SCL(EQ): Simple Clause Learning in First-Order Logic with Equality

Hendrik Leidinger

Dissertation

zur Erlangung des Grades
Doktor der Naturwissenschaften (Dr. rer. nat.)
der Fakultät für Mathematik und Informatik
der Universität der Saarlandes

Saarbrücken
2024

Tag des Kolloquiums:	21. Mai 2025
Dekan:	Prof. Dr. Roland Speicher

Prüfungsausschuss

Vorsitzender:	Prof. Dr. Sebastian Hack
Berichterstatter:	Prof. Dr. Christoph Weidenbach
	Prof. Dr. Pascal Fontaine
Akademischer Mitarbeiter:	Dr. Matthias Fassl

Abstract

I propose the SCL(EQ) calculus that lifts SCL for first-order logic to first-order logic with equality. SCL(EQ) learns non-redundant clauses only. It builds a trail of annotated ground literals, representing the model assumption for non-ground input clauses. The trail includes propagations (inferred literals) and decisions (guessed literals). When a clause is false under the model assumption, SCL(EQ) derives a new non-ground clause via paramodulation. The new clause is non-redundant under a dynamic ordering, which, along with a maximum term, limits ground literals and ensures termination. I prove SCL(EQ) to be sound and refutationally complete.

SCL(EQ) may use congruence closure (CC) to identify propagations and conflicts efficiently. However, exhaustive propagation of unit clauses already causes a worst case exponential blowup in ground instances. To address this, I propose CC($\mathcal{X}$), a generalization of CC with variables. It creates an explicit representation of constrained congruence classes of the whole ground input space smaller than the maximum term. I prove CC($\mathcal{X}$) sound and complete, implement it, and evaluate its performance against state-of-the-art CC. Joint work with Yasmine Briefs integrates Knuth-Bendix ordering into CC($\mathcal{X}$).

Zusammenfassung

Ich schlage den SCL(EQ) Kalkül vor, der SCL für die Prädikatenlogik erster Ordnung auf die Prädikatenlogik erster Ordnung mit Gleichheit erweitert. SCL(EQ) lernt ausschließlich nicht-redundante Klauseln. Es erstellt eine Folge annotierter Grundliterale, die die Modellannahme für nicht-grund Eingangsklauseln repräsentiert. Sie umfasst Propagationen (abgeleitete Literale) und Entscheidungen (geratene Literale). Wenn eine Klausel unter der Modellannahme falsch ist, leitet SCL(EQ) eine neue nicht-grund Klausel mittels Paramodulation ab. Die neue Klausel ist nicht redundant gemäß einer dynamischen Ordnung, die zusammen mit einem maximalen Term Grundliterale begrenzt und die Terminierung sicherstellt. Ich beweise, dass SCL(EQ) korrekt und widerspruchsvollständig ist.

SCL(EQ) kann zur effizienten Identifikation von Propagationen und Konflikten den Kongruenzabschluss (CC) verwenden. Die vollständige Propagation von Einheitsklauseln bewirkt bereits im schlimmsten Fall ein exponentielles Anwachsen der Grundinstanzen. Um dies zu adressieren, schlage ich CC(X) vor, eine Verallgemeinerung von CC mit Variablen. CC(X) erstellt eine explizite Repräsentation eingeschränkter Kongruenzklassen des gesamten Grundeingaberaums, die kleiner als der maximale Term ist. Ich beweise, dass CC(X) korrekt und vollständig ist, implementiere es und evaluiere seine Leistung im Vergleich zu modernen CC-Methoden. In Zusammenarbeit mit Yasmine Briefs wird die Knuth-Bendix-Ordnung in CC(X) integriert.

Contents

List of Figures

List of Tables

Acknowledgments

Writing a dissertation has been one of the most challenging experiences of my life—testing me in ways I never imagined. The constant fear of falling short of my own expectations weighed heavily on me, often leading to moments of psychological strain. When attempting to create something entirely new, failure is always a possibility— sometimes an inescapable aspect of the process.

In such moments, having a stable, supportive environment becomes invaluable. I am deeply grateful to be surrounded by people who uplift me and remind me that what I do has meaning. In particular, I owe heartfelt thanks to my family and my girlfriend, Lea Gröber. Lea's unwavering encouragement and belief in me were instrumental in helping me persevere through this phase of my life. Her motivation and confidence in my abilities pushed me to expand my boundaries and not give up.

I am also profoundly thankful to my colleagues in the Automation of Logic group. My supervisor, Christoph Weidenbach, deserves special mention for his incredible patience and guidance, always providing calm, composed advice while steering me toward solutions. I am equally grateful to Martin Bromberger for his willingness to lend an empathetic ear and for his insights on implementation and formalization.

I would like to extend my gratitude to my reviewer, Pascal Fontaine, for taking the time to carefully read and evaluate my dissertation.

Finally, I would like to thank all my friends who, with their surprising facility always gave me the feeling that everything is all right.

Hendrik Leidinger

Introduction

Equational logic is widely used in almost all aspects of formal reasoning about systems. Significant research has focused on the development of sound and complete calculi for first-order logic with equality [PZ00, BFP07, BW09, BPT12, BG94, KS10]. Among these, the Superposition calculus [BG94, HW07, Wei01] is considered to be state-of-the art. It employs ordering restrictions to guide paramodulation inferences [RW69] and utilizes an abstract redundancy framework to enable clause simplification and deletion techniques such as rewriting and subsumption [BG94]. However, the Superposition calculus generates a large number of redundant clauses. Infact, modern theorem provers [KV13, KS10, BBB+22] dedicate substantial computational effort to identifying and eliminating such redundancies. While the completeness proof for Superposition offers a "semantic" mechanism to derive only non-redundant clauses, this approach relies on an underlying ground model assumption that is not computable in general [Teu18]. More specifically, it necessitates the ordered enumeration of infinitely many ground instances of the clause set, which is infeasible in practice. Many other complete calculi for first-order logic with equality have been proposed in the past [BT05, BPT12, BW09, BFP07, PZ00, Kor13, KS10]. However, none of these approaches are able to incrementally learn new clauses that are non-redundant according to the current model assumption. For other logics there exist such algorithms, e.g. Conflict Driven Clause Learning (CDCL) [SS96, JS96, MMZ+01, BHvMW09, Wei15] in propositional logic or Simple Clause Learning (SCL) [AW15, FW19, BFW21, BSW23], Model Evolution ($\mathcal{ME}$) with lemma learning [BT03, BFT06] and SGGS [BP15] in first-order logic.

This thesis overcomes the aforementioned issues by (1) presenting Simple Clause Learning for First-Order Logic with Equality (SCL(EQ)) [LW23], a calculus that provides an effective way of generating ground model assumptions that then guarantee non-redundant inferences on the original clauses with variables, and (2) providing first steps towards an efficient implementation with my new calculus Non-Ground Congruence Closure (CC($\mathcal{X}$)) [LW24], that is a generalization of the congruence closure algorithm [NO80, DST80, Sho84] for non-ground equations.

The $SCL(EQ)$ calculus is designed to learn only non-redundant clauses. The underlying ordering is determined by the sequence of ground literals in the model assumption and, consequently, evolves dynamically during the execution of the calculus. Since it incorporates a standard rewrite ordering, this means that both rewriting and redundancy notions that are based on literal subset relations are permitted to dynamically simplify or eliminate clauses. Since newly generated clauses are guaranteed to be non-redundant, redundancy tests are only required backwards. Moreover, the ordering is generated automatically

from the structure of the clause set, rather than being fixed in advance as in the case of Superposition. The calculus adjusts its ordering dynamically to align with the most effective strategy for making progress, analogous to CDCL. As in CDCL and other SCL-based approaches, SCL(EQ) constructs a model assumption through decisions and propagations. A decision guesses a ground literal to be true, while propagation derives the truth of a ground literal from clauses that would otherwise be false. Unlike CDCL in propositional logic, where the number of propositional variables is finite, first-order logic can involve infinite propagation sequences [FW19]. To overcome this issue, SCL(EQ) restricts model assumptions to a finite number of ground literals at any point in time, effectively limiting the scope to a finite set of ground instances of the clause set. With this restriction, the calculus either discovers a refutation or reaches a *stuck state*, where the current model assumption satisfies all considered ground instances. In such a case, one can either try to generalize the model assumption for the entire clause set or use the information from the stuck state to increase the number of considered ground literals and resume the search for a refutation. SCL(EQ) does not require exhaustive propagation; it only prohibits the decision of the complement of a literal that could otherwise be propagated.

For completeness, SCL(EQ) requires exhaustive propagation of units before it allows for the first decision. As a result, it needs to propagate all smaller instances of a non-ground equation. This necessitates the development of a generalized congruence closure algorithm with support for variables and constraints, because ground instantiation of all units already leads to a worst-case exponential blow up. CC($\mathcal{X}$) is a novel calculus designed to handle non-ground equations by maintaining a set of classes, similar to standard congruence closure. Each class, referred to as a *constrained class*, consists of a set of *constrained terms*. A *constraint* is defined as a conjunction of inequations bounded by a specified maximum term. This *constraint* limits the ground instances of a constrained term to those less than or equal to the maximum term. Initially, the set of constrained classes includes one class for each input equation and a single-term constrained class for each occurring non-variable symbol. The purpose of non-ground congruence closure is to compute a solution that covers the entire ground input space, bounded by the maximum term. This process relies on two primary rules, *Merge* and *Deduction*, to construct the congruence classes: *Merge* creates a new class by unifying two terms in different classes and applying this unifier to the Union of these two classes. *Deduction* creates a new class by simultaneously unifying the arguments of two terms with the same top symbol in different classes with terms in the same class. Termination is guaranteed by a definition of subsumption between two classes and the fact that all terms are constrained by a maximum term. I have implemented CC($\mathcal{X}$) on the basis of the SPASS workbench infrastructure [BFSW19, WDF+09]. The algorithm outperforms classical ground congruence closure if grounding gets too expensive. In collaboration with Yasmine Briefs [BLW23] we show on a early version of CC($\mathcal{X}$) that a Knuth-Bendix Ordering (KBO) [KB70] fulfills the requirement of SCL(EQ) on effective algorithms for a term order.

The following two Sections provide an overview of the two calculi on a more formal level.

SCL(EQ): SCL for First-Order Logic with Equality

In this Chapter, I describe the SCL(EQ) calculus and illustrate its rules and states by means of an example. Moreover, I compare the run of SCL(EQ) to Superposition.

SCL(EQ) operates on a set of non-ground first-order clauses consisting of equational literals. Consider the following three clauses.

$$C_1 := h(x) \approx g(x) \vee c \approx d \qquad C_2 := f(x) \approx g(x) \vee a \approx b$$
$$C_3 := f(x) \not\approx h(x) \vee f(x) \not\approx g(x)$$

and a KBO [KB70] with unique weight 1, and precedence $d \prec c \prec b \prec a \prec g \prec h \prec f$. An application of the Superposition calculus would result in a Superposition Left [BG94] inference between C_2 and C_3:

$$C_4' := h(x) \not\approx g(x) \vee f(x) \not\approx g(x) \vee a \approx b.$$

$SCL(EQ)$ is a set of deduction rules applied to a six-tuple $(\Gamma; N; U; \beta; k; D)$ as we will see in more detail in Section 2.2.1. Γ is a sequence of ground equations and inequations, also called a trail. I refer to the equations and inequations as literals. The literals in the trail are annotated by a level and a closure, i.e. a clause C and a grounding substitution σ, denoted by $C\cdot\sigma$. N is the set of input clauses, in our example $N = \{C_1, C_2, C_3\}$. Note that any set of formulas can be transformed into an equisatisfiable set of clauses and $SCL(EQ)$ only operates on sets of clauses. β is the maximum term that restricts the number of ground instances of the set of input clauses to a finite number. This already gives a hint of the requirement for the term order, namely the number of ground terms, literals and clauses smaller than or equal to β has to be finite. U is the set of clauses that were learned during the deduction process. k is the level which will be explained in more detail later. In principle, $SCL(EQ)$ can be in two states. Either Γ does not contradict $N \cup U$, i.e. assuming that Γ holds, there is no clause in $N \cup U$ that is *false*. Or there is a clause in $N \cup U$ that is *false* under the assumption of Γ. These two states are represented by D. D is either $\top$, which means that the calculus did not yet find a clause in N that is *false* under the assumption of Γ, or D is a closure. In this case D is called the conflict clause initially from $N \cup U$, that is *false* under the assumption of Γ. A special clause that D may contain is the empty clause, also represented as $\bot$. In this case $SCL(EQ)$ terminates with the result that the set N of input clauses is unsatisfiable.

Initially, the state is $(\epsilon; N; \emptyset; \beta; 0; \top)$, meaning that there is no literal in the sequence of equations, no clause was learned yet and there is no conflicting clause. In our example this is the initial state for some sufficiently large β and $N = \{C_1, C_2, C_3\}$. The applicable rules when there is no conflict clause are *Propagate*, *Decide* and *Conflict* as we will see in Section 2.2.1.

1. *Propagate* adds a ground literal (equation or inequation) to the sequence Γ if there exists a clause $C = \{L_1, ..., L_n\}$ in $N \cup U$ and a grounding substitution σ such that w.l.o.g. $L_1\sigma$ is satisfiable under Γ and $L_2\sigma, ..., L_n\sigma$ are all unsatisfiable under Γ. The added literal is the normal form of $L_1\sigma$ with respect to the convergent rewrite system of Γ, denoted by $L_1\sigma{\downarrow_{conv(\Gamma)}}$.

Propagations are annotated with the current level and a closure resulting from the corresponding paramodulation steps between the closure $C\cdot\sigma$ and the closures annotated to the literals involved in the deduction of the normal form of $L_1\sigma$ (see Definition 2.2.16).

2. *Decide* adds a ground literal $L\sigma$ to the sequence Γ without any such restrictions, except that $(C \vee L)$ has to occur as a clause in $N \cup U$. Let $(C_{new} \vee K)\delta$ be the clause resulting from the application of multiple paramodulation steps to $(C \vee L)\sigma$ such that $K\delta = L\sigma\downarrow_{conv(\Gamma)}$ and k the current level. Then, *Decide* adds the literal $K\delta$ to the trail, annotated by $(K \vee comp(K))\cdot\delta$ and the level $k+1$. Note, that the application of *Decide* is restricted by the definition of a regular run (Def. 2.3.5), i.e. *Propagate* has precedence in many cases.

3. *Conflict* replaces D by a closure $C\cdot\sigma$, where $C \in N \cup U$, if there exists a substitution σ such that $C\sigma$ is *false* under Γ. *Conflict* always has precedence over *Decide* and *Propagate*, i.e. whenever there is a conflicting clause, *Conflict* has to be applied immediately (cf. Def. 2.3.5).

With these rules we could now start to build our trail Γ in the example above with two decisions

$$\Gamma := [h(a) \approx g(a)^{1:(h(x)\approx g(x) \vee h(x) \not\approx g(x))\cdot\sigma}, f(a) \approx g(a)^{2:(f(x)\approx g(x) \vee f(x) \not\approx g(x))\cdot\sigma}]$$

where $\sigma := \{x \mapsto a\}$. Now, with respect to Γ clause C_3 is false with grounding σ, and rule *Conflict* is applicable. This adds the closure $C_3\cdot\sigma$ as conflict clause.

For the conflict resolution the rules *Skip*, *Explore-Refutation*, *Factorize*, *Equality-Resolution* and *Backtrack* are applicable as we will see in Section 2.2.1. The idea of conflict resolution is to create a clause from the conflicting clause that would have made the last decision impossible. In contrast to DPLL [DLL62b, DP60] style algorithms, where the last decision is always flipped, this can lead to a clause that would result in a conflict multiple decisions in advance, allowing us to skip multiple decisions at once.

1. The rules *Factorize* and *Equality-Resolution* simplify the conflicting closure $D\cdot\sigma$. If there exist $\{L, L'\} \subseteq D$ such that $L\sigma = L'\sigma$, then *Factorize* creates the new conflicting clause $(D \setminus \{L'\})\mu\cdot\sigma$, where $\mu = mgu(L, L')$, the most general unifier of L and L'. If there exist $\{s \not\approx s'\} \subseteq D$ such that $s\sigma = s'\sigma$, then *Equality-Resolution* creates the new conflicting clause $(D \setminus \{s \not\approx s'\})\mu\cdot\sigma$, where $\mu = mgu(s, s')$.

2. The rule *Skip* is used to remove the rightmost literal on the trail if it is not involved into the actual conflict of the conflicting closure. This is the case if the closure is still false in the remaining trail.

3. *Explore-Refutation* is applicable if the last literal L on the trail Γ, L is the defining literal of a literal L' in $D\sigma$ that is maximal in $D\sigma$, i.e. L' is false in Γ, L but not false in Γ. The rule now creates a refutation of L' meaning that it uses literals from Γ, L to infer $\bot$ from L'. To deduce the new clause for the closure $D\cdot\sigma$, paramodulation is applied with the annotated clauses of all involved literals of the refutation in the trail (Definition 2.2.17).

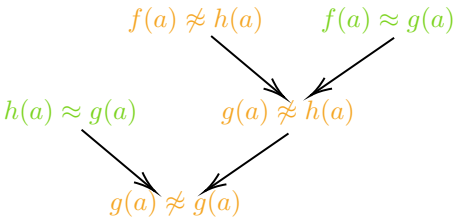

Figure 1: Ground refutation of the example. Green literals are from the trail, orange literal is the literal we want to refute.

4. *Backtrack* is applicable if the conflict clause reaches a level that is smaller than the current level for all but one literal by prior applications of *Explore-Refutation*. The rule than adds the new clause to the set of learned clauses and jumps back to the point on the trail where this new clause is again not conflicting with Γ for any possible substitution. The newly learned clause is non-redundant according to my trail induced ordering as defined in 2.2.9.

To come back to our example the maximal literal in $C_3\sigma$ is $(f(x) \not\approx h(x))\sigma$ and a rewrite refutation for this literal consists of two paramodulation steps first with the annotated clause of the second literal in the trail, which results in

$$(g(x) \not\approx h(x) \vee f(x) \not\approx g(x) \vee f(x) \not\approx g(x))\cdot \sigma$$

Then, with the annotated clause of the first literal in the trail to get

$$(g(x) \not\approx g(x) \vee f(x) \not\approx g(x) \vee f(x) \not\approx g(x) \vee h(x) \not\approx g(x))\cdot \sigma$$

where for the refutation justification clauses and all otherwise inferred clauses I use the grounding σ for guidance, but operate on the clauses with variables. Figure 1 shows the ground refutation of the literal of the conflicting clause. Figure 2 shows the corresponding non-ground refutation as shown above with paramodulation steps.

The respective ground clause is smaller than $(f(x) \not\approx h(x))\sigma$, false with respect to Γ and becomes our new conflict clause by an application of *Explore-Refutation*. It is simplified by *Equality-Resolution* and *Factorize*, resulting in the finally learned clause

$$C_4 := h(x) \not\approx g(x) \vee f(x) \not\approx g(x)$$

which is then used to apply rule *Backtrack* to the trail. Observe, that C_4 is strictly stronger than C_4' the clause inferred by Superposition and that C_4 cannot be inferred by Superposition. Thus, SCL(EQ) can infer stronger clauses than Superposition for this example.

In Section 2.3, I prove SCL(EQ) to be sound (Theorem 2.3.4) and refutation complete (Theorem 2.3.17). For soundness (Definition 2.3.1) I have to prove that

1. Γ is always consistent,

2. propagated and decided literals $L\sigma$ have to be undefined and irreducible and in the annotated clause $(C \vee L)\cdot \sigma$, C has to be false according to the preceding literals in the trail and $(C \vee L)\sigma$ has to be smaller than β,

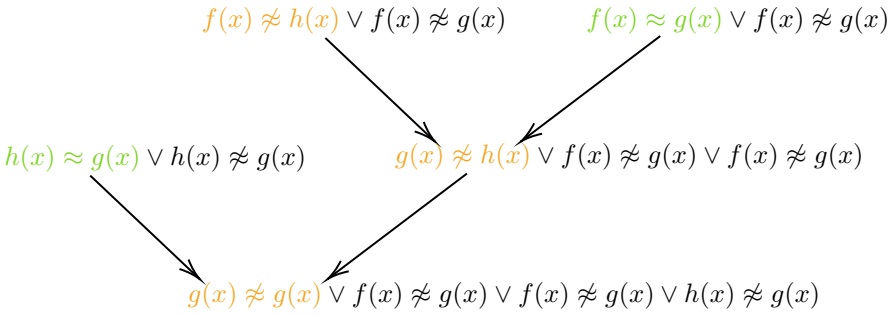

Figure 2: Non-ground refutation of the example with corresponding paramodulation steps as done by the *Explore-Refutation* rule. Clauses with green literals are annotated clauses from the trail, the orange literal is in the clause that we want to refute.

3. the set of learned clauses U has to follow from the set of input clauses N,
4. if there is a conflict clause $C \cdot \sigma$, then $C\sigma$ is false in Γ and C follows from N.

The proof is by structural induction on the state $(\Gamma; N; U; \beta; k; D)$ starting from the initial state. The proof is straightforward by checking if the state remains sound after application of the different rules.

For completeness I first have to give a definition of a regular run. A run is a sequence of applications of the SCL(EQ) rules (Definition 2.3.3). The run is called regular (Definition 2.3.5) if the following restrictions are applied:

1. The rules *Conflict* and *Factorize* have precedence over all other rules,
2. On level 0, *Propagate* has precedence over *Decide*,
3. If a literal L is propagatable, then $comp(L)$ may not be added by *Decide*,
4. If the application of *Decide* with new trail literal L immediately results in a conflict with conflict closure $D \cdot \sigma$, then *Backtrack* is only applicable if $comp(L) \in D\sigma$.
5. During conflict resolution *Skip* is applied at least once except if the conflict is an immediate result of *Decide*.

Now I show that under the assumption of a regular run and a large enough β such that the set of all ground instances $< \beta$ of an input set of clauses N is unsatisfiable, SCL(EQ) will terminate by deriving $\bot$. The proof is done step-by-step by first showing that any SCL(EQ) run terminates under the assumption that β is never increased in Lemma 2.3.15. Either the algorithm gets stuck, which means that no rule of the calculus is applicable anymore, or $D = \bot$, which means that the input set is unsatisfiable. In Lemma 2.3.11, I show that stuck states never occur during conflict resolution and only if all occuring literals $< \beta$ are defined in Γ. Furthermore, I show that conflict resolution will always result in an application of *Backtrack* (Lemma 2.3.12). This shows that SCL(EQ) will terminate by deriving $\bot$, since otherwise *Propagate* or *Decide* would be

applicable, which cannot be the case since then all literals would be defined in Γ and no conflict exists, which contradicts the assumption that N is unsatisfiable.

CC($\mathcal{X}$): Non-Ground Congruence Closure

In this Section I provide a more detailed description of my CC($\mathcal{X}$) calculus, again illustrated by an example. CC($\mathcal{X}$) takes as input non-ground equations and builds non-ground congruence classes, similar to ground congruence closure. As an example, consider the terms $g(x) \approx h(x)$, $h(y) \approx f(y)$ and $a \approx b$ and a maxterm $\beta = f(a)$. Furthermore, assume an ordering that orders the terms by the number of symbols they contain (see Definition 3.4.5). As described in Section 3.2, CC($\mathcal{X}$) operates on a set of constrained classes $\Pi = \{A_1, ..., A_n\}$ (Definition 3.2.3). A constraint class A is defined as a set of constraint terms of the form $\Gamma \parallel s$, where Γ is a constraint and s is a term. Constraints are of the form $t_1 \preceq \beta, ..., t_n \preceq \beta$. The constraint Γ restricts the number of ground instances of a term s to those instances $s\sigma$ such that $\Gamma\sigma$ evaluates to true. For readability I simply write $\{g(x) \parallel g(x)\}$ instead of $\{g(x) \preceq \beta \parallel g(x)\}$ omitting $\preceq \beta$ within the constraint and use the notation $\{\Gamma \parallel s_1, \ldots, s_n\}$ to denote that the constraint Γ holds for all terms $s_1, ..., s_n$. Initially, the set of classes consists of one constrained class for each input equation and, for technical reasons, one single term constrained class for each occurring non-variable symbol.

