtrsim \neg p(v,a,a)$ holds, there is a literal in Λ, namely $p(v,a,x)$, that strongly covers $p(v,a,a)$ in Λ and that shields $\neg p(v,a,a)$ from $\neg p(x,u,a)$. □

A consequence of the presence of the pseudo-literal $\neg v$ in every context Λ is that Λ produces L or $\overline{L}$ for every literal L. We can use this fact to associate to Λ a unique Herbrand interpretation.

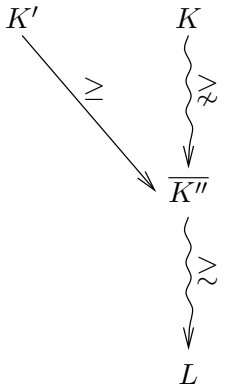
Definition 3.8 (Induced interpretation) *Let* Λ *be a non-contradictory context with signature* Σ^{sko}. *The* interpretation induced by Λ, *denoted by* I_Λ, *is the Herbrand* Σ*-interpretation that satisfies a positive ground* Σ*-literal* L *iff* L *is produced by* Λ.

For simplicity, we will sometimes say that a context *satisfies/falsifies* a literal or a clause if its induced interpretation does so.

Note that while a context can contain literals with Skolem constants with respect to some *original* signature Σ, the induced interpretation is over the original signature only. Also note that since it is possible for a context Λ to produce both a positive ground literal L and its complement $\overline{L}$, the above

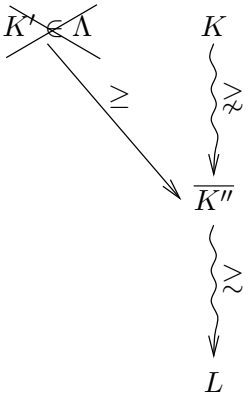


(a) K' strongly shields L from K.

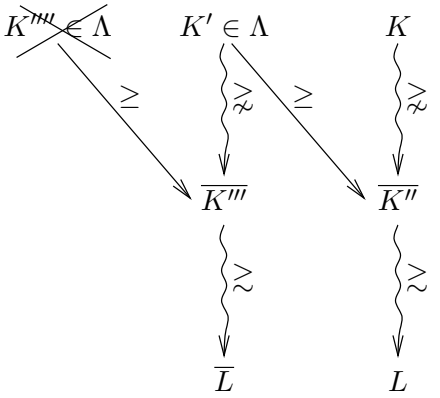


(b) K' shields L from K.

(c) K covers L in Λ.



(d) K strongly covers L in Λ.



(e) The only possible scenario when K covers L in Λ but K does *not* produce L in Λ: some literal $K' \in \Lambda$ shields L from K, and K' strongly covers $\overline{L}$ in Λ (and hence produces $\overline{L}$ in Λ).

Fig. 2. Illustrations of the notions "shields", "covers" and "produces". See Definitions 3.4, 3.5 and 3.6 for notation.

definition is asymmetric, because in that case I_Λ always chooses to satisfy L over $\overline{L}$. Contrapositively, this means that if I_Λ satisfies a ground literal L and L is positive, then L and possibly also $\overline{L}$ are produced by Λ. If on the other hand L is negative, then L but not $\overline{L}$ is produced by Λ.

It should be clear now that the purpose of the pseudo-literal $\neg v$ in a context Λ is to provide a *default* truth-value to those ground literals whose value is not determined by the rest of the context. In fact, consider a ground Σ-literal L such that neither L nor $\overline{L}$ is produced by $\Lambda \setminus \{\neg v\}$. If L is positive, then it is false in I_Λ because it is not produced by Λ at all. If L is negative, then it is true in I_Λ because it is produced by $\neg v$.

At this point a clarification on the complexity of the definition of productivity is perhaps in order. One might think that the more intuitive definition stating that

K *produces* L *in* Λ iff K strongly covers L in Λ,

is good enough to support Definition 3.8. While simpler, this definition is however not adequate for our purposes. The reason is that there exist (somewhat complicated) contexts Λ and ground literals L such that Λ produces neither L or $\overline{L}$ according to the simpler definition above. [8] Now, the requirement that any context Λ produce L or $\overline{L}$ for every ground literal L is fundamental for us, because it is used in the calculus to identify context literals that cause ground clause instances to be falsified by the current induced interpretation (see later). This requirement is indeed satisfied by the given Definition 3.6: should a candidate literal $K \in \Lambda$ cover L in Λ but not produce L in Λ, then there will be a literal $K' \in \Lambda$ that shields L from K *and produces* $\overline{L}$ *in* Λ (cf. Figure 2-(e)).

We refer the reader to (Fermüller and Pichler, 2005) for a study on the complexity of basic reasoning tasks on contexts [9] and their relation to other model representation formalisms.

For a given sequent $\Lambda \vdash \Phi$ the interpretation induced by the context Λ may falsify a clause of Φ. This situation is detectable through the computation of *context unifiers*.

Definition 3.9 (Context Unifier) *Let* Λ *be a context and*

$$C = L_1 \vee \cdots \vee L_m \vee L_{m+1} \vee \cdots \vee L_n$$

a parameter-free clause, where $0 \leq m \leq n$. *A substitution* σ *is* a context unifier of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ *iff there are fresh p-variants* $K_1, \ldots, K_n \in_\simeq \Lambda$ *such that*

[8] In essence, this is possible because context literals may shield each other in a cyclic way, preventing each one from producing L or $\overline{L}$.

[9] However, note that the context literals in (Fermüller and Pichler, 2005) are all variable-free or parameter-free. The "mixed" setting, where context literals may contain both variables and parameters is being introduced with this paper.

(1) σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,
(2) for all $i = 1, \ldots, m$, $(\mathcal{P}ar(K_i))\sigma \subseteq V$,
(3) for all $i = m+1, \ldots, n$, $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$.

We say, in addition, that σ is productive *iff K_i produces $\overline{L_i}\sigma$ in Λ for all $i = 1, \ldots, n$.*

For $i = 1, \ldots, n$, we call K_i a context literal of σ. *A context unifier σ of C against Λ with remainder $L_{m+1}\sigma \vee \cdots \vee L_n\sigma$ is* admissible (for Split) *iff for all distinct $i, j = m+1, \ldots, n$, $\mathcal{V}ar(L_i\sigma) \cap \mathcal{V}ar(L_j\sigma) = \emptyset$.*

Note that each context unifier has a unique remainder. If σ is a context unifier of a clause C with remainder D we call each literal of D *a remainder literal* of σ.

Example 3.10 Let $\Lambda := \{\neg v,\ p(v_1, u_1),\ \neg p(x_1, g(x_1)),\ q(v_2, g(v_2))\}$ and $C_1 = r(x) \vee \neg p(x, y)$. Then, the substitutions

$$\sigma_1 := \{v \mapsto r(x),\ v_1 \mapsto x,\ u_1 \mapsto y\}$$
$$\sigma_2 := \{v \mapsto r(v_1),\ x \mapsto v_1,\ u_1 \mapsto y\}$$

are both context unifiers of C_1 against Λ with respective remainders $r(x) \vee \neg p(x, y)$ and $r(v_1) \vee \neg p(v_1, y)$. While both σ_1 and σ_2 are productive, only σ_2 is admissible. The context unifier σ_1 is not admissible because its remainder literals are not variable-disjoint. By contrast, the substitution

$$\sigma_3 := \{v \mapsto r(v_1),\ x \mapsto v_1,\ y \mapsto u_1\}$$

is a context unifier of C_1 against Λ, this time with remainder $r(v_1)$, that is both productive and admissible.

Consider now the clause $C_2 = \neg p(x, y) \vee \neg q(x, y)$. The substitution

$$\sigma_4 := \{v_1 \mapsto v_2,\ u_1 \mapsto g(v_2),\ x \mapsto v_2,\ y \mapsto g(v_2)\}$$

is a context unifier of C_2 against Λ with remainder $\neg p(v_2, g(v_2))$. This context unifier is admissible but it is not productive because the literal $p(v_1, u_1)$ of Λ chosen to unify with $\overline{\neg p(x, y)}$ does not produce $\overline{\neg p(x, y)}\sigma_4 = p(v_2, g(v_2))$. □

We point out for later comparisons with the DPLL calculus that when, in Definition 3.9, C is ground and $\neg v$ is the only non-ground literal of Λ, the substitution σ is a context unifier of C against Λ with remainder $(L_{m+1}\sigma \vee \cdots \vee L_n\sigma) = (L_{m+1} \vee \cdots \vee L_n)$ iff

(1) for all $i = 1, \ldots, m$, $K_i = \overline{L_i}$ and

(2) for all $i = m+1, \ldots, n$, L_i is a positive literal and K_i is a p-variant of $\neg v$.

Admissible context unifiers are fundamental in the Model Evolution calculus. In fact, with a context Λ and a clause C, the existence of an admissible context unifier of C against Λ is a sign that I_Λ might not be a model of C. This is because it is possible to compute an admissible context unifier of C against Λ whenever Λ is non-contradictory and I_Λ falsifies C. The discovery by the calculus of an admissible context unifier σ of C against the current context Λ prompts a modification of Λ that involves adding a literal of $C\sigma$, with the goal of making C valid in the new I_Λ. This literal is chosen only among the remainder literals of σ, the reason being essentially that non-remainder literals can be ignored with no loss of completeness.

Note that while the existence of an admissible context unifier σ of C against Λ is necessary for the unsatisfiability of C in I_Λ, it is not sufficient unless σ is also productive. As a matter of fact, for completeness the calculus needs to add to the context only remainder literals of admissible unifiers that are also productive. For greater flexibility, however, we allow it to add remainder literals of non-productive admissible unifiers as well. The reason is mostly practical and twofold: first, when implementing the calculus, insisting on computing only productive context unifiers can be considerably more expensive than computing context unifiers that are usually, although not always, productive; second, sometimes "repairing" candidate models with remainder literals from non-productive context unifiers can produce more constrained contexts, as illustrated in the example that follows.

Example 3.11 Consider the context $\Lambda := \{\neg v,\ p(u),\ \neg q(g(y))\}$ and the clause $C := p(x) \vee q(x)$. The substitution

$$\sigma := \{v \mapsto p(g(y)),\ x \mapsto g(y)\}$$

is a context unifier of C against Λ with remainder $p(g(y))$, but it is not productive. As a matter of fact, I_Λ satisfies C, and so $C\sigma$, because Λ produces every ground instance of $p(x)$. This means that there is no need to repair I_Λ with the addition of $p(g(y))$ to Λ. However, as we explain in Section 3.2, having the universal literal $p(g(y))$ in Λ along with $p(u)$ considerably constrains further repairs involving instances of $p(u)$, with a corresponding reduction in the search space. □

Productivity issues aside, we point out that although context unifiers for a given clause C and context Λ are easily computable (they are just simultaneous most general unifiers), they are not unique and may not be admissible. Nevertheless, the calculus does not need to search for all admissible context unifiers. For completeness purposes *any* admissible context unifier of C against Λ will do. Furthermore, and more important, admissible context unifiers are

easily derived from non-admissible ones. In fact, let σ be a context unifier of C against Λ with remainder D. If σ has a remainder literal L that shares variables with another remainder literal, one can compose σ with a substitution that moves the variables of L to fresh parameters and fixes everything else. It is easy to see that a repeated application of this process leads to an admissible context unifier $\sigma\rho$ of C whose remainder is included in $D\rho$. For instance, the non-admissible context unifiers σ_1 in Example 3.10 can be turned into the admissible one σ_3 by this kind of process.

Now, while the choice of an admissible context unifier over another is irrelevant for completeness, some context unifiers are better than others for efficiency purposes. A context unifier with an empty remainder for instance is always preferrable to one with an non-empty remainder, because it lets the calculus stop the derivation right away, as we will see. In general, context unifiers with a smaller remainder are preferable over context unifiers with a longer remainder because offer less choices for repairing the current model. Also, context unifiers with parameter-free remainder literals are preferable over context unifiers with variable-free remainder literals only. As we explain later, the addition of a parameter-free literal to a context imposes more constraints on later additions than the addition of a variable-free literal, leading in principle to shorter derivations.

3.2 Parameters vs. Variables

Before moving to describe the rules of the Model Evolution calculus, it is important to clarify the respective rôles that parameters and variables play in it.

We said that the calculus manipulates sequents of the form $\Lambda \vdash \Phi$, where Φ is a clause set and Λ is a context providing a candidate model for Φ. Each derivation in the calculus starts with a sequent of the form $\neg v \vdash \Phi_0$, where Φ_0 contains only standard clauses, i.e., clauses with no parameters—but possibly with variables. Similarly, all sequents generated during a derivation have clause sets consisting of standard clauses only. Variables then can appear both in clause sets and in contexts. Parameters instead can appear only in contexts.

The rôle of variables within a clause is the usual one: they stand for all ground terms. In contrast, the rôle of variables and parameters within a context is to constrain, in different ways, how a candidate model can be repaired.

The current context Λ needs repairing whenever it falsifies a clause C in Φ. As we observed earlier, in that case there is an admissible context unifier σ of C against Λ such that Λ falsifies $C\sigma$ as well. To satisfy C it is then necessary to modify Λ so that it satisfies (at least) $C\sigma$. One way to do that is to pick

from $C\sigma$ a literal $L\sigma$ *non-contradictory with* Λ and add it to Λ. When $L\sigma$ contains no parameters, that is, is a *universal literal* in FDPLL terminology, the addition of $L\sigma$ will indeed make all ground instances of $L\sigma$ satisfied by the new context. Moreover, it will make such instances *permanently satisfied* in the sense that any further additions of literals to the context that do not make it contradictory will preserve the satisfiability of those instances.

In contrast, when $L\sigma$ contains parameters, the assertion of all the ground Σ-instances of $L\sigma$ is provisional: it can be retracted later, in whole or in part. With the addition of $L\sigma$ to the context, the calculus is in essence making the assumption that there is a model of C that satisfies all ground instances of $L\sigma$. This assumption, however, is just a working hypothesis, subject to be revised when evidence against it is found. That happens if the calculus, to satisfy some other clause, later adds to the current context Λ' a literal K' that is a (proper) instance of $\overline{L\sigma}$.[10] After the addition, the new context satisfies only those instances of $L\sigma$ that are not an instance of $\overline{L}\sigma'$.

We observe that the less parameters a context literal has, the more of its instances are parameter preserving—with the extreme cases of variable-free literals on one side, having *no* p-instances other than p-variants, and parameter-free literals on the other, having *only* p-instances. This means that the less the parameters in a context literal the more difficult it is to change the truth value of its instances by extending the context. That can be seen with the following simple example.

Example 3.12 Consider a signature Σ containing a constant symbol a, and the contexts $\Lambda_1 = \{\neg v, L_1\}$, $\Lambda_2 = \{\neg v, L_2\}$, and $\Lambda_3 = \{\neg v, L_3\}$, where $L_1 = p(u, v)$, $L_2 = p(x, v)$, and $L_3 = p(x, y)$. All ground Σ-instances of L_i are true in I_{Λ_i} for $i = 1, \ldots, 3$. In the first context, since $p(u, v)$ is variable-free, it is possible to change the truth of all literals of the form $p(a, t)$ or $p(t, a)$, with t a ground term, by adding to Λ_1 the literals $\neg p(a, v)$ and $\neg p(u, a)$. In the second context, this is already not possible because the literal $\neg p(a, v)$ is contradictory with $p(x, v)$. Nevertheless, it is still possible to change the truth value of all the instances of $p(x, v)$ of the form $p(t, a)$, with the addition of $\neg p(u, a)$. In the last context, nothing can be done with either $\neg p(a, v)$ or $\neg p(u, a)$ because they are both contradictory with $p(x, y)$. □

Because of the stronger restrictions they impose on the possible evolutions of a context, variables in effect help prune the search space during derivations. However, they cannot be added indiscriminately in place of parameters in context literals without making the calculus incomplete. By using *admissible*

[10] More generally, it happens if the calculus adds a literal K one of whose p-preserving instances K' is an instance of $\overline{L}\sigma$. Note that in any case, K' must be an non-parameter-preserving instance of $\overline{L\sigma}$, otherwise the new context would be contradictory, which is not permitted.

context unifiers the $\mathcal{ME}$ calculus is able to introduce a fair amount of variables in contexts without loss of completeness.

3.3 Derivation Rules

The $\mathcal{ME}$ calculus consists of three basic derivation rules: Split, Assert and Close, and three optional rules: Resolve, Subsume, and Compact. We define and discuss them in the following. In the process, We also compare them with the rules of the DPLL calculus to show that, modulo a technicality, $\mathcal{ME}$ reduces to DPLL when the input clause set is ground.[11] The technicality is simply that, contrary to DPLL, contexts in our calculus contain the pseudo literal $\neg v$. Except for that, the two calculi operate on the same kind of sequents in the ground case, and stepwise simulate each other.

The Split *Rule*

$$\textsf{Split}\quad \frac{\Lambda \vdash \Phi,\, C \vee L}{\Lambda,\, L\sigma \vdash \Phi,\, C \vee L \qquad \Lambda,\, (\overline{L\sigma})^{\text{sko}} \vdash \Phi,\, C \vee L}\quad \text{if } (*)$$

$$\text{where } (*) = \begin{cases} C \neq \Box, \\ \sigma \text{ is an admissible context unifier of } C \vee L \text{ against } \Lambda \\ \text{with remainder literal } L\sigma, \\ \text{neither } L\sigma \text{ nor } (\overline{L\sigma})^{\text{sko}} \text{ is contradictory with } \Lambda \end{cases}$$

We say that the clause $C \vee L$ above is the *selected clause*, the literal L is the *selected literal*, and σ is the *context unifier* of Split.

The Split rule is the analog of the Split' rule in DPLL. As in DPLL, this is the only (*don't-know*) non-deterministic rule of the calculus, the one that drives the search for a model for the input clause set. Split is the rule that discovers when the current candidate model falsifies one of the clauses in the current clause set. It does that by computing a context unifier σ with a non-empty remainder for a clause with at least two literals. The rule attempts to repair the candidate model by selecting a remainder literal $L\sigma$ and adding either $L\sigma$ or its complement to the context. The reason for adding the complement of $L\sigma$ in alternative to $L\sigma$ is of course that the current clause set may have no models that satisfy $L\sigma$. Obviously, the addition of $L\sigma$'s complement to the context will not make the selected clause $C \vee L$ valid in the new candidate model. But it will make sure that no context unifier of $C \vee L$ has $L\sigma$ in its

[11] More precisely, it reduces to the version of DPLL, described at the end of Section 2, that uses the rules Split', Assert' and Close in place of Split, Assert, and Empty, respectively.

remainder, forcing the calculus to select other remainder literals, if any, to make $C \vee L$ valid.

Note that Split does not exactly add the complement of $L\sigma$ to the current context, but rather a suitably Skolemized version of it: one that replaces every variable of $L\sigma$ by a fresh Skolem constant.[12] This is in accordance with our treatment context literals as universally quantified wrt. their variables, which essentially become existentially quantified after negation, thus leading to fresh Skolem constants.

Example 3.13 If $P(x, f(x), y, v, w)$ is the selected literal of a Split inference with, say, empty context unifier, then this literal $P(x, f(x), y, v, w)$ will be added to the context in the left sequent, and the literal $\neg P(c, f(c), d, v, w)$ will be added to the context in the right sequent, where c and d are fresh constants. □

We point out that a Split inference cannot be followed on either branch by another Split inference with the same literal $L\sigma$ (or a p-variant of it) because then either $L\sigma$ or $(\overline{L\sigma})^{\text{sko}}$ will be contradictory with the context. Since Split is the only rule that introduces Skolemized literals into contexts, this implies in particular that no context can contain more than one Skolemized version of the same literal. It is not too difficult to check that these properties hold even in presence of the Compact rule below, which removes literals from a context.

In the ground case—that is, when both $\Lambda \setminus \{\neg v\}$ and $\Phi \cup \{C \vee L\}$ are ground—the Split rule reduces exactly to the Split' rule of DPLL in Section 2. To see that it is enough to recall that in the ground case, if $L\sigma = L$ is a remainder literal of a context unifier σ of $C \vee L$ against Λ, then L must have been unified by σ with a variant of $\overline{\neg v}$, which implies that L is positive. Moreover, L (respectively, $(\overline{L\sigma})^{\text{sko}} = \overline{L}$) is contradictory with Λ, in the sense of Definition 3.2, iff $\overline{L} \in \Lambda$ (respectively, $L \in \Lambda$).

The Assert *Rule*

$$\textsf{Assert} \quad \frac{\Lambda \quad \vdash \Phi,\ C \vee L}{\Lambda, L\sigma \ \vdash \Phi,\ C \vee L} \quad \text{if} \quad \begin{cases} \sigma \text{ is a context unifier of } C \text{ against} \\ \Lambda \text{ with an empty remainder,} \\ L\sigma \text{ is parameter-free and} \\ \text{non-contradictory with } \Lambda, \\ \text{there is no } K \in \Lambda \text{ s. t. } K \geq L\sigma \end{cases}$$

We say that the clause $C \vee L$ above is the *selected clause*, and L is the *selected literal* of Assert.

[12] More precisely, one that does not occur in the current context yet.

As in DPLL, the Assert rule is extremely useful in reducing the non-determinism of the calculus. When the first of its side conditions holds, the candidate model induced by (any extension of) Λ *must* make $L\sigma$ valid to become a model of $\Phi \cup \{C \vee L\}$. The Assert rule achieves just that by adding $L\sigma$ to the context. Note that since $L\sigma$ is parameter-free, its addition to the context is not retractable. Also note that the rule does not apply if the permanent validity of $L\sigma$ has been already established. This is the case when Λ contains a—necessarily parameter-free—literal K such that $K \geq L\sigma$. The rule does not apply also if $L\sigma$ is contradictory with Λ. In that case, however, the candidate model is unrepairable. The Close rule, described later, will detect that and cause the calculus to stop working on $\Lambda \vdash \Phi,\ L \vee C$.

When $C = \Box$, that is, when the selected clause of Assert is just a unit clause L, the empty substitution is a context unifier of C against Λ with an empty remainder. In that case, the effect of the rule is simply to add L to the context. For greater flexibility, the Assert rule is defined for clauses with an arbitrary number of literals. As we will see in Section 4.3, however, for completeness purposes it is enough to restrict its applications to unit clauses only.

In the ground case, Assert reduces exactly to Assert' in DPLL. The reason is that, in the ground case (i) σ is a context unifier of C against Λ with an empty remainder iff σ is the empty substitution and Λ contains the complement of each literal of C, (ii) $L\sigma$ is trivially parameter-free, (iii) there is no $K \in \Lambda$ s.t. $K \geq L\sigma$ iff $L \notin \Lambda$, and (iv) $L\sigma$ is not contradictory with Λ iff $\overline{L} \notin \Lambda$.

The Close *Rule*

$$\textsf{Close}\quad \frac{\Lambda \vdash \Phi,\ C}{\Lambda \vdash \Box}\quad \text{if}\ \begin{cases} \Phi \neq \emptyset \text{ or } C \neq \Box, \\ \text{there is a context unifier } \sigma \text{ of } C \text{ against } \Lambda \\ \text{with an empty remainder} \end{cases}$$

We say that the clause C above is the *selected clause* of Close, and σ is the *context unifier* of Close.

The idea behind Close is that when its precondition holds there is no way to repair the current candidate model to make it satisfy C. The replacement of the current close set by the empty clause signals that the calculus has given up on that candidate model. Note that, because of Resolve below, it is possible for the calculus to generate a sequent containing an empty clause. The Close rule recognizes such sequents and applies to them as well. To see that it is enough to observe that, for any context Λ, the empty substitution is a context unifier of $\Box$ against Λ with an empty remainder.

In the ground case, the Close rule reduces to its namesake in DPLL, because then C has a context unifier against Λ with an empty remainder iff $\overline{L} \in \Lambda$ for

every literal L of C.

The Subsume *Rule*

$$\textsf{Subsume} \quad \frac{\Lambda,\ K \vdash \Phi,\ L \vee C}{\Lambda,\ K \vdash \Phi} \quad \text{if} \quad K \geq L$$

We say that the clause $L \vee C$ above is the *selected clause* of Subsume.

The purpose of Subsume is the same as in DPLL: to get rid of clauses that are valid in the current candidate model, and are guaranteed to stay so. [13] These are exactly those clauses one of whose literals is a p-instance of a literal in the current context. Although Subsume is not needed for completeness, it is potentially useful in practice since it reduces the size of the current clause set.

In the ground case, the Subsume rule reduces to its namesake in DPLL because, then, $K \geq L$ iff $K = L$.

The Resolve *Rule*

$$\textsf{Resolve} \quad \frac{\Lambda \vdash \Phi,\ L \vee C}{\Lambda \vdash \Phi,\ C} \quad \text{if} \quad \begin{cases} \text{there is a context unifier } \sigma \text{ of } L \\ \text{against } \Lambda \text{ with an empty remainder} \\ \text{such that } C\sigma = C \end{cases}$$

We say that the clause $L \vee C$ above is the *selected clause* and L is the *selected literal* of Resolve.

This rule is similar to Subsume in that it is not needed for completeness but is useful to reduce the complexity of the current clause set. Since Resolve is in a sense dual to Subsume, it would be reasonable to expect its precondition to be simply that there is a literal $\overline{K} \in \Lambda$ such that $K \geq L$. This precondition, however, is a special case of the one provided. The given precondition makes Resolve more widely applicable, allowing for more frequent simplifications. Observe that Resolve is a special case of unit resolution (with backward subsumption): the one in which the resolvent of a unit clause $\overline{K}$ and a clause $L \vee C$ is exactly C—as opposed to a proper instance of C.

In the ground case, the Resolve rule as well reduces to its namesake in DPLL. To see why it is enough to observe that in that case Resolve's precondition holds iff σ is the empty substitution and $\overline{L} \in \Lambda$.

[13] Note that, as L is parameter-free, a necessary condition for $K \geq L$ is that K be parameter-free, which implies that none of its instances can made false by subsequent contexts.

The Compact *Rule*

$$\textsf{Compact}\quad \frac{\Lambda,\ K,\ L\ \vdash\ \Phi}{\Lambda,\ K\ \vdash\ \Phi}\quad \text{if}\ \ K \geq L$$

We say that the literal L above is the *selected literal* and the literal K is the *subsuming literal* of Compact. [14]

The Compact rule is another simplification rule that is not nedeed for completeness but is useful in practice. Its intended application is after the addition of a literal K that is parameter-preserving more general than other literals in the context. After this addition, all such literals become superfluous since they can be replaced by K for all purposes. Hence Compact allows their elimination.

There is no rule in DPLL corresponding to Compact. However, it is easy to see that Compact never applies in the ground case.

3.4 Derivation Examples

For a better idea on how the various rules of the calculus apply, we now describe informally a couple of examples of derivations—a formal definition of derivation will be given in the next section.

In the first example we consider a satisfiable clause set from the Bernays-Schönfinkel class, showing how the calculus computes a model of the clause set. In the second example we consider an unsatisfiable set whose unsatisfiability can be proven in the calculus deterministically—that is, without applying Split, thanks to the Assert rule.

Example 3.14 Consider the following initial sequent:

$$\neg v\ \vdash\ p(x) \vee q(x),\ \neg p(y) \vee q(y) \vee \neg p(c),\ \neg p(z) \vee \neg q(z)$$

One rule (in fact, the only rule) that applies to this sequent is Split, with selected clause $p(x) \vee q(x)$. In fact, with the fresh variants $\neg v_1$ and $\neg v_2$ of the context literal $\neg v$, the substitution

$$\sigma = \{v_1 \mapsto p(x),\ v_2 \mapsto q(x)\}$$

is a context unifier of $p(x) \vee q(x)$ with remainder $p(x) \vee q(x)$. While this context unifier is not admissible (because the remainder literals share a variable),

[14] The literals K and L are meant to be distinct.

we can generate an admissible one from it by composition with the substitution $\{x \mapsto u\}$, say. The new context unifier $\sigma' := \sigma\{x \mapsto u\}$ has remainder $p(u) \vee q(u)$. Since the remainder literal $p(u)$ and its complement are both non-contradictory with the context, we can add $p(u)$ to the context by (the left conclusion of) one application of Split, obtaining

$$\neg v,\ p(u)\ \vdash\ p(x) \vee q(x),\ \neg p(y) \vee q(y) \vee \neg p(c),\ \neg p(z) \vee \neg q(z)$$

Note that σ' is a context unifier of $p(x) \vee q(x)$ against the new context as well, and $p(u)$ and $q(u)$ are still remainder literals for σ'. However, Split does not apply with selected literal $p(x)$ anymore, because the complement of $p(u) = p(x)\sigma'$ is now contradictory with the contex. The Split rule does apply with selected literal $q(x)$, but in a sense this application is useless because $p(x) \vee q(x)$ is now satisfied by the current context, which makes every ground instance of $p(u)$ true. The uselessness of applying Split with selected literal $q(x)$ is witnessed by the fact that σ' is a non-productive context unifier. In fact, the literal $\neg v_1$, the context literal variant paired with $p(x)$ by the unifier, does not produce $\neg p(x)$ anymore because of the presence of $p(u)$ in the context.

We point out that using a fresh variant $p(u_1)$ of the context literal $p(u)$, there are now context unifiers of the subclause $\neg p(z_1)$ of $\neg p(z) \vee \neg q(z)$. Hence, we could think of applying Assert with selected clause $\neg p(z) \vee \neg q(z)$ and selected literal $\neg q(z)$. However, that is not possible because all these unifiers either have a non-empty remainder, like for instance the unifier $\{u_1 \mapsto z\}$, or instantiate $\neg q(z)$ to a non-parameter-free literal, like for instance the unifier $\{z \mapsto u_1\}$.

Now, using the context literal variants $p(u_1)$, $\neg v_1$ and $p(u_2)$, the substitution

$$\sigma = \{y \mapsto u_1,\ v_1 \mapsto q(u_1), u_2 \mapsto c\}$$

say, is an admissible contex unifier of $\neg p(y) \vee q(y) \vee \neg p(c)$ with remainder $q(u_1) \vee \neg p(c)$. Since neither $q(u_1)$ nor its complement is contradictory with the context, we can apply Split with selected clauses $\neg p(y) \vee q(y) \vee \neg p(c)$ and literal $q(y)$.[15] Choosing again the left conclusion of Split, which adds $q(y)\sigma$ to the context, we then obtain

$$\neg v,\ p(u),\ q(u_1)\ \vdash\ p(x) \vee q(x),\ \neg p(y) \vee q(y) \vee \neg p(c),\ \neg p(z) \vee \neg q(z)$$

Now the Close rule applies with selected clause $\neg p(z) \vee \neg q(z)$. In fact, using

[15] The substitution $\{u_1 \mapsto y,\ v_1 \mapsto q(y), u_2 \mapsto c\}$ is also a context unifier of $\neg p(y) \vee q(y) \vee \neg p(c)$, but it is not admissible because its remainder is $\neg p(y) \vee q(y) \vee \neg p(c)$.

the context literal variants $p(u_2)$ and $q(u_3)$, the substitution

$$\sigma = \{z \mapsto u_2,\ u_3 \mapsto u_2\}$$

is a context unifier of $\neg p(z) \vee \neg q(z)$ with an empty remainder. Let's consider then the right conclusion of the last Split application. With that conclusion we get the sequent

$$\neg v,\ p(u),\ \neg q(u_1)\ \vdash\ p(x) \vee q(x),\ \neg p(y) \vee q(y) \vee \neg p(c),\ \neg p(z) \vee \neg q(z).$$

Using now the context literal variants $p(u_2)$ and $\neg q(u_3)$, the substitution

$$\sigma = \{y \mapsto u_2,\ u_3 \mapsto u_2\}$$

is a context unifier with an empty remainder of the subclause $\neg p(y) \vee q(y)$ of $\neg p(y) \vee q(y) \vee \neg p(c)$. Moreover, $\neg p(c) = \neg p(c)\sigma$ is paramenter-free and non-contradictory with the context. Finally, there is no context literal K such that $K \geq \neg p(c)$.[16] Hence Assert applies with selected clause $\neg p(y) \vee q(y) \vee \neg p(c)$ and literal $\neg p(c)$, yielding the sequent

$$\neg v,\ p(u),\ \neg q(u_1),\ \neg p(c)\ \vdash\ p(x) \vee q(x),\ \neg p(y) \vee q(y) \vee \neg p(c),\ \neg p(z) \vee \neg q(z)$$

to which Subsume immediately applies with selected clause $\neg p(y) \vee q(y) \vee \neg p(c)$, yielding the sequent

$$\neg v,\ p(u),\ \neg q(u_1),\ \neg p(c)\ \vdash\ p(x) \vee q(x),\ \neg p(z) \vee \neg q(z).$$

At this point, because of the context literal $\neg p(c)$, we can apply Assert with selected clause $p(x) \vee q(x)$, obtaining

$$\neg v,\ p(u),\ \neg q(u_1),\ \neg p(c),\ q(c)\ \vdash\ p(x) \vee q(x),\ \neg p(z) \vee \neg q(z).$$

It is easy to see that no rules apply to this sequent. For Split in particular the reason is that every possible remainder literal is contradictory with one of the context literals or their complements—for Assert the argument is similar.

For any signature Σ that includes the symbols of the original clause set, the final context induces a Herbrand Σ-interpretation in which all ground

[16] The literal $\neg p(c)$ is an instance of the context literal $\neg v$, but it is not a p-instance of $\neg v$.

instances of $p(u)$ except $p(c)$ are true, and all ground instances of $q(u_1)$ except $q(c)$ are false. For illustration purposes, if Σ contains only the symbols of the original clause set, the induced interpretation is simply $\{q(c)\}$. If Σ also contains a functions symbols f of arity 1, say, the induced interpretation is $\{q(c)\} \cup \{p(f^n(c)) \mid n > 0\}$. We leave it to the reader to verify that these interpretations are indeed a model of the original clause set.

Example 3.15 Now consider the following initial sequent, where we use the usual mathematical notation for greater clarity:

$$\neg v \;\vdash\; \begin{array}{l} \neg(x \geq y) \vee \neg(y \geq z) \vee (x \geq z),\ (x \geq 0) \vee (0 \geq x),\ |x| \geq 0,\ 0 \geq -|x|, \\ \neg(x \geq 0) \vee (|x| \geq x),\ \neg(0 \geq x) \vee (|x| \geq x),\ \neg(|c| \geq c) \vee \neg(|c| \geq -|c|) \end{array}$$

Recalling an earlier observation on the applicability of Assert to unit clauses, we can immediately add each unit clause in the clause set to the context by means of Assert, and then remove it from the set by means of Subsume. This results in the sequent:

$$\neg v,\ |x| \geq 0,\ 0 \geq -|x| \;\vdash\; \begin{array}{l} \neg(x \geq y) \vee \neg(y \geq z) \vee (x \geq z),\ (x \geq 0) \vee (0 \geq x), \\ \neg(x \geq 0) \vee (|x| \geq x),\ \neg(0 \geq x) \vee (|x| \geq x), \\ \neg(|c| \geq c) \vee \neg(|c| \geq -|c|) \end{array}$$

Now consider the clause $\neg(x \geq y) \vee \neg(y \geq z) \vee (x \geq z)$ and its subclause $\neg(x \geq y) \vee \neg(y \geq z)$. With the context literal variants $|x_1| \geq 0$ and $0 \geq -|x_2|$, the substitution

$$\sigma = \{x \mapsto |x_1|,\ y \mapsto 0,\ z \mapsto -|x_2|\}$$

is an admissible context unifier of $\neg(x \geq y) \vee \neg(y \geq z)$ with an empty remainder. Moreover, the literal $|x_1| \geq -|x_2| = (x \geq z)\sigma$ is parameter-free and non-contradictory with Λ. Finally, there is no context literal K such that $K \geq (|x_1| \geq -|x_2|)$. Hence, we can add $|x_1| \geq -|x_2|$ to the context by one application of Assert.

With this new literal we can then simplify $\neg(|c| \geq c) \vee \neg(|c| \geq -|c|)$ to $\neg(|c| \geq c)$ with Resolve, obtaining

$$\begin{array}{l} \neg v,\ |x| \geq 0,\ 0 \geq -|x|, \\ |x_1| \geq -|x_2| \end{array} \;\vdash\; \begin{array}{l} \neg(x \geq y) \vee \neg(y \geq z) \vee (x \geq z),\ (x \geq 0) \vee (0 \geq x), \\ \neg(x \geq 0) \vee (|x| \geq x),\ \neg(0 \geq x) \vee (|x| \geq x), \\ \neg(|c| \geq c) \end{array}$$

We can then move $\neg(|c| \geq c)$ to the context by means of Assert and Subsume, obtaining

$$\begin{array}{l} \neg v,\ |x| \geq 0,\ 0 \geq -|x|, \\ |x_1| \geq -|x_2|,\ \neg(|c| \geq c) \end{array} \quad \vdash \quad \begin{array}{l} \neg(x \geq y) \vee \neg(y \geq z) \vee (x \geq z),\ (x \geq 0) \vee (0 \geq x), \\ \neg(x \geq 0) \vee (|x| \geq x),\ \neg(0 \geq x) \vee (|x| \geq x), \end{array}$$

With $\neg(|c| \geq c)$ in the context, we can apply Assert with selected clause $\neg(x \geq 0) \vee (|x| \geq x)$ and substitution $\sigma_1 = \{x \mapsto c\}$, adding $\neg(|c| \geq 0)$ to the context. Similarly, we can apply Assert with selected clause $\neg(0 \geq x) \vee (|x| \geq x)$ and add $\neg(0 \geq |c|)$, obtaining:

$$\ldots,\ \neg(|c| \geq 0),\ \neg(0 \geq |c|) \ \vdash\ \ldots,\ (x \geq 0) \vee (0 \geq x),\ \ldots$$

With $\neg(|c| \geq 0)$ in the context, the substitution $\sigma_2 = \{x \mapsto |c|\}$ is a context unifier of $(x \geq 0)$ with an empty remainder. Note that we cannot apply Resolve with selected clause $(x \geq 0) \vee (0 \geq x)$ and substitution σ_2 because $(0 \geq x)\sigma \neq (0 \geq x)$. Similarly, we cannot apply Assert either with selected clause $(x \geq 0) \vee (0 \geq x)$ and substitution σ_2 because $(0 \geq x)\sigma$ is contradictory with the context literal $\neg(0 \geq |c|)$. However, we can apply Close with selected clause $(x \geq 0) \vee (0 \geq x)$ and substitution σ_2, obtaining the sequent

$$\ldots \vdash \Box\ .$$

This is because, thanks to the context literals $\neg(|c| \geq 0)$ and $\neg(0 \geq |c|)$, σ_2 is a context unifier of $(x \geq 0) \vee (0 \geq x)$ with an empty remainder.

Note that other sequences of rule applications are possible for the given clause set. However, as we will prove in Section 4, since the described one contained no applications of Split, all those other sequences too are guaranteed to lead to an application of Close. But then, as we will also prove in Section 4, we can conclude that the original clause set is unsatisfiable. □

3.5 Derivations

We now provide a formal definition of derivation in the Model Evolution calculus. As customary in sequent-style calculi, derivations in $\mathcal{ME}$ are defined in terms of *derivation trees* where each node corresponds to a particular application of a derivation rule, and each of the node's children corresponds to one of the conclusions of the rule.

Definition 3.16 (Derivation Tree) *A* derivation tree (in $\mathcal{ME}$) *is a labeled tree inductively defined as follows:*

(1) a one-node tree is a derivation tree iff its root is labeled with a sequent of the form $\Lambda \vdash \Phi$, where Λ is a context and Φ is a clause set;
(2) A tree $\mathbf{T}'$ is a derivation tree iff it is obtained from a derivation tree $\mathbf{T}$ by adding to a leaf node N in $\mathbf{T}$ new children nodes $N_1, \ldots, N_m$ so that the sequents labeling $N_1, \ldots, N_m$ can be derived by applying a rule of the calculus to the sequent labeling N. In this case, we say that $\mathbf{T}'$ is derived from $\mathbf{T}$.

We say that a derivation tree $\mathbf{T}$ is a *derivation tree of a clause set* Φ iff its root node tree is labeled with $\neg v \vdash \Phi$.

Let us call a non-leaf node in a derivation tree a Split *node* if the sequents labelling its children are obtained by applying the Split rule to the sequent labeling the node. (Similarly for nodes to which other rules are applied.) Observe that every non-leaf node in a derivation tree has only one child unless it is a Split node, in which case it has two children. When it is convenient and it does not cause confusion, we will identify the nodes of a derivation tree with their labels.

Definition 3.17 (Open, Closed) *A branch in a derivation tree is* closed *if its leaf is labeled by a sequent of the form $\Lambda \vdash \Box$; otherwise, the branch is* open*. A derivation tree is* closed *if each of its branches is closed, and it is* open *otherwise.*

We say that a derivation tree (of a clause set Φ) is a *refutation tree (of Φ)* iff it is closed.

In the rest of the paper, the letters i and n will denote finite ordinal numbers, whereas the letter κ will denote an ordinal smaller than or equal to the first infinite ordinal. For every κ then, we will denote a possibly infinite sequence $a_0, a_1, a_2, \ldots$ of κ elements by $(a_i)_{i<\kappa}$.

Definition 3.18 (Derivation) *A* derivation (in $\mathcal{ME}$) *is a possibly infinite sequence of derivation trees $(\mathbf{T}_i)_{i<\kappa}$, such that for all i with $0 < i < \kappa$, $\mathbf{T}_i$ is derived from $\mathbf{T}_{i-1}$.*

We say that a derivation $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa}$ is a *derivation of a clause set* Φ iff $\mathbf{T}_0$ is a one-node tree with label $\{\neg v\} \vdash \Phi$. We say that $\mathcal{D}$ is a *refutation of* Φ iff $\mathcal{D}$ is finite and ends with a refutation tree of Φ.

We show in the next sections that the Model Evolution calculus is sound and complete in the following sense: for all sets Φ_0 of Σ-clauses with no parameters, Φ_0 is unsatisfiable iff Φ_0 has a refutation in the calculus.

To prove the calculus' completeness we will introduce the notion of an *exhausted branch*, a derivation tree branch that cannot be extended any further by the calculus. A by-product of the completeness proof will be to show that the interpretation induced by the context in the leaf of an open exhausted branch is a model of the clause set in the branch's root. This means that whenever a derivation of a clause set Φ_0 produces a tree with an open exausted branch, it is possible not only to state that Φ_0 is satisfiable, but also to provide (a finite description) of a model of Φ_0.

4 Correctness of the Calculus

In this section, we prove the soundness and completeness of the Model Evolution calculus.

4.1 Soundness

To prove that the calculus is sound we first prove that each of its derivation rules preserves a particular notion of satisfiability that we call *a-satisfiability*, after (Baumgartner, 2000).