For our example equations CC($\mathcal{X}$) would initially create the classes:

$$\{\{g(x) \parallel g(x)\}, \{h(x) \parallel h(x)\}, \{f(x) \parallel f(x)\}, \{a \parallel a\},$$
$$\{b \parallel b\}, \{a, b \parallel a, b\}, \{g(x), h(x) \parallel g(x), h(x)\},$$
$$\{h(y), f(y) \parallel h(y), f(y)\}\}$$

For a constraint class A it is important to distinguish between separating and free variables as properly defined in Definition 3.2.3. Separating variables occur in every term of the class, whereas free variables are all the remaining variables.

In Definition 3.2.5, I create a mapping from constraint congruence classes to its corresponding ground classes. Intutitively speaking, we first create different classes for all possible instantiations of the separating variables. Then, for each resulting class we create one class for all possible instantiations of the free variables. I use the terminology $gnd(A)$ to denote the set of ground classes for a constraint class A. Figure 3 shows the process on an example constraint class.

The rules are dependant on the concept of a normal class which is a class where every term with free variables is duplicated with all free variables replaced by fresh variables (Definition 3.2.7). For example, let $A = \{f(x), g(y) \parallel f(x), g(y)\}$ be a class. Then the normal class is $norm(A) = \{f(x), g(y), f(x'), g(y') \parallel f(x), g(y), f(x'), g(y')\}$.

The *Merge* rule now takes two classes from Π that contain two unifiable terms, creates their corresponding normal classes, build the unifier and unions the classes with the unifier applied. Figure 4 shows an example of the application of the *Merge* rule on two classes.

In the example, CC($\mathcal{X}$) can now merge the classes $\{g(x), h(x) \parallel g(x), h(x)\}$ and $\{h(y), f(y) \parallel h(y), f(y)\}$ by unifying $h(x)$ with $h(y)$. So we get

$$A_1 = \{g(x), h(x), f(x) \parallel g(x), h(x), f(x)\}$$

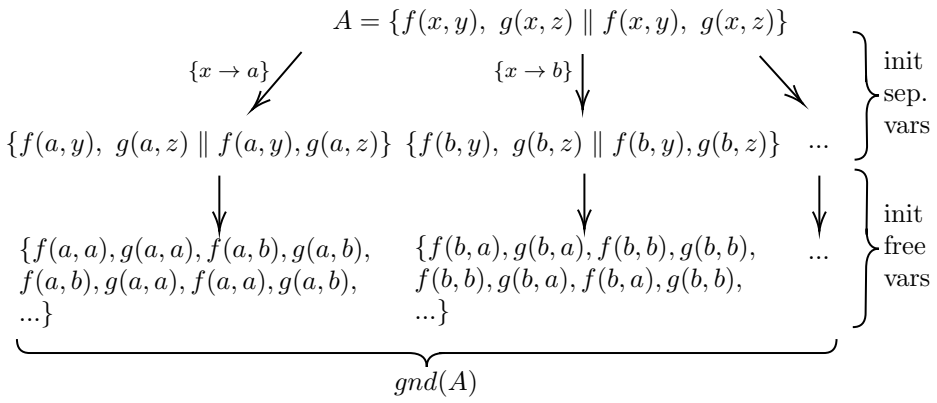

Figure 3: Illustration of the mapping from constrained congruence classes to ground congruence classes on an example

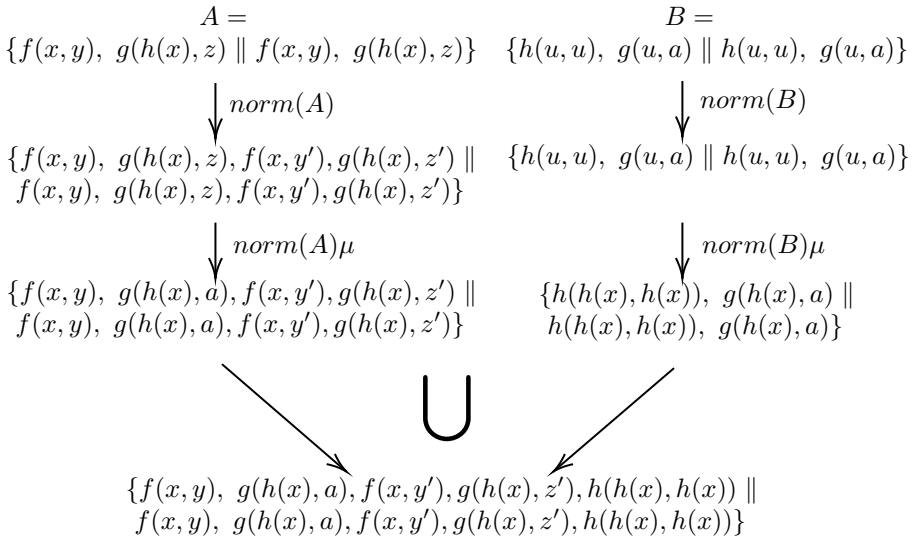

Figure 4: Illustration of the *Merge* rule on example classes A and B, where $\mu = mgu(g(h(x),z), g(u,a)) = \{u \to h(x), z \to a\}$

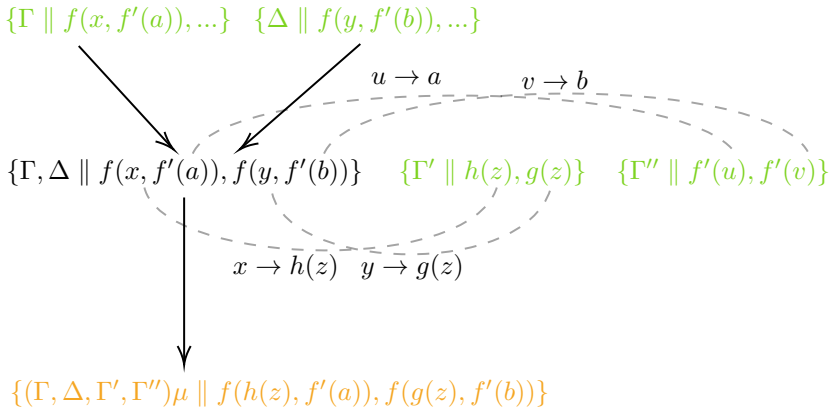

Figure 5: Illustration of the *Deduction* rule on example classes. Green classes are classes from the current Π, the orange class is the new class created by the rule. $\mu = \{u \to a, v \to b, x \to h(z), y \to g(z)\}$ is the simultaneous mgu.

The *Deduction* rule takes two terms from two different classes with the same top level function symbol. Then, for each argument, it tries to find a class such that the arguments are simultaneously unifiable with two terms in that class. The new class then consists of the two terms with the unifier applied and the constraints of all involved classes. Figure 5 shows an example of the application of the *Deduction* rule.

In our example, we can apply *Deduction* followed by two merges to A_1 by creating a variable disjoint copy $\{g(y), h(y), f(y) \parallel g(y), h(y), f(y)\}$ and unifying the arguments, e.g., of $f(x)$ and $f(y)$ with a and b and merging the resulting $f(a)$ and $f(b)$ with the new class to create $A_2 = \{g(a), h(a), f(a), g(b), h(b), f(b) \parallel g(a), h(a), f(a), g(b), h(b), f(b)\}$.

For termination it is crucial to have a proper definition of subsumption. Since I defined a mapping from constraint congruence classes to ground classes, a possible definition is by means of those ground classes, which are sets of ground terms. A ground class B subsumes another ground class A iff $A \subseteq B$. Consequently, a set of ground classes $\mathcal{B}$ subsumes a set of ground classes $\mathcal{A}$ if for all $A \in \mathcal{A}$ there exists a $B \in \mathcal{B}$ such that $A \subseteq B$. Thus, a constrained class B subsumes another constrained class A iff $gnd(B)$ subsumes $gnd(A)$ (Definition 3.2.8). The class A_2 subsumes the class A_1 by the above definition. Also the classes involved in the very first application of *Merge* to create A_1 are subsumed by A_1. Now the algorithm terminates, since the maxterm β restricts the creation of any new class. The resulting class set is:

$\{\{a, b \parallel a, b\}, \{g(a), h(a), f(a), g(b), h(b), f(b) \parallel g(a), h(a), f(a), g(b), h(b), f(b)\}\}$

In Section 3.3, I prove CC($\mathcal{X}$) to be sound (Lemma 3.3.4), complete (Lemma 3.3.6) and terminating (Lemma 3.3.5). Termination is shown by proving that there exist only finitely many constraint classes that are not subsumed by another class. This is the case, since there are only finitely many possible ground congruence classes. Soundness is shown by structural induction on the state. I show that a *Merge* step follows by transitivity of equality and a *Deduction* step follows by congruence of equality. Finally, I prove completeness by structural

induction on the inference system of equational logic [BN98] (see Section 1.1.2). I show that any rule of the inference system can be simulated by a rule of our congruence closure algorithm.

As described in Section 3.4, for the implementation of CC($\mathcal{X}$) I made use of the standard procedure to work through the classes, namely by maintaining a *worked-off* and a *usable* queue. *worked-off* contains all classes where the rules have already been applied to, whereas *usable* contains all classes where the rules have not yet been applied to.

The ordering that was chosen for the constraints is the symbol count order (Definition 3.4.5) as used in the example, where $s \prec t$ iff the number of symbols in s is smaller than the number of symbols in t. It turns out that the constraints together with the symbol count order reduce to linear arithmetic constraints (Lemma 3.4.7). Thus, I made use of the SPASS linear arithmetic solver [BFSW19] for constraint solving. I added several more optimisations to the presented algorithm which I will briefly present below.

Firstly, it is necessary to generalize the subsumption check defined for ground classes to non-ground classes, as the above check is infeasible in practice. I define subsumption by matching (Definition 3.4.2), which first fixes a matcher for the separating variables. Then it checks for each term in the subsumed class whether there is a term and a matcher for the free variables in the subsuming class so that this term matches the term in the subsumed class. I show that subsumption by matching implies subsumption (Lemma 3.4.4), but not vice-versa.

In our example, the class $\{g(x), h(x), f(x) \parallel g(x), h(x), f(x)\}$ actually is not subsumed by matching by the last class. But we have seen that it is subsumed in the ground case.

In the implementation I also have to take care of the constraints. There are cases where variables only occur within constraints but not within the constrained terms of a class. This leads to ever-increasing constraints that the LIA solver can no longer solve although the additional constraints do not cause a further restriction to the constrained term. The solution is to map all the variables that only occur in the constraints to exactly one fresh variable (Lemma 3.4.9). Now we can remove duplicate constraints, preserving their informative value but drastically reducing their size.

Another important optimization is *Condensation*. This new rule reduces the size of a class by matching one term in a class to another and checking if the result subsumes the original class. If this is the case, we can remove the larger class. *Condensation* is an expensive operation, but very important to keep the size of the classes small.

Finally, I make use of bit vectors, path indexes and discrimination trees [McC92] to reduce the number of subsumption checks by filtering out classes. For example, a class A cannot subsume a class B if B contains a term t where no term in A can match t.

In Section 3.5, the evaluation shows that CC($\mathcal{X}$) can outperform CC if grounding gets infeasible. I observed that CC is faster than CC($\mathcal{X}$) if the set of ground instances is small. However, as β increases, the tide turns further and further in favour of CC($\mathcal{X}$). For a very large β, CC($\mathcal{X}$) can at least solve 299 of 2900 problems whereas CC can solve only 102. It is also interesting to see that the number of classes in CC($\mathcal{X}$) is almost always way smaller than the number of classes in CC.

In joint work with Yasmine Briefs [BLW23], we also showed that constraints in

$\mathrm{CC}(\mathcal{X})$ can be effectively computed with her new algorithm for KBO constraints, as described in Section 3.6. In Section 3.6.1, I show that the constraint solver is very fast for an early version of $\mathrm{CC}(\mathcal{X})$, and calculates solutions for the constraints in a reasonable amount of time.

Contribution and Structure

This dissertation makes the following key contributions:

1. In Chapter 2, I present SCL(EQ), the first approach to conflict-driven clause learning in first-order logic with equality. SCL(EQ) ensures learning of non-redundant clauses only, according to my newly defined trail induced ordering (Definition 2.2.9). I prove SCL(EQ) sound (Theorem 2.3.4) and refutationally complete (Theorem 2.3.17).

2. In Chapter 3, I present first steps towards an efficient implementation with the help of my new calculus $\mathrm{CC}(\mathcal{X})$ that is a generalization of the congruence closure algorithm. I prove $\mathrm{CC}(\mathcal{X})$ sound (Lemma 3.3.4), complete (Lemma 3.3.6) and terminating (Lemma 3.3.5). I provide an implemenation that incorporates multiple optimizations as described in Section 3.4. The evaluation in Section 3.5 shows that $\mathrm{CC}(\mathcal{X})$ outperforms standard congruence closure if grounding gets infeasible. In Section 3.6, with joint work with Yasmine Briefs [BLW23], I show that KBO fulfills the requirement of SCL(EQ) on effective algorithms for a term order.

The work is composed from the following papers:

- Hendrik Leidinger and Christoph Weidenbach. SCL(EQ): SCL for first-order logic with equality. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 228–247. Springer, 2022

- Hendrik Leidinger and Christoph Weidenbach. SCL(EQ): SCL for first-order logic with equality. *Journal of Automated Reasoning*, 67(3):22, 2023

- Hendrik Leidinger and Christoph Weidenbach. Non-ground congruence closure. *arXiv preprint arXiv:2412.10066*, 2024

- Yasmine Briefs, Hendrik Leidinger, and Christoph Weidenbach. KBO constraint solving revisited. In Uli Sattler and Martin Suda, editors, *Frontiers of Combining Systems*, volume 14279 of *Lecture Notes in Computer Science*, pages 81–98, Cham, 2023. Springer Nature Switzerland

This thesis largely consists of the contents of these papers, but is extended by more detailed background information, related work and evaluation sections as well as examples that aid understanding. Furthermore, the text has been adapted to create a consistent storyline.

The remainder of this dissertation is now structured as follows. In Chapter 1, I provide all the necessary background knowledge required to understand this dissertation. Chapter 2 presents SCL(EQ) in full detail, including a detailed description of related work and all proofs for the correctness of the calculus.

In Chapter 3, I present CC($\mathcal{X}$) and its implementation, again accompanied by related work, proofs for soundness and completeness and a detailed evaluation of the implementation. Finally I conclude in Chapter 4 and outline opportunities for future work.

Chapter 1

Preliminaries

1.1 Fundamentals

In this Section, I present the basics of automated reasoning that are necessary to understand this thesis, namely first-order logic, equational logic and orderings.

1.1.1 First-Order Logic

I will start with the specification of the syntax and semantics of first-order logic.

Syntax

I first need to define a tuple $\Sigma = (\Omega, \Pi)$, which is the signature of my language. Ω is a finite set of tuples of the form f/n, where f is a function symbol and n is the arity of f, also denoted by $arity(f)$. Π is the finite set of tuples of predicate symbols of the form P/n, where P is the predicate and n its arity. I use P, Q, R for predicate symbols, f, g, h for function symbols of arity at least 1 and a, b, c, d for function symbols of arity 0, also called constants. By $\mathcal{X}$, I denote an infinite set of variables. x, y, z, u, v denote variables from $\mathcal{X}$. I assume that Σ contains at least one constant symbol. A term is either a variable x or a function, where the arguments are again terms. More formally:

Definition 1.1.1 (Term)**.** Let $\Sigma = (\Omega, \Pi)$ be a signature. A term t is recursively defined as follows:

t	$:= f(s_1, ..., s_n)$,	where $f \in \Omega$, $arity(f) = n$ and $s_1, ..., s_n$ are terms,
t	$:= x$	for some $x \in \mathcal{X}$.

For example, let $\Omega = \{f/2, g/1, a/0\}$ and $\mathcal{X} = \{x, y, ...\}$. Then $f(x, a), g(f(a, a))$ and $f(g(x), g(y))$ are all terms. I use s, t, l, r to denote terms. The set of all terms is denoted by $T(\Sigma, \mathcal{X})$. The set of all ground terms is denoted by $T(\Sigma, \emptyset)$ or just $T(\Sigma)$. The set of variables occuring in a term t (in a set of terms $\mathcal{T}$) is denoted by $vars(t)$ (respectively $vars(\mathcal{T})$). The term t is called ground if $vars(t) = \emptyset$. The number of occurences of a variable x in t is denoted by $\#(x, t)$.

Since equality is a particularly important function, especially in this thesis, I will treat it separately instead of representing it as a simple symbol in Π:

Definition 1.1.2. Let s, t be terms from $T(\Sigma, \mathcal{X})$. Then $s \approx t$ is an equation over Σ. I use $s \not\approx t$ as a shortcut for $\neg(s \approx t)$ and write $s \# t$ to denote a literal that is either $s \approx t$ or $s \not\approx t$.

A literal is a positive or negative predicate or equation. I use symbols L, K, H to denote (equational) literals. The function *comp* computes the complement of a literal.

Definition 1.1.3 (Formulas)**.** A formula ψ is inductively defined as follows

ψ	$:= \top$,	
ψ	$:= \bot$,	
ψ	$:= P(t_1, ..., t_n)$	for terms $\{t_1, ..., t_n\} \subseteq T(\Sigma, \mathcal{X})$ and $P \in \Pi$,
ψ	$:= s \approx t$	for $\{s, t\} \subseteq T(\Sigma, \mathcal{X})$,
ψ	$:= \neg\phi$	for a formula ϕ,
ψ	$:= \phi \circ \chi$	for formulas ϕ, χ and $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$,
ψ	$:= \forall x.\phi$	for a formula ϕ,
ψ	$:= \exists x.\phi$	for a formula ϕ,

For a signature $\Sigma = \{\{f/1, g/1, h/1, a/0\}, \{P/2, Q/1\}\}$ a valid formula would be $\forall x.\exists y.P(f(a), a) \wedge f(x) \approx g(h(y)) \vee Q(a)$. The set of all formulas is denoted by $F(\Sigma, \mathcal{X})$.

Definition 1.1.4 (Conjunctive Normal Form)**.** A formula ψ is in Conjunctive Normal Form (CNF) iff ψ is of the form $\forall x_1, ..., x_n. \bigwedge_{0 \leq i \leq m} (\bigvee_{0 \leq j \leq l} L_{i,j})$, where the $L_{i,j}$'s are literals.

Any first-order formula can be transformed into an equisatisfiable formula in CNF. This thesis only operates on formulas that are in Conjunctive Normal Form. A clause is a disjunction of literals and can be represented as a set of literals. I use symbols C, D to denote clauses, write $\{C_1, ..., C_n\}$ for a CNF formula $C_1 \wedge ... \wedge C_n$ and use N to denote such sets of clauses.

Note, that any signature $\Sigma = (\Omega, \Pi)$ with $|\Omega| > 1$ can be turned into a signature $\Sigma' = (\Omega', \emptyset)$, where $\Omega' = \Omega \cup \{f_P \mid P \in \Pi, f_P \text{ fresh}\} \cup \{true\}$. Replacing all occurences of a predicate $P(t_1, ..., t_n)$ in a formula by $f_P(t_1, ..., t_n) \approx true$ we get a semantically equivalent formula. In this thesis the signature is always assumed to have only one predicate symbol $\approx$.

Substitutions are a very important function in first-order logic. They replace variables in $\mathcal{X}$ with terms in $T(\Sigma, \mathcal{X})$

Definition 1.1.5 (Substitutions)**.** A substitution σ is a mapping $\mathcal{X} \rightarrow T(\Sigma, \mathcal{X})$. Its finite domain is defined as $dom(\sigma) := \{x \mid x\sigma \neq x\}$, and its codomain is defined as $cdom(\sigma) := \{t \mid x\sigma = t, x \in dom(\sigma)\}$. We recursively extend its application to terms and formulas as follows:

$\top\sigma$	$:= \top$,
$\bot\sigma$	$:= \bot$,
$f(t_1, ..., t_n)\sigma$	$:= f(t_1\sigma, ..., t_n\sigma)$,
$P(t_1, ..., t_n)\sigma$	$:= P(t_1\sigma, ..., t_n\sigma)$,
$(s \approx t)\sigma$	$:= s\sigma \approx t\sigma$,
$(\neg\psi)\sigma$	$:= \neg(\psi\sigma)$,
$(\psi \circ \phi)\sigma$	$:= \psi\sigma \circ \phi\sigma$,
$(Qx.\psi)\sigma$	$:= Qz.(\psi\sigma[x \mapsto z])$, where $Q \in \{\forall, \exists\}$, z fresh

where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

A substitution σ is called ground if $cdom(\sigma)$ is ground. A substitution σ is called grounding for some term t (formula ψ) if $t\sigma$ ($\psi\sigma$) is ground.

I use symbols $\sigma, \tau, \delta, \mu$ to denote substitutions. The function *gnd* computes the set of all ground instances of a term or formula.

As an example, assume $\sigma = \{x \rightarrow a, y \rightarrow g(z)\}$. Then $f(x, y)\sigma = f(a, g(z))$. Closures are denoted by $C\cdot\sigma$ and $L\cdot\sigma$ for a clause C, a literal L and a grounding substitution σ. A closure does not apply the substitution to the clause or the literal, but is more of a storage of the combination consisting of the clause and the substitution for possible later applications. Special forms of substitution are matchers and most general unifiers, which are defined as follows:

Definition 1.1.6 (Matchers and Unifiers)**.** Let σ be a substitution and s, t terms. The substitution σ is called a matcher from s to t if $s\sigma = t$. The substitution σ is called a unifier of s and t if $s\sigma = t\sigma$. The substitution σ is called the most general unifier (mgu) of s and t if it is a unifier and for all other unifiers μ of s and t it holds that it can be represented as $\mu = \sigma\mu'$, where μ' is a substitution.

I assume that mgus do not introduce fresh variables and that they are idempotent. While substitutions can replace variables with terms, they are not able to replace subterms of a term with other terms. Therefore, I need one operator to get subterms of terms and one to replace subterms in a term by other terms. To achieve this, I first need a definition of term positions.

Definition 1.1.7. The set of positions of a term, equation or formula is inductively defined as follows:

$pos(x)$	$:= \{\epsilon\}$
$pos(\psi)$	$:= \{\epsilon\}$ if $\psi \in \{\top, \bot\}$
$pos(\neg\psi)$	$:= \{\epsilon\} \cup \{1p \mid p \in pos(\psi)\}$
$pos(\psi \circ \phi)$	$:= \{\epsilon\} \cup \{1p \mid p \in pos(\psi)\} \cup \{2p \mid p \in pos(\phi)\}$
$pos(s \approx t)$	$:= \{\epsilon\} \cup \{1p \mid p \in pos(s)\} \cup \{2p \mid p \in pos(t)\}$
$pos(f(t_1, ..., t_n))$	$:= \{\epsilon\} \cup \bigcup_{i=0}^{n} \{ip \mid p \in pos(t_i)\}$
$pos(P(t_1, ..., t_n))$	$:= \{\epsilon\} \cup \bigcup_{i=0}^{n} \{ip \mid p \in pos(t_i)\}$
$pos(Qx.\psi)$	$:= \{\epsilon\} \cup \{1p \mid p \in pos(\psi)\}$, where $Q \in \{\forall, \exists\}$

where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

Any subterm (subformula) of a term (formula) can now be obtained using the $|$ operator, which returns the subterm (subformula) of a formula or term at some position.

Definition 1.1.8. The subterm $t|_p$ (subformula $\psi|_p$) of a term t (formula ψ) at position p is recursively defined as

$t\|_\epsilon$	$:= t$,	
$\psi\|_\epsilon$	$:= \psi$,	
$\neg\psi\|_{1p}$	$:= \psi\|_p$,	
$s_1 \approx s_2\|_{ip}$	$:= s_i\|_p$,	$i \in \{1, 2\}$,
$\psi_1 \circ \psi_2\|_{ip}$	$:= \psi_i\|_p$,	$i \in \{1, 2\}$,
$f(s_1, ..., s_n)\|_{ip}$	$:= s_i\|_p$,	$i \in \{1, ...n\}$,
$P(s_1, ..., s_n)\|_{ip}$	$:= s_i\|_p$,	$i \in \{1, ...n\}$,
$Qx.\psi\|_{1p}$	$:= \psi\|_p$,	$Q \in \{\forall, \exists\}$

where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

Analogously, I can replace any subterm or subformula with the [] operator, which replaces a subterm (subformula) of a term or formula by another term (formula) at some position.