Let us fix a constant a from the signature $\Sigma^{\text{sko}} \setminus \Sigma$ and consider the substitution $\alpha := \{v \mapsto a \mid v \in V\}$ mapping every parameter to a.[17] Given a literal L, we denote by L^a the literal $L\alpha$. Note that L^a is ground if, and only if, L is variable-free. Similarly, given a context Λ, we denote by Λ^a the set of *unit clauses* obtained from Λ by removing the pseudo-literal $\neg v$, replacing each literal L of Λ with L^a, and considering it as a unit clause. Finally, if σ is a substitution, we denote by σ^a the composed substitution $\sigma\alpha$. We point out for later that for all literals L and substitutions σ such that $(\mathcal{P}ar(L))\sigma \subseteq V$ (which includes all parameter-preserving substitutions), $L\sigma^a = L^a\sigma^a$.

We say that a sequent $\Lambda \vdash \Phi$ is *a-(un)satisfiable* iff the clause set $\Lambda^a \cup \Phi$ is (un)satisfiable in the standard sense—that is, it has (no) Herbrand models.

Lemma 4.1 *For each rule of the $\mathcal{ME}$ calculus, if the premise of the rule is a-satisfiable, then one of its conclusions is a-satisfiable as well.*

Proof. We prove the claim only for the rules Split, Assert, Resolve, and Close. For the other rules the claim holds trivially.

[17] Strictly speaking, α is not a substitution in the standard sense because $\mathcal{D}om(\alpha)$ is not finite. But this will cause no problems here.

Split) The premise of Split has the form $\Lambda \vdash \Psi$, while its conclusions have respectively the form $\Lambda, K \vdash \Psi$ and $\Lambda, \overline{K}^{\text{sko}} \vdash \Psi$. Suppose that $\Lambda \vdash \Psi$ is a-satisfiable. Now let $\mathbf{x} := (x_1, \ldots, x_n)$ be an enumeration of all the variables of K and note that K and K^a have exactly the same variables. Then consider the unit clause K^a (or, more explicitly, $\forall \mathbf{x}\, K^a$) and its negation $\neg\forall \mathbf{x}\, K^a$. Clearly, one of the two sets

$$S_1 := \Lambda^a \cup \{K^a\} \cup \Psi \text{ and } S_2 := \Lambda^a \cup \{\neg\forall \mathbf{x}\, K^a\} \cup \Psi$$

must be satisfiable. If S_1 is satisfiable, we have immediately that $\Lambda, K \vdash \Psi$ is a-satisfiable. If S_2 is satisfiable, then its Skolemized form $\Lambda^a \cup \{(\overline{K^a})^{\text{sko}}\} \cup \Psi$ is also satisfiable. Since $(\overline{K^a})^{\text{sko}} = (\overline{K}^{\text{sko}})^a$, as one can easily see, we then have that $\Lambda, \overline{K}^{\text{sko}} \vdash \Psi$ is a-satisfiable.

Assert) The premise of Assert has the form $\Lambda \vdash \Phi,\ L_1 \vee \cdots \vee L_n \vee L$, while its conclusion has the form $\Lambda, L\sigma \vdash \Phi,\ L_1 \vee \cdots \vee L_n \vee L$, where $L\sigma$ is parameter-free and not contradictory with Λ, $n \geq 0$, and σ is a context unifier of $L_1 \vee \cdots \vee L_n$ against Λ with an empty remainder. This means that there are fresh $K_1, \ldots, K_n \in_\simeq \Lambda$ such that σ is a simultaneous unifier of $\{\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}\}$ and $(\mathcal{P}ar(K_i))\sigma \subseteq V$ for all $i = 1, \ldots, n$.

Suppose $\Lambda \vdash \Phi,\ L_1 \vee \cdots \vee L_n \vee L$ is a-satisfiable, that is, $\Lambda^a \cup \Phi \cup \{L_1 \vee \cdots \vee L_n \vee L\}$ is satisfiable. Observing that $(\mathcal{P}ar(K_i))\sigma \subseteq V$ and L_i is parameter-free for each i, it is easy to see that σ^a is a simultaneous unifier of $\{\{K_1^a, \overline{L_1}\}, \ldots, \{K_n^a, \overline{L_n}\}\}$.

Since $K_i^a \in_\simeq \Lambda^a$ for each i, it follows from the soundness of resolution that $\Lambda^a \cup \Phi \cup \{L_1 \vee \cdots \vee L_n \vee L,\ L\sigma^a\}$ is satisfiable. Noting that $L\sigma^a = (L\sigma)^a$, we can then conclude that $\Lambda, L\sigma \vdash \Phi,\ L_1 \vee \cdots \vee L_n \vee L$ is a-satisfiable.

Resolve) The premise of Resolve has the form $\Lambda \vdash \Phi, L \vee C$, while its conclusion has the form $\Lambda \vdash \Phi, C$, and there is a most general unifier σ of $\{K, \overline{L}\}$ for some $K \in_\simeq \Lambda$ such that (i) $(\mathcal{P}ar(K))\sigma \subseteq V$, and (ii) $C\sigma = C$. Suppose $\Lambda \vdash \Phi,\ L \vee C$ is a-satisfiable, which means that $\Lambda^a \cup \Phi \cup \{L \vee C\}$ is satisfiable. It is easy to see that because of point (i) above and the fact that L is parameter-free, σ^a is a unifier of $\{K^a, \overline{L}\}$. Observing that $K^a \in_\simeq \Lambda^a$, it follows by the soundness of standard resolution that $\Lambda^a \cup \Phi \cup \{L \vee C,\ C\sigma^a\}$ is also satisfiable. By point (ii) above and the fact that C is parameter-free, we have that $C\sigma^a = (C\sigma)^a = C^a = C$. But this entails that $\Lambda^a \cup \Phi \cup \{C\}$ is satisfiable, and so $\Lambda \vdash \Phi, C$ is a-satisfiable.

Close) The premise of Close has the form $\Lambda \vdash \Phi, C$, while its conclusion

has the form $\Lambda \vdash \Box$, and there is a context unifier σ of C against Λ with an empty remainder. As $\Lambda \vdash \Box$ is a-unsatisfiable, we must show that $\Lambda \vdash \Phi, C$ is a-unsatisfiable as well. We do that by proving that $\Lambda^a \cup \{C\}$ is unsatisfiable.

Let $C = L_1 \vee \cdots \vee L_n$ for some $n \geq 0$. Since σ is a context unifier σ of C against Λ with an empty remainder, we know that there are fresh variants $K_1, \ldots, K_n \in_{\simeq} \Lambda$ such that σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$, and $(\mathcal{P}ar(K_i))\sigma \subseteq V$ for all $i = 1, \ldots, n$. Let us fix the literals $K_1, \ldots, K_n$.

Clearly, σ^a is a simultaneous unifier of $\{K_1, \overline{L_1}\}$, $\ldots$, $\{K_n, \overline{L_n}\}$. By an earlier observation we know that $K_i\sigma^a = K_i^a\sigma^a$ for all $i = 1, \ldots, n$. It follows that σ^a is a simultaneous unifier of

$$\{K_1^a, \overline{L_1}\},\ \{K_2^a, \overline{L_2}\},\ \ldots,\ \{K_n^a, \overline{L_n}\}.$$

This entails that $\{K_1^a, \ldots, K_n^a, L_1 \vee \cdots \vee L_n\}$ is unsatisfiable. From the fact that $K_1^a, \ldots, K_n^a \in_{\simeq} \Lambda^a$ it then immediately follows that $\Lambda^a \cup \{C\}$ is unsatisfiable. $\Box$

Proposition 4.2 (Soundness) *For all sets Φ_0 of parameter-free Σ-clauses, if Φ_0 has a refutation tree, then Φ_0 is unsatisfiable.*

Proof. Let $\mathbf{T}_0$ be a refutation tree of Φ_0. We prove below by structural induction that the root of any subtree of a refutation tree is a-unsatisfiable. This will entail in particular that $\neg v \vdash \Phi_0$, the root of $\mathbf{T}_0$, is a-unsatisfiable. The claim will then follow from the immediate fact that the sequent $\neg v \vdash \Phi_0$ is a-unsatisfiable iff Φ_0 is unsatisfiable.

Let $\mathbf{T}$ be a subtree of a refutation tree and let N be its root. If $\mathbf{T}$ is a one-node tree, N can only have the form $\Lambda \vdash \Box$, which is trivially a-unsatisfiable. If $\mathbf{T}$ has more than one node, we can assume by induction that all the children nodes of N are a-unsatisfiable. But then we can conclude that N is a-unsatisfiable as well by the contrapositive of Lemma 4.1. $\Box$

4.2 Fairness

As customary, we will prove the completeness of the calculus with respect to *fair derivations*. The specific notion of fairness that we adopt is defined formally in the following. For that, it will be convenient to describe a tree $\mathbf{T}$ as the pair $(\mathbf{N}, \mathbf{E})$, where $\mathbf{N}$ is the set of the nodes of $\mathbf{T}$ and $\mathbf{E}$ is the set of the edges of $\mathbf{T}$.

Each derivation $\mathcal{D}$ in the Model Evolution calculus determines a *limit tree* with respect to all the derivation trees in $\mathcal{D}$.

Definition 4.3 (Limit Tree) *Let $\mathcal{D} = (\mathbf{T}_i)_{i<\kappa}$ be a derivation, where $\mathbf{T}_i = (\mathbf{N}_i, \mathbf{E}_i)$ for all $i < \kappa$. We say that*

$$\mathbf{T} := (\bigcup_{i<\kappa} \mathbf{N}_i, \bigcup_{i<\kappa} \mathbf{E}_i)$$

is the limit tree of $\mathcal{D}$.

It is easy to show that a limit tree of a derivation $\mathcal{D}$ is indeed a tree. But note that it will not be a derivation tree unless $\mathcal{D}$ is finite.

Definition 4.4 (Persistency) *Let $\mathbf{T}$ be the limit tree of some derivation, and let $\mathbf{B} = (N_i)_{i<\kappa}$ be a branch in $\mathbf{T}$ with κ nodes. Let $\Lambda_i \vdash \Phi_i$ be the sequent labeling node N_i, for all $i < \kappa$. We define the following sets of* persistent context literals *and* persistent clauses*, respectively:*

$$\Lambda_{\mathbf{B}} := \bigcup_{i<\kappa} \bigcap_{i\leq j<\kappa} \Lambda_j \qquad \Phi_{\mathbf{B}} := \bigcup_{i<\kappa} \bigcap_{i\leq j<\kappa} \Phi_j$$

In words, a context literal is persistent in the considered branch $\mathbf{B}$ iff it appears in the context of some node and in the context of all the node's descendants (and similarly for persistent clauses).

Where Σ is the signature of the first clause set of a derivation, we will also consider the set $\Lambda^{\Sigma}_{\mathbf{B}}$ of all the Σ-literals in $\Lambda_{\mathbf{B}}$ (which excludes any literal with Skolem constants). Although, strictly speaking, $\Lambda_{\mathbf{B}}$ and $\Lambda^{\Sigma}_{\mathbf{B}}$ are not contexts because they may be infinite, for the purposes of the completeness proof we can treat them as such. We note that all the definitions introduced in Section 3.1 can be applied without change to $\Lambda_{\mathbf{B}}$ and $\Lambda^{\Sigma}_{\mathbf{B}}$ as well.

Fair derivations in the $\mathcal{ME}$ calculus are defined in terms of *exhausted branches.*

Definition 4.5 (Exhausted branch) *Let $\mathbf{T}$ be a limit tree, and let $\mathbf{B} = (N_i)_{i<\kappa}$ be a branch in $\mathbf{T}$ with κ nodes. For all $i < \kappa$, let $\Lambda_i \vdash \Phi_i$ be the sequent labeling node N_i. The branch $\mathbf{B}$ is* exhausted *iff for all $i < \kappa$, all of the following hold:*

(i) For all $C \in \Phi_{\mathbf{B}}$, if Split *is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C and productive context unifier σ such that $K \in_{\simeq} \Lambda^{\Sigma}_{\mathbf{B}}$ for every context literal K of σ, then there is a remainder literal L of σ and a j with $i \leq j < \kappa$ such that Λ_j produces L but does not produce $\overline{L}$.*

(ii) For all unit clauses $L \in \Phi_{\mathbf{B}}$, if Assert *is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause L, selected literal L and empty context unifier, then there is a j*

with $i \leq j < \kappa$ such that for any literal K with $L \geq K$, Λ_j produces K but does not produce $\overline{K}$.

(iii) *For all $C \in \Phi_{\mathbf{B}}$, Close is not applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C and a context unifier σ such that $K \in_{\simeq} \Lambda_{\mathbf{B}}$ for every context literal K of σ.*

(iv) $\Phi_i \neq \{\Box\}$.

It is worth noticing that Point (i) in Definition 4.5 does *not* require that Split be eventually applied with selected clause C and context unifier σ, for the branch to be exhausted. It only requires that the *intended effect* of applying Split with selected clause C and context unifier σ be achieved, namely that some literal L of $C\sigma$ is permanently produced *and* $\overline{L}$ *is not produced*. Only with the latter property it is guaranteed that the interpretation induced by the limit context assigns true to every ground Σ-instance of L and hence to every ground Σ-instance of $C\sigma$. A similar observation can be made about Point (ii) and the effect of applying Assert with selected unit clause L, namely that all Σ-instances of L and no Σ-instance of $\overline{L}$ is produced. To make Point (ii) operational, Lemma 8.6 below can be used to provide a sufficient condition. According to that lemma it is enough to add L to a context to achieve the desired effect.

Furthermore, as stated in Point (i) in Definition 4.5, concerning the context unifier σ mentioned there it suffices to consider only a persistent clause C and as context literals of σ only *persistent* context Σ-literals (and similarly in Point (iii) in Definition 4.5). That only Σ-*literals* from $\Lambda_{\mathbf{B}}$ need to be considered is justified, intuitively, by the fact that in order to determine the satisfiability of a input clause, which is built over the signature Σ, it is enough to find a Herbrand Σ-model for it.[18] That only *persistent* literals from $\Lambda_{\mathbf{B}}$ need to be considered results in an important consequence for the design of proof procedures: for completeness purposes, neither clauses nor contexts need to be stored over time; instead, it suffices to maintain a *current context* and a *current clause set*—in addition to backtracking information for recovering from Split applications that have led to a closed branch. See (Baumgartner et al., 2006a) for a proof procedure along these lines.

Definition 4.6 (Fairness) *A limit tree of a derivation is* fair *iff it is a refutation tree or it has an exhausted branch. A derivation is* fair *iff its limit tree is fair.*

We point out that fair derivations as defined above do exist and are computable for any set of (parameter-free) Σ-clauses. A proof of this fact can be given by adapting a technique used in (Baumgartner, 2000) to show the computability of fair derivations in FDPLL. Moreover, and similarly to FDPLL, fair deriva-

[18] Notice, however, that Close cannot be restricted to work with Σ-literals from the context only.

tions need not be searched. As we will see, the calculus is *proof convergent*, that is, if a set Φ of Σ-clauses is unsatisfiable, then *every* fair derivation of Φ is a refutation.

4.3 Completeness

Our proof that the Model Evolution calculus is complete is based on showing that the set $\Lambda^{\Sigma}_{\mathbf{B}}$ of persistent context Σ-literals along an exhausted branch $\mathbf{B}$ in a limit tree denotes a model of the clause set at the root of the tree. We provide below only a sketch of the completeness proof by proving just the main results. A complete proof of all the auxiliary results on properties of contexts and derivation rules stated here can be found in the appendix.

4.3.1 Properties of Contexts

We start with some general properties of contexts that we will use in the following.

Lemma 4.7 *Let Λ be a non-contradictory context. Then, for any literal L, Λ produces L or Λ produces $\overline{L}$ (or both).*

This lemma is needed in the proof of the following proposition.

Proposition 4.8 *Let Λ be a non-contradictory context and L a ground literal. If I_Λ satisfies L then Λ produces L.*

Observe that the converse of this proposition does not hold in general. This can be seen by considering the context $\{\neg v, P(a,u), \neg P(v,b)\}$. While the context produces both $P(a,b)$ and $\neg P(a,b)$, its induced interpretation satisfies only $P(a,b)$. The converse of Proposition 4.8 does hold for *positive* literals however.

Proof. If L is a positive literal then the claim follows trivially from Definition 3.8. Hence suppose that L is a negative literal. It is impossible that Λ produces $\overline{L}$, which is a positive literal, because then again by Definition 3.8 the interpretation I_Λ would satisfy $\overline{L}$, and thus not satisfy L. Now, since Λ does not produce $\overline{L}$, it follows by Lemma 4.7 that Λ produces L. □

4.3.2 Properties of Inference Rules

The following lemmas provide sufficient conditions for the applicability of the main rules of the calculus to a given context. We will refer to these conditions to prove the completeness of the calculus.

We need to characterize conditions under which Split is applicable. Their proof will be facilitated by the next two general lemmas. The first one shows how unification can be used to identify clause instances that are false in the interpretation induced by the current context.

Lemma 4.9 (Lifting Lemma) *Let Λ be a non-contradictory context. Let $C = L_1 \vee \cdots \vee L_n$ be a Σ-clause and $C\gamma$ a ground Σ-instance. If Λ produces $\overline{L_1}\gamma,\ldots,\overline{L_n}\gamma$, then there are fresh variants $K_1, \ldots, K_n \in_{\simeq} \Lambda^{\Sigma}$ and a substitution σ such that*

(1) σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,
(2) for all $i = 1, \ldots, n$, $L_i \gtrsim L_i\sigma \gtrsim L_i\gamma$,
(3) for all $i = 1, \ldots, n$, K_i produces $\overline{L_i}\sigma$ in Λ.

In Section 3 we mentioned that the calculus does not need to search for admissible context unifiers, and that any context unifier can be composed with a renaming substitution, obtained deterministically, such that the resulting context unifier is admissible. This fact is expressed by the following lemma.

Lemma 4.10 (Existence of Admissible Context Unifiers) *Let Λ be a context, C a clause and σ a context unifier of C against Λ. Then, there is a renaming ρ such that $\sigma' := \sigma\rho$ is an admissible context unifier of C against Λ with the same context literals as σ.*

It should be mentioned that the purpose of this lemma is just to show the existence of an admissible context unifier based on a possibly non-admissible context unifier. A realistic implementation would compute a clever renaming, one that tries to maximize the parameter-free literals in the resulting remainder.[19] For completeness purposes, however, *any* renaming that yields an admissible context unifier will do, as it will be clear from the proof of Proposition 4.16 below.

Now we can turn to the lemma stating conditions under which the Split rule is applicable. Roughly, Split is applicable if its selected clause admits a context unifier, it does not overlap with Assert, and Close is not applicable with the selected clause.

Lemma 4.11 (Split Applicability) *Let $\Lambda \vdash \Psi, C$ be a sequent with a non-contradictory context Λ, where C contains at least two literals. If all context unifiers of C against Λ have a non-empty remainder, and σ is an admissible context unifier of C against Λ such that Λ produces $\overline{L}$ for every remainder literal L of σ, then* Split *is applicable to $\Lambda \vdash \Psi, C$ with selected clause C and context unifier σ.*

[19] See (Fuchs, 2004) for a discussion of how to compute such a renaming.

The next lemma provides sufficient conditions for the applicabity of Assert to unit clauses, which is enough for completeness.

Lemma 4.12 (Assert Applicability) *Let $\Lambda \vdash \Psi$, L be a sequent with a non-contradictory context Λ. If all context unifiers of L against Λ have a non-empty remainder and there is an instance $L\sigma$ of L such that Λ produces $\overline{L}\sigma$, then* Assert *is applicable to $\Lambda \vdash \Psi$, L with selected clause L, selected literal L and the empty substitution as context unifier.*

4.3.3 Main Result

In this section, let Φ be a set of parameter-free Σ-clauses and assume that $\mathcal{D}$ is a fair derivation of Φ that is not a refutation. Observe that $\mathcal{D}$'s limit tree must have at least one exhausted branch. We denote this branch by $\mathbf{B} = (N_i)_{i<\kappa}$. Then, by $\Lambda_i \vdash \Phi_i$, we will always mean the sequent labeling the node N_i in $\mathbf{B}$, for all $i < \kappa$. (As a consequence, we will also have that $\Lambda_0 = \{\neg v\}$ and $\Phi_0 = \Phi$.)

Quite often we will appeal to the following compactness property of $\Lambda_{\mathbf{B}}$. By definition, $L \in \Lambda_{\mathbf{B}}$ holds iff there is an $i < \kappa$ such that $L \in \Lambda_j$ for all $j \geq i$ with $j < \kappa$.

Similarly, if $L \in_{\simeq} \Lambda_{\mathbf{B}}$ (meaning, by definition, that $L \simeq K$ for some literal $K \in \Lambda_{\mathbf{B}}$), then there is an $i < \kappa$ such that $K \in \Lambda_j$ for all $j \geq i$ with $j < \kappa$, which entails that $L \in_{\simeq} \Lambda_j$, for all $j \geq i$ with $j < \kappa$. More generally then, if $L_1, \ldots, L_n \in \Lambda_{\mathbf{B}}$ (or $L_1, \ldots, L_n \in_{\simeq} \Lambda_{\mathbf{B}}$) for some $n \geq 0$, then there is an $i < \kappa$ such that $L_1, \ldots, L_n \in \Lambda_j$ (or $L_1, \ldots, L_n \in_{\simeq} \Lambda_j$) for all $j \geq i$ with $j < \kappa$.[20]

Being non-contradictory is a fundamental property of the contexts manipulated by the calculus. Essentially, because the derivation rules can produce only non-contradictory contexts from non-contradictory contexts we obtain the following result:

Lemma 4.13 *$\Lambda_{\mathbf{B}}$ is not contradictory.*

The following lemma reduces productivity in the limit for the given branch to productivity in contexts within the branch.

Lemma 4.14 *Let K, L be two literals with $K \in \Lambda_{\mathbf{B}}$. If K produces L in $\Lambda_{\mathbf{B}}$, then there is an i such that for all $j \geq i$ with $j < \kappa$, $K \in \Lambda_j$ and K produces L in Λ_j.*

[20] It is easy to see that this index i can be determined by taking the maximum of the i-indices associated individually to the literals $L_1, \ldots, L_n$, as just described.

Lemma 4.15 (Close Applicability) *Let $C \in \Phi_{\mathbf{B}}$ and $i < \kappa$ such that* Close *is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C. Then, for some j with $i \leq j < \kappa$,* Close *is applicable to $\Lambda_j \vdash \Phi_j$ with selected clause C and a context unifier σ such that $K \in_{\simeq} \Lambda_{\mathbf{B}}$ for each context literal K of σ.*

The following proposition is fundamental as it states that that the calculus computes a model, in the limit, for any persistent clause set not containing the empty clause.

Proposition 4.16 *If $\Box \notin \Phi_{\mathbf{B}}$, then $I_{\Lambda_{\mathbf{B}}}$ is a model of $\Phi_{\mathbf{B}}$.*

Proof. Suppose ad absurdum that $\Phi_{\mathbf{B}}$ does not contain the empty clause, but $I_{\Lambda_{\mathbf{B}}}$ is not a model of $\Phi_{\mathbf{B}}$. This means that there is a ground Σ-instance $C\gamma$ of a clause $C = L_1 \vee \cdots \vee L_n$ with $n \geq 1$ from $\Phi_{\mathbf{B}}$ that is not satisfied by $I_{\Lambda_{\mathbf{B}}}$.

Since $C\gamma$ is not satisfied by $I_{\Lambda_{\mathbf{B}}}$, the literals $\overline{L_1}\gamma, \ldots, \overline{L_n}\gamma$ are all satisfied by $I_{\Lambda_{\mathbf{B}}}$. By Lemma 4.13, $\Lambda_{\mathbf{B}}$ is non-contradictory, and so by Proposition 4.8 it follows that $\Lambda_{\mathbf{B}}$ produces $\overline{L_1}\gamma, \ldots, \overline{L_n}\gamma$.

We distinguish two complementary cases below, depending on whether $n = 1$ or $n > 1$, and show that they both lead to a contradiction. In both cases we need the fact that Close is not applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C, for any $i < \kappa$. This follows immediately from Lemma 4.15: for, if Close were applicable to Λ_i with selected clause C, for some $i < \kappa$, then Close would be also applicable to Λ_j, for some $j \geq i$ with $j \leq \kappa$ and such that $K \in_{\simeq} \Lambda_{\mathbf{B}}$ for each context literal K of its context unifier. This, however, would contradict Definition 4.5-(iii).

($n = 1$) In this case, C consists of the single literal L_1. For $\Lambda_{\mathbf{B}}$ to produce $\overline{L_1}\gamma$ it must contain a literal K that produces $\overline{L_1}\gamma$ in $\Lambda_{\mathbf{B}}$. By Lemma 4.14 then there is an i such that

$$\text{for all } j \geq i \text{ with } j < \kappa,\ K \in \Lambda_j \text{ and } K \text{ produces } \overline{L_1}\gamma \text{ in } \Lambda_j. \qquad (1)$$

Since L_1 is a (unit) clause from $\Phi_{\mathbf{B}}$, there is a i' such that $L_1 \in \Phi_{j'}$ for all $j' \geq i'$. Without loss of generality assume that $i \geq i'$ (otherwise i' can be used instead of i in the sequel).

As shown above, Close is in particular not applicable to $\Lambda_i \vdash \Phi_i$ with selected clause L_1. Since $L_1 \in \Phi_i$, all context unifiers of L_1 against Λ_i have a non-empty remainder. Together with (1), this implies by Lemma 4.12 that Assert is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause L_1, selected literal L_1 and empty context unifier.

According to Definition 4.5-(ii) then, there is a $j \geq i$ with $j < \kappa$ such that for any literal L with $L_1 \geq L$, Λ_j produces L but does not produce $\overline{L}$. Recall

that clauses in sequents are parameter-free, which implies that $L_1 \geq L_1\gamma$. But then, taking $L = L_1\gamma$ we have a contradiction with assertion (1) above which implies that Λ_j produces $\overline{L_1}\gamma$.

($n > 1$) By the Lifting Lemma (Lemma 4.9), there are fresh p-variants $K_1, \ldots, K_n \in_\simeq \Lambda^\Sigma_{\mathbf{B}}$ and a substitution σ such that

(1) σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,
(2) for all $k = 1, \ldots, n$, $L_k \gtrsim L_k\sigma \gtrsim L_k\gamma$,
(3) for all $k = 1, \ldots, n$, K_k produces $\overline{L_k}\sigma$ in $\Lambda_{\mathbf{B}}$.

By Definition 3.9, σ is a productive context unifier of C against $\Lambda_{\mathbf{B}}$.

By Lemma 4.10, an admissible context unifier of C against $\Lambda_{\mathbf{B}}$ can be obtained as $\sigma' = \sigma\rho$, for some renaming ρthat has the same context literals $K_1, \ldots, K_n$ as σ.

Let $k \in \{1, \ldots, n\}$ and observe that a literal K produces a literal L in a context Λ iff K produces a variant of L in Λ. From the fact that K_k produces $\overline{L_k}\sigma$ in $\Lambda_{\mathbf{B}}$, we have that K_k produces $\overline{L_k}\sigma'$ in $\Lambda_{\mathbf{B}}$ as well. By applying Lemma 4.14 to every K_k and $\overline{L_k}\sigma'$ individually (for $k = 1, \ldots, n$), and taking the maximum of the indices i mentioned in the lemma's statement, we conclude that there is an i such that

$$\text{for all } j \geq i \text{ with } j < \kappa,\ K_k \in_\simeq \Lambda^\Sigma_j \text{ and } K_k \text{ produces } \overline{L_k}\sigma' \text{ in } \Lambda_j. \qquad (2)$$

By assumption, C is a clause of $\Phi_{\mathbf{B}}$. Hence, there is a i' such that $C \in \Phi_{j'}$ for all $j' \geq i'$. Without loss of generality suppose that $i \geq i'$ (otherwise i' can be used instead of i in the sequel).

As shown above, Close is in particular not applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C. Therefore, all context unifiers of C against Λ_i must have a non-empty remainder.

By (2) and the generality of k we have that $K_k \in_\simeq \Lambda^\Sigma_i$ produces $\overline{L_k}\sigma'$ in Λ_i for all $k = 1, \ldots, n$, and so, in particular, Λ_i produces all remainder literals of σ'. By Lemma 4.11 then, Split is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C and productive context unifier σ'. Recall that each literal K_k has a p-variant in $\Lambda^\Sigma_{\mathbf{B}}$, and by (2) it has one in Λ_i as well. Because of Definition 4.5-(i), there is a remainder literal L of σ' and a $j \geq i$ such that Λ_j produces L but Λ_j does not produce $\overline{L}$. However, this contradicts conclusion (2) above which also entails that Λ^Σ_j produces L. □

The completeness of the calculus is a consequence of Proposition 4.16. We state it here in its contrapositive form to underline the model computation abilities of $\mathcal{ME}$.

Theorem 4.17 (Completeness) *Let Φ be a parameter-free Σ-clause set, and let $\mathcal{D}$ be a fair derivation of Φ with limit tree $\mathbf{T}$. If $\mathbf{T}$ is not a refutation tree, then Φ is satisfiable; more specifically, for every exhausted branch $\mathbf{B}$ of $\mathbf{T}$, $I_{\Lambda_\mathbf{B}}$ is a model of Φ.*

Let $\top$ be the universally true clause. For every clause $C \in \Phi$, we define $C^0 := C$, and for all $i > 0$

$$C^i := \begin{cases} D & \text{if } C^{i-1} \text{ is of the form } L \vee D \text{ and Resolve is applied with selected clause } C^{i-1} \text{ and selected literal } L \text{ to } \Lambda_{i-1} \vdash \Phi_{i-1} \text{ to obtain } \Lambda_i \vdash \Phi_i \\ \top & \text{if } C^{i-1} \text{ is of the form } L \vee D \text{ and Subsume is applied with selected clause } C^{i-1} \text{ and selected literal } L \text{ to } \Lambda_{i-1} \vdash \Phi_{i-1} \text{ to obtain } \Lambda_i \vdash \Phi_i \\ C^{i-1} & \text{otherwise} \end{cases}$$

Observe that for all $i \geq 0$, $\{C^i \mid C \in \Phi\} = \Phi_i \cup \{\top\}$.

Proof. Let C be any clause in Φ and $\mathbf{B}$ an exhausted branch of $\mathbf{T}$.

It is enough to show that $I_{\Lambda_\mathbf{B}}$ is a model of C. Now, it is easy to see that there is a smallest j such that $C^i = C^{i-1}$ for all $i > j$ with $i < \kappa$, which means that C^j is either $\top$ or a persistent clause of $\mathbf{B}$. Let us fix that j. We show below by induction on i that $I_{\Lambda_\mathbf{B}}$ is a model of C^i for all $i \leq j$, from which it will immediately follow that $I_{\Lambda_\mathbf{B}}$ is a model of $C = C^0$.

($i = j$) If C^i is $\top$, $I_{\Lambda_\mathbf{B}}$ is trivially a model of C^i. Hence assume that C^i is a persistent clause of $\mathbf{B}$, that is, $C^i \in \Phi_\mathbf{B}$. By Proposition 4.16, it is enough to show that $\Phi_\mathbf{B}$ does not contain the empty clause. Assume by contradiction that it does, i.e., that $\Phi_\mathbf{B} = \Phi' \cup \{\Box\}$ for some clause set Φ'.

That $\Phi_\mathbf{B}$ contains the empty clause entails trivially that Φ_k contains the empty clause, for some $k \geq 0$ with $k < \kappa$. That is, there must be a k such that $\Lambda_k \vdash \Phi_k$ has the form $\Lambda_k \vdash \Phi'_k, \Box$. That $\Phi'_k = \emptyset$ holds is impossible by Definition 4.5-(iv). Hence suppose that $\Phi'_k \neq \emptyset$. But then, since the empty substitution is certainly a context unifier of $\Box$ against Λ_k with an empty remainder, Close is applicable to $\Lambda_k \vdash \Phi'_k, \Box$ with selected clause $\Box$, which is impossible by Definition 4.5-(iii). It follows that $I_{\Lambda_\mathbf{B}}$ is a model of C^j.

($i < j$) Assume by induction hypothesis that $I_{\Lambda_\mathbf{B}}$ is a model of C^{i+1}, and consider the following three cases, depending on the definition of C^{i+1}.

(i) If $C^i = C^{i+1}$, we can conclude immediately that $I_{\Lambda_\mathbf{B}}$ is a model of C^i.

(ii) If C^i is of the form $L \vee D$ and Resolve is applied with selected literal L to

$\Lambda_i \vdash \Phi_i$ to obtain $\Lambda_{i+1} \vdash \Phi_{i+1}$, then $C^{i+1} = D$. It follows immediately that $I_{\Lambda_\mathbf{B}}$ is a model of C^i.

(iii) If C^i is of the form $L \vee D$ and Subsume is applied with selected clause C^i to $\Lambda_i \vdash \Phi_i$ to obtain $\Lambda_{i+1} \vdash \Phi_{i+1}$, then $C^{i+1} = \top$. By the definition of Subsume, there is a $K \in \Lambda_i$ such that $K \geq L$. By Lemma 8.15, there is a $K' \in \Lambda_\mathbf{B}$ such that $K' \geq K$. It follows that there is a $K' \in \Lambda_\mathbf{B}$ such that $K' \geq L$.

Recalling that $C \in \Phi$ is a parameter-free Σ-clause and that, by definition, C^i is a sub-clause of C, we have that C^i is a parameter-free Σ-clause and that L is a parameter-free Σ-literal. From the fact that $K' \geq L$, it follows that K' is also parameter-free and that $K' \geq L\gamma$, for any grounding substitution γ. Let $L\gamma$ be any such ground Σ-instance. Now, since $K' \in \Lambda_\mathbf{B}$ and $K' \geq L\gamma$, we have by Lemma 8.6 that $\Lambda_\mathbf{B}$ produces $L\gamma$ but does not produce $\overline{L}\gamma$. It follows by definition of $I_{\Lambda_\mathbf{B}}$ that $I_{\Lambda_\mathbf{B}}$ satisfies $L\gamma$. Because $L\gamma$ was an arbitrary ground Σ-instance of L, we can deduce that $I_{\Lambda_\mathbf{B}}$ is a model of L, and so of C^i. □

When the branch $\mathbf{B}$ in Theorem 4.17 is finite, $\Lambda_\mathbf{B}$ coincides with the context Λ_n, say, in $\mathbf{B}$'s leaf. From a model computation perspective, this is a crucial point because it means that a model of the original clause set—or rather, a finite representation of it, Λ_n—is readily available at the end of the derivation; it does not have to be computed from the branch, as in other model generation calculi.

The calculus is proof confluent (Bibel, 1982): any derivation of an unsatisfiable clause set extends to a refutation. In fact, because of the strong completeness result in Theorem 4.17, the calculus satisfies an even stronger property, which we refer to as *proof convergence*.

Corollary 4.18 (Proof Convergence) *Let Φ be a a parameter-free clause set over the signature Σ. If Φ is unsatisfiable, then every fair derivation of Φ is a refutation.*

In practical terms, the above corollary implies that as long as a derivation strategy guarantees fairness, the order of application of the rules of the calculus is irrelevant for proving an input clause set unsatisfiable, giving to the $\mathcal{ME}$ calculus the same kind of flexibility enjoyed by the DPLL calculus at the propositional level.

5 Implementation

At the theoretical level, the development of the $\mathcal{ME}$ calculus was motivated by the desire to close an existing gap in the theorem proving landscape and provide a proper lifting to first-order logic of a popular refutation method for propositional logic, DPLL. At a practical level, the calculus was also motivated by the conjecture that the very successful improvements developed by the SAT community for DPLL could be lifted to a suitable first-order version of it, and prove themselves similarly effective. To verify such a conjecture we have devised a proof procedure for the $\mathcal{ME}$ calculus and turned it into an implementation, the *Darwin* theorem prover. [21]

We have evaluated *Darwin* experimentally over the TPTP problem library (Sutcliffe and Suttner, 1998), comparing it with state-of-the-art theorem provers based on other calculi for first-order logic. Our experiments have shown that the $\mathcal{ME}$ calculus lends itself to competitive implementations for first-order logic without equality. In particular, *Darwin* is currently very competitive for input problems with a large percentage of non-Horn clauses. Furthermore, it is the best prover for function-free clause sets, which correspond to problems belonging to Bernay-Schönfikel class, for which *Darwin* is in fact a decision procedure.

In this section we describe the main aspects of *Darwin*'s proof procedure and implementation. For a more detailed account of *Darwin*'s general architecture, proof procedure, heuristics, and implementation details, as well as a detailed experimental evaluation, we refer the reader to (Baumgartner et al., 2006a).

Iterative Deepening Proof Procedure

Similarly to the DPLL procedure, *Darwin*'s proof procedure can be seen as exploring in a depth-first fashion the limit tree of a derivation in the calculus. Since the $\mathcal{ME}$ calculus is refutationally complete only for *fair* derivations, the proof procedure must make sure it gives rise only to fair derivations. This is achieved by performing a sort of iterative deepening search, however not on the depth of the search tree but of the *term depth* of certain literals, a complexity measure based on the depth of a term's tree representation.

Specifically, at any moment *Darwin* maintains a current context and clause set, corresponding to a node of the derivation tree, and a current set of *candidate literals*, literals that can be added to the context by an Assert or Split application. The proof procedure chooses for addition to the context only

[21] This latter work was in collaboration with Alexander Fuchs, *Darwin*'s main developer.

among those candidate literals whose term depth does not exceed a current term depth bound. Since by design of the inference rules it is impossible for a context to contain two or more p-variants of the same literal, this selection strategy implies the termination of any exhaustive sequence of inference rule applications under the term depth bound. [22]

After applying exhaustively all inferences rules with respect to the current bound without being able to close the current branch of the derivation tree, *Darwin*'s proof procedure checks whether the branch is *incomplete*. This is the case if during the generation of the branch a candidate literal was computed that exceeded the current depth bound. If the current branch is not incomplete it denotes a model of the input set, and the proof procedure reports that. Otherwise, the procedure behaves according to one of several strategies, as initially specified by the user. With the simplest of these strategies, the procedure just restarts the derivation from scratch, but with an increased depth bound.

Backtracking

In exploring a derivation tree, *Darwin*'s proof procedure generates a choice point for each (left) application of the Split rule. The depth-first exploration, within the term depth bound, of the derivation tree is then achieved by backtracking to a previous choice point every time a branch is closed. Instead of always going back to the most recent choice point, *Darwin* implements *backjumping*, a more effective form of chronological backtracking that takes into account dependencies between choice points. The idea of backjumping is best explained in terms of the calculus: suppose the derivation subtree below a left node introduced by a Split rule application is closed *and* the literal added on the left conclusion by that application is not needed to establish that the subtree is closed. Then, the Split rule application can be viewed as not being carried out at all. The proof procedure thus may skip the corresponding choice point on backtracking and proceed to the previous one.

Backjumping is well known to be one of the most effective improvements for DPLL-based SAT solvers. Its implementation for $\mathcal{ME}$ is not too difficult and relies on keeping track of which context literals and clauses are involved in particular in Assert and Close rule applications. Backjumping is an example of a successful propositional technique that directly lifts to the proof procedure of *Darwin*.

[22] This termination property is not immediately obvious because of the infinite supply of Skolem constants that can be used in Split inferences. Referring back to Example 3.13, however, where we argued that no branch can contain more than one Skolemized version of the same literal, one can see that Split inferences are not problematic in this regard.

Darwin also features *dynamic backtracking* (Ginsberg, 1993), a sophisticated form of non-chronological backtracking. See (Baumgartner et al., 2006a) for more details.

Conflict-based Learning

Another major conceptual improvement in DPLL-based solvers in the last years has been *lemma learning*, a mechanism for generating new propositional clauses that prevent later in the search combinations of split decisions that have already led to closed subtrees in the derivation.

Something similar can be done in $\mathcal{ME}$-based prover by analyzing the sequence of rule applications of a closed branch. The analysis determines which of the Split inferences along the branch were really relevant in allowing the application of Close and saves this information so that the same split choices, or similar choices that would also lead to a conflict, are avoided later in the search. As in DPLL SAT solvers, a convenient way to save such information is in the form of a clause added to the clause set so that applications of Assert with this clause block prevent the repetition later of the Split inference that caused the conflict.

In contrast to backjumping, adapting DPLL learning methods to an $\mathcal{ME}$-based prover is not immediate, first because one needs to lift properly to the first-order level the lemma generation process so that it generates lemmas that do prune the search space, and second because any such process,, when carried over at a first-order level, is bound to add a significant computational overhead that can offset in practice the advantages of pruning. On the other hand, working at the first-order level offers the enticing possibility of achieving learning in the more proper sense of word, with lemmas helping prune also areas of the search space that do not duplicate previously explored ones.

Darwin successfully implements two variants of a learning mechanism that lifts the main features of learning methods for DPLL procedures. In both variants, lemmas are generated by a guided resolution process that starts with the selected clause of the Close inference closing a branch and uses a selected number of clauses involved in Assert inference along the branch. A description of these variants and their positive effects on *Darwin*'s performance is given in (Baumgartner et al., 2006b).

Context Unifiers and Selection Heuristics

The central operation in *Darwin*'s proof procedure is the computation of all possible context unifiers of current clauses against the current context. The system computes context unifiers of current clauses to identify literals that

can be added to the context by the Split rule (Split candidates), and context unifiers of subsets of input clauses to identify literals that can be added by the Assert rule (Assert candidates). The set of such context unifiers is built incrementally, but exhaustively, as the context grows. With this technique, all possible Assert candidates can be eagerly added to a context, which correspond to the eager unit propagation mechanism of DPLL. Also, all theoretically necessary Split candidates at any point are available for inspection, allowing the implementation of a heuristic selection mechanisms for choosing the *best* literals to split with. The current selection heuristics in *Darwin* is based on several considerations, such as whether a candidate contains variables only or whether adding it will cause no proper branching in the derivation tree.

Darwin uses special data structures and a few dynamic programming techniques to compute and store context unifiers, with the goal of limiting runtime and memory requirements. In addition, it uses term indexing techniques on context to support fast checking of the preconditions of the Split, Assert, and Subsume rules. [23] More details on context unification in *Darwin* can be found again in (Baumgartner et al., 2006a).

6 Related work

Approaches that have features in common with $\mathcal{ME}$ come from the following four categories: *first-order DPLL methods*, *instance-based methods*, *Resolution methods* and *Tableau methods*.