Definition 1.1.9. The replacement of a formula or term is inductively defined as follows

$t[s]_\epsilon$	$:= s$,	
$\psi[\phi]_\epsilon$	$:= \phi$,	
$\neg\psi[\phi]_{1p}$	$:= \psi[\phi]_p$,	
$\neg\psi[s]_{1p}$	$:= \psi[s]_p$,	
$s_1 \approx s_2[s]_{ip}$	$:= s_i[s]_p$,	$i \in \{1, 2\}$,
$\psi_1 \circ \psi_2[s]_{ip}$	$:= \psi_i[s]_p$,	$i \in \{1, 2\}$,
$\psi_1 \circ \psi_2[\phi]_{ip}$	$:= \psi_i[\phi]_p$,	$i \in \{1, 2\}$,
$f(s_1, ..., s_n)[s]_{ip}$	$:= s_i[s]_p$,	$i \in \{1, ...n\}$,
$P(s_1, ..., s_n)[s]_{ip}$	$:= s_i[s]_p$,	$i \in \{1, ...n\}$,
$Qx.\psi[\phi]_{1p}$	$:= \psi[\phi]_p$,	$Q \in \{\forall, \exists\}$
$Qx.\psi[s]_{1p}$	$:= \psi[s]_p$,	$Q \in \{\forall, \exists\}$

where $\circ \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

For example, the term $f(a, g(x))$ has the positions $\{\epsilon, 1, 2, 21\}$, $f(a, g(x))|_{21} = x$ and $f(a, g(x))[b]_2$ denotes the term $f(a, b)$.

Another important operator that can be defined by positions is the size of a term, which corresponds to the number of its symbols. It turns out that the number of symbols is exactly the number of positions of t.

Definition 1.1.10. The size $|t|$ of a term t or $|\psi|$ of a formula ψ is defined as $|pos(t)|$ and $|pos(\psi)|$, respectively.

Semantics

The semantics of first-order logic are defined by a Σ-Algebra.

Definition 1.1.11 (Σ-Algebra)**.** Let $\Sigma = (\Omega, \Pi)$ be a signature. The Σ-Algebra $\mathcal{A}$ is defined as a triple

$$(U_\mathcal{A}, \{f_\mathcal{A} : U_\mathcal{A}^n \rightarrow U_\mathcal{A} \mid f \in \Omega, arity(f) = n\}, \{P_\mathcal{A} \subseteq U_\mathcal{A}^m \mid P \in \Pi, arity(P) = m\})$$

where $U_\mathcal{A}$ is called the universe of $\mathcal{A}$ and is non-empty.

Definition 1.1.12 (Variable Assignment)**.** A variable assignment over a Σ-Algebra $\mathcal{A}$ is a map $\beta : \mathcal{X} \rightarrow U_\mathcal{A}$.

Now I can define the value of a term and the truth value of a formula with a given Σ-Algebra $\mathcal{A}$ and a variable assignment β.

Definition 1.1.13 (Value of a Term)**.** Let $\mathcal{A}$ be a Σ-Algebra and β a variable assignment. The function $\mathcal{A}(\beta) : T(\Sigma, \mathcal{X}) \rightarrow U_\mathcal{A}$ is recursively defined as follows:

$\mathcal{A}(\beta)(x) = \beta(x)$,
$\mathcal{A}(\beta)(f(t_1, ..., t_n)) = f_\mathcal{A}(\mathcal{A}(\beta)(t_1), ..., \mathcal{A}(\beta)(t_n))$

Definition 1.1.14 (Truth Value of a Formula)**.** Let $\mathcal{A}$ be a Σ-Algebra, β a variable assignment. The function $\mathcal{A}(\beta) : F(\Sigma, \mathcal{X}) \to \{0, 1\}$ is recursively defined as follows:

$\mathcal{A}(\beta)(\top)$	$:= 1$,
$\mathcal{A}(\beta)(\bot)$	$:= 0$,
$\mathcal{A}(\beta)(P(s_1, ..., s_n))$	$:=$ if $(\mathcal{A}(\beta)(s_1), ..., \mathcal{A}(\beta)(s_n)) \in P_{\mathcal{A}}$ then 1 else 0,
$\mathcal{A}(\beta)(s \approx t)$	$:=$ if $\mathcal{A}(\beta)(s) = \mathcal{A}(\beta)(t)$ then 1 else 0,
$\mathcal{A}(\beta)(\neg\psi)$	$:= 1 - \mathcal{A}(\beta)(\psi)$,
$\mathcal{A}(\beta)(\psi \wedge \phi)$	$:= min(\mathcal{A}(\beta)(\psi), \mathcal{A}(\beta)(\phi))$,
$\mathcal{A}(\beta)(\psi \vee \phi)$	$:= max(\mathcal{A}(\beta)(\psi), \mathcal{A}(\beta)(\phi))$,
$\mathcal{A}(\beta)(\psi \to \phi)$	$:= max(1 - \mathcal{A}(\beta)(\psi), \mathcal{A}(\beta)(\phi))$,
$\mathcal{A}(\beta)(\psi \leftrightarrow \phi)$	$:=$ if $\mathcal{A}(\beta)(\psi) = \mathcal{A}(\beta)(\phi)$ then 1 else 0,
$\mathcal{A}(\beta)(\forall x.\psi)$	$:= min_{a \in U_{\mathcal{A}}}\{\mathcal{A}(\beta[x \mapsto a])(\psi)\}$,
$\mathcal{A}(\beta)(\exists x.\psi)$	$:= max_{a \in U_{\mathcal{A}}}\{\mathcal{A}(\beta[x \mapsto a])(\psi)\}$,

A formula ψ is called satisfiable by $\mathcal{A}$ under β if $\mathcal{A}(\beta)(\psi) = 1$

Definition 1.1.15 (Entailment)**.** Let ψ, ϕ be two formulas. If for all Σ-Algebras $\mathcal{A}$ and all variable assignments β it holds that if $\mathcal{A}(\beta)(\psi) = 1$, then $\mathcal{A}(\beta)(\phi) = 1$ then ψ entails ϕ. It is written $\psi \models \phi$.

1.1.2 Equational Logic

As already mentioned, equality occupies a special status in first-order logic. Naively, one could add equality as a symbol in Π and add the equality axioms to the set of input clauses. In practice, however, this turns out to be extremely inefficient. In the past, therefore, a more efficient way was found, namely to represent the equations as a term rewrite system. This Section focuses on such term rewrite systems to solve the equational problems.

Term Rewrite Systems

Rewrite systems are defined as follows:

Definition 1.1.16 (Rewrite System)**.** A rewrite system is a tuple $(M, \to)$, where M is a set of elements and $\to \subseteq M \times M$ is a binary relation.

Definition 1.1.17. Let $(M, \to)$ be a rewrite system. Then

$\to^0$	$:= \{(a, a) \mid a \in M\}$
$\to^{i+1}$	$:= \to^i \circ \to$
$\to^*$	$:= \bigcup_{i \geq 0} \to^i$
$\leftarrow$	$:= \{(b, c) \mid (c, b) \in \to\}$
$\leftrightarrow$	$:= \to \cup \leftarrow$
$\leftrightarrow^*$	$:= (\leftrightarrow)^*$

Assume that $M = \{a, b, c\}$ and $\to = \{a \to b, b \to c, c \to a\}$. Then $\to^0 = \{a \to a, b \to b, c \to c\}$, $\to^1 = \{a \to b, b \to c, c \to a\}$, $\to^2 = \{a \to c, b \to a, c \to b\}$.

Definition 1.1.18. Let $(M, \to)$ be a rewrite system. An element $a \in M$ is called reducible if $a \to b$ for some $b \in M$, otherwise a is called in *normal form*. An element b is called a normal form of a, denoted $a\downarrow = b$, iff $a \to^* b$ and b is in normal form. Two elements $\{b, c\} \subseteq M$ are called joinable, denoted $b \downarrow c$, iff there exists an $a \in M$ such that $b \to^* a$ and $c \to^* a$.

Definition 1.1.19. Let $(M, \rightarrow)$ be a rewrite system. $\rightarrow$ is called

1. *locally confluent* if $a \rightarrow b$ and $a \rightarrow c$ implies $b \downarrow c$.
2. *confluent* if $a \rightarrow^* b$ and $a \rightarrow^* c$ implies $b \downarrow c$.
3. *terminating* if there exists no infinite descending chain $a_0 \rightarrow a_1 \rightarrow ...$
4. *convergent* if it is confluent and terminating.

Definition 1.1.20 (Rewrite Rule)**.** A rewrite rule is a equation $s \approx t$ such that $vars(t) \subseteq vars(s)$ and s is not a variable. A term rewrite system (TRS) is a set of rewrite rules.

Definition 1.1.21 (Rewrite Relation)**.** Let E be a set of equations. The rewrite relation $\rightarrow_E \subseteq T(\Sigma, \mathcal{X}) \times T(\Sigma, \mathcal{X})$ is defined as $s \rightarrow_E t$ iff there exists a $l \approx r \in E$, $p \in pos(s)$ and a matcher σ such that $s|_p = l\sigma$ and $t = s[r\sigma]_p$.

A TRS R is terminating, confluent, convergent, if the rewrite relation $\rightarrow_R$ is terminating, confluent, convergent.

Definition 1.1.22 (Term Normal Form)**.** Let E be a TRS and $\rightarrow_E$ be the corresponding rewrite relation. We write $s = t{\downarrow_E}$ if s is the normal form of t in the rewrite relation $\rightarrow_E$. We write $s\#t = (s'\#t'){\downarrow_E}$ if s is the normal form s' and t is the normal form of t'.

Definition 1.1.23 (Reducibility and Irreducibility)**.** Let R be a set of rewrite rules. A term t is called irreducible by R if no rule in R rewrites t. Otherwise it is called reducible. A literal, clause is called irreducible if all of its terms are irreducible, and reducible otherwise. A substitution σ is called irreducible if any $t \in cdom(\sigma)$ is irreducible, otherwise this substitution is called reducible.

E-Algebras

Let E be a set of equations over $T(\Omega, \mathcal{X})$ where all variables are implicitly universally quantified. The well-known inference system of equational logic comprises the following rules [BN98]

Reflexivity $E \Rightarrow_{\mathrm{EQ}} E \cup \{t \approx t\}$
provided t is a term.

Symmetry $E \cup \{t \approx t'\} \Rightarrow_{\mathrm{EQ}} E \cup \{t \approx t', t' \approx t\}$

Transitivity $E \cup \{t \approx t', t' \approx t''\} \Rightarrow_{\mathrm{EQ}} E \cup \{t \approx t', t' \approx t'', t \approx t''\}$

Congruence $E \cup \{t_1 \approx t'_1, ..., t_n \approx t'_n\} \Rightarrow_{\mathrm{EQ}} E \cup \{t_1 \approx t'_1, ..., t_n \approx t'_n, f(t_1, ..., t_n) \approx f(t'_1, ..., t'_n)\}$

Instance $E \cup \{t \approx t'\} \Rightarrow_{\mathrm{EQ}} E \cup \{t \approx t, t\sigma \approx t'\sigma\}$
provided σ is a substitution.

and by Bikhoff's theorem [BN98] we get:

Theorem 1.1.24 (Bikhoff's theorem [BN98])**.** Let E be a set of equations, Σ a signature and $\mathcal{X}$ a countably inifinite set of variables. Then for all $\{s,t\} \subseteq T(\Sigma,\mathcal{X})$ the following properties are equivalent:

1. $s \leftrightarrow_E^* t$,
2. $E \Rightarrow_{EQ}^* \{s \approx t\} \cup E'$ is derivable,
3. $E \models \forall x_1,..,x_n.s \approx t$.

where $\Rightarrow_{EQ}^*$ denotes multiple applications of $\Rightarrow_{EQ}$.

1.1.3 Orderings

One question that arises with term rewrite systems is in which direction a term should be rewritten. Ideally, a term should become smaller and smaller until it has reached its normal form. However, the question is when a term is actually smaller than another term. At this point, orderings come into play that can at least partially answer this question. The first step is to define various orderings.

Definition 1.1.25 (Orderings)**.** Let M be a set. A *partial ordering* $\preceq \subseteq M \times M$ on a set M is a reflexive, antisymmetric and transitive binary relation on M. A *total ordering* is a partial ordering that is total. A strict partial ordering $\prec \subseteq M \times M$ is a transitive and irreflexive binary relation on M.

Definition 1.1.26. Let $\prec, \preceq$ be orderings. Then $\succ = \{(b,a) \mid a \prec b\}$, $\succeq = \{(b,a) \mid a \preceq b\}$

As already mentioned, ideally I want a term rewrite system to make a term smaller and smaller up to a fixed point. For an ordering this means that it has to be well-founded.

Definition 1.1.27 (Well-Foundedness)**.** Let M be a set. A strict ordering $\prec \subseteq M \times M$ is *well-founded* if there is no infinite descending chain $a_0 \succ a_1 \succ ...$ with $a_i \in M$.

Furthermore, I need the following two properties of a rewrite system.

Definition 1.1.28. A binary relation $\sqsupset \subseteq T(\Sigma,\mathcal{X}) \times T(\Sigma,\mathcal{X})$ is *compatible with* Σ*-operations* if $s \sqsupset t$ implies $f(t_1,...,s,...,t_n) \sqsupset f(t_1,...,t,...,t_n)$ for all $f \in \Omega$ and $\{s,t,t_1,...,t_n\} \subseteq T(\Sigma,\mathcal{X})$

Definition 1.1.29 (Substitution Stable Relation)**.** A binary relation $\sqsupset \subseteq T(\Sigma,\mathcal{X}) \times T(\Sigma,\mathcal{X})$ is *stable under substitutions* if $s \sqsupset t$ implies $s\sigma \sqsupset t\sigma$ for all $\{s,t\} \subseteq T(\Sigma,\mathcal{X})$ and substitutions σ.

Definition 1.1.30 (Rewrite Relation, Rewrite Ordering)**.** A binary relation $\sqsupset \subseteq T(\Sigma,\mathcal{X}) \times T(\Sigma,\mathcal{X})$ is a *rewrite relation* if it is compatible with Σ-operations and stable under substitutions. A *rewrite ordering* is an ordering that is a rewrite relation.

Definition 1.1.31 (Reduction Ordering)**.** A well-founded rewrite ordering is called a *reduction ordering*.

Definition 1.1.32 (Lexicographic Ordering)**.** Let $(M_1, \prec_1), ..., (M_n, \prec_n)$ be strict orderings. The lexicographic ordering $\prec_{lex}$ is defined as $(m_1, ..., m_n) \prec (m'_1, ..., m'_n)$ iff $m_1 = m'_1, ..., m_i = m'_i$ and $m_{i+1} \prec_{i+1} m'_{i+1}$ for some $0 \leq i \leq n$.

Definition 1.1.33 (Multisets)**.** Let M be a set. A multiset S of M is a mapping $S : M \to \mathbb{N}$.

Definition 1.1.34 (Multiset Ordering)**.** Let $(M, \prec)$ be a strict ordering. The multiset ordering $\prec_{mul}$ to multisets over M is defined by $S_1 \prec_{mul} S_2$ iff $S_1 \neq S_2$ and $\forall m \in M.[S_2(m) < S_1(m) \to \exists m' \in M.(m' \prec m \wedge S_1(m') < S_2(m'))]$

Let $\prec$ be a well-founded, total, strict ordering on ground literals, which is lifted to clauses and clause sets by its respective multiset extension. I overload $\prec$ for literals, clauses, clause sets if the meaning is clear from the context. The ordering is lifted to the non-ground case via instantiation: I define $C \prec D$ if for all grounding substitutions σ it holds that $C\sigma \prec D\sigma$. Then, I define $\preceq$ as the reflexive closure of $\prec$ and $N^{\preceq C} := \{D \mid D \in N \text{ and } D \preceq C\}$ and use the standard Superposition style notion of redundancy [BG94].

Definition 1.1.35 (Clause Redundancy [BG94])**.** A ground clause C is *redundant* with respect to a set N of ground clauses and an ordering $\prec$ if $N^{\preceq C} \models C$. A clause C is *redundant* with respect to a clause set N and an ordering $\prec$ if for all $C' \in gnd(C)$, C' is redundant with respect to $gnd(N)$.

The Knuth-Bendix Ordering

The ordering that is used extensively in this thesis is the Knuth-Bendix Ordering (KBO) [KB70] as defined in the following.

Definition 1.1.36 (KBO Weight Function)**.** Let $\Sigma = (\Omega, \Pi)$ be a signature and $\mathcal{X}$ a set of variables. We define the *KBO Weight Function* $w_{kbo} : \Omega \cup X \to \mathbb{R}^+$ such that

1. $w_{kbo}(x) = w_0 \in \mathbb{R}^+$ for all variables $x \in \mathcal{X}$,
2. $w_{kbo}(c) \geq w_0$ for all constants $c \in \Omega$ and
3. $w_{kbo}(f(t_1, ..., t_n)) = w_{kbo}(f) + \Sigma_{1 \leq i \leq n} w_{kbo}(t_i)$.

Definition 1.1.37 (Knuth-Bendix Ordering [KB70])**.** Let $\Sigma = (\Omega, \Pi)$ be a finite signature,w_{kbo} a *KBO Weight Function* and $\prec$ be a strict partial ordering on Ω. Then $s \succ_{kbo} t$ iff

1. $\#(x, s) \geq \#(x, t)$ for all variables x and $w_{kbo}(s) > w_{kbo}(t), or$
2. $\#(x, s) \geq \#(x, t)$ for all variables x, $w_{kbo}(s) = w_{kbo}(t)$ and
   (a) $s = f(s_1, .., s_n)$, $t = g(t_1, ..., t_m)$ and $f \succ g$, or
   (b) $s = f(s_1, .., s_n)$, $t = f(t_1, ..., t_n)$ and $(s_1, ..., s_n)(\succ_{kbo})_{lex}(t_1, ..., t_n)$.

As an example consider a signature $\Sigma = (\Omega, \Pi)$, where $\Omega = \{f, g, h\}$. Further assume a weight funtion such that

$$w_{kbo}(f) = 2, w_{kbo}(g) = w_{kbo}(h) = w_{kbo}(x) = 1 \text{ for all } x \in \mathcal{X}$$

and an ordering on Ω such that $f \succ g \succ h \succ a$. Then $f(x, y) \succ_{kbo} g(x, y)$ by the first condition, $g(x, y) \succ_{kbo} h(x, y)$ by condition $2a$ and $f(f(x, y), z) \succ_{kbo} f(x, f(y, z))$ by condition $2b$.

1.2 Basic Calculi

This Section presents basic calculi. The new calculi presented in this dissertation are strongly inspired by these basic calculi and try to extend or improve them.

1.2.1 CDCL

CDCL (Conflict Driven Clause Learning) [SS96, JS96, MMZ+01, BHvMW09, Wei15] is a SAT solver for propositional clauses. The signature in propositional logic is always of the form $\Sigma = (\emptyset, \Pi)$, where for all $P \in \Pi$ it holds $arity(P) = 0$. Formulas are built without quantifiers and equations. Propositional logic is a decidable fragment of first-order logic.

In a refined and verified version of CDCL as described in [BFLW18] a state in CDCL is a quintuple $(M; N; U; k; D)$, where M is a sequence of annotated literals, N is the set of input clauses, U is the set of learned clauses, k a level and D a conflict clause. Initially the state is $(\epsilon; N; \emptyset; 0; \top)$. The rules of CDCL are as follows:

Propagate $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (ML^{C \vee L}; N; U; k; \top)$
provided $C \vee L \in (N \cup U)$, $M \models \neg C$, and L is undefined in M

Decide $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (ML^{k+1}; N; U; k+1; \top)$
provided L is undefined in M

Conflict $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (M; N; U; k; D)$
provided $D \in (N \cup U)$ and $M \models \neg D$

Skip $(ML^{C \vee L}; N; U; k; D) \Rightarrow_{\text{CDCL}} (M; N; U; k; D)$
provided $D \notin \{\top, \bot\}$, and $comp(L)$ does not occur in D

Resolve $(ML^{C \vee L}; N; U; k; D \vee comp(L)) \Rightarrow_{\text{CDCL}} (M; N; U; k; D \vee C)$
provided D is of level k

Backtrack $(M_1 K^{i+1} M_2; N; U; k; D \vee L) \Rightarrow_{\text{CDCL}} (M_1 L^{D \vee L}; N; U \cup \{D \vee L\}; i; \top)$
provided L is of level k and D is of level i

Restart $(M; N; U; k; \top) \Rightarrow_{\text{CDCL}} (\epsilon; N; U; 0; \top)$
provided $M \not\models N$

Forget $(M; N; U \cup \{C\}; k; \top) \Rightarrow_{\text{CDCL}} (M; N; U; k; \top)$
provided $M \not\models N$

A literal L is of level i if L^i or $comp(L)^i$ occurs in the trail M. A literal L is also of level i if L^C or $comp(L)^C$ occurs in M and its preceding literal in M is of level i. If there is no preceding literal, then it is of level 0.

CDCL can terminate in different states depending on the input set N. The state $(M; N; U; k; \bot)$ indicates that the input clause set is unsatisfiable. The

state $(M; N; U; k; \top)$ indicates that the input clause set is satisifiable and M is a model of N if no rule is applicable anymore.

As an example consider three clauses

$$P \vee Q,\ \neg Q \vee R \text{ and } \neg R \vee \neg Q$$

A run of CDCL could now for example decide Q. So Q^1 is added to the trail. Then, the first clause gets *true*. Since the second clause has only one undefined literal R and all the other literals are *false* in M (namely $\neg Q$), we have to apply Propagate to add $R^{\neg Q \vee R}$ to the trail. This leads to a conflict with the third clause, so we have to apply the same-named rule Conflict. Since both literals in the conflict clause are of level 1 we can not apply Backtrack. Also, Skip is not applicable since $comp(\neg R)$ is the last literal on the trail. Thus we have to apply Resolve. This rule creates the new clause $\neg Q$ on which we can now apply Backtrack. Now, we have to Propagate $\neg Q^{\neg Q}$ and $P^{P \vee Q}$ to get the final solution to the input clause set.

1.2.2 Superposition

Ganzinger et al [BG94] describe the Superposition calculus upon which many of today's approaches are based. It can be seen as an integration of classical term rewriting into the resolution calculus. Given a reduction order $\succ$ on terms which is extended to an ordering on literals and clauses as described in Section 1.1.3, the basic Superposition rules are as follows:

Superposition $(N \cup \{D \vee t \approx t', C \vee s[u] \ \# \ s'\}) \ \Rightarrow_{\text{SUP}} \ (N \cup \{D \vee t \approx t', C \vee s[u] \ \# \ s'\} \cup \{(D \vee C \vee s[t'] \ \# \ s')\sigma\})$

provided $\sigma = mgu(t, u)$ and u is not a variable, $t\sigma \not\preceq t'\sigma, s\sigma \not\preceq s'\sigma$, $(t \approx t')\sigma$ strictly maximal in $(D \vee t \approx t')\sigma$, $(s \approx s')\sigma$ maximal in $(C \vee s \ \# \ s')\sigma$ and $(C \vee s \ \# \ s')\sigma \not\preceq (D \vee t \approx t')\sigma$

Equality-Resolution $(N \cup \{C \vee s \not\approx s'\}) \ \Rightarrow_{\text{SUP}} \ (N \cup \{C \vee s \not\approx s'\} \cup \{C\sigma\})$

where σ is the mgu of s, s', $(s \not\approx s')\sigma$ maximal in $(C \vee s \not\approx s')\sigma$

Equality-Factoring $(N \cup \{C \vee s' \approx t' \vee s \approx t\}) \ \Rightarrow_{\text{SUP}} \ (N \cup \{C \vee s' \approx t' \vee s \approx t\} \cup \{(C \vee t \not\approx t' \vee s \approx t')\sigma\})$

where σ is the mgu of s, s', $s'\sigma \not\preceq t'\sigma$, $s\sigma \not\preceq t\sigma$, $(s \approx t)\sigma$ maximal in $(C \vee s' \approx t' \vee s \approx t)\sigma$

Now let N be a set of clauses. A run is a sequence $N_0, N_1, ..., N_n$ such that $N = N_0$ and $N_{i+1} = N_i \cup \{C\}$, where C is a new clause generated by applying one of the above rules to clauses in N_i. For completeness it is shown that if N_n is saturated up to redundancy then $\bot \in N_n$ if and only if N is unsatisifiable.

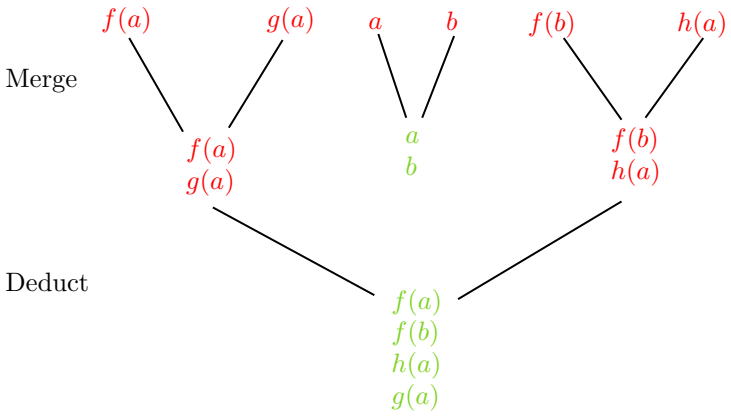

Figure 1.1: Steps of the congruence closure algorithm for example 1.2.1. Green classes represent the final result. Red classes are classes that no longer exist in the final result.

1.2.3 Congruence Closure

Congruence Closure [NO80, DST80, Sho84] is an algorithm for deciding the satisfiability of ground equations. It is well suited as an incremental algorithm that decides the validity of ground equations and is able to find a small subset of input equations that serves as a proof, all in time $\mathcal{O}(n\ log(n))$ [NO07]. In this thesis, I present the abstract version of congruence closure as described by Fontaine [Fon04] and based on the algorithm by Nelson and Oppen [NO80]. The initial state is (Π, E), where Π is a partition of all ground terms, such that every term is in its own class, and E is the set of ground equations. The algorithm consists of the following three inference rules.

Delete $(\{A\} \cup \Pi, E \cup \{s \approx t\}) \Rightarrow_{\text{CC}} (\{A\} \cup \Pi, E)$
provided $\{s, t\} \subseteq A$.

Merge $(\{A, B\} \cup \Pi, E \cup \{s \approx t\}) \Rightarrow_{\text{CC}} (\{A \cup B\} \cup \Pi, E)$
provided $s \in A$, $t \in B$ and $A \neq B$.

Deduction $(\{A, B\} \cup \Pi, E) \Rightarrow_{\text{CC}} (\{A, B\} \cup \Pi, E \cup \{f(s_1, ..., s_n) \approx f(t_1, ..., t_n)\})$
provided $f(s_1, ..., s_n) \in A$, $f(t_1, ..., t_n) \in B$, $A \neq B$ and for each i, there exists a $D_i \in \{A, B\} \cup \Pi$ such that $\{s_i, t_i\} \in D_i$ and $f(s_1, ..., s_n) \approx f(t_1, ..., t_n) \notin E$.

The algorithm terminates if no rule is applicable anymore. The resulting set Π represents the set of congruence classes.

Example 1.2.1. Assume for example the set $E = \{a \approx b, f(a) \approx g(a), f(b) \approx h(a)\}$. Initially, the algorithm creates classes for each occuring term and subterm. Then, Merge can be applied three times for the three equations. Since $a \approx b$ Deduct is applicable as well for $f(a)$ and $f(b)$. The steps and the final result are depicted in figure 1.1.

Chapter 2

$SCL(EQ)$: SCL for First-Order Logic with Equality

In this Chapter, I provide a detailed description of my new calculus SCL(EQ) [LW23] accompanied by comprehensive examples, proofs for soundness and completeness as well as an extensive description of related work. I start with related work in Section 2.1. Then, I present my calculus in Section 2.2 and proof soundness and completeness in Sections 2.3 and 2.4. Finally, I conclude and discuss improvements and future work in Section 2.5. This work was mainly conducted by me. Christoph Weidenbach was involved in the exchange of ideas and the final polishing of the paper.

2.1 Related Work

SCL(EQ) is based on the idea of Simple Clause Learning (SCL) by Bromberger et al. [BSW23]. Similar to my approach, a simple model representation is established, namely a sequence of ground literals. The problem state is given by a six-tuple $(\Gamma; N; U; \beta; k; D)$, where Γ is the aforementioned sequence of ground literals, N and U are the sets of initial and learned clauses, β restricts the number of ground clauses and literals to those smaller than β, k is the current decision level and D is the current conflict clause, wich is $\top$ if there is no conflict. Similar to CDCL, the rules are now divided into two parts: *conflict search* and *conflict resolution* rules. Conflict search rules propagate and decide ground literals until a conflicting ground instance of a clause is found. They are defined as a set of deduction rules as follows:

Propagate $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL}} (\Gamma, L\sigma^{(C_0 \vee L)\delta \cdot \sigma}; N; U; \beta; k; \top)$
provided $C \vee L \in (N \cup U)$, $C = C_0 \vee C_1$, $C_1\sigma = L\sigma \vee ... \vee L\sigma$, $C_0\sigma$ does not contain $L\sigma$, δ is the mgu of the literals in C_1 and L, $(C \vee L)\sigma$ is ground, $(C \vee L)\sigma \prec_B \{\beta\}$, $C_0\sigma$ is false under Γ and $L\sigma$ is undefined in Γ

Decide $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL}} (\Gamma, L\sigma^{k+1}; N; U; \beta; k+1; \top)$

provided $L \in C$ for a $C \in (N \cup U)$, $L\sigma$ is a ground literal undefined in Γ and $L\sigma \prec_B \{\beta\}$

Conflict $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL}} (\Gamma; N; U; \beta; k; D \cdot \sigma)$
provided $D \in (N \cup U)$, $D\sigma$ false in Γ for a grounding substitution σ

As one can see, the propagated literal $L\sigma$ is ground. However the annotated clause is a closure $(C_0 \vee L)\delta \cdot \sigma$. Furthermore the conflict clause $D \cdot \sigma$ is also a closure. This allows to perform resolution inferences during conflict resolution on the non-ground clauses. Conflict resolution rules consist of Skip, Factorize, Resolve and Backtrack. Skip allows to skip literals on the trail that are no longer dependant on the conflict clause. Factorize allows to remove duplicate literals from the conflict clause. Resolve performs a resolution step on the conflict clause and the leading clause on the trail. And finally Backtrack allows a non-chronological backjump. The rules need to be adapted to non-ground clauses:

Skip $(\Gamma, L; N; U; \beta; k; D \cdot \sigma) \Rightarrow_{\text{SCL}} (\Gamma; N; U; \beta; k - i; D \cdot \sigma)$
provided $\mathsf{comp}(L)$ does not occur in $D\sigma$, if L is a decision literal then $i = 1$, otherwise $i = 0$.

Factorize $(\Gamma; N; U; \beta; k; (D \vee L \vee L') \cdot \sigma) \Rightarrow_{\text{SCL}} (\Gamma; N; U; \beta; k; (D \vee L)\eta \cdot \sigma)$
provided $L\sigma = L'\sigma$, $\eta = mgu(L, L')$

Resolve $(\Gamma, L\delta^{(C \vee L) \cdot \delta}; N; U; \beta; k; (D \vee L') \cdot \sigma) \Rightarrow_{\text{SCL}} (\Gamma, L\delta^{(C \vee L) \cdot \delta}; N; U; \beta; k; (D \vee C)\eta \cdot \sigma\delta)$
provided $L\delta = \mathsf{comp}(L'\sigma)$, $\eta = mgu(L, \mathsf{comp}(L'))$

Backtrack $(\Gamma, K, \Gamma', comp(L\sigma)^k; N; U; \beta; k; (D \vee L) \cdot \sigma) \Rightarrow_{\text{SCL}} (\Gamma_0; N; U \cup \{D \vee L\}; \beta; j; \top)$
provided $D\sigma$ is of level $i' < k$, and Γ_0, K is the minimal trail subsequence such that there is a grounding substitution τ with $(D \vee L)\tau$ is false in Γ_0, K but not in Γ_0, and Γ_0 is of level j

The authors show that their approach is sound and complete for full first-order logic. In SCL and in SCL(EQ) propagations need not to be exhaustively applied. Both approaches only learn non-redundant clauses, but for the first time conflicts resulting out of a decision have to be considered in SCL(EQ), due to the nature of the equality relation.

There have been suggested several approaches to lift the idea of an inference guiding model assumption from propositional to full first-order logic [FW19, BFT06, BP15, BFSS15]. However, they do not provide a native treatment of equality, e.g., via paramodulation or rewriting. Thus, in the following, we solely concentrate on approaches that are designed specifically for first-order logic with equality.

Baumgartner et al. [BT03] describe the Model Evolution calculus ($\mathcal{ME}$), which lifts the DPLL calculus [DLL62a, DP60] to first-order logic. An extension of $\mathcal{ME}$ to first-order logic with equality ($\mathcal{ME}_E$) was presented in [BT05], which was again revised and implemented in [BPT12]. Here I will concentrate on the latest description of $\mathcal{ME}_E$ as described in [BPT12]. Similar to my approach, a

candidate model is constructed until a clause instance contradicts this model or all instances are satisfied by the model. The candidate model results from the so-called *context*, which consists of a finite set of non-ground rewrite literals. Roughly speaking, a context literal specifies the truth value of all its ground instances unless a more specific literal specifies the complement. Initially the context consists of a pseudo literal $\neg v$, which represents all negative literals. Literals within a context may be universal or parametric, where universal literals guarantee all its ground instances to be true. The algorithm now operates on sequents $\Lambda \vdash \Phi$, where Λ is the context and Φ is the set of constrained clauses. Constrained clauses are of the form $C \cdot \Gamma$, where C is a clause and Γ is a constraint. Intuitively speaking, constraints can be seen as ground facts that held in the model when the clause was derived. Constrained clauses, where at least one constraint does no longer hold in the current model are not considered anymore. The Superposition and Equality Resolution rule are now adapted to constrained clauses:

Superposition $(\Lambda \cup \{l \to r\} \vdash \Phi \cup \{s[l']_p \# t \vee C \cdot \Gamma\}) \Rightarrow_{\text{SUP}} (\Lambda \cup \{l \to r\} \vdash \Phi \cup \{s[l']_p \# t \vee C \cdot \Gamma, (s[r]_p \# t \vee C \cdot \Gamma, l \to r)\sigma\})$
provided $\sigma = mgu(l, l'), r\sigma \not\succeq l\sigma, t\sigma \not\succeq s\sigma$, l' not a variable

Equality-Resolution $\Lambda \vdash \Phi \cup \{s \not\approx t \vee C \cdot \Gamma\} \Rightarrow_{\text{SUP}} \Lambda \vdash \Phi \cup \{s \not\approx t \vee C \cdot \Gamma, (C \cdot \Gamma)\sigma\}$
where σ is the mgu of s, t

Additionally, a negative resolution rule is needed to allow inferences with inequations within the context.

Neg-Res $\Lambda \cup \{\neg A\} \vdash \Phi \cup \{s \approx t \vee C \cdot \Gamma\} \Rightarrow_{\text{SUP}} \Lambda \cup \{\neg A\} \vdash \Phi \cup \{s \approx t \vee C \cdot \Gamma, (C \cdot \Gamma, s \not\to t)\sigma\}$
where $\neg A$ is $\neg v$ or a negative rewrite literal $l \not\to r$, $\sigma = mgu(A, s \approx t)$ and $t\sigma \not\succeq s\sigma$.

The main calculus now roughly consists of three rules: *Deduce*, *Split* and *Close*. *Deduce* generates new clauses based on the current model assumption using the aforementioned rules. For the Superposition and Equality Resolution rules the right premise is from Φ and the left premise from Λ. And for the Neg-Res rule the premise is from Φ. In case the empty clause is derived in this way, *Split* is applied. It basically repairs the trail by adding a literal from the constraint of the empty clause to the context on one branch and its negation on the other branch. If the negation of all literals in a constraint of an empty clause are contradictory with the current model assumption, the branch is closed with the *Close* rule. If all branches are closed, the clause set is unsatisfiable. Universal literals also allow optional rules such as *Assert*, *Compact* and more powerful redundancy criteria. *Assert* allows to add a literal to the context without branching. *Compact* allows to remove superfluous literals from the context. Powerful redundancy criteria allow to avoid redundant inferences, but also to simplify or delete clauses from the clause set. The authors show their approach to be sound and complete for full first-order logic.

Another approach by Baumgartner and Waldmann [BW09] combined the Superposition calculus with the Model Evolution calculus with equality. In this

calculus the atoms of the clauses are labeled as "split atoms" or "superposition atoms". The Superposition part of the calculus then generates a model for the Superposition atoms while the model evolution part generates a model for the split atoms. Conversely, this means that if all atoms are labeled as "split atom", the calculus behaves similar to the model evolution calculus. If all atoms are labeled as "superposition atom", it behaves like the Superposition calculus.

In [BFT06] Baumgartner et al. show how to integrate learning into the original Model Evolution calculus. It would be interesting to see this integration also in the $\mathcal{ME}_E$ calculus as this would be very close to my approach.

Baumgartner et al. [BFP07] present another approach to equality based on hyper tableaux [Bau98]. In general, a hyper tableaux works as follows: Assume clauses in the form $L_1, ..., L_n \Leftarrow K_1, ..., K_m$, where $L_1, ..., L_n$ are called the head literals and $K_1, ..., K_m$ are called the body literals, and an initial tableaux tree with only branch $\top$, which is open. Now assume that there is an open branch $\mathcal{B}$, a clause $L_1, ..., L_n \Leftarrow K_1, ..., K_m$, a set $K'_1, ..., K'_m \in \mathcal{B}$ and a substitution σ such that $K'_1 = K_1\sigma, ..., K'_m = K_m\sigma$. Then extend the branch with new nodes $L_1\sigma\phi, ..., L_n\sigma\phi$, where ϕ is a substitution which ensures that $vars(L_i) \cap vars(K_j) = \emptyset$ for all $i \neq j \leq n$. A branch is closed if it can be extended with a clause $\Leftarrow K_1, ..., K_m$. If all branches are closed, the clause set is unsatisfiable. For the handling of equality, the nodes in the hyper tableaux for equality do not consist of literals but clauses. The reason is that, in contrast to the standard hyper tableaux calculus, new clauses are generated using (unit) Superposition inferences which are dependant on the current context given by a branch. The algorithm allows three types of Superposition inferences, called *sup-left* and *unit-sup-right* and *ref*. *sup-left* allows a Superposition inference between a body literal and a positive unit clause. *unit-sup-right* allows a Superposition inference between two positive unit clauses. *ref* allows an equality resolution of a body literal.
The algorithm now consists of four derivation rules, which modifiy individual branches of the tableaux. Extension rules extend the branch in a given tableau: They consist of the *Split* and *Equality* rule. The *Split* rule branches out on an instance of a clause that contains only head literals. The *Equality* rule appends to the current branch the result of an application of one of the aforementioned Superposition rules with clauses of the current branch. Apart from these rules there exist two other rules, namely *Del* and *Simp*. *Del* deletes redundant clauses by replacing them by a trivial equation. *Simp* replaces a clause by another one that is smaller according to the reduction ordering that is used for the Superposition inferences and is defined in the same way as in the Superposition calculus. An E-hyper-tableau derivation on a set of clauses $C_1, ..., C_n$ is a possibly infinite sequence of tableaux $T_1, T_2, ...,$, where T_0 is the tableau that consists of a single branch of length n with clauses $C_1, ..., C_n$ and T_i is the result of applying a single derivation rule. A branch is closed iff it contains the empty clause. A tableau is closed iff all its branches are closed. The authors show that their approach is sound and complete.

Both the hyper tableaux calculus with equality and the model evolution calculus with equality allow only unit Superposition applications, while SCL(EQ) inferences are guided paramodulation inferences on clauses of arbitrary length. The model evolution calculus with equality was revised and implemented in 2011 [BPT12] and compares its performance with that of hyper tableaux. Model evolution performed significantly better, with more problems solved in all rele-

vant TPTP [Sut17] categories, than the implementation of the hyper tableaux calculus.

Plaisted et al. [PZ00] present the Ordered Semantic Hyper-Linking (OSHL) calculus. OSHL is an instantiation based approach that repeatedly chooses ground instances of a non-ground input clause set such that the current model does not satisfy the current ground clause set. A further step repairs the current model such that it satisfies the ground clause set again. The algorithm terminates if the set of ground clauses contains the empty clause. OSHL supports rewriting and narrowing, but only with unit clauses. In order to handle non-unit clauses it makes use of other mechanisms such as Brand's Transformation [BGV98].

Inst-Gen by Korovin [Kor13] handles equations only axiomatically. However, in [KS10] Korovin and Sticksel sketched the idea of *Inst-Gen-Eq* which is described in the following. The basic idea of Inst-Gen is to create propositional instances of first-order clauses by mapping all variable occurences to a constant $\bot$. Unsatisfiability of these ground instances implies that the original first-order clauses are also unsatisfiable. If the ground instances are satisifiable, more instances need to be added to the original clause set. The Inst-Gen rule creates new First-order clauses which are entailed by the original set. The basic Inst-Gen rule is as follows:

Inst-Gen $(N \cup \{L \vee C, L' \vee D\}) \Rightarrow_{\text{SUP}} (N \cup \{L \vee C, L' \vee D, (L \vee C)\sigma, (L' \vee D)\sigma\})$
where σ is a mgu of L and L' and σ maps at least one variable to a term.

The rule is somehow similar to the basic Resolution rule, but no actual resolution is performed. This leaves it up to the propositional SAT solver to decide how to proceed with these new clauses. The authors describe many refinements to this basic rule and other techniques such as redundancy elimination and literal selection. The idea of Inst-Gen-Eq is to extract useful instances from a unit Superposition proof of a contradiction of selected literals. The ground abstraction is then extended by the extracted clauses and an SMT solver then checks the satisfiability of the resulting set. Consider the following example set N as seen in [Sti11]:

$$C_1 := h(x) \approx x \vee x \not\approx a \tag{2.1}$$
$$C_2 := f(h(y)) \approx g(z) \tag{2.2}$$
$$C_3 := f(a) \not\approx g(u) \tag{2.3}$$

Now assume that the left literals are selected. Obviously, the ground instantiation with $\bot$ is not unsatisfiable. Thus new instances must be created. We need to create a unit Superposition refutation of the set of selected literals M:

$$\begin{array}{lll} M & \Rightarrow_{SUP} & M \cup \{f(x) \approx g(z)\} \\ & \Rightarrow_{SUP} & M \cup \{f(x) \approx g(z), g(z) \not\approx g(u)\} \\ & \Rightarrow_{SUP} & M \cup \{f(x) \approx g(z), g(z) \not\approx g(u), \bot\} \end{array}$$

Now for each selected literal we can extract the substitutions made for variables within these literals in the respective branch. Then we get new instances:

$$h(a) \approx a \vee a \not\approx a \tag{2.4}$$
$$f(h(a)) \approx g(z) \tag{2.5}$$
$$f(a) \not\approx g(z) \tag{2.6}$$

Since the last literal is only a variant of $f(a) \not\approx g(u)$ it can be ignored. The corresponding ground instances are now:

$$h(\bot) \approx \bot \vee \bot \not\approx a$$
$$f(h(\bot)) \approx g(\bot)$$
$$f(a) \not\approx g(\bot)$$
$$h(a) \approx a \vee a \not\approx a$$
$$f(h(a)) \approx g(\bot)$$

An underlying SMT solver can now show that this set is unsatisfiable. Similar to the approaches above, *Inst-Gen-Eq* only allows for unit Superposition inferences in contrast to SCL(EQ).

On ground equational clauses, the behavior of SCL(EQ) is similar to SMT (Satisfiability Modulo Theories) [NOT06]. SCL(EQ) rigorously searches for implied equalities and does not explicitely consider the propositional abstraction that drives SMT. Therefore, SCL(EQ) only learns non-redundant clauses that is not guaranteed by standard SMT reasoning. On the other hand the level of laziness in reasoning that is offered by SMT is currently not supported by SCL(EQ). On equational clauses with variables, SCL(EQ) learns only non-redundant clauses with variables whereas SMT solely operates on ground instances.

2.2 The Calculus

I start the introduction of the calculus [LW23] by defining the ingredients of an SCL(EQ) state. First, I extend reduction orderings on terms to literals and clauses. In the end, I want to ensure that both literals and clauses are comparable with terms. This is particularly important for the comparison with the maximum term β. This is achieved by encoding literals and clauses in multisets and taking the multiset extension of the term order $\prec_T$.

Definition 2.2.1 (Term Ordering)**.** Let $\prec_T$ be a reduction ordering on terms, which is total on ground terms and for all ground terms t there exist only finitely many ground terms $s \prec_T t$. I extend its application to literals $s \approx t$, $s \not\approx t$ by encoding the terms as multisets $\{s, t\}, \{s, s, t, t\}$ and using the multiset extension $\prec_{mul}$ of $\prec_T$. I further extend its application to clauses by comparing the multisets of literals using the multiset extension $(\prec_{mul})_{mul}$ of $\prec_{mul}$. I overload $\prec_T$ by using $\prec_T$ for literals and clauses if it is clear from the context.

For a (multi)set of terms $\{t_1, ..., t_n\}$ and a term t, I define $\{t_1, ..., t_n\} \prec_T t$ if $\{t_1, ..., t_n\} \prec_T \{t\}$. For a (multi)set of literals $\{L_1, ..., L_n\}$ and a term t, I define $\{L_1, ..., L_n\} \prec_T t$ if $\{L_1, ..., L_n\} \prec_T \{\{t\}\}$. Given a ground term β then $gnd_{\prec_T}\beta$ computes the set of all ground instances of a literal, clause, or clause set where the groundings are smaller than β according to the ordering $\prec_T$.

Now I define the trail, which is the current model assumption of the set of input clauses. Similar to SCL a trail consists of annotated ground literals. In contrast to SCL, however, each literal in Γ is annotated with its level and with a clause. We will see later why this is needed in SCL(EQ). Another distinctive feature of SCL(EQ) is that a ground literal on the trail has to be irreducible by the convergent rewrite system resulting from its preceding literals. This is necessary to learn only non-redundant clauses as we will see later on.

Definition 2.2.2 (Trail)**.** A *trail* $\Gamma := [L_1^{i_1:C_1\cdot\sigma_1}, ..., L_n^{i_n:C_n\cdot\sigma_n}]$ is a consistent sequence of ground equations and inequations where L_j is annotated by a level i_j with $i_{j-1} \leq i_j$, and a closure $C_j\cdot\sigma_j$. I omit the annotations if they are not needed in a certain context. A ground literal L is true in Γ if $\Gamma \models L$. A ground literal L is false in Γ if $\Gamma \models comp(L)$. A ground literal L is undefined in Γ if $\Gamma \not\models L$ and $\Gamma \not\models comp(L)$. Otherwise it is defined. For each literal L_j in Γ it holds that L_j is undefined in $[L_1, ..., L_{j-1}]$ and irreducible by $\text{conv}(\{L_1, ..., L_{j-1}\})$.

The above definition of truth and undefinedness is extended to clauses in the obvious way. The notions of true, false, undefined can be parameterized by a ground term β by saying that L is β-undefined in a trail Γ if $\beta \prec_T L$ or L is undefined. The notions of a β-true, β-false term are restrictions of the above notions to literals smaller β, respectively. All SCL(EQ) reasoning is layered with respect to a ground term β.

Given a set (sequence) of ground literals Γ let $conv(\Gamma)$ be a convergent rewrite system out of the positive equations in Γ using $\prec_T$.

Next I will define cores. A core is a minimal subsequence of a trail Γ that makes a literal L true or false. As a consequence, there exists no core for undefined literals. Since there can be several cores, I also give a description for defining cores and defining literals. A defining core is a core whose literal that is furthest to the right in the trail is furthest to the left compared to all other rightmost literals of the other cores. This literal is called a defining literal because without this literal L is undefined according to the preceding literals of the defining literal.

Definition 2.2.3. Let Γ be a trail and L a ground literal such that L is defined in Γ. By $\text{core}(\Gamma; L)$ I denote a minimal subsequence $\Gamma' \subseteq \Gamma$ such that L is defined in Γ'. By $\text{cores}(\Gamma; L)$ I denote the set of all cores.

As already mentioned, $core(\Gamma; L)$ is not necessarily unique. There can be multiple cores for a given trail Γ and ground literal L.