6.1 First-Order DPLL Methods

A "lifted" version of the DPLL method has been described in the early textbook on automated reasoning by Chang and Lee (Chang and Lee, 1973). It uses the device of pseudosemantic trees which, like $\mathcal{ME}$, realize splits at the non-ground level. Nethertheless, the pseudosemantic tree method is very different from our approach: in sharp contrast to $\mathcal{ME}$, a variable is treated rigidly there, i.e. as a placeholder for a (one) not-yet-known term. [24] Section 6.4 below discusses rigid variable methods, and what is said there applies to the method in (Chang and Lee, 1973) as well.

A more recent attempt to incorporate first-order reasoning into DPLL has

[23] These preconditions require, in essence, to search the context for literals that unify with, subsume, or are subsumed by a given literal.
[24] However the term "rigid" is not used there, as it had not been yet introduced at the time the book (Chang and Lee, 1973) was written.

been made in (Parkes, 1999; Ginsberg and Parkes, 2000). Instead of instantiating the input clauses into ground ones before applying a DPLL method, the modified DPLL method in (Parkes, 1999; Ginsberg and Parkes, 2000) directly takes advantange of the (first-order) clauses of the input clause set. More specifically, these clauses are used for *unit propagation* on the basis of the current partial (propositional) model candidate. In our terms, this corresponds to working with ground contexts and using the Assert rule similarly as in $\mathcal{ME}$, but always adding a *ground* instance of the Assert literal to the context. In (Parkes, 1999; Ginsberg and Parkes, 2000) it is proven that unit propagation of this kind itself includes an NP-complete search problem (which, of course, also applies to $\mathcal{ME}$). However, it is argued that at the same time the much more compact representations enabled by using first-order logic may well pay off. A more fundamental difference between our approach and that in (Parkes, 1999; Ginsberg and Parkes, 2000) is that the latter is restricted to checking the satisfiability of quantifier formulas in finite models only, whereas $\mathcal{ME}$ works with full first-order logic.

The closest relative of the $\mathcal{ME}$ calculus is the FDPLL calculus developed by one of us (Baumgartner, 2000). As mentioned in the introduction, $\mathcal{ME}$ is loosely based on FDPLL. More precisely, the $\mathcal{ME}$ calculus can be specialized to the core FDPLL calculus by

(1) removing the Subsume, the Resolve and the Compact inference rules (which are optional in $\mathcal{ME}$), and
(2) restricting Split to use only admissible context unifiers with a variable-free remainder. [25]

In terms of the present paper, the core calculus of FDPLL does not have simplification rules and does not deal with variables – it just uses parameters. However, in (Baumgartner, 2000) an extension of the core FDPLL calculus to include reasoning with variables is sketched. Contrary to $\mathcal{ME}$, mixed literals are not allowed, and so the literals used there for splits are of the same type—variable-free or parameter-free (or both). Even ignoring this aspect, $\mathcal{ME}$ is much stronger than that version of FDPLL. Expressed in $\mathcal{ME}$ terms, the rules mentioned under (1) above are still not available in FDPLL. Furthermore, admissible context unifiers are defined to be those that either satisfy the restriction (2) above or have a unit remainder and use only parameter-free context literals. In resolution terminology, FDPLL mimics unit-resulting resolution, roughly, on the Horn clause subset of the input clause set.

The impact of the more limited capabilities of FDPLL can be seen by looking at some examples. For instance, if the current context is just $\Lambda = \{\neg v\}$ and there is a given clause $P(x) \vee Q(y) \vee R(z)$, FDPLL will consider the clause

[25] Using only such admissible context unifiers preserves completeness in $\mathcal{ME}$.

instance $P(u) \vee Q(v) \vee R(w)$ and split based on its literals, which contain *parameters*. In contrast, $\mathcal{ME}$ will in essence carry out a case analysis according to the three literals $P(x)$, $Q(y)$ and $R(z)$, which have the advantage of containing variables instead of parameters. [26]

As another example consider the context $\Lambda = \{\neg v, P(u_1, u_2), Q(x, a, z)\}$ and the clause $R(y, z) \vee \neg P(x, x) \vee \neg Q(x, y, z)$. Based on the admissible context unifier $\sigma = \{v \mapsto R(a, z),\ u_1 \mapsto u_2,\ x \mapsto u_2,\ y \mapsto a\}$ [27] whose sole remainder literal is $R(a, z)$, the Split rule is applicable in $\mathcal{ME}$. A comparable inference step is not possible with FDPLL, as one of the involved context literals (namely, $P(u_1, u_2)$) is not parameter-free.
In conclusion, due to the presence of the simplification inference rules Subsume, Resolve and Compact and the better treatment of variables, the $\mathcal{ME}$ calculus improves significantly on FDPLL.

6.2 Instance-Based Methods

Besides the FDPLL calculus, $\mathcal{ME}$ is related to the family of *instance-based methods*. Proof search in instance-based methods relies on maintaining a set of instances of input clauses and analyzing it for satisfiability until completion. We point out that $\mathcal{ME}$ is *not* an instance-based method in this sense, as clause instances are used only temporarily within the Split inference rule and can be forgotten after the split has been carried out.

The contemporary stream of research on instance-based methods was initiated with the Hyperlinking calculus (HL) (Lee and Plaisted, 1992). This calculus is based on the idea of steadily growing a set of instances of input clauses in an intelligent way, and regularly testing it for propositional unsatisfiability by an integrated DPLL procedure. An important conceptual difference between $\mathcal{ME}$ and HL is that the latter *includes* a DPLL procedure but does not (directly) *extend* it to first-order clause logic.

The current successor of HL is the Ordered Semantic Hyperlinking calculus (OSHL) (Plaisted and Zhu, 1997, 2000). OSHL has many interesting features, for instance "semantical guidance" by assuming a procedural representation of an (any) interpretation, that just has to be capable to decide if a given ground literal is true in the interpretation. As in $\mathcal{ME}$, the main operation in OSHL is to detect an instance of a clause that is false in a current interpretation, and

[26] As discussed in Section 3, the more variables a context literal has in place of parameters, the more constraints it imposes on a derivation.
[27] For simplicity, and without loss of generality in this case, we are not taking fresh variants of the context literals in computing the context unifiers.

then repair the interpretation (in the sense given here). However, unlike $\mathcal{ME}$, the repairs are carried out through *ground literals*.

Some instance-based calculi have been formulated within the (clausal) tableau framework. Similar to $\mathcal{ME}$, and unlike HL and its successors, they extend a propositional method—this time propositional clausal tableaux—to the first order level without resorting to a separate propositional solver.

The initial work in this direction is Billon's disconnection method (Billon, 1996), followed by the calculus described in (Baumgartner, 1998) which relates to the disconnection method pretty much in the same way as the hyper-resolution calculus relates to the resolution calculus.

The disconnection method has been picked up by Letz and Stenz for further improvements and efficiently implemented into a competitive prover (Stenz, 2002). The disconnection calculus, as they call it, uses clausal tableau as the primary data structure. The tableau structure represents an exhaustive search through all possible connections between literals in clauses; the (single) inference rule extends the current tableau by two clause instances found via a connection on the branch. Thus, the disconnection calculus is conceptually rather different to $\mathcal{ME}$ in that the main derivation rule there is based on resolving *pairs* of complementary literals from *two* clauses, whereas $\mathcal{ME}$'s splitting rule is based on evaluating *all* literals of a *single* clause against a candidate model.

In (Letz and Stenz, 2001), further improvements on the disconnection calculus are discussed. Among them is a dedicated inference rule for deriving unit clauses. Interestingly, all variables in such a derived unit clause have to be identified for soundness reasons. Still, this inference rule represents a limited and local form of "lemma learning" that does not have a direct counterpart in the $\mathcal{ME}$ calculus. Directly comparable to $\mathcal{ME}$ are the more recent developments introduced in (Stenz and Letz, 2004). The disconnection calculus there works with *two* kinds of variables: *shared* and *local* ones. The shared variables correspond to what we call parameters—which are present in one form or the other in all instance-based methods. A variable qualifies as a local variable if it occurs in *just one* literal of a clause instance considered for tableau expansion. There is a roughly corresponding requirement in what we call admissible context unifiers, where a variable occuring in a (remainder) literal must not occur in another remainder literal. The correspondence is not perfect, however, as clause instances for tableau expansion are derived differently from $\mathcal{ME}$ in (Stenz and Letz, 2004). Nevertheless, the concepts are comparable, and their use in subsumption tests is similar.

Two variants of an instance-based method are described by Hooker *et al.* (Hooker et al., 2002). One of them, the "Primal Approach" seems to be very similar to the disconnection method although, unfortunately, the relation with

this method is not made explicit in (Hooker et al., 2002). The other variant, the "Dual Approach", differs from the former by the presence of *auxiliary* clauses of the form $K \rightarrow L$ generated during the proof search, where (K, L) is a connection of literals occurring in the current clause set. No simplification mechanisms is described, such as for instance those based on unit propagation rules. Both methods compare to $\mathcal{ME}$ in the same way as the disconnection method, discussed above.

Finally, a rather abstract framework for instance-based calculi which also admits simplification techniques is described in (Ganzinger and Korovin, 2003). The underlying idea is to work with a propositional abstraction of a candidate model for the input clause set. That abstraction is used to guide the search for a refutation in a rather flexible way. As with the Hyperlinking calculi, the perhaps most significant difference between $\mathcal{ME}$ (or the disconnection calculus for that matter) and the framework in (Ganzinger and Korovin, 2003) is that the latter relies on the execution of propositional satisfiability tests. This has the advantage that off-the-shelf SAT solvers can be readily used for those tests. On the other hand, it is unclear how to exploit first-order features like our variables (or the "local variables" of the disconnection calculus) when relying on a propositional solver. However, as shown in (Ganzinger and Korovin, 2003) by treating certain variable occurences in a special way it is sometimes possible to replace the SAT solver by a decision procedure for some fragment of first-order logic. This way, such a decision procedure can sometimes be lifted to work on input formulas outside its fragment.

6.3 Resolution Methods

Resolution calculi are conceptually very different from the $\mathcal{ME}$ calculus, which makes a comparison difficult. A common feature is model generation. Modern completeness proofs for resolution calculi provide a method for constructing a model of any saturated clause set not containing the empty clause (see (Bachmair and Ganzinger, 2001)). However, this is a *conceptual construction*, and non-trivial postprocessing is necessary to extract a model in practive from a failed refutation (but see (Ganzinger et al., 1997)). Typically, a model is computed by enumerating all true *ground* literals, thereby interleaving this enumeration with calls to the resolution procedure again in order to determine the "next" ground literal (Fermüller and Leitsch, 1993, 1996).

6.4 Tableau Methods

Apart from clausal tableau methods that are also instance-based methods, which we have already discussed, clausal tableau calculi in general are related

to $\mathcal{ME}$ for encoding, like $\mathcal{ME}$, a model of the given clause set in an exhausted open branch. For comparison purposes, it is useful to classify these calculi according to whether they treat their variables *rigidly* or they instantiate them by ground terms.

In tableau calculi with *rigid variables*, a variable in a tableau is a placeholder for a (one) not-yet-known term (see e.g. (Fitting, 1990) for a basic version). The meaning of rigid variables can also be captured by constraints (Peltier, 1999; Giese, 2001; van Eijck, 2001). Although (most) tableau calculi are proof confluent, practically usable fair strategies to achieve proof convergence (cf. Corollary 4.18) are hard to devise (but see (Beckert, 2003)).

Another drawback of the rigid variables is that they make it difficult to find useful redundancy mechanisms such as, for instance, one that would in general prevent to have an unbounded number of variants of the same literal along a branch. More concretely, given the unit clause $P(x)$, say, there seems to be no simple justification for not enumerating variants $P(x), P(x'), P(x''), \ldots$ of $P(x)$ along a branch. In fact, in general, and contrary to $\mathcal{ME}$, one variant is not enough for completeness, and it is difficult to (automatically) determine sharp bounds on their number—a discussion on various options in the design of rigid tableau calculi, including constraint-based approaches, can be found in (Giese, 2002).

Ground-level tableau calculi avoid the problems with rigid variables by resorting to the propositional level. While analytic tableau with the *classical* γ-rule do not seem a suitable basis to build competitive theorem provers, there are structural refinements for clause logic that are also related to hyper resolution (Manthey and Bry, 1988; Baumgartner et al., 1996) or to Ordered Semantic Hyper Tableaux (Yahya and Plaisted, 2002). However, these methods suffer from an, generally unavoidable, don't-know nondeterminism, which can lead to an enumeration of the whole Herbrand base along a branch.

Finally, in contrast to $\mathcal{ME}$, tableau calculi (which includes the disconnection calculus) branch on subformulas, or, the literals of a clause in the clausal case, as opposed to complementary literals. For the propositional case it is easy to see that branching on complementary literals as done in $\mathcal{ME}$ is more general than branching on clauses. In fact, each branching on a clause with n literals can be simulated by n splits with complementary literals. Furthermore, some improvements like factoring (see (Letz et al., 1994)) are *automatically* realized by the branching on complementary literals approach. A systematic investigation on how this fact exactly carries over to the first-order case—i.e. $\mathcal{ME}$ vs. certain clausal tableau calculi—is left for future work.

7 Conclusions

In this paper we introduced the Model Evolution ($\mathcal{ME}$) calculus, a refutation calculus for first-order clausal logic. The $\mathcal{ME}$ calculus extends the well-known (propositional part of the) DPLL procedure to first-order logic by means of unification-based, first-order versions of DPLL's main inference rules. In fact, the calculus is a lifting of DPLL to the first-order level and reduces to DPLL in case of ground clause sets. Compared to its most immediate predecessor, FDPLL (Baumgartner, 2000), $\mathcal{ME}$ is a more faithful lifting of DPLL because it also lifts DPLL's inference rules for unit propagation. Except for termination, which is unachievable in the general first-order case, the calculus enjoys the same theoretical properties of DPLL: it is sound, complete for fair derivations, and proof convergent. The latter property in particular implies that any fair proof procedure for $\mathcal{ME}$ is guaranteed to produce a refutation for unsatisfiable input clause sets. Also like DPLL, the calculus is such that terminating derivations of a satisfiable clause set effectively compute a model of the set.

Our experience with implementing the $\mathcal{ME}$ calculus confirms the validity of the idea of lifting DPLL, and its improvements, to the (full) first-order level. The performance of the *Darwin* theorem prover for $\mathcal{ME}$ compares favorably with much more mature state-of-the-art provers over the whole class of TPTP problems without equality, and is superior than most of them over selected non-trivial subclasses. Furthermore, much of the potential of the lifted DPLL idea is still untapped given that at the moment *Darwin* implements only some of the major improvements developed for DPLL. In general, there is still room for further research on how to properly lift and adapt more DPLL improvements to first-order logic, and or on devising new improvements directly for the first-order level.

7.1 Further Research

We envision several directions for future theoretical work on the $\mathcal{ME}$ calculus and its implementations. A few of them are listed below.

Semantical Guidance

As presented here, the $\mathcal{ME}$ calculus always starts with an interpretation that assigns false to all ground atoms. By simply replacing the pseudo-literal $\neg v$ by v, it is possible to have the calculus start instead with a complementary initial interpretation. The kind of semantic guidance achieved in OSHL (Plaisted and Zhu, 2000) by means of a user-defined initial interpretation, is trivially achievable in $\mathcal{ME}$ when this interpretation is denotable by a context: one

simply starts the derivation with that context. More work is needed, however, to allow $\mathcal{ME}$ to start with arbitrary interpretations, in particular, ones that cannot be encoded into a (finite) context.

Equational Theories and Equality

In many theorem proving applications, an explicit treatment of equality is mandatory. To our knowledge there are only two instance-based methods that have been extended with dedicated equality inference rules for full equational clausal logic. One is the disconnection calculus in (Letz and Stenz, 2002) and the other is the instance-based method in (Ganzinger and Korovin, 2003). Both of the extended calculi are based on superposition-style inference rules (Bachmair and Ganzinger, 1998), with the second one also including rather powerful redundancy criteria. Our initial results on extension of $\mathcal{ME}$ with inference rules for equality reasoning are presented in (Baumgartner and Tinelli, 2005). Further work will concentrate on building an efficient and competitive implementation of the new calculus.

$\mathcal{ME}$ *as a decision procedure*

A deduction system capable of deciding relevant classes of formulas is usually of greater practical interest than a mere refutation system (e.g., to disprove false "theorems" in a software verification context).
The $\mathcal{ME}$ calculus is guaranteed to terminate for clauses resulting from the translation to clausal form of conjunctions of Bernays-Schönfinkel formulas [28] and hence gives a decision procedure [29]. The same holds for many instance-based methods (see Section 6.2), but, interestingly, not for any known refinement of the resolution calculus. On the other hand, there are refinements of the resolution calculus that decide (see (Fermüller et al., 2001)) classes of formulas not obviously decidable by $\mathcal{ME}$ or other instance based methods. It would be interesting to investigate ways to refine $\mathcal{ME}$ so that it can decide some of these classes or other classes not currently decided by resolution methods.

Acknowledgments

We thank Alexander Fuchs for his helpful comments on this paper and for all his outstanding work on the *Darwin* prover. We are also thankful to Thomas

[28] Such clauses contain no function symbols, but no other restrictions apply.
[29] This is an easy consequence of the fact that there are no two p-variants of a literal on any sequent derivable by $\mathcal{ME}$ (cf. Lemma 8.14).

Hillenbrand, Yevgeny Kazakov and Konstantin Korovin for being always open for discussions on $\mathcal{ME}$.

References

Bachmair, L., Ganzinger, H., 1998. Chapter 11: Equational Reasoning in Saturation-Based Theorem Proving. In: Bibel, W., Schmitt, P. H. (Eds.), Automated Deduction. A Basis for Applications. Vol. I: Foundations. Calculi and Refinements. Kluwer Academic Publishers, pp. 353–398.

Bachmair, L., Ganzinger, H., 2001. Resolution Theorem Proving. In: Robinson, A., Voronkov, A. (Eds.), Handbook of Automated Reasoning. North Holland.

Baumgartner, P., 1998. Hyper Tableaux — The Next Generation. In: de Swaart, H. (Ed.), Automated Reasoning with Analytic Tableaux and Related Methods. Vol. 1397 of Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Heidelberg, New-York, pp. 60–76.

Baumgartner, P., 2000. FDPLL – A First-Order Davis-Putnam-Logeman-Loveland Procedure. In: McAllester, D. (Ed.), CADE-17 – The 17th International Conference on Automated Deduction. Vol. 1831 of Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Heidelberg, New-York, pp. 200–219.

Baumgartner, P., Fuchs, A., Tinelli, C., 2006a. Implementing the model evolution calculus. International Journal of Artificial Intelligence Tools 15 (1), 21–52.

Baumgartner, P., Fuchs, A., Tinelli, C., 2006b. Lemma learning in the model evolution calculus. In: Hermann, M., Voronkov, A. (Eds.), Proceedings of the 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR'06), Phnom Penh, Cambodia. Vol. 4246 of Lecture Notes in Computer Science. Springer, pp. 572–586.

Baumgartner, P., Furbach, U., Niemelä, I., 1996. Hyper Tableaux. In: Proc. JELIA 96. No. 1126 in Lecture Notes in Artificial Intelligence. European Workshop on Logic in AI, Springer.

Baumgartner, P., Tinelli, C., 2005. The model evolution calculus with equality. In: (Nieuwenhuis, 2005), pp. 392–408.

Beckert, B., 2003. Depth-first proof search without backtracking for free-variable clausal tableaux. Journal of Symbolic Computation 36, 117–138.

Bibel, W., 1982. Automated Theorem Proving. Vieweg.

Billon, J.-P., 1996. The Disconnection Method. In: Miglioli, P., Moscato, U., Mundici, D., Ornaghi, M. (Eds.), Theorem Proving with Analytic Tableaux and Related Methods. No. 1071 in Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Heidelberg, New-York, pp. 110–126.

Chang, C., Lee, R., 1973. Symbolic Logic and Mechanical Theorem Proving. Academic Press.

Davis, M., Logemann, G., Loveland, D., Jul. 1962. A machine program for theorem proving. Communications of the ACM 5 (7), 394–397.
Davis, M., Putnam, H., Jul. 1960. A computing procedure for quantification theory. Journal of the ACM 7 (3), 201–215.
Eder, E., March 1985. Properties of Substitutions and Unifications. Journal of Symbolic Computation 1 (1).
Fermüller, C., Leitsch, A., 1993. Model building by resolution. In: Börger, E., Jäger, G., Kleine-Büning, H., Martini, S., Richter, M. (Eds.), Computer Science Logic – CSL'92. Vol. 702 of Lecture Notes in Computer Science. Springer Verlag, Berlin, Heidelberg, New-York, pp. 134–148.
Fermüller, C., Leitsch, A., 1996. Hyperresolution and automated model building. Journal of Logic and Computation 6 (2), 173–230.
Fermüller, C., Leitsch, A., Hustadt, U., Tammet, T., 2001. Resolution Decision Procedures. Elsevier and MIT Press, pp. 1791–1850.
Fermüller, C., Pichler, R., 2005. Model representation via contexts and implicit generalizations. In: (Nieuwenhuis, 2005), pp. 409–423.
Fitting, M., 1990. First Order Logic and Automated Theorem Proving. Texts and Monographs in Computer Science. Springer.
Fuchs, A., 2004. Darwin: A Theorem Prover for the Model Evolution Calculus. Master's thesis, University of Koblenz-Landau.
Ganzinger, H., Korovin, K., 2003. New directions in instance-based theorem proving. In: LICS - Logics in Computer Science.
Ganzinger, H., Meyer, C., Weidenbach, C., Jul. 1997. Soft typing for ordered resolution. In: McCune, W. (Ed.), Automated Deduction — CADE 14. LNAI 1249. Springer-Verlag, Townsville, North Queensland, Australia, pp. 321–335.
Giese, M., 2001. Incremental closure of free variable tableaux. In: Proc. International Joint Conference on Automated Reasoning. Vol. 2083 of Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Heidelberg, New-York.
Giese, M., 2002. A model generation style completeness proof for constraint tableaux with superposition. In: Egly, U., Fermu"ller, C. G. (Eds.), Proc. Intl. Conf. on Automated Reasoning with Analytic Tableaux and Related Methods, Copenhagen, Denmark. Vol. 2381 of LNCS. Springer-Verlag.
Ginsberg, M. L., 1993. Dynamic backtracking. Journal of Artificial Intelligence Research 1, 25–46.
Ginsberg, M. L., Parkes, A. J., 2000. Satisfiability algorithms and finite quantification. In: Cohn, A. G., Giunchiglia, F., Selman, B. (Eds.), Principles of Knowledge Representation and Reasoning: Proceedings of the Seventh International Conference (KR'2000). Morgan Kauffman, pp. 690–701.
Goldberg, E., Novikov, Y., 2002. Berkmin: A fast and robust sat solver.
Hooker, J., Rago, G., Chandru, V., Shrivastava, A., 2002. Partial Instantiation Methods for Inference in First Order Logic. Journal of Automated Reasoning 28 (4), 371–396.
Jackson, D., 2000. Automating first-order relational logic. In: Proceedings of

the ACM SIGSOFT Conference on Foundations of Software Engineering. pp. 130–139.