Definition 2.2.4 (Trail Ordering)**.** Let $\Gamma := [L_1, ..., L_n]$ be a trail. The (partial) trail ordering $\prec_\Gamma$ is the sequence ordering given by Γ, i.e., ($L_i \prec_\Gamma L_j$ if $i < j$) for all $1 \leq i, j \leq n$.

Definition 2.2.5 (Defining Core and Defining Literal)**.** For a trail Γ and a sequence of literals $\Delta \subseteq \Gamma$ I write $max_{\prec_\Gamma}(\Delta)$ for the largest literal in Δ according to the trail ordering $\prec_\Gamma$. Let Γ be a trail and L a ground literal such that L is defined in Γ. Let $\Delta \in cores(\Gamma; L)$ be a sequence of literals where $max_{\prec_\Gamma}(\Delta) \preceq_\Gamma max_{\prec_\Gamma}(\Lambda)$ for all $\Lambda \in cores(\Gamma; L)$, then $\max_\Gamma(L) := \max_{\prec_\Gamma}(\Delta)$ is called the *defining literal* and Δ is called a *defining core* for L in Γ. If $cores(\Gamma; L)$ contains only the empty core, then L has no *defining literal* and no *defining core*.

As an example, consider the trail $\Gamma := [L_1, L_2, L_3]$ and literal L, such that $[L_1, L_2] \models L$ and $[L_2, L_3] \models L$. Then both $[L_1, L_2]$ and $[L_2, L_3]$ are cores, but only $[L_1, L_2]$ is a defining core and only L_2 is the defining literal. Note that there can be multiple defining cores but only one defining literal for any defined literal L. For example, consider a trail $\Gamma := [f(a) \approx f(b)^{1:C_1\cdot\sigma_1}, a \approx b^{2:C_2\cdot\sigma_2}, b \approx c^{3:C_3\cdot\sigma_3}]$ with an ordering $\prec_T$ that orders the terms of the equations from left to right, and a literal $g(f(a)) \approx g(f(c))$. Then the defining cores are $\Delta_1 := [a \approx b, b \approx c]$ and $\Delta_2 := [f(a) \approx f(b), b \approx c]$. The defining literal, however, is in both cases $b \approx c$.

Defined literals that have no defining core and therefore no defining literal are literals that are trivially false or true. Consider, for example, $g(f(a)) \approx g(f(a))$. This literal is trivially true in Γ. Thus an empty subset of Γ is sufficient to show that $g(f(a)) \approx g(f(a))$ is defined in Γ.

With the help of defining literals, I can now determine the level of literals and clauses. This is important for finding conflicting clauses and learning from them, similar to SCL.

Definition 2.2.6 (Literal Level)**.** Let Γ be a trail. A ground literal $L \in \Gamma$ is of *level* i if L is annotated with i in Γ. A defined ground literal $L \notin \Gamma$ is of level i if the defining literal of L is of level i. If L has no defining literal, then L is of level 0. A ground clause D is of level i if i is the maximum level of a literal in D.

The restriction to minimal subsequences for the defining literal and definition of a level eventually guarantees that learned clauses are smaller in the trail ordering. This enables completeness in combination with learning non-redundant clauses as shown later.

Lemma 2.2.7. Let Γ_1 be a trail and K a defined literal that is of level i in Γ_1. Then K is of level i in a trail $\Gamma := \Gamma_1, \Gamma_2$.

Definition 2.2.8. Let Γ be a trail and $L \in \Gamma$ a literal. L is called a *decision literal* if $\Gamma = \Gamma_0, K^{i:C\cdot\tau}, L^{i+1:C'\cdot\tau'}, \Gamma_1$. Otherwise L is called a *propagated literal*.

In other words: L is a decision literal if the level of L is one greater than the level of the preceeding literal K. In the above example $g(f(a)) \approx g(f(c))$ is of level 3 since the defining literal $b \approx c$ is annotated with 3. $a \not\approx b$ on the other hand is of level 2.

I define a well-founded total strict ordering which is induced by the trail and with which non-redundancy is proven in Section 2.3. Unlike SCL [FW19,BFW21] I use this ordering for the inference rules as well. In previous SCL calculi, conflict resolution automatically chooses the greatest literal and resolves with this literal. In SCL(EQ) this is generalized. Coming back to the running example above, suppose we have a conflict clause $f(b) \not\approx f(c) \vee b \not\approx c$. The defining literal for both inequations is $b \approx c$. So we could do paramodulation inferences with both literals. The following ordering makes this non-deterministic choice deterministic.

Definition 2.2.9 (Trail Induced Ordering)**.** Let $\Gamma := [L_1^{i_1:C_1\cdot\sigma_1}, ..., L_n^{i_n:C_n\cdot\sigma_n}]$ be a trail, β a ground term such that $\{L_1, ..., L_n\} \prec_T \beta$ and $M_{i,j}$ all β-defined ground literals not contained in $\Gamma \cup comp(\Gamma)$: for a defining literal $max_\Gamma(M_{i,j}) = L_i$ and for two literals $M_{i,j}$, $M_{i,k}$ I have $j < k$ if $M_{i,j} \prec_T M_{i,k}$. The trail induces a total well-founded strict order $\prec_{\Gamma^*}$ on β-*defined* ground literals $M_{k,l}, M_{m,n}$, L_i, L_j of level greater than zero, where

1. $M_{i,j} \prec_{\Gamma^*} M_{k,l}$ if $i < k$ or ($i = k$ and $j < l$)
2. $L_i \prec_{\Gamma^*} L_j$ if $L_i \prec_\Gamma L_j$
3. $comp(L_i) \prec_{\Gamma^*} L_j$ if $L_i \prec_\Gamma L_j$
4. $L_i \prec_{\Gamma^*} comp(L_j)$ if $L_i \prec_\Gamma L_j$ or $i = j$
5. $comp(L_i) \prec_{\Gamma^*} comp(L_j)$ if $L_i \prec_\Gamma L_j$

6. $L_i \prec_{\Gamma^*} M_{k,l}$, $comp(L_i) \prec_{\Gamma^*} M_{k,l}$ if $i \leq k$

7. $M_{k,l} \prec_{\Gamma^*} L_i$, $M_{k,l} \prec_{\Gamma^*} comp(L_i)$ if $k < i$

and for all β-*defined* literals L of level zero:

8. $\prec_{\Gamma^*} := \prec_T$

9. $L \prec_{\Gamma^*} K$ if K is of level greater than zero and K is β-*defined*

and can eventually be extended to β-*undefined* ground literals K, H by

10. $K \prec_{\Gamma^*} H$ if $K \prec_T H$

11. $L \prec_{\Gamma^*} H$ if L is β-*defined*

The literal ordering $\prec_{\Gamma^*}$ is extended to ground clauses by multiset extension and identified with $\prec_{\Gamma^*}$ as well.

Note, that in the above definition, for a given i, I can always put the $M_{i,j}$ in an order corresponding to the integers. This is due to the fact that the total number of ground literals is countable and since there are only finitely many β-defined literals.

Lemma 2.2.10 (Properties of $\prec_{\Gamma^*}$)**.** 1. $\prec_{\Gamma^*}$ is well-defined.

2. $\prec_{\Gamma^*}$ is a total strict order, i.e. $\prec_{\Gamma^*}$ is irreflexive, transitive and total.

3. $\prec_{\Gamma^*}$ is a well-founded ordering.

Example 2.2.11. Assume a trail $\Gamma := [a \approx b^{1:C_0 \cdot \sigma_0}, c \approx d^{1:C_1 \cdot \sigma_1}, f(a') \not\approx f(b')^{1:C_2 \cdot \sigma_2}]$, select KBO as the term ordering $\prec_T$ where all symbols have weight one and $a \prec a' \prec b \prec b' \prec c \prec d \prec f$ and a ground term $\beta := f(f(a))$. According to the trail induced ordering we have that $a \approx b \prec_{\Gamma^*} c \approx d \prec_{\Gamma^*} f(a') \not\approx f(b')$ by 2.2.9.2. Furthermore we have that

$$a \approx b \prec_{\Gamma^*} a \not\approx b \prec_{\Gamma^*} c \approx d \prec_{\Gamma^*} c \not\approx d \prec_{\Gamma^*} f(a') \not\approx f(b') \prec_{\Gamma^*} f(a') \approx f(b')$$

by 2.2.9.3 and 2.2.9.4. Now for any literal L that is β-*defined* in Γ and the defining literal is $a \approx b$ it holds that $a \not\approx b \prec_{\Gamma^*} L \prec_{\Gamma^*} c \approx d$ by 2.2.9.6 and 2.2.9.7. This holds analogously for all literals that are β-*defined* in Γ and the defining literal is $c \approx d$ or $f(a') \not\approx f(b')$. Thus we get:

$$\begin{gathered} L_1 \prec_{\Gamma^*} \ldots \prec_{\Gamma^*} a \approx b \prec_{\Gamma^*} a \not\approx b \prec_{\Gamma^*} f(a) \approx f(b) \prec_{\Gamma^*} f(a) \not\approx f(b) \prec_{\Gamma^*} \\ c \approx d \prec_{\Gamma^*} c \not\approx d \prec_{\Gamma^*} f(c) \approx f(d) \prec_{\Gamma^*} f(c) \not\approx f(d) \prec_{\Gamma^*} \\ f(a') \not\approx f(b') \prec_{\Gamma^*} f(a') \approx f(b') \prec_{\Gamma^*} a' \approx b' \prec_{\Gamma^*} a' \not\approx b' \prec_{\Gamma^*} K_1 \prec_{\Gamma^*} \ldots \end{gathered}$$

where K_i are the β-*undefined* literals and L_j are the trivially defined literals.

Table 2.1 summarizes the various orders presented so far.

In CDCL or SCL new clauses are learned by applying resolution to the conflicting clause and the annotated clauses in the trail. In SCL(EQ) this is generalized again. Here I need paramodulation inferences to learn new clauses.

Definition 2.2.12 (Rewrite Step)**.** A *rewrite step* is a five-tuple $(s\#t \cdot \sigma, s\#t \vee C \cdot \sigma, R, S, p)$ and inductively defined as follows. The tuple $(s\#t \cdot \sigma, s\#t \vee C \cdot \sigma, \epsilon, \epsilon, \epsilon)$ is a rewrite step. Given rewrite steps R, S and a position p then $(s\#t \cdot \sigma, s\#t \vee C \cdot \sigma, R, S, p)$ is a *rewrite step*. The literal $s\#t$ is called the *rewrite literal*. In case R, S are not ϵ, the rewrite literal of R is an equation.

Table 2.1: Summary of the orderings presented so far.

Order		Description
Term order	$\prec_T$	• well-founded rewrite ordering on terms • total on ground terms • for all ground terms t there exist only finitely many ground terms $s \prec_T t$
Trail order	$\prec_\Gamma$	• sequence ordering given by the trail Γ
Trail induced order	$\prec_{\Gamma^*}$	• extends the sequence ordering to all literals implicitly defined due to literals of level >0 • uses $\prec_T$ for literals of level 0 and undef. literals • literals of level 0 are smaller and undef. literals are greater than literals of level >0

Note that R and S in the above definition describe the "history" of a rewrite step, i.e., they contain all preceeding rewrite steps.

We assume two rewrite steps $I_1 := (L_1 \cdot \sigma_1, (L_1 \vee C_1) \cdot \sigma_1, R_1, L_1, p_1)$ and $I_2 := (L_2 \cdot \sigma_2, (L_2 \vee C_2) \cdot \sigma_2, R_2, L_2, p_2)$ to be variable disjoint iff $L_1 \vee C_1$ and $L_2 \vee C_2$ are variable disjoint.

Rewriting is one of the core features of my calculus. The following definition describes a rewrite inference between two clauses. Note that unlike the Superposition calculus I allow rewriting below variable level.

Definition 2.2.13 (Rewrite Inference)**.** Let $I_1 := (l_1 \approx r_1 \cdot \sigma_1, l_1 \approx r_1 \vee C_1 \cdot \sigma_1, R_1, L_1, p_1)$ and $I_2 := (l_2 \# r_2 \cdot \sigma_2, l_2 \# r_2 \vee C_2 \cdot \sigma_2, R_2, L_2, p_2)$ be two variable disjoint rewrite steps where $r_1\sigma_1 \prec_T l_1\sigma_1$, $(l_2 \# r_2)\sigma_2|_p = l_1\sigma_1$ for some position p. I distinguish two cases:

1. if $p \in \text{pos}(l_2 \# r_2)$ and $\mu := \mathsf{mgu}((l_2 \# r_2)|_p, l_1)$ then $(((l_2 \# r_2)[r_1]_p)\mu \cdot \sigma_1\sigma_2, ((l_2 \# r_2)[r_1]_p)\mu \vee C_1\mu \vee C_2\mu \cdot \sigma_1\sigma_2, I_1, I_2, p)$ is the result of a rewrite inference.

2. if $p \notin \text{pos}(l_2 \# r_2)$ then let $(l_2 \# r_2)\delta$ be the most general instance of $l_2 \# r_2$ such that $p \in \text{pos}((l_2 \# r_2)\delta)$, δ introduces only fresh variables and $(l_2 \# r_2)\delta\sigma_2\rho = (l_2 \# r_2)\sigma_2$ for some minimal ρ. Let $\mu := \mathsf{mgu}((l_2 \# r_2)\delta|_p, l_1)$. Then $((l_2 \# r_2)\delta[r_1]_p\mu \cdot \sigma_1\sigma_2\rho, (l_2 \# r_2)\delta[r_1]_p\mu \vee C_1\mu \vee C_2\delta\mu \cdot \sigma_1\sigma_2\rho, I_1, I_2, p)$ is the result of a rewrite inference.

Note that case 1 describes rewriting above or at a variable and case 2 describes rewriting inside a variable.

Example 2.2.14. Consider two rewrite steps

$$I_1 := (x \approx b \cdot \sigma_1, x \approx b \vee g(x) \approx c \cdot \sigma_1, R_1, L_1, p_1)$$

and

$$I_2 := (f(y) \not\approx f(f(b)) \cdot \sigma_2, f(y) \not\approx f(f(b)) \cdot \sigma_2, R_2, L_2, p_2)$$

where $\sigma_1 = \{x \to a\}$ and $\sigma_2 = \{y \to f(a)\}$. This is an example where we have to rewrite below variable level. Since the position $p = 111$ does not exist

yet in $f(y) \approx f(f(b))$ we first have to create a most general instance with $\delta = \{y \to f(z)\}$. A minimal ρ would now be $\rho = \{z \to a\}$. A most general unifier could be $\mu = \{x \to z\}$. Now we can create our new rewrite step:

$$I_3 := (f(f(b)) \not\approx f(f(b)) \cdot \sigma_1\sigma_2\rho, f(f(b)) \not\approx f(f(b)) \vee g(z) \approx c \cdot \sigma_1\sigma_2\rho, I_1, I_2, p)$$

Lemma 2.2.15. Let $I_1 := (l_1 \approx r_1 \cdot \sigma_1, l_1 \approx r_1 \vee C_1 \cdot \sigma_1, R_1, L_1, p_1)$ and $I_2 := (l_2 \# r_2 \cdot \sigma_2, l_2 \# r_2 \vee C_2 \cdot \sigma_2, R_2, L_2, p_2)$ be two variable disjoint rewrite steps where $r_1\sigma_1 \prec_T l_1\sigma_1, (l_2 \# r_2)\sigma_2|_p = l_1\sigma_1$ for some position p. Let $I_3 := (l_3 \# r_3 \cdot \sigma_3, l_3 \# r_3 \vee C_3 \cdot \sigma_3, I_1, I_2, p)$ be the result of a rewrite inference. Then:

1. $C_3\sigma_3 = (C_1 \vee C_2)\sigma_1\sigma_2$ and $l_3 \# r_3\sigma_3 = (l_2 \# r_2)\sigma_2[r_1\sigma_1]_p$.
2. $(l_3 \# r_3)\sigma_3 \prec_T (l_2 \# r_2)\sigma_2$
3. If $N \models (l_1 \approx r_1 \vee C_1) \wedge (l_2 \# r_2 \vee C_2)$ for some set of clauses N, then $N \models l_3 \# r_3 \vee C_3$

Now that I have defined rewrite inferences I can use them to define a *reduction chain application* and a *refutation*, which are sequences of rewrite steps. Intuitively speaking, a *reduction chain application* reduces a literal in a clause with terms in $conv(\Gamma)$ until it is irreducible. A *refutation* for a literal L that is β-*false* in Γ for a given β, is a sequence of rewrite steps with literals in Γ, L such that $\bot$ is inferred. Refutations for the literals of the conflict clause will be examined during conflict resolution by the rule *Explore-Refutation*.

Definition 2.2.16 (Reduction Chain)**.** Let Γ be a trail. A *reduction chain* $\mathcal{P}$ from Γ is a sequence of rewrite steps $[I_1, ..., I_m]$ such that for each $I_i = (s_i \# t_i \cdot \sigma_i, s_i \# t_i \vee C_i \cdot \sigma_i, I_j, I_k, p_i)$ either

1. $s_i \# t_i^{n_i : s_i \# t_i \vee C_i \cdot \sigma}$ is contained in Γ and $I_j = I_k = p_i = \epsilon$ or
2. I_i is the result of a rewriting inference from rewrite steps I_j, I_k out of $[I_1, ..., I_m]$ where $j, k < i$.

Let $(l \# r)\delta^{o : l \# r \vee C \cdot \delta}$ be an annotated ground literal. A *reduction chain application* from Γ to $l \# r$ is a reduction chain $[I_1, ..., I_m]$ from $\Gamma, (l \# r)\delta^{o : l \# r \vee C \cdot \delta}$ such that $l\delta\downarrow_{\text{conv}(\Gamma)} = s_m\sigma_m$ and $r\delta\downarrow_{\text{conv}(\Gamma)} = t_m\sigma_m$. I assume reduction chain applications to be minimal, i.e., if any rewrite step is removed from the sequence it is no longer a reduction chain application.

Definition 2.2.17 (Refutation)**.** Let Γ be a trail and $(l \# r)\delta^{o : l \# r \vee C \cdot \delta}$ an annotated ground literal that is β-*false* in Γ for a given β. A *refutation* $\mathcal{P}$ from Γ and $l \# r$ is a reduction chain $[I_1, ..., I_m]$ from $\Gamma, (l \# r)\delta^{o : l \# r \vee C \cdot \delta}$ such that $(s_m \# t_m)\sigma_m = s \not\approx s$ for some s. I assume refutations to be minimal, i.e., if any rewrite step I_k, $k < m$ is removed from the refutation, it is no longer a refutation.

2.2.1 The SCL(EQ) Inference Rules

I can now define the rules of my calculus based on the previous definitions. A *state* is a six-tuple $(\Gamma; N; U; \beta; k; D)$ similar to the SCL calculus, where Γ a sequence of annotated ground literals, N and U the sets of initial and learned

clauses, β is a ground term such that for all $L \in \Gamma$ it holds that $L \prec_T \beta$, k is the decision level, and D a status that is $\top$, $\bot$ or a closure $C \cdot \sigma$. Before I propagate or decide any literal, I make sure that it is irreducible in the current trail. Together with the design of $\prec_{\Gamma^*}$ this eventually enables rewriting as a simplification rule.

Propagate
$(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL(EQ)}} (\Gamma, s_m \# t_m \sigma_m^{k:(s_m \# t_m \vee C_m)\cdot\sigma_m}; N; U; \beta; k; \top)$
provided there is a $C \in (N \cup U)$, σ grounding for C, $C = C_0 \vee C_1 \vee L$, $\Gamma \models \neg C_0\sigma$, $C_1\sigma = L\sigma \vee ... \vee L\sigma$, $C_1 = L_1 \vee ... \vee L_n$, $\mu = mgu(L_1, ..., L_n, L)$ $L\sigma$ is *β-undefined* in Γ, $(C_0 \vee L)\mu\sigma \prec_T \beta$, σ is irreducible by $conv(\Gamma)$, $[I_1, \ldots, I_m]$ is a reduction chain application from Γ to $L\sigma^{k:(L \vee C_0)\mu\cdot\sigma}$ where $I_m = (s_m \# t_m \cdot \sigma_m, s_m \# t_m \vee C_m \cdot \sigma_m, I_j, I_k, p_m)$.

The rule *Propagate* finds a ground instance of a clause which can be propagated, i.e. it contains (possibly multiple occurences of) a literal that is *undefined* and all other literals are *false* in the trail. The multiple occurences of the undefined literal are factored. Then it adds the normal form of this literal to the trail. The propagating clause is reduced by the corresponding paramodulation steps. Note that the definition of *Propagate* also includes the case where $L\sigma$ is irreducible by Γ. In this case $L = s_m \# t_m$ and $m = 1$. The rule *Decide* below, is similar to *Propagate*, except for the subclause C_0 which must be *β-undefined* or *β-true* in Γ, i.e., *Propagate* cannot be applied and the decision literal is annotated by a tautology.

Decide
$(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL(EQ)}} (\Gamma, s_m \# t_m \sigma_m^{k+1:(s_m \# t_m \vee comp(s_m \# t_m))\cdot\sigma_m}; N; U; \beta; k+1; \top)$
provided there is a $C \in (N \cup U)$, σ grounding for C, $C = C_0 \vee L$, $C_0\sigma$ is *β-undefined* or *β-true* in Γ, $L\sigma$ is *β-undefined* in Γ, $(C_0 \vee L)\sigma \prec_T \beta$, σ is irreducible by $conv(\Gamma)$, $[I_1, \ldots, I_m]$ is a reduction chain application from Γ to $L\sigma^{k+1:L \vee C_0\cdot\sigma}$ where $I_m = (s_m \# t_m \cdot \sigma_m, s_m \# t_m \vee C_m \cdot \sigma_m, I_j, I_k, p_m)$.

Conflict $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL(EQ)}} (\Gamma; N; U; \beta; k; D)$
provided there is a $D' \in (N \cup U)$, σ grounding for D', $D'\sigma$ is *β-false* in Γ, σ is irreducible by $conv(\Gamma)$, $D = \bot$ if $D'\sigma$ is of level 0 and $D = D' \cdot \sigma$ otherwise.

For the non-equational case, when a conflict clause is found by an SCL calculus [FW19, BFW21], the complements of its first-order ground literals are contained in the trail. For equational literals this is not the case, in general. The proof showing D to be *β-false* with respect to Γ is a rewrite proof with respect to conv(Γ). This proof needs to be analyzed to eventually perform paramodulation steps on D or to replace D by a $\prec_{\Gamma^*}$ smaller *β-false* clause showing up in the proof.

Skip $(\Gamma, K^{l:C\cdot\tau}, L^{k:C'\cdot\tau'}; N; U; \beta; k; D \cdot \sigma) \Rightarrow_{\text{SCL(EQ)}} (\Gamma, K^{l:C\cdot\tau}; N; U; \beta; l; D \cdot \sigma)$
if $D\sigma$ is *β-false* in $\Gamma, K^{l:C\cdot\tau}$.

The *Explore-Refutation* rule is the FOL with Equality counterpart to the resolve rule in CDCL or SCL. While in CDCL or SCL complementary literals of

the conflict clause are present on the trail and can directly be used for resolution steps, this needs a generalization for FOL with Equality. Here, in general, I need to look at (rewriting) refutations of the conflict clause and pick an appropriate clause from the refutation as the next conflict clause.

Explore-Refutation
$(\Gamma, L; N; U; \beta; k; (D \vee s \# t) \cdot \sigma)) \Rightarrow_{\text{SCL(EQ)}} (\Gamma, L; N; U; \beta; k; (s_j \# t_j \vee C_j) \cdot \sigma_j)$
if $(s \# t)\sigma$ is strictly $\prec_{\Gamma^*}$ maximal in $(D \vee s \# t)\sigma$, L is the defining literal of $(s \# t)\sigma$, $[I_1, ..., I_m]$ is a refutation from Γ and $(s \# t)\sigma$, $I_j = (s_j \# t_j \cdot \sigma_j, (s_j \# t_j \vee C_j) \cdot \sigma_j, I_l, I_k, p_j)$, $1 \leq j \leq m$, $(s_j \# t_j \vee C_j)\sigma_j \prec_{\Gamma^*} (D \vee s \# t)\sigma$, $(s_j \# t_j \vee C_j)\sigma_j$ is β-*false* in Γ.

Factorize
$(\Gamma; N; U; \beta; k; (D \vee L \vee L') \cdot \sigma) \Rightarrow_{\text{SCL(EQ)}} (\Gamma; N; U; \beta; k; (D \vee L)\mu \cdot \sigma)$
provided $L\sigma = L'\sigma$, and $\mu = \mathsf{mgu}(L, L')$.