Joshi, R., Nelson, G., Randall, K., July 2001. Denali: a goal-directed superoptimizer. Tech. rep., Compaq SRC.

Lee, S.-J., Plaisted, D., 1992. Eliminating Duplicates with the Hyper-Linking Strategy. Journal of Automated Reasoning 9, 25–42.

Letz, R., Mayr, K., Goller, C., 1994. Controlled Integrations of the Cut Rule into Connection Tableau Calculi. Journal of Automated Reasoning 13.

Letz, R., Stenz, G., 2001. Proof and Model Generation with Disconnection Tableaux. In: Nieuwenhuis, R., Voronkov, A. (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba. Vol. 2250 of Lecture Notes in Computer Science. Springer Verlag, Berlin, Heidelberg, New-York.

Letz, R., Stenz, G., 2002. Integration of Equality Reasoning into the Disconnection Calculus. In: Egly, U., Fermüller, C. G. (Eds.), TABLEAUX. Vol. 2381 of Lecture Notes in Computer Science. Springer Verlag, Berlin, Heidelberg, New-York, pp. 176–190.

Manthey, R., Bry, F., 1988. SATCHMO: a theorem prover implemented in Prolog. In: Lusk, E., Overbeek, R. (Eds.), Proceedings of the 9^{th} Conference on Automated Deduction, Argonne, Illinois, May 1988. Vol. 310 of Lecture Notes in Computer Science. Springer Verlag, Berlin, Heidelberg, New-York, pp. 415–434.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., Malik, S., Jun. 2001. Chaff: Engineering an Efficient SAT Solver. In: Proceedings of the 38th Design Automation Conference (DAC'01).

Nieuwenhuis, R. (Ed.), 2005. Automated Deduction – CADE-20. Vol. 3632 of Lecture Notes in Artificial Intelligence. Springer.

Parkes, A. J., June 1999. Lifted search engines for satisfiability. Ph.D. thesis, University of Oregon.

Peltier, N., 1999. Pruning the search space and extracting more models in tableaux. Logic Journal of the IGPL 7 (2), 217–251.

Plaisted, D. A., Zhu, Y., 1997. Ordered Semantic Hyper Linking. In: Proceedings of Fourteenth National Conference on Artificial Intelligence (AAAI-97).

Plaisted, D. A., Zhu, Y., 2000. Ordered Semantic Hyper Linking. Journal of Automated Reasoning 25 (3), 167–217.

Stenz, G., 2002. DCTP 1.2 - System Abstract. In: Egly, U., Fermüller, C. G. (Eds.), Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX 2002, Copenhagen, Denmark, July 30 - August 1, 2002, Proceedings. Vol. 2381 of Lecture Notes in Computer Science. Springer Verlag, Berlin, Heidelberg, New-York, pp. 335–340.

Stenz, G., Letz, R., 2004. Generlized handling of variables in disconnection tableaux. In: Proc. International Joint Conference on Automated Reasoning. Vol. 3097 of Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Heidelberg, New-York, pp. 289–306.

Strichman, O., Seshia, S., Bryant, R., 2002. Deciding separation formulas

with sat. In: Proceedings of the Computer Aided Verification conference (CAV'02).
Sutcliffe, G., Suttner, C., 1998. The TPTP Problem Library: CNF Release v1.2.1. Journal of Automated Reasoning 21 (2), 177–203.
Tinelli, C., 2002. A DPLL-based calculus for ground satisfiability modulo theories. In: Ianni, G., Flesca, S. (Eds.), Proceedings of the 8th European Conference on Logics in Artificial Intelligence (Cosenza, Italy). Vol. 2424 of Lecture Notes in Artificial Intelligence. Springer Verlag, Berlin, Heidelberg, New-York.
van Eijck, J., 2001. Constrained Hyper Tableaux. In: Fribourg, L. (Ed.), Computer Science Logic. pp. 232–246, 15th International Workshop, CSL 2001 (LNCS 2142).
Yahya, A., Plaisted, D., 2002. Ordered Semantic Hyper-Tableaux. Journal of Automated Reasoning 29 (1), 17–57.
Zhang, H., Stickel, M. E., 1996. An efficient algorithm for unit propagation. In: Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96). Fort Lauderdale (Florida USA).

8 Appendix

This appendix contains auxiliary lemmas, their proofs, and proofs of the results stated in the main part of this paper. It is structured in three parts. Section 8.1 is a collection of results about contexts in general, not necessarily about contexts as they evolve in derivations. The latter is the subject of Section 8.2. The subsequent Section 8.3 then contains lemmas stating conditions under which the mandatory inference rules of $\mathcal{ME}$ are applicable. The results collected up to then were employed in Section 4.3 to prove our main theorem, the completeness of $\mathcal{ME}$.

8.1 Properties of Contexts

The first lemma is not concerned with contexts; it will be needed, however, for some proofs below.

Lemma 8.1 *For any literal L, the sets $\{K \mid K \geq L\} / \simeq$ and $\{K \mid K \gtrsim L\} / \simeq$ are finite.*

That is, for a given literal L, there are only finitely many more general literals wrt. $\geq$ of L modulo p-variantship, and similarly for $\gtrsim$. A similar result formulated in terms of $\gtrsim$ and $\approx$ is proven in (Eder, 1985).

Proof. Since $K \gtrsim L$ whenever $K \geq L$, it is enough to show that the set $\{K \mid K \gtrsim L\} \,/\, \simeq$ is finite.

In the following argumentation we will prove the claim using a tree representation of literals: if L is a literal, its tree representation is the (ordered) tree, the root of which is labeled with the predicate symbol of the literal, inner nodes are labeled with function symbols, and leaf nodes are labeled with constant or variable symbols, all in the obvious way.

Recall that $K \geq L$ means there is a (p-preserving) substitution σ such that $K\sigma = L$. This means that the tree for L is obtained by replacing each variable leaf node x in the tree for K by the tree for $x\sigma$, and similarly for parameters. Clearly, the number of nodes in K is less than or equal to the number of nodes in L.

Now let $\{x_1, \ldots, x_n\} \subset X$ and $\{u_1, \ldots, u_n\} \subset V$ be finite sets of any n pairwise different variables and any n pairwise different parameters, respectively,where n is the number of nodes of the tree representation of L.

Let K be any literal such that $K \gtrsim L$ and such that K contains variables and parameters from the finite sets just mentioned only.

Because the number of nodes in K is less than or equal to the number of nodes in L, it follows together with the assumed finite sets of variables and parameters that only finitely many such literals $K \gtrsim L$ exist. Let $\mathbf{K}$ be the finite set of all these literals.

Notice that $\mathbf{K}$ is finite also if the signature under consideration contains *infinitely* many function symbols. This holds, because K cannot contain any function symbol not occurring in L (because then K could not be instantiated to L), and there occur only finitely many function symbols in L.

Clearly, every literal K with $K \gtrsim L$ is a $\simeq$-variant of some literal in $\mathbf{K}$. Therefore, with $\mathbf{K}$ being finite, so is $\{K \mid K \gtrsim L\} \,/\, \simeq$. □

Lemma 8.2 *Let Λ be a context and let K, L be literals. If $K \geq L$ and $\overline{L} \in_{\leq} \Lambda$, then K is contradictory with Λ.*

In words, if the complement of some literal from Λ and K admit a common p-instance, then K is contradictory with Λ. Most of the times this lemma is used only in a weaker form, where $K = L$.

Since the calculus works on non-contradictory contexts only, the lemma above implies that no context contains a literal and the complement of one of its p-instances.

Proof. Suppose that $K' \in \Lambda$, $K' \geq \overline{L}$ and $K \geq L$. Let $K'' \simeq K'$ be a fresh p-variant of K'. With $K' \geq \overline{L}$ it follows $K'' \geq \overline{L}$. Let σ be a p-preserving substitution such that $K''\sigma = \overline{L}$. Since K'' is fresh, σ may be assumed to move only the variables and parameters of K''. Therefore σ will not modify L, and so $L = L\sigma$ follows. Altogether then $K''\sigma = \overline{L}\sigma$.

Because of $K \geq L$ there is a p-preserving substitution σ' such that $K\sigma' = L$. This implies $K\sigma'\sigma = L\sigma$. Since K'' is fresh, σ' may be assumed not to modify K'', and $K'' = K''\sigma'$ follows. With $K''\sigma = \overline{L}\sigma$ from above it follows $K''\sigma'\sigma = \overline{L}\sigma$. Together with $K\sigma'\sigma = L\sigma$ it follows $K''\sigma'\sigma = \overline{K}\sigma'\sigma$. But then, since both σ' and σ are p-preserving and because of $K'' \in_{\simeq} \Lambda$, K is contradictory with Λ. □

Lemma 8.3 *Let Λ be a context, $K \in \Lambda$ and K', L literals. If $K \geq L$ and K' shields L from K then K' is contradictory with Λ.*

The previous lemma implies that no literal in a context produced by the $\mathcal{ME}$ calculus can shield a p-instance of another literal in the context. This fact easily entails the next two results below.

Proof. Suppose that $K \geq L$ and K' shields L from K. This means there is literal K'' such that $K' \geq K''$ and $K \gtrsim \overline{K''} \gtrsim L$. Let σ, σ' be p-preserving substitutions such that $K\sigma = L$ and $K'\sigma' = K''$. Without loss of generality assume that K'' does not contain a single variable. (If this is not the case, let ρ be a (p-preserving) substitution that maps each of K'''s variables to a fresh parameter, and observe that $K' \geq K''\rho$ and $K \gtrsim \overline{K''\rho} \gtrsim L$ hold; in the sequel K'' itself will be used to denote $K''\rho$ then.)

We distinguish two exhaustive cases, where the first case will directly lead to a proof of the conclusion, while the second case will be shown to be impossible.

If $K \geq \overline{K''}$, this implies $\overline{K''} \in_{\leq} \Lambda$, and so with $K' \geq K''$ it follows immediately with Lemma 8.2 that K' is contradictory with Λ.

If $K \not\geq \overline{K''}$, then from $K \gtrsim \overline{K''}$ and $\overline{K''} \gtrsim L$ conclude that there are substitutions δ and δ' such that $K\delta = \overline{K''}$ and $\overline{K''}\delta' = L$, where δ is not p-preserving. For later use note $K\delta\delta' = L$.

Let $U := \{u_1, \ldots, u_n\} = \mathcal{P}ar(K)$ be the (pairwise different) parameters of K, for some parameters $u_1, \ldots, u_n$ and $n \geq 0$. With $K\sigma = L$ from above, and since σ is p-preserving, there is a subset $\{v_1, \ldots, v_n\}$ of L's parameters that σ maps U onto, for some parameters $v_1, \ldots, v_n$. Without loss of generality assume that $u_i\sigma = v_i$, for $i = 1, \ldots, n$. Of course, the parameters in $\{v_1, \ldots, v_n\}$ must be pairwise different, too.

From $K\delta\delta' = L$ and $K\sigma = L$ it follows $u_i\delta\delta' = v_i$. Clearly, $u_i\delta$ is a variable or

a parameter, because otherwise $u_i\delta\delta' = v_i$ would be impossible. However, K'' was assumed above to contain no single variable. With $K\delta = \overline{K''}$ it follows the stronger result that $u_i\delta$ is a parameter. Now, it is impossible that a bijection from U onto $U\delta$ exists, because then, with U being the set of all parameters in K, this substitution, say, δ_U, and the substitution $\delta_{|X}$ could be combined as the p-preserving substitution $\delta_U\delta_{|X}$, and $K(\delta_U\delta_{|X}) = \overline{K''}$ would follow, contradicting the current case $K \not\geq \overline{K''}$. This implies that δ identifies at least two parameters in U, say u_1 and u_2. But then, with $u_1\delta = u_2\delta$ it is impossible to have $(u_1\delta)\delta' = v_1$, $(u_2\delta)\delta' = v_2$ and $v_1 \neq v_2$. Hence the case $K \not\geq \overline{K''}$ is impossible, which remained to be shown. □

Lemma 8.4 *Let Λ be a non-contradictory context and $K \in \Lambda$. Then, for any literal L with $K \geq L$, K strongly covers L in Λ.*

Proof. Let L be a literal with $K \geq L$. It follows immediately $K \gtrsim L$. If K does not strongly cover L in Λ, then there is a literal $K' \in \Lambda$ that shields L from K in Λ. But then, by Lemma 8.3 K' is contradictory with Λ. Since Λ is given as non-contradictory, there is no such literal $K' \in \Lambda$, and so K strongly covers L in Λ. □

Lemma 8.5 *Let Λ be a non-contradictory context and K, L be literals. If K strongly covers L in Λ then K produces L in Λ.*

This lemma states that the "strongly covers" relation is stronger than the "produces" relation. That the converse of the lemma does not hold can be seen from Example 3.7.

Proof. Suppose that K strongly covers L in Λ. It follows that K covers L in Λ. If there is a $K' \in \Lambda$ that shields L from K, then by Lemma 8.3 K' is contradictory with Λ. Since Λ is given as non-contradictory, there is no such literal $K' \in \Lambda$, and so K produces L in Λ. □

Lemma 8.6 *Let Λ be a non-contradictory context and $K \in \Lambda$. Then, for any literal L with $K \geq L$,*

(i) K produces L in Λ, and
(ii) Λ does not produce $\overline{L}$.

Proof. Let L be a literal such that $K \geq L$. That K produces L in Λ follows immediately with Lemmas 8.4 and 8.5.

To show that Λ does not produce $\overline{L}$, let $K' \in \Lambda$ be any literal such that $K' \gtrsim \overline{L}$.[30] We will show that K' does not produce $\overline{L}$ in Λ. From the given assumption $K \geq L$ it follows (a) with Lemma 8.4 that K strongly covers L in

[30] If such a literal does not exist, Λ cannot produce $\overline{L}$.

Λ, and (b) that K shields $\overline{L}$ from K'. Together with $K' \in \Lambda$ this implies that K' does not produce $\overline{L}$ in Λ. □

Lemma 4.7 *Let Λ be a non-contradictory context. Then, for any literal L, Λ produces L or Λ produces $\overline{L}$ (or both).*

Proof. Let L be any literal. We will directly prove that Λ produces L or Λ produces $\overline{L}$.

Due to the presence of the pseudo-literal $\neg v$ in Λ, Λ covers L (the case that $\neg v$ covers L in Λ is possible). More precisely, there is a $K \in \Lambda$ such that there is no literal $K'' \in \Lambda$ with $K \gtrsim \overline{K''} \gtrsim L$ or $K \gtrsim \overline{K''} \gtrsim \overline{L}$. This is possible, because there are only finitely many literals K'' (modulo renaming) such that $\overline{K''} \gtrsim L$ or $K'' \gtrsim L$ (cf. Lemma 8.1), and because the relation $\gtrsim$ is a strict, partial ordering, and hence does not admit cycles.

For reasons of symmetry we consider in the sequel only the case where $K \gtrsim L$ and there is no $K'' \in \Lambda$ with $K \gtrsim \overline{K''} \gtrsim L$. Observe this just means that K covers L in Λ.

By condition (2) in Definition 3.6, if there is no $K' \in \Lambda$ that strongly covers $\overline{L}$ in Λ and that shields L from K, then K produces L in Λ. Otherwise, if there is such a $K' \in \Lambda$, then by Lemma 8.5 K' produces $\overline{L}$ in Λ. Together we have thus shown that Λ produces L or Λ produces $\overline{L}$. □

Lemma 8.8 *Let Λ be a context and K, K', L literals. If K produces L in Λ and $K \gtrsim K' \gtrsim L$ then K produces K' in Λ.*

Proof. Suppose that K produces L in Λ and $K \gtrsim K' \gtrsim L$. We will show that K produces K' in Λ.

Clearly, K cover K' in Λ (for, if it did not, K would not cover L in Λ either, contradicting that K produces L in Λ). Now, if K would not produce K' in Λ, then there is a $K'' \in_{\leq} \Lambda$ such that $K \gtrsim \overline{K''} \gtrsim K'$. But with $K' \gtrsim L$ it would follow $K \gtrsim \overline{K''} \gtrsim L$, and so K would not produce L in Λ either. □

The next two lemmas complement each other. Their prerequisites mean that L is contradictory with Λ, as witnessed by a literal $K \in_{\simeq} \Lambda$. The first lemma considers the case that L is parameter-free, and the second lemma considers the case that L is variable-free. Both lemmas express how a literal $K' \simeq K$ can take the rôle of K.

Derivations are about about stepwise modifications of sequents. This subsection contains lemmas mainly describing constraints on how contexts in sequents can evolve in a derivation. One such constraint, for instance, is that it is impossible to derive a sequent with a context that contains two p-variants of the same literal.

For the rest of this section, we make the same assumptions as stated in the beginning of Section 4.3.3. In particular thus let Φ be a set of parameter-free Σ-clauses and assume that $\mathcal{D}$ is a fair derivation of Φ that is not a refutation. Furthermore, let $\mathbf{B} = (N_i)_{i<\kappa}$ be an exhausted branch of $\mathcal{D}$'s limit tree and let $\Lambda_i \vdash \Phi_i$ denote the sequent labeling the node N_i in $\mathbf{B}$, for all $i < \kappa$.

Lemma 8.9 *For all $i < \kappa$, Λ_i is not contradictory.*

Being non-contradictory is a fundamental property of the contexts manipulated by the calculus. Lemma 8.9 essentially says that rule applications produce non-contradictory contexts from non-contradictory contexts.

Proof. The proof is by induction on i. For the base we have $\Lambda_0 = \{\neg v\}$ and this set is trivially not contradictory. For the induction step we take as induction hypothesis the claim of the lemma. Observe that each inference rule that extends a context includes as an applicability condition that the resulting context(s) is (are) not contradictory. With this observation the induction step follows immediately. □

The next lemma extends the previous one to the limit case.

Lemma 4.13 *$\Lambda_{\mathbf{B}}$ is not contradictory.*

Proof. Suppose that $\Lambda_{\mathbf{B}}$ is contradictory. Then there are literals $L \in \Lambda_{\mathbf{B}}$ and $K \in_{\simeq} \Lambda_{\mathbf{B}}$ and there is a p-preserving substitution σ such that $L\sigma = \overline{K}\sigma$. By the compactness property, there is a j such that both $L \in \Lambda_j$ and $K \in_{\simeq} \Lambda_j$. By virtue of the substitution σ, Λ_j is contradictory then. However, this is impossible by Lemma 8.9. □

Lemma 8.11 *The sequent $\Lambda, L \vdash \Psi$ is not derivable from a sequent $\Lambda \vdash \Psi$ in $\mathbf{B}$ if $L \in_{\leq} \Lambda$ or $\overline{L} \in_{\leq} \Lambda$.*

Lemma 8.11 states that the inference rules of the $\mathcal{ME}$ calculus never add a literal to a context in presence of a more general one (wrt. $\geq$). One could say that this fact expresses a kind of *loop check*.

Proof. It suffices to consider potential applications of the Split or the Assert

inference rule to Λ, because these are the only rules that can extend a context.

By the applicability conditions of both the Split and the Assert inference rules, the context Λ can be extended with the literal L only if L is not contradictory with L. Now, if $\overline{L} \in_{\leq} \Lambda$ then by Lemma 8.2 L is contradictory with Λ and so in this case neither Split nor Assert is applicable. Hence it only remains to be shown that the sequent $\Lambda,\ L \vdash \Psi$ is not derivable from $\Lambda \vdash \Psi$ if $L \in_{\leq} \Lambda$.

Split) Recall that the Split rule is applicable only if neither K nor $\overline{K}^{\text{sko}}$ is contradictory with Λ, where K is the remainder literal to split with. We consider two cases, corresponding to the case that the literal L in the lemma statement is K or is $\overline{K}^{\text{sko}}$. In both cases we will show that Split is not applicable by showing that K or $\overline{K}^{\text{sko}}$ is contradictory with Λ.

In the first case the literal L in the lemma statement is K. That $K \in_{\leq} \Lambda$ holds means there is a literal $K' \in \Lambda$ and a p-preserving substitution σ such that $K'\sigma = K$.

Let μ be the Skolemizing substitution used, i.e. the substitution μ such that $K\mu = K^{\text{sko}}$. From $K'\sigma = K$ it follows trivially that $K'\sigma\mu = K\mu$. Since $\sigma\mu$ is p-preserving, we then have that $K' \geq K\mu$. With $K\mu = K^{\text{sko}}$ we get $K' \geq K^{\text{sko}}$, or equivalently $K' \geq \overline{\overline{K}^{\text{sko}}}$. Since $K' \in \Lambda$, this means in other words $\overline{\overline{K}^{\text{sko}}} \in_{\leq} \Lambda$. But now, by Lemma 8.2, $\overline{K}^{\text{sko}}$ is contradictory with Λ. This completes the proof for the first case.

In the second case the literal L in the lemma statement is $\overline{K}^{\text{sko}}$. That $\overline{K}^{\text{sko}} \in_{\leq} \Lambda$ holds means there is a literal $K' \in \Lambda$ and a p-preserving substitution σ such that $K'\sigma = \overline{K}^{\text{sko}}$.

Let μ be the Skolemizing substitution used, i.e. the substitution μ such that $K\mu = K^{\text{sko}}$. It can be written as $\mu = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$, where $x_1, \ldots, x_n$ are all the variables occurring in K, and $a_1, \ldots, a_n$ are fresh constants. Now, because the constants $a_1, \ldots, a_n$ are fresh, none of them will occur in K. This means that we can consider the "substitution" $\mu' = \{a_1 \mapsto x_1, \ldots, a_n \mapsto x_n\}$ and have that $K = K\mu\mu' = K^{\text{sko}}\mu'$. From $K'\sigma = \overline{K}^{\text{sko}}$ it follows trivially $K'\sigma\mu' = \overline{K}^{\text{sko}}\mu'$.

Recall that the substitution σ is p-preserving. We may assume that all the variables moved by σ are just the variables of K', and each variable in K' is moved by σ to some Skolem constant a_i, so that $K'\sigma = \overline{K}^{\text{sko}}$ holds. It follows that the substitution $\sigma\mu'$ is a renaming on V and each variable in K' is moved to some variable x_i (not neccessarily in an injective way). More formally, $K' \geq K'\sigma\mu'$ holds.

Now, from $K'\sigma\mu' = \overline{K}^{\text{sko}}\mu'$ and with $K = K^{\text{sko}}\mu'$ we get easily $K'\sigma\mu' = \overline{K}$. Since $K' \in \Lambda$ this means in other words $\overline{K} \in_{\leq} \Lambda$. But now, by Lemma 8.2, K is contradictory with Λ. This completes the proof for the second case.

Together, thus, K or $\overline{K}^{\text{sko}}$ is contradictory with Λ, which remained to be shown.

Assert) The proof is immediate from the applicability condition from Assert, which explicitly demands that there is no $K \in \Lambda$ such that $K \geq L$. □

In the course of the development of a branch, a literal in a sequent's context may be deleted by means of the Compact rule. Such a deletion is only possible in presence of a p-subsuming literal, which takes the rôle of the deleted literal. This process may continue and is formalized in the following definition.

Definition 8.12 *Let K be a literal. For all $i < \kappa$, if $L \in \Lambda_i$ then the* trace of L from Λ_i *is the sequence $(L^j)_{i \leq j < \kappa}$, where $L^j := L$ if $j = i$, and for all $j > i$,*

$$L^j := \begin{cases} K & \text{if Compact is applied with selected literal } L^{j-1} \text{ and} \\ & \text{subsuming literal } K \text{ to } \Lambda_{j-1} \vdash \Phi_{j-1} \text{ to obtain } \Lambda_j \vdash \Phi_j \\ L^{j-1} & \text{otherwise} \end{cases}$$

Lemma 8.13 *Let $i < \kappa$ and $L \in \Lambda_i$. For every j with $i \leq j < \kappa$, there is a $K \in \Lambda_j$ such that $K \geq L$.*

While growing a branch, the $\mathcal{ME}$ calculus can delete a literal from the current context by means of the Compact rule. Such a deletion is only possible after the addition of a p-subsuming literal, which takes the rôle of the deleted literal. The p-subsuming literal itself may be deleted later, in a similar way. Lemma 8.13 is a formal statement of this process.

Proof. Consider the trace $(L^j)_{i \leq j < \kappa}$ of L from Λ_i. Each literal L^j from the trace, where $i \leq j < \kappa$, is contained in Λ_j, and $L^j \geq L$ holds by construction of the trace. □

Lemma 8.14 *For all distinct literals $K, L \in \bigcup_{i<\kappa} \Lambda_i$ it holds that $K \not\simeq L$.*

Lemma 8.14 states that if some context in the branch $\mathbf{B}$ contains a literal K, then neither this nor any other context can contain a p-variant L of$K$. This holds even if K is deleted at some point along the branch.

Proof. Assume by way of contradiction that $K \simeq L$ for some different literals $K, L \in \bigcup_{i<\kappa} \Lambda_i$. Notice first that not both K and L can be pseudo-literals, i.e. of the form $\neg v$, because the context Λ_0 contains exactly one such pseudo-literal and the calculus has no inference rules to add (or to delete) those. If

only one of K and L is a pseudo-literal, the lemma holds trivially. Hence, from now on assume that neither K nor L is a pseudo-literal.

There must be finite ordinals j and k such that $L \in \Lambda_j$ and $K \in \Lambda_k$. W.l.o.g. assume that $j \geq k$. Furthermore j and k may be chosen minimal, i.e. $L \notin \Lambda_{j-1}$ and $K \notin \Lambda_{k-1}$. This means that some inference rule is applied to the node N_{j-1} that extends the context Λ_{j-1} with L to obtain Λ_j (and similarly for K). Observe that the inference rules of $\mathcal{ME}$ extend the given context by at most one literal. In particular, K cannot have been added to Λ_{j-1} by the considered inference rule application. Thus, not only $j \geq k$ but also $j > k$ must hold.

But then, by Lemma 8.13 there is a literal $K' \in \Lambda_{j-1}$ such that $K' \geq K$. Together with $K \simeq L$ it follows immediately that $K' \geq L$. However, according to Lemma 8.11 the considered inference rule application that extends Λ_{j-1} by L is not possible. A plain contradiction. Thus, we must have $K \not\simeq L$. □

Lemma 8.15 *For all $i < \kappa$ and $L \in \Lambda_i$ there is a $K \in \Lambda_{\mathbf{B}}$ such that $K \geq L$.*

Lemma 8.15 essentially states that the set of persistent context literals of the branch $\mathbf{B}$ contains generalizations of all the context literals along $\mathbf{B}$.

Proof. Consider the trace $(L^j)_{i \leq j < \kappa}$ of L from Λ_i. All its consecutive different elements L_j and L_{j+1} are those where the Compact rule is applied to the sequent $\Lambda_j \vdash \Phi_j$ labeling the node N_j. That Compact is applied means $L_{j+1} \geq L_j$. Of course, both $L_{j+1} \in \Lambda_j$ and $L_j \in \Lambda_j$ must hold as well. By Lemma 8.14 we can conclude that $L_{j+1} \not\simeq L_j$. Together with $L_{j+1} \geq L_j$ this entails that $L_{j+1} \gneq L_j$.

In other words, the considered consecutive different elements from the trace determine a sequence of increasing literals wrt. $\gneq$. With Lemma 8.1 it follows immediately that this sequence is finite. If the sequence is non-empty, let K be its last element. Otherwise, let $K = L$. In both cases it is easy to see that $K \in \Lambda_{\mathbf{B}}$ and $K \geq L$. □

Lemma 8.16 *If $L \in \Lambda_{\mathbf{B}}$ and $L \in \Lambda_i$, for some $i < \kappa$, then $L \in \Lambda_j$, for all $j \geq i$ with $j < \kappa$.*

This means if a persistent literal of a context is present at some time i, then it is present from that time on. This is not trivial, as it does not follow from the definition of "persistency" alone: that definition is consistent with having, say, $L \in \Lambda_i$, $L \notin \Lambda_{i+1}$ and $L \in \Lambda_{j+2}$ for $j \geq i$. The result above instead really expresses a property of $\mathcal{ME}$ calculus.

Proof. Suppose $L \in \Lambda_{\mathbf{B}}$ and $L \in \Lambda_i$ for some i as stated. Suppose, to the contrary of the lemma conclusion, there is a $j \geq i$ such that $L \notin \Lambda_j$. By Lemma 8.13 there is a $K \in \Lambda_j$ such that $K \geq L$. By induction we now show

that $L \notin \Lambda_k$ for all $k \geq j$ with $k < \kappa$. Once shown, this completes the proof, because this results contradicts the given assumption $L \in \Lambda_{\mathbf{B}}$. We further take the invariant that, for every k as stated, there is a literal $K \in \Lambda_k$ such that $K \geq L$.

$k = j$) Trivial, as $L \notin \Lambda_j$ and $K \in \Lambda_j$ with $K \geq L$ is assumed.

$k \mapsto k + 1$) We consider the possible inference rule applications to derive $\Lambda_{k+1} \vdash \Phi_{k+1}$ from $\Lambda_k \vdash \Phi_k$, thereby concentrating on the non-trivial cases.

If Compact is applied to $\Lambda_k \vdash \Phi_k$ with selected literal K, both Λ_k and Λ_{k+1} must contain a literal $K' \geq K$. By the invariant we know $K \geq L$, and so $K' \geq L$ follows. Therefore chose now $K := K'$ to preserve the invariant.

If some inference rule is applied to extend Λ_k to Λ_{k+1}, L', for some literal L', then with Lemma 8.11 it follows $K \not\geq L$. This implies $L \notin \Lambda_{k+1}$. The invariant is trivially preserved.

This completes the induction, and hence the whole proof. □

Lemma 8.17 *Let K, L be two literals. If K does not strongly cover L in $\Lambda_{\mathbf{B}}$, then there is an $i < \kappa$ such that for all j with $i \leq j < \kappa$, K does not strongly cover L in Λ_j.*

Proof. Suppose that K does not strongly cover L in $\Lambda_{\mathbf{B}}$. We will directly prove the conclusion.

If $K \not\gtrsim L$ then the lemma holds trivially (take $i = 0$). Hence assume $K \gtrsim L$ from now on. That K does not strongly cover L in $\Lambda_{\mathbf{B}}$ then implies that $\Lambda_{\mathbf{B}}$ shields L from K. This means, there is a $K' \in \Lambda_{\mathbf{B}}$ and a p-preserving substitution σ such that $K \gtrsim \overline{K'}\sigma \gtrsim L$. Since $K' \in \Lambda_{\mathbf{B}}$ there is a $k < \kappa$ such that $K' \in \Lambda_k$. By Lemma 8.15 there is a $K'' \in \Lambda_{\mathbf{B}}$ with $K'' \geq K'$. Since $K'' \in \Lambda_{\mathbf{B}}$ there is an $i < \kappa$ such that $K'' \in \Lambda_j$, for all $j \geq i$ with $j < \kappa$.

Let σ' be a p-preserving substitution such that $K''\sigma' = K'$. From this, $K \gtrsim \overline{K''}\sigma'\sigma \gtrsim L$ follows immediately. Because both σ' and σ are p-preserving, $\sigma'\sigma$ is p-preserving as well. Thus, K'' shields L from K. Since $K'' \in \Lambda_j$, for all j with $i \leq j < \kappa$, Λ_j shields L from K, and so K does not strongly cover L in Λ_j. □

Lemma 4.14 *Let K, L be two literals with $K \in \Lambda_{\mathbf{B}}$. If K produces L in $\Lambda_{\mathbf{B}}$, then there is an $i < \kappa$ such that for all $j \geq i$ with $j < \kappa$, $K \in \Lambda_j$ and K produces L in Λ_j.*

Proof. Assume that K produces L in $\Lambda_{\mathbf{B}}$. We will directly prove the conclusion.

Since $K \in \Lambda_{\mathbf{B}}$ there is a $k < \kappa$ such that $K \in \Lambda_k$, for all $j \geq k$. However,

there is no guarantee that k is the index i we are looking for. Informally, it might be the case that a literal K' shielding L from K and strongly covering $\overline{L}$ is present in Λ_k or some successor context. While there is no way then of preventing any such context to shield L from K, we will show that neccessarily none of the *finitely many* literals K' shielding L from K will strongly cover $\overline{L}$ from some timepoint on (Lemma 8.17 will be used for this). More formally, let

$$M = \{K' \mid \text{there is an } m \geq k \text{ with } m < \kappa \text{ such that } K' \in_{\leq} \Lambda_m \text{ and } K \gtrsim\!\!\!\!\!\!\sim\; K' \gtrsim L\}$$

be those literals in Λ_m that shield L from K, for some $m \geq k$.

Since $K' \gtrsim L$ for each $K' \in M$, the set $M \,/\, \simeq$ must be finite by Lemma 8.1. Moreover, M is trivially a subset of $\bigcup_{i<\kappa} \Lambda_i$. Therefore Lemma 8.14 is applicable, and it gives us $K'_1 \not\simeq K'_2$ for any two different literals $K'_1, K'_2 \in M$. This means that each element of $M \,/\, \simeq$ is a singleton. In sum, $M \,/\, \simeq$ is a finite set of (singleton) equivalence classes. Therefore M itself is finite.

Let

$$M' = M \,\cap\, \Lambda_{\mathbf{B}}$$

be the set of persistent literals of $\Lambda_{\mathbf{B}}$ and of M that shield L from K. Clearly, with M being a finite set, M' also is. Let $k' < \kappa$ be the smallest index such that $M' \subseteq \Lambda_{j'}$, for all $j' \geq k'$ (such a k' must exist by the compactness property).

It is impossible that $\Lambda_{k'}$ is extended later with a new literal that shields L from K. More formally, for all $j' \geq k'$, any literal $K'' \in \Lambda_{j'}$ that shields L from K is contained in M'. This is, because for any such literal K'', by Lemma 8.15 there is a $K' \in \Lambda_{\mathbf{B}}$, with $K' \geq K''$. It is easy to see that then K' shields L from K, and hence K' is contained in M', and so is contained in $\Lambda_{k'}$.

Because of this property, that any literal in $\Lambda_{j'}$ (for any $j' \geq k'$) that shields L from K is already contained in M', to prove the claim it suffices by definition of productivity to show that from some timepoint on equal or later than k', none of the literals in M' strongly produces $\overline{L}$ in that contexts.

More precisely, for any literal $K' \in M'$, from $K' \in \Lambda_{\mathbf{B}}$, the fact that K' shields L from K, and the assumption that K produces L in $\Lambda_{\mathbf{B}}$, conclude by the definition of productivity that K' does not strongly cover $\overline{L}$ in $\Lambda_{\mathbf{B}}$. Then, by applying Lemma 8.17 to each literal $K' \in M'$, we can conclude from the finiteness of M' that there is an index $i \geq k'$ such that for all $j \geq i$ and for each $K' \in M'$, K' does not strongly cover $\overline{L}$ in Λ_j. Together with the conclusion above that any literal in $\Lambda_{j'}$ (for any $j' \geq k'$) that shields L from K is already contained in M', this implies that K produces L in Λ_j, for all $j \geq i$. To complete the proof it is enough to recall that $K \in \Lambda_j$ for all $j \geq k$ with $j < \kappa$ and that $i \geq k' \geq k$. □

The following lemmas provide sufficient conditions for the applicability of the main rules of the calculus to a given context.

Lemma 4.9 (Lifting Lemma) *Let Λ be a non-contradictory context. Let $C = L_1 \vee \cdots \vee L_n$ be a Σ-clause and $C\gamma$ a ground Σ-instance. If Λ produces $\overline{L_1}\gamma,\ldots,\overline{L_n}\gamma$, then there are fresh variants $K_1, \ldots, K_n \in_\simeq \Lambda^\Sigma$ and a substitution σ such that*

(1) σ is a most general simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,
(2) for all $i = 1, \ldots, n$, $L_i \gtrsim L_i\sigma \gtrsim L_i\gamma$,
(3) for all $i = 1, \ldots, n$, K_i produces $\overline{L_i}\sigma$ in Λ.

Proof. Let $i \in \{1, \ldots, n\}$ and assume that Λ produces $\overline{L_i}\gamma$. Then, there are literals $K_i' \in \Lambda$ such that K_i' produces $\overline{L_i}\gamma$ in Λ. Let $K_i \simeq K_i'$ be fresh variants of K_i'. It is easy to see that $K_i \in_\simeq \Lambda$ produces $\overline{L_i}\gamma$ in Λ. Because all the K_i's are fresh, they are pairwise disjoint, and each K_i is disjoint from C. Furthermore, each K_i must be a Σ-literal (and not a Σ^{sko}-literal), because if K_i contains some Skolem constant, $K_i \gtrsim \overline{L_i}\gamma$ would not hold (recall that $\overline{L_i}\gamma$ is a ground Σ-literal) and thus K_i would not produce $\overline{L_i}\gamma$ in Λ.

By definition of productivity, $K_i \gtrsim \overline{L_i\gamma}$, that is, there is a substitution π_i such that $K_i\pi_i = \overline{L_i\gamma}$. Since K_i is variable disjoint from C, we can assume that π_i moves only the variables and the parameters of K_i. Now, since K_i is disjoint from K_j for $j \in \{1, \ldots, n\}$ distinct from i, and π_i is a ground substitution for K_i, we have that $K_i\pi_i = K_i\pi$ where $\pi := \pi_1 \cdots \pi_i \cdots \pi_n$. Since $L_i\gamma$ is ground, it follows immediately that $L_i\gamma = L_i\gamma\pi$.

We may assume that all variables moved by γ occur in C only (otherwise restrict γ respectively). Together with the assumptions made it follows that $K_i = K_i\gamma$, which implies trivially that $K_i\pi = K_i\gamma\pi$.

Putting together all results obtained so far together, we get that $K_i\gamma\pi = \overline{L_i}\gamma\pi$ for all $i = 1, \ldots, n$. In other words, $\gamma\pi$ is a simultaneous unifier of $\{K_1, \overline{L_1}\}$ $\ldots$, $\{K_n, \overline{L_n}\}$. It follows that $\{K_1, \overline{L_1}\}$ $\ldots$, $\{K_n, \overline{L_n}\}$ admits a simultaneous mgu σ, which proves item 1 in the statement of the lemma.

Now, to prove item 2 observe that since $L_i\gamma$ is ground, $L_i\gamma\pi = L_i\gamma$. Since σ is a more general substitution than $\gamma\pi$ we know that $\gamma\pi = \sigma\delta$ for some substitution δ. It follows that $L_i\sigma\delta = L_i\gamma\pi = L_i\gamma$. In other words, $L_i\sigma \gtrsim L_i\gamma$. But then $L \gtrsim L_i\sigma \gtrsim L_i\gamma$ as desired.

To prove item 3 first observe that $K_i \gtrsim \overline{L_i}\sigma$ because $K_i\sigma = \overline{L_i}\sigma$. By item 2 we then have that $K_i \gtrsim \overline{L_i}\sigma \gtrsim \overline{L_i}\gamma$. Recalling that the literal K_i produces $\overline{L_i}\gamma$ in

Λ, it follows by Lemma 8.8 that K_i produces $\overline{L_i}\sigma$ in Λ as well. □

Lemma 4.10 (Existence of Admissible Context Unifiers) *Let Λ be a context, C a clause and σ a context unifier of C against Λ. Then, there is a renaming ρ such that $\sigma' := \sigma\rho$ is an admissible context unifier of C against Λ with the same context literals as σ.*

Proof. Let $C = L_1 \vee \cdots \vee L_n$ for some $n \geq 0$. By Definition 3.9 of context unifier, for all $i = 1, \ldots, n$ there is a $K_i \in_{\simeq} \Lambda$ such that $K_i\sigma = \overline{L_i}\sigma$. Moreover, there is an $m \in \{1, \ldots, n\}$ such that $(\mathcal{P}ar(K_i))\sigma \subseteq V$ for all $i = 1, \ldots, m$ and $(\mathcal{P}ar(K_i))\sigma \not\subseteq V$ for all $i = m+1, \ldots, n$.

We are going to construct a renaming substitution ρ as stated. Let $x_1, \ldots, x_k$ be the variables such that $\{x_1, \ldots, x_k\} = \mathcal{V}ar(L_{m+1}\sigma \vee \cdots \vee L_n\sigma)$, i.e. all variables occurring in the remainder. Define $\rho := \{x_1 \mapsto u_1, \ldots, x_k \mapsto u_k, u_1 \mapsto x_1, \ldots, u_k \mapsto x_k, \}$, where $u_1, \ldots, u_k$ are pairwise different and fresh parameters[31] .

Clearly, ρ is a renaming. It remains to show that $\sigma\rho$ is admissible for Split. Recall that $(\mathcal{P}ar(K_i))\sigma \subseteq V$ holds, for $i = 1, \ldots, m$. By construction, all the parameters moved by ρ are fresh parameters, none of which therefore can occur in K_i. In other words, $(\mathcal{P}ar(K_i))\rho = \mathcal{P}ar(K_i)$ holds, which entails $(\mathcal{P}ar(K_i))\sigma\rho = (\mathcal{P}ar(K_i))\sigma$. (However, $(\mathcal{P}ar(K_i))\sigma\rho \not\subseteq V$, for $i = m+1, \ldots, n$, will in general not hold). Therefore, there is a m' with $m \leq m' \leq n$ such that $(\mathcal{P}ar(K_i))\sigma\rho \subseteq V$, for $i = 1, \ldots, m'$ and $(\mathcal{P}ar(K_i))\sigma\rho \not\subseteq V$, for $i = m'+1, \ldots, n$.