Equality-Resolution
$(\Gamma; N; U; \beta; k; (D \vee s \not\approx s') \cdot \sigma) \Rightarrow_{\text{SCL(EQ)}} (\Gamma; N; U; \beta; k; D\mu \cdot \sigma)$
provided $s\sigma = s'\sigma$, $\mu = mgu(s, s')$.

For backtracking I have to make sure, that the learned clause is not *false* in the resulting trail. It is not sufficient to backtrack to the point where the clause with the current substitution is no longer false, but where it is no longer false with all possible substitutions.

Backtrack $(\Gamma, K, \Gamma'; N; U; \beta; k; (D \vee L) \cdot \sigma) \Rightarrow_{\text{SCL(EQ)}} (\Gamma; N; U \cup \{D \vee L\}; \beta; j - i; \top)$
provided $D\sigma$ is of level i' where $i' < k$, K is of level j and Γ, K the minimal trail subsequence such that there is a grounding substitution τ with $(D \vee L)\tau$ β-false in Γ, K but not in Γ; $i = 1$ if K is a decision literal and $i = 0$ otherwise.

Grow $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\text{SCL(EQ)}} (\epsilon; N; U; \beta'; 0; \top)$
provided $\beta \prec_T \beta'$.

In contrast to Superposition I do not need an equality factoring rule as the following example illustrates.

Example 2.2.18 (Equality Factoring)**.** Assume clauses:

$$\begin{aligned} C_1 &:= b \approx c \vee c \approx d \\ C_2 &:= a_1 \approx b \vee a_1 \approx c \\ &\quad ... \\ C_{n+1} &:= a_n \approx b \vee a_n \approx c \end{aligned}$$

The completeness proof of Superposition requires that adding a new literal to an interpretation does not make any smaller literal true. In this example, however, after adding $b \approx c$ to the interpretation, we cannot add any further literal, since it breaks this invariant. So in Superposition we would have to add the following clauses with the help of the Equality Factoring rule:

$$\begin{aligned} C_{n+2} &:= b \not\approx c \vee a_1 \approx c \\ &\ldots \\ C_{2n+1} &:= b \not\approx c \vee a_n \approx c \end{aligned}$$

In SCL(EQ) on the other hand we can just decide a literal in each clause to get a model for this clause set. As we support undefined literals we do not have to bother with this problem at all. For example if we add $b \approx c$ to our model, both literals $a_1 \approx b$ and $a_1 \approx c$ are undefined in our model. Thus we need to decide one of these literals to add it to our model.

In addition to soundness and completeness of the SCL(EQ) rules their tractability in practice is an important property for a successful implementation. In particular, finding propagating literals or detecting a false clause under some grounding. It turns out that these operations are NP-complete, similar to first-order subsumption which has been shown to be tractable in practice.

Lemma 2.2.19. Assume that all ground terms t with $t \prec_T \beta$ for any β are polynomial in the size of β. Then testing *Propagate* (*Conflict*) is NP-Complete, i.e., the problem of checking for a given clause C whether there exists a grounding substitution σ such that $C\sigma$ propagates (is false) is NP-Complete.

In the rest of this Section I provide some examples that compare SCL(EQ) to Superposition.

Example 2.2.20 (SCL(EQ) vs. Superposition: Saturation)**.** Consider the following clauses:

$$N := \{C_1 := c \approx d \vee D, C_2 := a \approx b \vee c \not\approx d, C_3 := f(a) \not\approx f(b) \vee g(c) \not\approx g(d)\}$$

where again we assume a KBO with all symbols having weight one, precedence $d \prec c \prec b \prec a \prec g \prec f$ and $\beta := f(f(g(a)))$. Suppose that we first decide $c \approx d$ and then propagate $a \approx b$: $\Gamma = [c \approx d^{1:c\approx d \vee c \not\approx d}, a \approx b^{1:C_2}]$. Now we have a conflict with C_3. *Explore-Refutation* applied to the conflict clause C_3 results in a paramodulation inference between C_3 and C_2. Another application of *Equality-Resolution* gives us the new conflict clause $C_4 := c \not\approx d \vee g(c) \not\approx g(d)$. Now we can *Skip* the last literal on the trail, which gives us $\Gamma = [c \approx d^{1:c\approx d \vee c \not\approx d}]$. Another application of the *Explore-Refutation* rule to C_4 using the decision justification clause followed by *Equality-Resolution* and *Factorize* gives us $C_5 := c \not\approx d$. Thus with SCL(EQ) the following clauses remain:

$$\begin{aligned} C_1' = D \qquad C_5 = c \not\approx d \\ C_3 = f(a) \not\approx f(b) \vee g(c) \not\approx g(d) \end{aligned}$$

where we derived C_1' out of C_1 by subsumption resolution [Wei01] using C_5. Actually, subsumption resolution is compatible with the general redundancy notion of SCL(EQ), see Lemma 2.3.7. Now we consider the same example with Superposition and the very same ordering (N_i is the clause set of the previous step and N_0 the initial clause set N).

$$\begin{aligned} N_0 &\Rightarrow_{Sup(C_2,C_3)} N_1 \cup \{C_4 := c \not\approx d \vee g(c) \not\approx g(d)\} \\ &\Rightarrow_{Sup(C_1,C_4)} N_2 \cup \{C_5 := c \not\approx d \vee D\} \Rightarrow_{Sup(C_1,C_5)} N_3 \cup \{C_6 := D\} \end{aligned}$$

Thus Superposition ends up with the following clauses:

$$C_2 = a \approx b \vee c \not\approx d \qquad C_3 = f(a) \not\approx f(b) \vee g(c) \not\approx g(d)$$
$$C_4 = c \not\approx d \vee g(c) \not\approx g(d) \qquad C_6 = D$$

The Superposition calculus generates more and larger clauses.

Example 2.2.21 (SCL(EQ) vs. Superposition: Refutation)**.** Suppose the following set of clauses: $N := \{C_1 := f(x) \not\approx a \vee f(x) \approx b, C_2 := f(f(y)) \approx y, C_3 := a \not\approx b\}$ where again we assume a KBO with all symbols having weight one, precedence $b \prec a \prec f$ and $\beta := f(f(f(a)))$. A long refutation by the Superposition calculus results in the following (N_i is the clause set of the previous step and N_0 the initial clause set N):

$$\begin{aligned}
N_0 \quad &\Rightarrow_{Sup(C_1,C_2)} \quad N_1 \cup \{C_4 := y \not\approx a \vee f(f(y)) \approx b\} \\
&\Rightarrow_{Sup(C_1,C_4)} \quad N_2 \cup \{C_5 := a \not\approx b \vee f(f(y)) \approx b \vee y \not\approx a\} \\
&\Rightarrow_{Sup(C_2,C_5)} \quad N_3 \cup \{C_6 := a \not\approx b \vee b \approx y \vee y \not\approx a\} \\
&\Rightarrow_{Sup(C_2,C_4)} \quad N_4 \cup \{C_7 := y \approx b \vee y \not\approx a\} \\
&\Rightarrow_{EqRes(C_7)} \quad N_5 \cup \{C_8 := a \approx b\} \Rightarrow_{Sup(C_3,C_8)} N_6 \cup \{\bot\}
\end{aligned}$$

The shortest refutation by the Superposition calculus is as follows:

$$\begin{aligned}
N_0 \quad &\Rightarrow_{Sup(C_1,C_2)} \quad N_1 \cup \{C_4 := y \not\approx a \vee f(f(y)) \approx b\} \\
&\Rightarrow_{Sup(C_2,C_4)} \quad N_2 \cup \{C_5 := y \approx b \vee y \not\approx a\} \\
&\Rightarrow_{EqRes(C_5)} \quad N_3 \cup \{C_6 := a \approx b\} \Rightarrow_{Sup(C_3,C_6)} N_4 \cup \{\bot\}
\end{aligned}$$

In SCL(EQ) on the other hand we would always first propagate $a \not\approx b, f(f(a)) \approx a$ and $f(f(b)) \approx b$. As soon as $a \not\approx b$ and $f(f(a)) \approx a$ are propagated we have a conflict with $C_1\{x \to f(a)\}$. So suppose in the worst case we propagate:

$$\Gamma := [a \not\approx b^{0:a\not\approx b}, f(f(b)) \approx b^{0:(f(f(y))\approx y)\{y\to b\}}, f(f(a)) \approx a^{0:(f(f(y))\approx y)\{y\to a\}}]$$

Now we have a conflict with $C_1\{x \to f(a)\}$. Since there is no decision literal on the trail, the *Conflict* rule immediately returns $\bot$ and the algorithm terminates.

Example 2.2.22 (Intermediate Results in Refutation)**.** Consider the following ground clause set N:

$$C_1 := f(a,a) \not\approx f(b,b) \vee c \approx d \qquad C_2 := a \approx b \vee f(a,a) \approx f(b,b)$$

Suppose that we decide $f(a,a) \not\approx f(b,b)$. Then C_2 is false in Γ. The conflict state is as follows: $([f(a,a) \not\approx f(b,b)^{1:f(a,a)\not\approx f(b,b)\vee f(a,a)\approx f(b,b)}]; N; \{\}; 2; C_2)$. *Explore-Refutation* creates the following ground refutation for $a \approx b$, since it is greatest literal in the conflict clause:

$$\begin{aligned}
I_1 := \quad & (f(a,a) \not\approx f(b,b), f(a,a) \not\approx f(b,b) \vee f(a,a) \approx f(b,b), \epsilon, \epsilon, \epsilon) \\
I_2 := \quad & (a \approx b, C_2, \epsilon, \epsilon, \epsilon) \\
I_3 := \quad & (f(b,a) \not\approx f(b,b), f(b,a) \not\approx f(b,b) \vee f(a,a) \approx f(b,b) \vee f(a,a) \approx f(b,b), \\
& I_2, I_1, 11) \\
I_4 := \quad & (f(b,b) \not\approx f(b,b), f(b,b) \not\approx f(b,b) \vee f(a,a) \approx f(b,b) \vee f(a,a) \approx f(b,b) \\
& \vee f(a,a) \approx f(b,b), I_3, I_1, 12)
\end{aligned}$$

As one can see, the intermediate result $f(b,a) \not\approx f(b,b)$ is not false in Γ. Thus it is no candidate for the new conflict clause. We have to choose I_4. The new state is thus:

$$([f(a,a) \not\approx f(b,b)^{1:f(a,a)\not\approx f(b,b)\vee f(a,a)\approx f(b,b)}]; N; \{\}; 2;$$

$$f(b,b) \not\approx f(b,b) \vee f(a,a) \approx f(b,b) \vee f(a,a) \approx f(b,b) \vee f(a,a) \approx f(b,b))$$

Now we can apply *Equality-Resolution* and two times *Factorize* to get the final clause $f(a,a) \approx f(b,b)$ with which we can backtrack.

2.3 Correctness

In this Section, I show soundness and refutational completeness of SCL(EQ) under the assumption of a regular run. I provide the definition of a regular run and show that for a regular run all learned clauses are non-redundant according to the trail induced ordering. I start with the definition of a sound state.

Definition 2.3.1. A state $(\Gamma; N; U; \beta; k; D)$ is sound if the following conditions hold:

1. Γ is a consistent sequence of annotated literals,
2. for each decomposition $\Gamma = \Gamma_1, L\sigma^{i:(C \vee L)\cdot\sigma}, \Gamma_2$ where $L\sigma$ is a propagated literal, we have that $C\sigma$ is β-*false* in Γ_1, $L\sigma$ is β-*undefined* in Γ_1 and irreducible by $conv(\Gamma_1)$, $N \cup U \models (C \vee L)$ and $(C \vee L)\sigma \prec_T \beta$,
3. for each decomposition $\Gamma = \Gamma_1, L\sigma^{i:(L \vee comp(L))\cdot\sigma}, \Gamma_2$ where $L\sigma$ is a decision literal, we have that $L\sigma$ is β-*undefined* in Γ_1 and irreducible by $conv(\Gamma_1)$, $N \cup U \models (L \vee comp(L))$ and $(L \vee comp(L))\sigma \prec_T \beta$,
4. $N \models U$,
5. if $D = C \cdot \sigma$, then $C\sigma$ is β-*false* in Γ, $N \cup U \models C$,

Lemma 2.3.2. The initial state $(\epsilon; N; \emptyset; \beta; 0; \top)$ is sound.

Definition 2.3.3. A run is a sequence of applications of SCL(EQ) rules starting from the initial state.

Theorem 2.3.4. Assume a state $(\Gamma; N; U; \beta; k; D)$ resulting from a run. Then $(\Gamma; N; U; \beta; k; D)$ is sound.

Next, I give the definition of a regular run. Intuitively speaking, in a regular run I am always allowed to do decisions except if

1. a literal can be propagated before the first decision and
2. the negation of a literal can be propagated.

To ensure non-redundant learning I enforce at least one application of *Skip* during conflict resolution except for the special case of a conflict after a decision.

Definition 2.3.5 (Regular Run)**.** A run is called *regular* if

1. the rules *Conflict* and *Factorize* have precedence over all other rules,
2. If $k = 0$ in a state $(\Gamma; N; U; \beta; k; D)$, then *Propagate* has precedence over *Decide*,
3. If an annotated literal $L^{k:C\cdot\sigma}$ could be added by an application of *Propagate* on Γ in a state $(\Gamma; N; U; \beta; k; D)$ and $C \in N \cup U$, then the annotated literal $comp(L)^{k+1:C'\cdot\sigma'}$ is not added by *Decide* on Γ,

4. during conflict resolution *Skip* is applied at least once, except if *Conflict* is applied immediately after an application of *Decide*.

5. if *Conflict* is applied immediately after an application of *Decide*, then *Backtrack* is only applied in a state $(\Gamma, L'; N; U; \beta; k; D\cdot\sigma)$ if $L\sigma = comp(L')$ for some $L \in D$.

The following example shows an immediate conflict after a decision.

Example 2.3.6 (Implicit Conflict after Decision)**.** Consider the following clause set N

$$C_1 := h(x) \approx g(x) \vee c \approx d \qquad C_2 := f(x) \approx g(x) \vee a \approx b$$
$$C_3 := f(x) \not\approx h(x) \vee f(x) \not\approx g(x)$$

Suppose we apply the rule *Decide* first to C_1 and then to C_2 with substitution $\sigma = \{x \to a\}$. Then we yield a conflict with $C_3\sigma$, resulting in the following state:

$$([h(a) \approx g(a)^{1:(h(x)\approx g(x)\vee h(x)\not\approx g(x))\cdot\sigma}, f(a) \approx g(a)^{2:(f(x)\approx g(x)\vee f(x)\not\approx g(x))\cdot\sigma}];$$

$$N; \{\}; 2; C_3\cdot\sigma)$$

According to $\prec_{\Gamma^*}$, $f(a) \not\approx h(a)$ is the greatest literal in $C_3\sigma$. Since $f(a) \approx g(a)$ is the defining literal of $f(a) \not\approx h(a)$ we can not apply *Skip*. *Factorize* is also not applicable, since $f(a) \not\approx h(a)$ and $f(a) \not\approx g(a)$ are not equal. Thus we must apply *Explore-Refutation* to the greatest literal $f(a) \not\approx h(a)$. The rule first creates a refutation $[I_1, ..., I_5]$, where:

$$\begin{array}{ll}
I_1 := & ((f(x) \not\approx h(x))\cdot\sigma, C_3\cdot\sigma, \epsilon, \epsilon, \epsilon) \\
I_2 := & ((f(x) \approx g(x))\cdot\sigma, (f(x) \approx g(x) \vee f(x) \not\approx g(x))\cdot\sigma, \epsilon, \epsilon, \epsilon) \\
I_3 := & ((h(x) \approx g(x))\cdot\sigma, (h(x) \approx g(x) \vee h(x) \not\approx g(x))\cdot\sigma, \epsilon, \epsilon, \epsilon) \\
I_4 := & ((h(x) \not\approx g(x))\cdot\sigma, (h(x) \not\approx g(x) \vee f(x) \not\approx g(x) \vee f(x) \not\approx g(x))\cdot\sigma, I_2, I_1, 1) \\
I_5 := & ((g(x) \not\approx g(x))\cdot\sigma, (g(x) \not\approx g(x) \vee f(x) \not\approx g(x) \vee f(x) \not\approx g(x) \\
& \vee h(x) \not\approx g(x))\cdot\sigma, I_4, I_3, 1)
\end{array}$$

Explore-Refutation can now choose either I_4 or I_5. Both, $(h(x) \not\approx g(x))\sigma$ and $(g(x) \not\approx g(x))\sigma$ are smaller than $(f(x) \not\approx h(x))\sigma$ according to $\prec_{\Gamma^*}$ and false in Γ. Suppose we choose I_5. Now our new conflict state is:

$$([h(a) \approx g(a)^{1:(h(x)\approx g(x)\vee h(x)\not\approx g(x))\cdot\sigma}, f(a) \approx g(a)^{2:(f(x)\approx g(x)\vee f(x)\not\approx g(x))\cdot\sigma}];$$

$$N; \{\}; 2; (g(x) \not\approx g(x) \vee f(x) \not\approx g(x) \vee f(x) \not\approx g(x) \vee h(x) \not\approx g(x))\cdot\sigma)$$

Now we apply *Equality-Resolution* and *Factorize* to get the new state

$$([g(a) \approx h(a)^{1:(g(x)\approx h(x)\vee g(x)\not\approx h(x))\cdot\sigma}, f(a) \approx g(a)^{2:(f(x)\approx g(x)\vee f(x)\not\approx g(x))\cdot\sigma}];$$

$$N; \{\}; 2; (f(x) \not\approx g(x) \vee h(x) \not\approx g(x))\cdot\sigma)$$

Now we can backtrack. Note, that this clause is non-redundant according to our ordering, although conflict was applied immediately after decision.

Now I show that any learned clause in a regular run is non-redundant according to the trail induced ordering.

Lemma 2.3.7 (Non-Redundant Clause Learning)**.** Let N be a clause set. The clauses learned during a regular run in SCL(EQ) are not redundant with respect to $\prec_{\Gamma^*}$ and $N \cup U$. For the trail only non-redundant clauses need to be considered.

The proof of Lemma 2.3.7 is based on the fact that conflict resolution eventually produces a clause smaller then the original conflict clause with respect to $\prec_{\Gamma^*}$. All simplifications, e.g., contextual rewriting, as defined in [BG94, Wei01, WW08, WW10, Wis12, GKR20], are therefore compatible with Lemma 2.3.7 and may be applied to the newly learned clause as long as they respect the induced trail ordering. In detail, let Γ be the trail before the application of rule *Backtrack*. The newly learned clause can be simplified according to the induced trail ordering $\prec_{\Gamma^*}$ as long as the simplified clause is smaller with respect to $\prec_{\Gamma^*}$.

Another important consequence of Lemma 2.3.7 is that newly learned clauses need not to be considered for redundancy. Furthermore, the SCL(EQ) calculus always terminates, Lemma 2.3.15, because there only finitely many non-redundant clauses with respect to a fixed β.

For dynamic redundancy, I have to consider the fact that the induced trail ordering changes. At this level, only redundancy criteria and simplifications that are compatible with *all* induced trail orderings may be applied. Due to the construction of the induced trail ordering, it is compatible with $\prec_T$ for unit clauses.

Lemma 2.3.8 (Unit Rewriting)**.** Assume a state $(\Gamma; N; U; \beta; k; D)$ resulting from a regular run where the current level $k > 0$ and a unit clause $l \approx r \in N$. Now assume a clause $C \vee L[l']_p \in N$ such that $l' = l\mu$ for some matcher μ. Now assume some arbitrary grounding substitutions σ' for $C \vee L[l']_p$, σ for $l \approx r$ such that $l\sigma = l'\sigma'$ and $r\sigma \prec_T l\sigma$. Then $(C \vee L[r\mu\sigma\sigma']_p)\sigma' \prec_{\Gamma^*} (C \vee L[l']_p)\sigma'$.

In addition, any notion that is based on a literal subset relationship is also compatible with ordering changes. The standard example is subsumption.

Lemma 2.3.9. Let C, D be two clauses. If there exists a substitution σ such that $C\sigma \subset D$, then D is redundant with respect to C and any $\prec_{\Gamma^*}$.

The notion of redundancy, Definition 1.1.35, only supports a strict subset relation for Lemma 2.3.9, similar to the Superposition calculus. However, the newly generated clauses of SCL(EQ) are the result of paramodulation inferences [RW69]. In a recent contribution to dynamic, abstract redundancy [WTRB20] it is shown that also the non-strict subset relation in Lemma 2.3.9, i.e., $C\sigma \subseteq D$, preserves completeness.

If all stuck states, see below Definition 2.3.10, with respect to a fixed β are visited before increasing β then this provides a simple dynamic fairness strategy.

When unit reduction or any other form of supported rewriting is applied to clauses smaller than the current β, it can be applied independently from the current trail. If, however, unit reduction is applied to clauses larger than the current β then the calculus must do a restart to its initial state, in particular the trail must be emptied, as for otherwise rewriting may result generating a conflict that did not exist with respect to the current trail before the rewriting. This is analogous to a restart in CDCL once a propositional unit clause is derived and used for simplification. More formally, I add the following new Restart rule to the calculus to reset the trail to its initial state after a unit reduction.

Restart $(\Gamma; N; U; \beta; k; \top) \Rightarrow_{\mathrm{SCL(EQ)}} (\epsilon; N; U; \beta; 0; \top)$

Next I show refutation completeness of SCL(EQ). To achieve this I first give a definition of a stuck state. Then I show that stuck states only occur if all ground literals $L \prec_T \beta$ are β-*defined* in Γ and not during conflict resolution. Finally I show that conflict resolution will always result in an application of *Backtrack*. This allows us to show termination (without application of Grow) and refutational completeness.

Definition 2.3.10 (Stuck State)**.** A state $(\Gamma; N; U; \beta; k; D)$ is called *stuck* if $D \neq \bot$ and none of the rules of the calculus, except for Grow, is applicable.

Lemma 2.3.11 (Form of Stuck States)**.** If a regular run (without rule Grow) ends in a stuck state $(\Gamma; N; U; \beta; k; D)$, then $D = \top$ and all ground literals $L\sigma \prec_T \beta$, where $L \vee C \in (N \cup U)$ are β-*defined* in Γ.

Lemma 2.3.12. Suppose a sound state $(\Gamma; N; U; \beta; k; D)$ resulting from a regular run where $D \notin \{\top, \bot\}$. If *Backtrack* is not applicable then any set of applications of *Explore-Refutation*, *Skip*, *Factorize*, *Equality-Resolution* will finally result in a sound state $(\Gamma'; N; U; \beta; k; D')$, where $D' \prec_{\Gamma^*} D$. Then *Backtrack* will be finally applicable.

Corollary 2.3.13 (Satisfiable Clause Sets)**.** Let N be a satisfiable clause set. Then any regular run without rule Grow will end in a stuck state, for any β.

Thus a stuck state can be seen as an indication for a satisfiable clause set. Of course, it remains to be investigated whether the clause set is actually satisfiable. Superposition is one of the strongest approaches to detect satisfiability and constitutes a decision procedure for many decidable first-order fragments [BGW93, GdN99]. Now given a stuck state and some specific ordering such as KBO, LPO [KB70, KAM80], or some polynomial ordering [DP01], it is decidable whether the ordering can be instantiated from a stuck state such that Γ coincides with the Superposition model operator on the ground terms smaller than β. In this case it can be effectively checked whether the clauses derived so far are actually saturated by the Superposition calculus with respect to this specific ordering. In this sense, SCL(EQ) has the same power to decide satisfiability of first-order clause sets than Superposition.

Definition 2.3.14. A regular run terminates in a state $(\Gamma; N; U; \beta; k; D)$ if $D = \top$ and no rule is applicable, or $D = \bot$.

Lemma 2.3.15. Let N be a set of clauses and β be a ground term. Then any regular run that never uses Grow terminates.

Lemma 2.3.16. If a regular run reaches the state $(\Gamma; N; U; \beta; k; \bot)$ then N is unsatisfiable.

Theorem 2.3.17 (Refutational Completeness)**.** Let N be an unsatisfiable clause set, and $\prec_T$ a desired term ordering. For any ground term β where $gnd_{\prec_T \beta}(N)$ is unsatisfiable, any regular SCL(EQ) run without rule Grow will terminate by deriving $\bot$.

2.4 Proofs

In this Section, I provide all the proofs of the Lemmas shown in the previous Sections.

Lemma 2.2.7 Let Γ_1 be a trail and K a defined literal that is of level i in Γ_1. Then K is of level i in a trail $\Gamma := \Gamma_1, \Gamma_2$.

Proof. Assume a trail Γ_1 and a literal K that is of level i in Γ_1. Let $\Gamma := \Gamma_1, \Gamma_2$ be a trail. Then we have two cases:

1. K has no defining literal in Γ_1. Then $cores(\Gamma_1; K) = \{[]\}$ contains only the empty core and K is of level 0 in Γ_1. Then $cores(\Gamma; K) = \{[]\}$ as well and thus K is of level 0 in Γ.

2. K has a defining literal $L := max_{\Gamma_1}(K)$ and L is of level i. Then there exists a core $\Delta \in cores(\Gamma_1; K)$ such that L is the maximum literal in Δ according to $\prec_\Gamma$ and for all $\Lambda \in cores(\Gamma_1; K)$ it holds $max_{\prec_\Gamma}(\Delta) \preceq_\Gamma max_{\prec_\Gamma}(\Lambda)$. Thus Δ is a defining core. Now any $\Lambda \in (cores(\Gamma; K) \setminus cores(\Gamma_1; K))$ has a higher maximum literal according to $\prec_\Gamma$. Thus Δ is also a defining core in Γ and L is the defining literal of K in Γ and thus K is of level i in Γ.

□

Auxiliary Lemmas for the Proofs of Lemma 2.2.10

Lemma 2.4.1. Let Γ be a trail. Then any literal in Γ occurs exactly once.

Proof. Let $\Gamma := [L_1^{i_1:C_1 \cdot \sigma_1}, ..., L_n^{i_n:C_n \cdot \sigma_n}]$. Now suppose there exist L_i, L_j with $i < j$ and $1 \leq i, j \leq n$ such that $L_i = L_j$. By definition of Γ, L_j is undefined in $[L_1, ..., L_i, ..., L_{j-1}]$. But obviously L_j is defined in Γ. Contradiction. □

Lemma 2.4.2. Let Γ be a trail. If a literal L is of level i, then it is not of level $j \neq i$.

Proof. Let Γ be a trail. By Lemma 2.4.1 any literal in Γ is unique. Suppose there exists a literal L such that L is of level i and of level j. If the core is empty for L then L is of level 0 by definition. Otherwise there must exist cores $\Delta, \Lambda \in cores(\Gamma; L)$ such that $max_{\prec_\Gamma}(\Delta) \preceq_\Gamma max_{\prec_\Gamma}(\Lambda')$ and $max_{\prec_\Gamma}(\Lambda) \preceq_\Gamma max_{\prec_\Gamma}(\Lambda')$ for all $\Lambda' \in cores(\Gamma; L)$. But then $max_{\prec_\Gamma}(\Lambda) = max_{\prec_\Gamma}(\Delta)$. Contradiction. □

Lemma 2.4.3. Let L be a ground literal and Γ a trail. If L is defined in Γ then L has a level.

Proof. Let Γ be a trail. Suppose that L is defined in Γ. Then it either has a defining literal or it has no defining literal. If it has a defining literal K, then the level of K is the level of L. Since $K \in \Gamma$ it is annotated by a level. Thus L has a level. If L does not have a defining literal, then L is of level 0 by definition of a literal level. □

Lemma 2.2.10-1 $\prec_{\Gamma^*}$ is well-defined.

Proof. Suppose a trail $\Gamma := [L_1^{i_1:C_1\cdot\sigma_1}, ..., L_n^{i_n:C_n\cdot\sigma_n}]$ and a term β such that $\{L_1, ..., L_n\} \prec_T \beta$. We have to show that the rules 2.2.9.1-2.2.9.11 are pairwise disjunct. Consider the rules 2.2.9.1-2.2.9.7. These rules are pairwise disjunct, if the sets $\{L_1, ..., L_n\}$, $\{comp(L_1), ..., comp(L_n)\}$ and $\{M_{i,j} \mid i \leq n\}$ are pairwise disjunct. Obviously, $\{L_1, ..., L_n\} \cap \{comp(L_1), ..., comp(L_n)\} = \emptyset$. Furthermore $(\{L_1, ..., L_n\} \cup$
$\{comp(L_1), ..., comp(L_n)\}) \cap \{M_{i,j} \mid i \leq n\} = \emptyset$ follows directly from the definition of a trail induced ordering. 2.2.9.8 and 2.2.9.9 are disjunct since $\{L \mid L \; is \; of \; level \; 0\}$ and $\{L \mid L \; is \; of \; level \; greater \; 0\}$ are disjunct by Lemma 2.4.2. It follows that 2.2.9.1-2.2.9.9 are pairwise disjunct, since all relations in 2.2.9.1-2.2.9.7 contain only *β-defined* literals of level 1 or higher and all relations in 2.2.9.8, 2.2.9.9 contain at least one *β-defined* literal of level 0. 2.2.9.10 and 2.2.9.11 are disjunct since a literal cannot be both *β-defined* and *β-undefined*. It follows that 2.2.9.1-2.2.9.11 are pairwise disjunct, since all relations in 2.2.9.1-2.2.9.9 contain only *β-defined* literals and all relations in 2.2.9.10, 2.2.9.11 contain at least one *β-undefined* literal. □

Lemma 2.2.10-2 $\prec_{\Gamma^*}$ is a total strict order, i.e. $\prec_{\Gamma^*}$ is irreflexive, transitive and total.

Proof. Suppose a trail $\Gamma := [L_1^{i_1:C_1\cdot\sigma_1}, ..., L_n^{i_n:C_n\cdot\sigma_n}]$ and a term β such that $\{L_1, ..., L_n\} \prec_T \beta$.
Irreflexivity. We have to show that there is no ground literal L such that $L \prec_{\Gamma^*} L$. Suppose two literals L and K such that $L \prec_{\Gamma^*} K$ and $L = K$. Now we have several cases:

1. Suppose that L, K are *β-defined* and of level 1 or higher. Then we have several cases:
   (a) $L = M_{i,j}$ and $K = M_{k,l}$. Then by 2.2.9.1 $M_{i,j} \prec_{\Gamma^*} M_{k,l}$ if $i < k$ or $(i = k$ and $j < l)$. Thus $i \neq k$ or $j \neq l$. We show that for $M_{i,j}$, $M_{k,l}$ with $i \neq k$ or $j \neq l$ it holds $M_{i,j} \neq M_{k,l}$. Assume that $M_{i,j} = M_{k,l}$ and $k \neq i$ or $j \neq l$. Assume that $k = i$. Then, by Definition 2.2.9 $M_{i,j} \prec_T M_{k,l}$ or $M_{k,l} \prec_T M_{i,j}$. Thus $M_{i,j} \neq M_{k,l}$ since $\prec_T$ is a rewrite ordering. Now assume that $k \neq i$. Since $M_{i,j} = M_{k,l}$ it holds $max_\Gamma(M_{i,j}) = max_\Gamma(M_{k,l})$, since both have the same level by Lemma 2.4.2. But then $k = i$. Thus $M_{i,j} \neq M_{k,l}$ for $k \neq i$ or $j \neq l$. Thus if by 2.2.9.1 $M_{i,j} \prec_{\Gamma^*} M_{k,l}$ if $i < k$ or $(i = k$ and $j < l)$, then $M_{i,j} \neq M_{k,l}$.
   (b) $L = L_i$ and $K = L_j$. Then by 2.2.9.2 $L_i \prec_{\Gamma^*} L_j$ if $L_i \prec_\Gamma L_j$. Then by Lemma 2.4.1 $L_i \neq L_j$.
   (c) $L = comp(L_i)$ and $K = L_j$. Then by 2.2.9.3 $comp(L_i) \prec_{\Gamma^*} L_j$ if $L_i \prec_\Gamma L_j$. $L_i \neq L_j$ by Lemma 2.4.1. $L \neq K$ has to hold since Γ is consistent.
   (d) $L = L_i$ and $K = comp(L_j)$. Then by 2.2.9.4 $L_i \prec_{\Gamma^*} comp(L_j)$ if $L_i \prec_\Gamma L_j$ or $i = j$. If $i \neq j$ then we can proceed analogous to the previous step. If $i = j$ then obviously $L_i \neq comp(L_i)$.
   (e) $L = comp(L_i)$ and $K = comp(L_j)$. Then by 2.2.9.5 $comp(L_i) \prec_{\Gamma^*} comp(L_j)$ if $L_i \prec_\Gamma L_j$. By Lemma 2.4.1 $L_i \neq L_j$. Thus $comp(L_i) \neq comp(L_j)$.

(f) $L = L_i$ and $K = M_{k,l}$. Then by 2.2.9.6 $L_i \prec_{\Gamma^*} M_{k,l}$, $\mathsf{comp}(L_i) \prec_{\Gamma^*} M_{k,l}$ if $i \leq k$. $M_{k,l} \neq L_i$ and $M_{k,l} \neq comp(L_i)$ follows directly from the Definition 2.2.9. Thus if $L_i \prec_{\Gamma^*} M_{k,l}$ or $\mathsf{comp}(L_i) \prec_{\Gamma^*} M_{k,l}$ if $i \leq k$ by 2.2.9.6, then $L_i \neq M_{k,l}$ and $\mathsf{comp}(L_i) \neq M_{k,l}$.

(g) $L = M_{k,l}$ and $K = L_i$. Then we can proceed analogous to the previous step for 2.2.9.7.

2. Suppose that L and K are *β-defined* and of level zero. Since $\prec_T$ is irreflexive, $L \not\prec_T K$ has to hold. Since $\prec_{\Gamma^*} = \prec_T$ for literals of level zero $L \not\prec_{\Gamma^*} K$ has to hold too.

3. Suppose that L, K are *β-defined* and L is of level zero and K is of level greater than zero. But then $L \neq K$ has to hold by Lemma 2.4.2. Thus $L \not\prec_{\Gamma^*} K$ for 2.2.9.9.

4. Suppose that L and K are *β-undefined*. Then by 2.2.9.10 $K \prec_{\Gamma^*} H$ if $K \prec_T H$. Since $\prec_T$ is a rewrite ordering $K \prec_T H$ iff $K \neq H$.

5. Suppose that L is *β-defined* and K is *β-undefined*. Then by 2.2.9.11 $L \prec_{\Gamma^*} K$. Then $L \neq K$ has to hold since otherwise L, K would be both *β-defined* and *β-undefined*, contradicting consistency of Γ.

Transitivity. Suppose there exist literals L, K, H such that $H \prec_{\Gamma^*} K$ and $K \prec_{\Gamma^*} L$ but not $H \prec_{\Gamma^*} L$. We have several cases:

1. Suppose all literals are *β-undefined*. Then $K \prec_T L$ and $H \prec_T K$. Otherwise $K \prec_{\Gamma^*} L$ and $H \prec_{\Gamma^*} K$ would not hold. Thus also $H \prec_T L$ by transitivity of $\prec_T$. Thus $H \prec_{\Gamma^*} L$ by 2.2.9.10.

2. Suppose two literals are *β-undefined*. If K would be *β-defined*, then $K \prec_{\Gamma^*} H$ by 2.2.9.11 contradicting assumption. If L would be *β-defined*, then $L \prec_{\Gamma^*} K$ by 2.2.9.11 again contradicting assumption. Thus H has to be *β-defined*. Then $H \prec_{\Gamma^*} L$ by Definition 2.2.9.11.

3. Suppose one literal is *β-undefined*. If K would be *β-undefined*, then $L \prec_{\Gamma^*} K$ by Definition 2.2.9.11 contradicting assumption. If H would be *β-undefined*, then $K \prec_{\Gamma^*} H$ by Definition 2.2.9.11 again contradicting assumption. Thus L has to be *β-undefined*. Then $H \prec_{\Gamma^*} L$ by Definition 2.2.9.11.

4. Suppose all literals are *β-defined*. Then we have multiple subcases:

   (a) Suppose all literals have the same defining literal L_i and L_i is of level 1 or higher. By 2.2.9.6 $L_i \prec_{\Gamma^*} M_{i,j}$ and $comp(L_i) \prec_{\Gamma^*} M_{i,j}$ for all j. By 2.2.9.4 $L_i \prec_{\Gamma^*} comp(L_i)$. Thus $L_i \prec_{\Gamma^*} comp(L_i) \prec_{\Gamma^*} M_{i,j}$ for all j. Since $K \prec_{\Gamma^*} L$ either $K = L_i$ and $L \neq L_i$ or $L = M_{i,j}$ and $K = comp(L_i)$ or $L = M_{i,j}$ and $K = M_{i,k}$ with $k < j$.

      i. Assume $K = L_i$ and $L \neq L_i$. Since K is the smallest literal with defining literal L_i, $K = H$ has to hold. But then $K \prec_{\Gamma^*} K$ contradicting irreflexivity.

      ii. Assume $L = M_{i,j}$ and $K = comp(L_i)$. Since $H \prec_{\Gamma^*} K$ and all literals have the same defining literal, $H = L_i$ has to hold by 2.2.9.6 and 2.2.9.4. Then, again by 2.2.9.6, $H \prec_{\Gamma^*} L$.

iii. $L = M_{i,j}$ and $K = M_{i,k}$ with $k < j$. Since $H \prec_{\Gamma^*} K$ and all have the same defining literal either $H = M_{i,l}$ with $l < k$ by 2.2.9.1, or $H = L_i$ or $H = comp(L_i)$ by 2.2.9.6. In both cases $H \prec_{\Gamma^*} L$ holds by 2.2.9.1 and 2.2.9.6.

(b) Suppose H, K, L have at least one different defining literal and $max_\Gamma(L) = L_i$ with L_i of level 1 or higher. First, we have to show that if $L_j = max_\Gamma(K') \prec_\Gamma max_\Gamma(L') = L_i$ and L_i is of level 1 or higher, then $K' \prec_{\Gamma^*} L'$. Suppose that L_j is of level 0. Then $K' \prec_{\Gamma^*} L'$ by 2.2.9.9. Suppose that $L' = M_{i,k}$ and $K' = M_{j,l}$. Then $K' \prec_{\Gamma^*} L'$ by 2.2.9.1. Suppose that $L' = M_{i,k}$ and $K' = L_j$ or $K' = comp(L_j)$. Then $K' \prec_{\Gamma^*} L'$ by 2.2.9.6. Suppose that $L' = L_i$ or $L' = comp(L_i)$ and $K' = L_j$ or $K' = comp(L_j)$. Then $K' \prec_{\Gamma^*} L'$ by 2.2.9.2-2.2.9.5. Suppose that $L' = L_i$ or $L' = comp(L_i)$ and $K' = M_{j,l}$. Then $K' \prec_{\Gamma^*} L'$ by 2.2.9.7.
Now by assumption $H \prec_{\Gamma^*} K$ and $K \prec_{\Gamma^*} L$. If $max_\Gamma(K) \prec_\Gamma max_\Gamma(H)$ then $K \prec_{\Gamma^*} H$ contradicting assumption. The same holds for L and K. Thus either $max_\Gamma(H) \prec_\Gamma max_\Gamma(L)$ or $max_\Gamma(K) \prec_\Gamma max_\Gamma(L)$. In the first case $H \prec_{\Gamma^*} L$ follows from above. In the second case $max_\Gamma(H) \preceq_\Gamma max_\Gamma(K)$ has to hold. Thus $H \prec_{\Gamma^*} L$ follows again.

(c) Suppose that $max_\Gamma(L) = L_i$ where L_i is of level 0. Since $K \prec_{\Gamma^*} L$, $max_\Gamma(K) = L_j$ with L_j of level 0 has to hold by 2.2.9.9 and 2.2.9.11. Now assume that $L \prec_T K$. Then $L \prec_{\Gamma^*} K$ by 2.2.9.8 contradicting assumption. Thus $K \prec_T L$ has to hold since $K \neq L$. Since $H \prec_{\Gamma^*} K$, $max_\Gamma(H) = L_k$ with L_k of level 0 has to hold by 2.2.9.9 and 2.2.9.11. Now assume that $K \prec_T H$. Then $K \prec_{\Gamma^*} H$ by 2.2.9.8 contradicting assumption. Thus $H \prec_T K$ has to hold since $H \neq K$. By transitivity of $\prec_T$, $H \prec_T L$ and thus $H \prec_{\Gamma^*} L$ has to hold.

Totality. First we show that any ground literal is either *β-defined* and has a level or *β-undefined*. Since Γ is consistent, a literal is either *β-defined* or *β-undefined*. We just need to show that if a literal is *β-defined*, it has a level. By Lemma 2.4.3 all defined literals have a level. *β-definedness* implies definedness. Thus all *β-defined* literals have a level. Now assume some arbitrary ground literals $L \neq K$. We have several cases:

1. L, K are *β-undefined*. Since $L \neq K$ we have $L \prec_T K$ or $K \prec_T L$ by totality of $\prec_T$ on ground literals. Thus by 2.2.9.10 $L \prec_{\Gamma^*} K$ or $K \prec_{\Gamma^*} L$.
2. One is *β-defined*. Then either $L \prec_{\Gamma^*} K$ or $K \prec_{\Gamma^*} L$ by 2.2.9.11.
3. Both are *β-defined*. Then we have several subcases:
   (a) L is of level zero and K is of level greater than zero or vice versa. Then by 2.2.9.9 $L \prec_{\Gamma^*} K$ or $K \prec_{\Gamma^*} L$ has to hold.
   (b) $max_\Gamma(L) = L_i$ and $max_\Gamma(K) = L_j$ and both are of level 1 or higher.
      i. $L = M_{i,k}$ and $K = M_{j,l}$. Then either $M_{i,k} \prec_{\Gamma^*} M_{j,l}$ or $M_{j,l} \prec_{\Gamma^*} M_{i,k}$ by 2.2.9.1.
      ii. $L = M_{i,k}$ and $K = L_j$ or $K = comp(L_j)$. If $i \geq j$ then by 2.2.9.6 $L_j \prec_{\Gamma^*} M_{i,k}$ or $comp(L_j) \prec_{\Gamma^*} M_{i,k}$. If $i < j$ then by 2.2.9.7 $M_{i,k} \prec_{\Gamma^*} L_j$ or $M_{i,k} \prec_{\Gamma^*} comp(L_j)$.

iii. $K = M_{i,k}$ and $L = L_j$ or $L = comp(L_j)$. Analogous to previous step.

iv. $L = L_i$ and $K = L_j$. Then if $i < j$ by 2.2.9.2 $L_i \prec_{\Gamma^*} L_j$ and $L_j \prec_{\Gamma^*} L_i$ otherwise.

v. $L = comp(L_i)$ and $K = comp(L_j)$ analogous to previous step for 2.2.9.5.

vi. $L = L_i$ and $K = comp(L_j)$. Then if $i \leq j$ by 2.2.9.4 $L_i \prec_{\Gamma^*} comp(L_j)$. If $j < i$ by 2.2.9.3 $comp(L_j) \prec_{\Gamma^*} L_i$.

vii. $L = comp(L_i)$ and $K = L_j$. Then if $i < j$ by 2.2.9.3 $comp(L_i) \prec_{\Gamma^*} L_j$. If $j \leq i$ by 2.2.9.4 $L_j \prec_{\Gamma^*} comp(L_i)$.

(c) $max_\Gamma(L) = L_i$ and $max_\Gamma(K) = L_j$ and both are of level 0. Now either $L \prec_T K$ or $K \prec_T L$. Thus by 2.2.9.8 $L \prec_{\Gamma^*} K$ or $K \prec_{\Gamma^*} L$.

□

Lemma 2.2.10-3 $\prec_{\Gamma^*}$ is a well-founded ordering.

Proof. Suppose some arbitrary subset M of all ground literals, a trail $\Gamma := [L_1^{i_1:C_1 \cdot \sigma_1}, ..., L_n^{i_n:C_n \cdot \sigma_n}]$ and a term β such that $\{L_1, ..., L_n\} \prec_T \beta$. We have to show that M has a minimal element. We have several cases:

1. L is β-*undefined* in Γ for all literals $L \in M$. Then $\prec_{\Gamma^*} = \prec_T$. Since $\prec_T$ is well-founded there exists a minimal element in M. Thus there exists a minimal element in M with regard to $\prec_{\Gamma^*}$.

2. there exists at least one literal in M that is β-*defined*. Then we have two cases:

   (a) there exists a literal in M that is of level zero. Then let $L \in M$ be the literal of level zero, where $L \prec_T K$ for all $K \in M$ with K of level zero. We show that L is the minimal element. Suppose there exists a literal $L' \in M$ that is smaller than L. Since L is of level zero, $L \prec_{\Gamma^*} K$ for all literals K of level greater than zero by 2.2.9.9 and $L \prec_{\Gamma^*} H$ for all β-*undefined* literals H by 2.2.9.11. Thus L' must be of level zero. But then $L' \prec_T L$ has to hold, contradicting assumption.

   (b) There exists no literal in M that is of level zero. Let $L \in M$ be the literal where $max_\Gamma(L) \preceq_\Gamma max_\Gamma(K)$ for all $K \in M$ and

      i. $L = max_\Gamma(L)$ or

      ii. $L = comp(max_\Gamma(L))$ and $max_\Gamma(L) \notin M$ or

      iii. $L \prec_T H$ for all $H \in M$ such that $max_\Gamma(L) = max_\Gamma(H)$ and $max_\Gamma(L) \notin M$ and $comp(max_\Gamma(L)) \notin M$.

      We show that L is the minimal element. Suppose there exists a literal $L' \in M$ that is smaller than L. We have three cases:

      i. $max_\Gamma(L) = L = L_i$. Since $L_i \prec_T \beta$ we have either $L' = L_j$ with $j < i$ by 2.2.9.2 or $L' = comp(L_j)$ with $j < i$ by 2.2.9.3 or $L' = M_{k,l}$ with $k < i$ by 2.2.9.7. In all three cases we have $max_\Gamma(L') \prec_\Gamma max_\Gamma(L)$ contradicting assumption that the defining literal of L is minimal in M.

ii. $L = comp(L_i) = comp(max_\Gamma(L))$ and $max_\Gamma(L) \notin M$. Since $L_i \prec_T \beta$ either $L' = L_j$ with $j < i$ by 2.2.9.4 or $L' = comp(L_j)$ with $j < i$ by 2.2.9.5 or $L' = M_{k,l}$ with $k < i$ by 2.2.9.7. In all three cases we have $max_\Gamma(L') \prec_\Gamma max_\Gamma(L)$ contradicting assumption that the defining literal of L is minimal in M.

iii. $L = M_{k,l}$ and $max_\Gamma(L) \notin M$ and $comp(max_\Gamma(L)) \notin M$. Then either $L' = M_{i,j}$ with $i < k$ or ($i = k$ and $j < l$) by 2.2.9.1 or $L' = L_i$ or $L' = comp(L_i)$ with $i \leq k$. Suppose that $L' = M_{i,j}$ and $i < k$. Then $max_\Gamma(L') \prec_\Gamma max_\Gamma(L)$ contradicting assumption. Suppose that $L' = M_{i,j}$ and $i = k$ and $j < l$. Then $L' \prec_T L$ and $max_\Gamma(L) = max_\Gamma(L')$. For L it holds $L \prec_T H$ for all $H \in M$ such that $max_\Gamma(L) = max_\Gamma(H)$. Contradiction. Suppose that $L' = L_i$ or $L' = comp(L_i)$ with $i = k$. Then $max_\Gamma(L) = L_i$. By assumption $max_\Gamma(L) \notin M$ and $comp(max_\Gamma(L)) \notin M$. Contradiction. Suppose that $L' = L_i$ or $L' = comp(L_i)$ with $i < k$. Then we have $max_\Gamma(L') \prec_\Gamma max_\Gamma(L)$ contradicting assumption that the defining literal of L is minimal in M.

□

Lemma 2.2.19 Assume that all ground terms t with $t \prec_T \beta$ for any β are polynomial in the size of β. Then testing *Propagate* (*Conflict*) is NP-Complete, i.e., the problem of checking for a given clause C whether there exists a grounding substitution σ such that $C\sigma$ propagates (is false) is NP-Complete.

Proof. Let $C\sigma$ be propagable (false). The problem is in NP because β is constant and for all $t \in cdom(\sigma)$ it holds that t is polynomial in the size of β. Checking if $C\sigma$ is propagable (false) can be done in polynomial time with congruence closure [NO80] since σ has polynomial size.