None of the remainder literals $K_i\sigma\rho$, for $i = m+1, \ldots, n$, contains a single variable. Hence the disjointness requirement in the definition of admissible context unifier is trivially satisfied. This concludes the proof of existence of a renaming ρ as claimed. □

The following lemma applies (in particular) to remainders of admissible context unifiers. The lemma implies that if a clause has an admissible context unifier, against a given context, with a remainder all of whose literals are *individually* contradictory with the context, then the clause has a context unifier with an empty remainder.

Lemma 8.21 *Let Λ be a context, $L_1 \vee \cdots \vee L_n$ be a clause, where $n \geq 0$, possibly containing mixed literals, and such that for all distinct $i, j = 1, \ldots, n$, $\mathcal{V}ar(L_i) \cap \mathcal{V}ar(L_j) = \emptyset$. If for all $i = 1, \ldots, n$, L_i is contradictory with Λ*

[31] That is, every variable in the remainder is renamed by ρ to a parameter. From a practical point of view this is absurd, and it is better to compute a renaming that keeps as many variables in the remainder as possible. For the purpose of the completeness proof, however, the renaming ρ as constructed will do.

then there are fresh literals $K_1, \ldots, K_n \in_\simeq \Lambda$ *and a substitution* δ *such that the following holds:*

(1) δ *is a simultaneous unifier of* $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$,
(2) for all $i = 1, \ldots, n$, $\mathcal{D}om(\delta) \cap \mathcal{P}ar(L_i) = \emptyset$, *(i.e.* δ *does not move any single parameter in the given clause)*
(3) for all $i = 1, \ldots, n$, $(\mathcal{P}ar(K_i))\delta \subseteq V$.

Proof. Let Λ and $L_1 \vee \cdots \vee L_n$ be as stated, and such that the condition in the lemma result is satisfied. The conclusions, items 1–3, are proven by induction on n.

Base) If $n = 0$ then the result follows trivially by choosing for δ the empty substitution.

Step) Suppose $n > 0$ and consider the clause $L_1 \vee \cdots \vee L_{n-1}$. Clearly, for all distinct $i, j = 1, \ldots, n-1$, $\mathcal{V}ar(L_i) \cap \mathcal{V}ar(L_j) = \emptyset$ holds. Therefore, by the induction hypothesis there are fresh literals $K_1, \ldots, K_{n-1} \in_\simeq \Lambda$ and a substitution δ' such that

(1) δ' is a simultaneous unifier of $\{K_1, \overline{L_1}\}, \ldots, \{K_{n-1}, \overline{L_{n-1}}\}$,
(2) for all $i = 1, \ldots, n-1$, $\mathcal{D}om(\delta') \cap \mathcal{P}ar(L_i) = \emptyset$,
(3) for all $i = 1, \ldots, n-1$, $(\mathcal{P}ar(K_i))\delta' \subseteq V$.

Since L_n is contradictory with Λ, there is a literal $K \in_\simeq \Lambda$ and a p-preserving substitution σ such that $L_n\sigma = \overline{K}\sigma$. Let K_n be a fresh p-variant of K. Let ρ be a renaming substitution such that $K_n\rho = K$. Let $\rho' = \rho_{|\mathcal{V}ar(K_n) \cup \mathcal{P}ar(K_n)}$ It follows easily $K_n\rho' = K$. Since K_n is fresh, ρ' will not move variables or parameters in L_n, i.e. $L_n = L_n\rho'$ holds. From $L_n\sigma = \overline{K}\sigma$ it follows $L_n\rho'\sigma = \overline{K}\sigma = \overline{K_n}\rho'\sigma$. Since σ is p-preserving, $\sigma_{|V}$ is a renaming on the parameters, and $(\sigma_{|V})^{-1}$ exists. This implies trivially $\mathcal{P}ar(L_n)\sigma(\sigma_{|V})^{-1} = \mathcal{P}ar(L_n)$, i.e. $\sigma(\sigma_{|V})^{-1}$ does not move any parameter in L_n. With $L_n = L_n\rho'$ it follows that the substitution $\sigma' := \rho'\sigma(\sigma_{|V})^{-1}$ does not move any parameter in L_n. From $L_n\rho'\sigma = \overline{K_n}\rho'\sigma$ it follows trivially $L_n\rho'(\sigma\sigma_{|V})^{-1} = \overline{K}\rho'(\sigma\sigma_{|V})^{-1}$, i.e. $L_n\sigma' = \overline{K_n}\sigma'$.

Now let $\sigma'' := \sigma'_{|\mathcal{V}ar(L_n) \cup \mathcal{V}ar(K_n) \cup \mathcal{P}ar(K_n)}$. From $L_n\sigma' = \overline{K_n}\sigma'$ it follows $L_n\sigma'' = \overline{K_n}\sigma''$ (recall that σ' does not move any parameter in L_n and hence $\mathcal{P}ar(L_n)$ need not be included in the domain restriction of σ' defining σ'').

Let $\delta := \sigma''\delta'$ be the substitution to prove the induction step. We have to show items 1 to 3 to hold. Since all the variables moved by σ'' occur in L_n or K_n, K_n is fresh and L_n is variable disjoint with L_i, for $i = 1, \ldots, n-1$, σ'' will not move any variable in any L_i or in any K_i. Since K_n is fresh, and all parameters moved by σ'' occur in K_n, σ'' will not move any parameter

in any L_i or in any K_i (nor in L_n, as concluded above). This implies that $L_i\sigma'' = L_i$ and $K_i\sigma'' = K_i$, for $i = 1, \ldots, n-1$. With $K_i\delta' = \overline{L_i}\delta'$ from the induction hypothesis conclude $K_i\sigma''\delta' = \overline{L_i}\sigma''\delta'$, i.e. Above we concluded $L_n\sigma'' = \overline{K_n}\sigma''$, which implies trivially $L_n\sigma''\delta' = \overline{K_n}\sigma''\delta'$. Together and using the identity $\delta = \sigma''\delta'$ this gives the proof for item 1 for the induction step.

That item 2 to be proven carries over from the induction hypothesis to the induction step follows immediately from the definition of σ'' (recall that the parameters moved by σ'' is a subset of $\mathcal{P}ar(K_n)$).

It is not difficult to see that all parameters moved by σ'' are moved to parameters: since σ'' is obtained from σ' be restricting the domain of σ', it suffices to consider σ'. Recall that $\sigma' = \rho'\sigma(\sigma_{|V})^{-1}$, where all the parameters moved by ρ' are moved to parameters. The substitution σ' is p-preserving. Together this implies that all the parameters moved by σ' are moved to parameters. But then, item 3 to be proven immediately carries over from the induction hypothesis to the induction step. □

Lemma 4.11 (Split Applicability) *Let $\Lambda \vdash \Psi, C$ be a sequent with a non-contradictory context Λ, where C contains at least two literals. If all context unifiers of C against Λ have a non-empty remainder, and σ is an admissible context unifier of C against Λ such that Λ produces $\overline{L}$, for every remainder literal L of σ, then* Split *is applicable to $\Lambda \vdash \Psi, C$ with selected clause C and context unifier σ.*

Proof. Suppose the condition of the lemma statement holds. The proof of the conclusion consists of two parts: in a first part, we will show that there is a remainder literal that is not contradictory with Λ. Then, in a second part we will show that for each remainder literal L, $\overline{L}^{\text{sko}}$ is not contradictory with Λ. This will immediately give a proof that Split is applicable to $\Lambda \vdash \Psi, C$ with selected clause C, context unifier σ and that mentioned remainder literal. Let $C = L_1 \vee \cdots \vee L_m \vee L_{m+1} \vee \cdots \vee L_m$, where $0 \leq m \leq n$ (and $n \geq 2$), where the remainder D is $(L_{m+1} \vee \cdots \vee L_m)\sigma$. Suppose, to the contrary of the statement for the first part that every literal $L_j\sigma$, for $j = m+1, \ldots, n$ is contradictory with Λ. Since σ is admissible, all prerequisites to apply Lemma 8.21 to D are satisfied. By this lemma then, there are fresh literals $K_{m+1}, \ldots, K_n \in_{\simeq} \Lambda$ and there is a simultaneous unifier δ of $\{K_{m+1}, \overline{L_{m+1}}\sigma\}$, $\ldots$, $\{K_n, \overline{L_n}\sigma\}$ (Lemma 8.21, item 1) such that for all $j = m+1, \ldots, n$, it holds $\mathcal{D}om(\delta) \cap \mathcal{P}ar(L_j) = \emptyset$ (item 2), and $(\mathcal{P}ar(K_j))\delta \subseteq V$ (item 3). We may assume that δ is restricted so that each parameter moved by it occurs only in some literal K_j, where $m+1 \leq j \leq n$. Otherwise restrict δ respectively by excluding from its domain all the parameters that do not occur in any K_j, and items 1–3 will still hold. In particular, δ will still be a simultaneous unifier as stated in item 1, because the unrestricted δ does not move the parameters in L_j anyway.

In the sequel let the index j always ranges from $m+1, \ldots, n$.

Since each literal K_j is fresh, we may assume that σ does not modify K_j, i.e. $K_j = K_j\sigma$ holds. Therefore, δ is a simultaneous unifier of $\{K_{m+1}\sigma, \overline{L_{m+1}\sigma}\}$, $\ldots$, $\{K_n\sigma, \overline{L_n\sigma}\}$. Equivalently, $\sigma\delta$ is a simultaneous unifier of $\{K_{m+1}, \overline{L_{m+1}}\}$, $\ldots$, $\{K_n, \overline{L_n}\}$.

Furthermore, from $(\mathcal{P}ar(K_j))\delta \subseteq V$ and $K_j = K_j\sigma$ it follows $(\mathcal{P}ar(K_j))\sigma\delta \subseteq V$.

We are given that σ is an (admissible) context unifier. This means in particular that σ is a simultaneous unifier of $\{K_1, \overline{L_1}\}$, $\ldots$, $\{K_m, \overline{L_m}\}$. Trivially, $\sigma\delta$ is a simultaneous unifier of these literals as well.

Above we assumed that δ is restricted so that each parameter moved by it occurs in some literal K_j, where $m+1 \leq j \leq n$. Since each literal K_j is fresh, δ will not move any parameter in any literal $K_i\sigma$, for all $i = 1, \ldots, m$. Since σ is a context unifier, we know $(\mathcal{P}ar(K_i))\sigma \subseteq V$, for all $i = 1, \ldots, m$. Together this implies $(\mathcal{P}ar(K_i))\sigma\delta \subseteq V$.

Summing up, there is a simultaneous unifier $\sigma\delta$ (of $\{K_1, \overline{L_1}\}$, $\ldots$, $\{K_n, \overline{L_n}\}$ – we will omit in the sequel the mentioning of these pairs if just these are meant) such that $(\mathcal{P}ar(K_i))\sigma\delta \subseteq V$, for all $i = 1, \ldots, n$.

However, there is no guarantee yet that $\sigma\delta$ will be a simultaneous most general unifier. We will show next that a simultaneous most general unifier exists, that, moreover will be a context unifier of C against Λ with empty remainder, contradicting the lemma statement.

Since $\sigma\delta$ is a simultaneous unifier, there is a most general simultaneous unifier σ' and a substitution δ' such that $\sigma'\delta' = \sigma\delta$. The same arguments as in the proof of the Lifting Lemma, Lemma 4.9, can be applied to show this. However, there is no guarantee that $(\mathcal{P}ar(K_i))\sigma' \subseteq V$, for all $i = 1, \ldots, n$. But it must hold $(\mathcal{P}ar(K_i))\sigma' \subseteq X \cup V$, for all $i = 1, \ldots, n$, because otherwise there would be a parameter u in some literal K_i, where $1 \leq i \leq n$ and that would be moved to a term $u\sigma' \notin X \cup V$, which implies $u\sigma'\delta' \notin V$. However, we know $u\sigma'\delta' = u\sigma\delta \in V$.

Let $x_1, \ldots, x_k$ be all the variables in $(\mathcal{P}ar(K_1))\sigma' \cup \cdots \cup (\mathcal{P}ar(K_n))\sigma'$ and define the renaming

$$\rho = \{x_1 \mapsto u_1, \ldots, x_k \mapsto u_k, u_1 \mapsto x_1, \ldots, u_k \mapsto x_k\} \ ,$$

where $u_1, \ldots, u_k$ are fresh parameters. By this construction, each variable in $(\mathcal{P}ar(K_i))\sigma'$ is moved to a parameter, and because $u_1, \ldots, u_k$ are fresh, each parameter in $(\mathcal{P}ar(K_i))\sigma'$ is moved to itself, for all $i = 1, \ldots, n$. This proves

$(\mathcal{P}ar(K_i))\sigma'\rho \subseteq V$, for all $i = 1, \ldots, n$. Furthermore, with σ' being a most general simultaneous unifier and ρ being a renaming, $\sigma'\rho$ is a most general simultaneous unifier, too. In other words, $\sigma'\rho$ is a context unifier of C against Λ with empty remainder. Since this plainly contradicts what is given in the lemma statement, the assumption that every literal $L_j\sigma$, for $j = m+1, \ldots, n$, is contradictory with Λ must be withdrawn. Hence, as claimed, there is a remainder literal $L\sigma$ that is not contradictory with Λ. This completes the first part of the proof.

For the second part, let $L \in D$ be any remainder literal. We have to show that $\overline{L}^{\text{sko}}$ is not contradictory with Λ. Suppose to the contrary that $\overline{L}^{\text{sko}}$ is contradictory with Λ. Then, there is a $K \in_{\simeq} \Lambda$ and a p-preserving substitution σ such that $\overline{L}^{\text{sko}}\sigma = \overline{K}\sigma$.

Since σ is p-preserving, $\sigma_{|V}$ exists and is a renaming on V. Therefore, also $\rho := (\sigma_{|V})^{-1}$ exists, and so $u = u\sigma\rho$ for any parameter u follows. Because of Skolemization, $\overline{L}^{\text{sko}}$ is variable-free. This implies $\overline{L}^{\text{sko}} = \overline{L}^{\text{sko}}\sigma\rho$, and with $\overline{L}^{\text{sko}}\sigma = \overline{K}\sigma$, it follows easily $\overline{L}^{\text{sko}} = \overline{K}\sigma\rho$.

Let $\mu = \{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$ be the Skolemizing substitution used, for some $n \geq 0$. Now, because the constants $a_1, \ldots, a_n$ are fresh, none of them will occur in L. This means, we can consider the "substitution" $\mu' = \{a_1 \mapsto x_1, \ldots, a_n \mapsto x_n\}$ and it will hold $L = L^{\text{sko}}\mu'$.

The substitution $\sigma\rho$ is p-preserving, because both σ and ρ are. We may assume that all the variables moved by $\sigma\rho$ are just the variables of K, and each variable in K is moved by $\sigma\rho$ to some Skolem constant a_i, so that $\overline{L}^{\text{sko}} = \overline{K}\sigma\rho$ holds. It follows that the substitution $\sigma\rho\mu'$ is a renaming on V and each variable in K is moved to some variable x_i (not neccessarily in an injective way). More formally, $K \geq K\sigma\rho\mu'$ holds. Now, from $\overline{L}^{\text{sko}} = \overline{K}\sigma\rho$ it follows trivially $\overline{L}^{\text{sko}}\mu' = \overline{K}\sigma\rho\mu'$, and with $L = L^{\text{sko}}\mu'$ we get $L = K\sigma\rho\mu'$. With $K \geq K\sigma\rho\mu'$ it follows $K \geq L$. With $K \in_{\simeq} \Lambda$ and Lemma 8.6, K produces L in Λ and Λ does not produce $\overline{L}$. This contradicts the lemma statement according to which Λ does produce $\overline{L}$. Therefore, the assumption that $\overline{L}^{\text{sko}}$ is contradictory with Λ is false, and so no remainder literal is contradictory with Λ. Since this is all that remained to be proven, the proof is complete now. □

Lemma 4.12 (Assert Applicability) *Let $\Lambda \vdash \Psi$, L be a sequent with a non-contradictory context Λ. If all context unifiers of L against Λ have a non-empty remainder and there is an instance $L\sigma$ of L such that Λ produces $\overline{L}\sigma$, then* Assert *is applicable to $\Lambda \vdash \Psi$, L with selected clause L, selected literal the empty substitution as context unifier.*

Proof. Suppose that all context unifiers of L against Λ have a non-empty remainder and there is an instance $L\sigma$ of L such that Λ produces $\overline{L}\sigma$. To

show that Assert is applicable as stated, we first have to show that there is no literal $K \in \Lambda$ such that $K \geq L$. Suppose there were such a literal K. Recall that the clauses in the sequents are parameter-free. With L being therefore parameter-free, it follows easily that $L \geq L\sigma$. Together with $K \geq L$ conclude $K \geq L\sigma$. But then, Lemma 8.6 can be applied to conclude that Λ produces $L\sigma$ but Λ does not produce $\overline{L}\sigma$ plainly contradicting to what was supposed. Therefore, there is no literal $K \in \Lambda$ such that $K \geq L$.

Next, we have to show that L is not contradictory with Λ. Suppose, to the contrary, there is a literal $K \in_{\simeq} \Lambda$ and a p-preserving substitution δ such that $L\delta = \overline{K}\delta$. Since δ is a unifier for L and $\overline{K}$, there is a most general unifier σ' and a substitution δ' such that $\sigma'\delta' = \delta$. The same arguments as in the proof of the Lifting Lemma, Lemma 4.9, can be applied to show this. However, there is no guarantee that $(\mathcal{P}ar(K))\sigma' \subseteq V$. But it must hold $(\mathcal{P}ar(K))\sigma' \subseteq X \cup V$, because otherwise there would be a parameter u in K that would be moved to a term $u\sigma' \notin X \cup V$, which implies $u\sigma'\delta' \notin V$. However, we know $u\sigma'\delta' = u\delta \in V$ since δ is p-preserving.

Let $x_1, \ldots, x_k$ be all the variables in $(\mathcal{P}ar(K))\sigma'$ and define the renaming

$$\rho = \{x_1 \mapsto u_1, \ldots, x_k \mapsto u_k, u_1 \mapsto x_1, \ldots, u_k \mapsto x_k\} \ ,$$

where $u_1, \ldots, u_k$ are fresh parameters. By this construction, each variable in $(\mathcal{P}ar(K))\sigma'$ is moved to a parameter, and because $u_1, \ldots, u_k$ are fresh, each parameter in $(\mathcal{P}ar(K))\sigma'$ is moved to itself. This proves $(\mathcal{P}ar(K))\sigma'\rho \subseteq V$. Furthermore, with σ' being a most general unifier and ρ being a renaming, $\sigma'\rho$ is a most general unifier, too. In other words, $\sigma'\rho$ is a context unifier of L against Λ with an empty remainder. This, however, plainly contradicts what was supposed above.

Finally, in reference to the definition of the Assert inference rule, the clause L to be selected can also be written as $\Box \vee L$. Because, trivially, the empty substitution is a context unifier of $\Box$ against any context with an empty remainder, this concludes the proof that Assert is applicable as stated. $\Box$

Lemma 4.15 (Close Applicability) *Let $C \in \Phi_{\mathbf{B}}$ and $i < \kappa$ such that* Close *is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C. Then, for some $j \geq i$ with $j \leq \kappa$,* Close *is applicable to $\Lambda_j \vdash \Phi_j$ with selected clause C and a context unifier σ such that $K \in_{\simeq} \Lambda_{\mathbf{B}}$ for each context literal K of σ.*

Proof. Assume that Close is applicable to $\Lambda_i \vdash \Phi_i$ with selected clause C. We will directly prove the conclusion.

Suppose that C is of the form $L_1 \vee \cdots \vee L_n$. Let σ' be the context unifier of the considered Close rule application and let $K'_1, \ldots, K'_n \in_{\simeq} \Lambda_i$ be the context literals of σ'. With Lemma 8.15 it follows there are literals $K_1, \ldots, K_n \in_{\simeq} \Lambda_{\mathbf{B}}$

such that $K_k \geq K'_k$, for $k = 1, \ldots, n$. With $C \in \Phi_{\mathbf{B}}$ and $K_1, \ldots, K_n \in \Lambda_{\mathbf{B}}$ it follows easily from the compactness property that there is an $j \geq i$ with $j < \kappa$ such that $C \in \Phi_j$ and $K_k \in_{\simeq} \Lambda_j$, for $k = 1, \ldots, n$.

With this, it is enough to show that there is a context unifier σ of C against Λ_j with an empty remainder and context literals $K_1, \ldots, K_n$. Without loss of generality assume that $K_1, \ldots, K_n$ have been chosen as fresh p-variants of literals in Λ_j. In the sequel let the index k allways ranges from $1, \ldots, n$.

From the existence of the context unifier σ' and the fact $K_k \geq K'_k$ it follows there is a most general simultaneous unifier σ'' of $\{K_1, \overline{L_1}\}, \ldots, \{K_n, \overline{L_n}\}$. The same arguments as in the proof of the Lifting Lemma, Lemma 4.9, can be applied to show this. However, there is no guarantee that $(\mathcal{P}ar(K_k))\sigma'' \subseteq V$. Using the same construction as in the proof of Lemma 4.11, it can be shown that there is a renaming substitution ρ such that $(\mathcal{P}ar(K_k))\sigma''\rho \subseteq V$. Setting $\sigma := \sigma''\rho$ thus gives the desired substitution. $\square$