We reduce 3-SAT to testing rule *Conflict*. Consider a 3-place predicate R, a unary function g, and a mapping from propositional variables P to first-order variables x_P. Assume a 3-SAT clause set $N = \{\{L_0, L_1, L_2\}, ..., \{L_{n-2}, L_{n-1}, L_n\}\}$, where L_i may denote both P_i and $\neg P_i$. Now we create the clause

$$\{R(t_0, t_1, t_2) \not\approx true, ..., R(t_{n-2}, t_{n-1}, t_n \not\approx true)\}$$

where $t_i := x_{P_i}$ if $L_i = P_i$ and $t_i := g(x_{P_i})$ otherwise. Now let $\Gamma := \{R(x_0, x_1, x_2) \mid x_i \in \{0, 1, g(0), g(1)\}$ such that $(x_0 \vee x_1 \vee x_2) \downarrow_{\{g(x) \mapsto (\neg x)\}}$ is true $\}$ be the set of all R-atoms that evaluate to true if considered as a three literal propositional clause. Now N is satisfiable if and only if *Conflict* is applicable to the new clause. The reduction is analogous for *Propagate*. □

Theorem 2.3.4 Assume a state $(\Gamma; N; U; \beta; k; D)$ resulting from a run. Then $(\Gamma; N; U; \beta; k; D)$ is sound.

Proof. Proof by structural induction on $(\Gamma; N; U; \beta; k; D)$. Let $(\Gamma; N; U; \beta; k; D) = (\epsilon, N, \emptyset, \beta, 0, \top)$, the initial state. Then it is sound according to Lemma 2.3.2. Now assume that $(\Gamma; N; U; \beta; k; D)$ is sound. We need to show that any application of a rule results in a sound state.

Propagate: Assume *Propagate* is applicable. Then there exists $C \in N \cup U$ such that $C = C_0 \vee C_1 \vee L$, $L\sigma$ is β-*undefined* in Γ, $C_1\sigma = L\sigma \vee ... \vee L\sigma$, $C_1 = L_1 \vee ... \vee L_n$,$\mu = mgu(L_1, ..., L_n, L)$ and $C_0\sigma$ is β-*false* in Γ. Then a reduction chain application $[I_1, ..., I_m]$ from Γ to $L\sigma^{k:(C_0 \vee L)\mu\cdot\sigma}$ is created with $I_m := (s_m\#t_m\cdot\sigma_m, s_m\#t_m \vee C_m\cdot\sigma_m, I_j, I_k, p_m)$. Finally $s_m\#t_m\sigma_m^{k:(s_m\#t_m \vee C_m)\cdot\sigma_m}$ is added to Γ.
By definition of a reduction chain application $(s_m\#t_m)\sigma_m = L\sigma{\downarrow_{conv(\Gamma)}}$. Thus, $(s_m\#t_m)\sigma_m$ must be β-*undefined* in Γ and irreducible by $conv(\Gamma)$, since $(C_0 \vee L)\mu\sigma \prec_T \beta$ by definition of *Propagate*.

- 2.3.1.1: Since $(s_m\#t_m)\sigma_m$ is β-*undefined* in Γ, adding $(s_m\#t_m)\sigma_m$ does not make Γ inconsistent. Thus $\Gamma, (s_m\#t_m)\sigma_m$ remains consistent.

- 2.3.1.2: $(s_m\#t_m)\sigma_m$ is β-*undefined* in Γ and irreducible by $conv(\Gamma)$. It remains to show that $C_m\sigma_m$ is β-*false* in Γ, $N \cup U \models s_m\#t_m \vee C_m$ and $(s_m\#t_m \vee C_m)\sigma_m \prec_T \beta$. By i.h. for all $L'\sigma'^{l:(L' \vee C')\cdot\sigma'} \in \Gamma$ it holds that $C'\sigma'$ is β-*false* in Γ, $(L' \vee C')\sigma' \prec_T \beta$ and $N \cup U \models (L' \vee C')$. By definition of *Propagate* $C_0\sigma$ is β-*false* in Γ and $C\sigma \prec_T \beta$ and $N \cup U \models C$. $(C_0 \vee C_1 \vee L)\mu$ is an instance of C. Thus $C \models (C_0 \vee C_1 \vee L)\mu$. $C_0\mu = L\mu \vee ... \vee L\mu$ by definition of *Propagate*. Thus $C \models (C_1 \vee L)\mu$ and by this $N \cup U \models (C_1 \vee L)\mu$. By definition of a reduction chain application I_j either contains a clause annotation from $\Gamma, L\sigma^{k:(C_0 \vee L)\cdot\sigma}$ or it is a rewriting inference from smaller rewrite steps for all $1 \leq j \leq m$. Thus, by Lemma 2.2.15 it follows by induction that for any rewriting inference $I_j := (s_j\#t_j\cdot\sigma_j, s_j\#t_j \vee C_j\cdot\sigma_j, I_i, I_k, p_j)$ it holds $C_j\sigma_j$ is β-*false* in Γ, $N \cup U \models s_j\#t_j \vee C_j$ and $(s_j\#t_j \vee C_j)\sigma_j \prec_T \beta$.

- 2.3.1.3 and 2.3.1.4 trivially hold by induction hypothesis.

- 2.3.1.5: trivially holds since $D = \top$.

Decide: Assume *Decide* is applicable. Then there exists $C \in N \cup U$ such that $C = C_0 \vee L$, $L\sigma$ is ground and β-*undefined* in Γ and $C_0\sigma$ is ground and β-*undefined* or β-*true* in Γ. Then a reduction chain application $[I_1, ..., I_m]$ from Γ to $L\sigma^{k+1:(C_0 \vee L)\cdot\sigma}$ is created with $I_m := (s_m\#t_m\cdot\sigma_m, s_m\#t_m \vee C_m\cdot\sigma_m, I_j, I_k, p_m)$. Finally $s_m\#t_m\sigma_m^{k+1:(s_m\#t_m \vee comp(s_m\#t_m))\cdot\sigma_m}$ is added to Γ.
By definition of a reduction chain application $(s_m\#t_m)\sigma_m = L\sigma{\downarrow_{conv(\Gamma)}}$. Thus, $(s_m\#t_m)\sigma_m$ must be β-*undefined* in Γ and irreducible by $conv(\Gamma)$, since $(C_0 \vee L)\sigma \prec_T \beta$ by definition of *Decide*.

- 2.3.1.1: Since $(s_m \# t_m)\sigma_m$ is β-*undefined* in Γ adding $(s_m\#t_m)\sigma_m$ does not make Γ inconsistent. Thus $\Gamma, (s_m\#t_m)\sigma_m$ remains consistent.

- 2.3.1.3: $(s_m\#t_m)\sigma_m$ is β-*undefined* in Γ and irreducible by $conv(\Gamma)$. $N \cup U \models (s_m\#t_m) \vee comp(s_m\#t_m)$ obviously holds. $(s_m\#t_m)\sigma_m \prec_T \beta$ holds inductively by Lemma 2.2.15 and since $L\sigma \prec_T \beta$.

- 2.3.1.2 and 2.3.1.4 trivially hold by induction hypothesis.

- 2.3.1.5: trivially holds since $D = \top$.

Conflict: Assume *Conflict* is applicable. Then there exists a $D'\sigma$ such that $D'\sigma$ is β-*false* in Γ. Then:

- 2.3.1.1 - 2.3.1.4 trivially hold by induction hypothesis
- 2.3.1.5: $D'\sigma$ is β-*false* in Γ by definition of *Conflict*. Now we have two cases:
  1. $D'\sigma$ is of level greater than zero. Then $N \cup U \models D'$ since $D' \in N \cup U$ by definition of *Conflict*.
  2. $D'\sigma$ is of level zero. Then we have to show that $N \cup U \models \bot$. For any literal $L_0^{0:(L_0 \vee D_0)\cdot\sigma} \in \Gamma$ it holds $N \models L_0$, since any literal of level 0 is a propagated literal. By definition of a level, for any $K \in D'\sigma$ there exists a core $core(\Gamma; K)$ that contains only literals of level 0. Thus $N \cup U \models core(\Gamma; K)$ and $core(\Gamma; K) \models \neg K$ for any such K. Then $N \cup U \models \neg D'\sigma$ and $N \cup U \models D'\sigma$ and therefore $N \cup U \models \bot$.

Skip: Assume *Skip* is applicable. Then $\Gamma = \Gamma', L$ and $D = D' \cdot \sigma$ and $D'\sigma$ is β-*false* in Γ'.

- 2.3.1.1: By i.h. Γ is consistent. Thus Γ' is consistent as well.
- 2.3.1.2- 2.3.1.4: trivially hold by induction hypothesis and since Γ' is a prefix of Γ.
- 2.3.1.5: By i.h. $D'\sigma$ is β-*false* in Γ and $N \cup U \models D'$. By definition of *Skip* $D'\sigma$ is β-*false* in Γ'.

Explore-Refutation: Assume *Explore-Refutation* is applicable. Then $D = (D' \vee s \# t)\cdot\sigma$, $(s \# t)\sigma$ is strictly $\prec_{\Gamma^*}$ maximal in $(D' \vee s \# t)\sigma$, $[I_1, ..., I_m]$ is a refutation from Γ and $(s \# t)\sigma$, $I_j = (s_j \# t_j \cdot \sigma_j, (s_j \# t_j \vee C_j)\cdot\sigma_j, I_l, I_k, p_j)$, $1 \leq j \leq m$, $(s_j \# t_j \vee C_j)\sigma_j \prec_{\Gamma^*} (D' \vee s \# t)\sigma$, $(s_j \# t_j \vee C_j)\sigma_j$ is β-*false* in Γ.

- 2.3.1.1-2.3.1.4 trivially hold by i.h.
- 2.3.1.5. By definition $(C_j \vee s_j \# t_j)\sigma_j$ is β-*false* in Γ. By i.h. for all $L'\sigma'^{l:(L' \vee C')\cdot\sigma'} \in \Gamma$ it holds that $N \cup U \models (L' \vee C')$. By i.h. $N \cup U \models D' \vee s \# t$. By definition of a refutation $I_j := (s_j \# t_j \cdot \sigma_j, s_j \# t_j \vee C_j \cdot \sigma_j, I_i, I_k, p_j)$ either contains a clause annotation from $\Gamma, (s \# t)\sigma^{k:(D' \vee s \# t)\cdot\sigma}$ or it is a rewriting inference from smaller rewrite steps for all $1 \leq j \leq m$. Thus it follows inductively by Lemma 2.2.15 that $N \cup U \models (s_j \# t_j \vee C_j)$.

Factorize: Assume *Factorize* is applicable. Then $D = D' \cdot \sigma$.

- 2.3.1.1 - 2.3.1.4 trivially hold by induction hypothesis.
- 2.3.1.5: By i.h. $D'\sigma$ is β-*false* in Γ and $N \cup U \models D'$. By the definition of *Factorize* $D' = D_0 \vee L \vee L'$ such that $L\sigma = L'\sigma$ and $\mu = mgu(L, L')$. $(D_0 \vee L \vee L')\mu$ is an instance of D'. Thus $N \cup U \models (D_0 \vee L \vee L')\mu$. Since $L\mu = L'\mu$, $(D_0 \vee L \vee L')\mu \models (D_0 \vee L)\mu$. Thus $N \cup U \models (D_0 \vee L)\mu$ and $(D_0 \vee L)\mu\sigma$ is β-*false* since $(D_0 \vee L)\mu\sigma = (D_0 \vee L)\sigma$ by definition of an mgu.

Equality-Resolution: Assume *Equality-Resolution* is applicable. Then $D = (D' \vee s \not\approx s')\sigma$ and $s\sigma = s'\sigma$, $\mu = mgu(s, s')$. Then

- 2.3.1.1 - 2.3.1.4 trivially hold by induction hypothesis.
- 2.3.1.5: By i.h. $(D' \vee s \not\approx s')\sigma$ is β-*false* in Γ and $N \cup U \models (D' \vee s \not\approx s')$. $D'\mu$ is an instance of $(D' \vee s \not\approx s')$. Thus $(D' \vee s \not\approx s') \models D'\mu$. Thus $N \cup U \models D'\mu$. $D'\mu\sigma$ is β-*false* since $(D' \vee s \not\approx s')\sigma$ is β-*false* and $D'\mu\sigma = D'\sigma$ by definition of a mgu.

Backtrack: Assume *Backtrack* is applicable. Then $\Gamma = \Gamma', K, \Gamma''$ and $D = (D' \vee L)\sigma$, where $L\sigma$ is of level k, and $D'\sigma$ is of level i.

- 2.3.1.1: By i.h. Γ is consistent. Thus $\Gamma' \subseteq \Gamma$ is consistent.
- 2.3.1.2 - 2.3.1.3: Since Γ' is a prefix of Γ by i.h. this holds.
- 2.3.1.4: By i.h. $N \cup U \models D' \vee L$ and $N \models U$. Thus $N \models U \cup \{D' \vee L\}$
- 2.3.1.5: trivially holds since $D = \top$ after backtracking.

□

Lemma 2.3.8 Assume a state $(\Gamma; N; U; \beta; k; D)$ resulting from a regular run where the current level $k > 0$ and a unit clause $l \approx r \in N$. Now assume a clause $C \vee L[l']_p \in N$ such that $l' = l\mu$ for some matcher μ. Now assume some arbitrary grounding substitutions σ' for $C \vee L[l']_p$, σ for $l \approx r$ such that $l\sigma = l'\sigma'$ and $r\sigma \prec_T l\sigma$. Then $(C \vee L[r\mu\sigma\sigma']_p)\sigma' \prec_{\Gamma^*} (C \vee L[l']_p)\sigma'$.

Proof. Let $(\Gamma; N; U; \beta; k; D)$ be a state resulting from a regular run where $k > 0$ and $\Gamma = [L_1, ..., L_n]$. Now we have two cases:

1. $\beta \prec_T (l \approx r)\sigma$. Since $(l \approx r)\sigma$ rewrites $L[l']_p\sigma'$, $\beta \prec_T L[l']_p\sigma'$ has to hold as well. Thus $(l \approx r)\sigma$ is β-*undefined* in Γ and $L[l']_p\sigma'$ is β-*undefined* in Γ. By definition of a trail induced ordering $\prec_{\Gamma^*} := \prec_T$ for β-*undefined* literals. Thus, in case that $L[r\mu]_p)\sigma\sigma'$ is still undefined, $(L[r\mu]_p)\sigma\sigma' \prec_{\Gamma^*} (L[l']_p)\sigma'$ has to hold since $(L[r\mu]_p)\sigma\sigma' \prec_T (L[l']_p)\sigma'$. Thus, according to the definition of multiset orderings, $(C \vee L[r\mu]_p)\sigma\sigma' \prec_{\Gamma^*} (C \vee L[l']_p)\sigma'$. In the case that $(L[r\mu]_p)\sigma\sigma'$ is defined, $(L[r\mu]_p)\sigma\sigma' \prec_{\Gamma^*} (L[l']_p)\sigma'$ has to hold as well by Definition 2.2.9.11. Thus, according to the definition of multiset orderings, $(C \vee L[r\mu]_p)\sigma\sigma' \prec_{\Gamma^*} (C \vee L[l']_p)\sigma'$.
2. $(l \approx r)\sigma \prec_T \beta$. Since propagation is exhaustive for literals of level 0 (cf. 2.3.5.2) $(l \approx r)\sigma$ is on the trail or defined and of level 0. Now we have two cases:
   (a) $(L[l']_p)\sigma'$ is of level 1 or higher. Since $(L[l']_p)\sigma'$ is reducible by $(l \approx r)\sigma$, $(L[l']_p)\sigma' \neq L_i$ and $(L[l']_p)\sigma' \neq comp(L_i)$ for all $L_i \in \Gamma$. Since $(L[l']_p)\sigma'$ is of level 1 or higher, rewriting with $(l \approx r)\sigma$ does not change the defining literal of $(L[l']_p)\sigma'$. Thus $(L[r\mu]_p)\sigma\sigma' \prec_{\Gamma^*} (L[l']_p)\sigma'$ has to hold since $(L[r\mu]_p)\sigma\sigma' \prec_T (L[l']_p)\sigma'$. Thus, according to the definition of multiset orderings, $(C \vee L[r\mu]_p)\sigma\sigma' \prec_{\Gamma^*} (C \vee L[l']_p)\sigma'$

(b) $(L[l']_p)\sigma'$ is of level 0. First we show that $(L[r\mu]_p)\sigma\sigma'$ is still of level 0. Suppose that $(L[l']_p)\sigma' = s \# s$. Then rewriting either the left or right side of the equation results in $(L[r\mu]_p)\sigma\sigma'$. Then $core(\Gamma; (l \approx r)\sigma)$ is also a core for $(L[r\mu]_p)\sigma\sigma'$ and thus $(L[r\mu]_p)\sigma\sigma'$ must be of level 0. Now suppose that $(L[r\mu]_p)\sigma\sigma' = s \# s$. Then it is of level 0 by definition of a level. Finally suppose that $(L[r\mu]_p)\sigma\sigma' \neq s \# s$ and $(L[l']_p)\sigma' \neq s \# s$. Then $core(\Gamma; (L[l']_p)\sigma') \cup core(\Gamma; (l \approx r)\sigma)$ is a core for $(L[r\mu]_p)\sigma\sigma'$. Thus $(L[r\mu]_p)\sigma\sigma'$ is of level 0. Since $(L[r\mu]_p)\sigma\sigma' \prec_T (L[l']_p)\sigma'$, $(L[r\mu]_p)\sigma\sigma' \prec_{\Gamma^*} (L[l']_p)\sigma'$ according to the definition of $\prec_{\Gamma^*}$. Thus, according to the definition of multiset orderings, $(C \vee L[r\mu]_p)\sigma\sigma' \prec_{\Gamma^*} (C \vee L[l']_p)\sigma'$.

□

Lemma 2.3.9 Let C, D be two clauses. If there exists a substitution σ such that $C\sigma \subset D$, then D is redundant with respect to C and any $\prec_{\Gamma^*}$.

Proof. Let τ be a grounding substitution for D. Since $C\sigma \subset D$, $C\sigma\tau \subset D\tau$. Thus, for any $L \in C\sigma\tau$ it holds $L \in D\tau$ and $C\sigma\tau \neq D\tau$. Thus, $C\sigma\tau \prec_{\Gamma^*} D\tau$ by definition of a multiset extension and $C\sigma\tau$ makes $D\tau$ redundant by Definition 1.1.35. □

Auxiliary Lemmas for the Proof of Lemma 2.3.7

Lemma 2.4.4. During a regular run, if $(\Gamma; N; U; \beta; k; \top)$ is the immediate result of an application of *Backtrack*, then there exists no clause $C \in N \cup U$ and a substitution σ such that $C\sigma$ is β-*false* in Γ.

Proof. We prove this by induction. For the induction start assume the state $(\Gamma'; N; U \cup \{D\}; \beta; i; \top)$ after the first application of *Backtrack* in a regular run, where D is the learned clause. Since *Backtrack* was not applied before, the previous (first) application of *Conflict* in a state $(\Gamma, K; N; U; \beta; k; \top)$ was immediately preceded by an application of *Propagate* or *Decide*. By the definition of a regular run there is no clause $C \in N$ with substitution σ such that $C\sigma$ is β-*false* in Γ. Otherwise *Conflict* would have been applied earlier. By the definition of *Backtrack*, there exists no substition τ such that $D\tau$ is β-*false* in Γ'. Since there existed such a substitution before the application of *Backtrack*, Γ' has to be a prefix of Γ and $\Gamma \neq \Gamma'$. Thus there exists no clause $C \in N \cup U \cup \{D\}$ and a grounding substitution δ such that $C\delta$ is β-*false* in Γ'.

For the induction step assume the state $(\Gamma'; N; U \cup \{D\}; \beta; i; \top)$ after nth application of *Backtrack*. By i.h. the previous application of *Backtrack* did not produce any β-*false* clause. It follows that the the previous application of *Conflict* in a state $(\Gamma, K; N; U; \beta; k; \top)$ was immediately preceded by an application of *Propagate* or *Decide*. By the definition of a regular run there is no clause $C \in N \cup U$ with substitution σ such that $C\sigma$ is β-*false* in Γ. Otherwise *Conflict* would have been applied earlier. By the definition of *Backtrack*, there exists no substition τ such that $D\tau$ is β-*false* in Γ'. Since there existed such a substitution before the application of *Backtrack*, Γ' has to be a prefix of Γ and $\Gamma \neq \Gamma'$. Thus there exists no clause $C \in N \cup U \cup \{D\}$ and a grounding substitution δ such that $C\delta$ is β-*false* in Γ'. □

Corollary 2.4.5. If *Conflict* is applied in a regular run, then it is immediately preceded by an application of *Propagate* or *Decide*, except if it is applied to the initial state.

Lemma 2.4.6. Assume a state $(\Gamma; N; U; \beta; k; D)$ resulting from a regular run. Then there exists no clause $(C \vee L) \in N \cup U$ and a grounding substitution σ such that $(C \vee L)\sigma$ is β-*false* in Γ, $comp(L\sigma)$ is a decision literal of level i in Γ and $C\sigma$ is of level $j < i$.

Proof. Proof is by induction. Assume the initial state $(\epsilon; N; \emptyset; \beta; 0; \top)$. Then any clause $C \in N$ is undefined in Γ. Then this trivially holds.

Now for the induction step assume a state $(\Gamma; N; U; \beta; k; D)$. Only *Propagate*, *Decide*, *Backtrack* and *Skip* change the trail and only *Backtrack* adds a new literal to U. By i.h. there exists no clause with the above properties in $N \cup U$.

Now assume that *Propagate* is applied. Then a literal L is added to the trail. Let $C_1 \vee L_1, ..., C_n \vee L_n$ be the ground clause instances that get β-*false* in Γ by the application such that L is the defining literal of $L_1, ..., L_n$. Then L_i is of level k for $1 \leq i \leq n$. Thus $L_i \neq comp(K)$ for the decision literal $K \in \Gamma$ of level k. Thus $C_1 \vee L_1, ..., C_n \vee L_n$ do not have the above properties.

Now assume that *Decide* is applied. Then a literal L of level $k+1$ is added to the trail. Let $C_1 \vee L_1, ..., C_n \vee L_n$ be the (ground) clause instances that get β-*false* in Γ by the application such that L is the defining literal of $L_1, ..., L_n$. By the definition of a regular run for all L_i with $1 \leq i \leq n$ it holds that $L_i \neq comp(L)$ or there exists another literal $K_i \in C_i$ such that K_i is of level $k+1$ and $L_i \neq K_i$, since otherwise *Propagate* must be applied. Thus $C_1 \vee L_1, ..., C_n \vee L_n$ do not have the above properties.

Now assume that *Skip* is applied. Then there are no new clauses that get β-*false* in Γ. Thus this trivially holds.

Now assume that *Backtrack* is applied. Then a new clause $D \vee L$ is added to U and $\Gamma = \Gamma', K, \Gamma''$ such that there is a grounding substitution τ with $(D \vee L)\tau$ β-*false* in Γ', K, there is no grounding substitution δ with $(D \vee L)\delta$ β-*false* in Γ'. Γ' is the trail resulting from the application of *Backtrack*. By Lemma 2.4.4, after application of *Backtrack* there exists no clause $C \in N \cup U$ and a substitution σ such that $C\sigma$ is β-*false* in Γ'. Thus there exists no clause with the above properties. □

Lemma 2.3.7 Let N be a clause set. The clauses learned during a regular run in SCL(EQ) are not redundant with respect to $\prec_{\Gamma^*}$ and $N \cup U$. For the trail only non-redundant clauses need to be considered.

I first prove that learned clauses are non-redundant and then that only non-redundant clauses need to be considered, Lemma 2.4.10, below.

Proof. Consider the following fragment of a derivation learning a clause:

$$\Rightarrow^{Conflict}_{\text{SCL(EQ)}} \qquad (\Gamma; N; U; \beta; k; D \cdot \sigma)$$

$$\Rightarrow^{\{Explore\text{-}Refutation, Skip, Eq\text{-}Res, Factorize\}^*}_{\text{SCL(EQ)}} \qquad (\Gamma'; N; U; \beta; l; C \cdot \sigma)$$

$$\Rightarrow^{Backtrack}_{\text{SCL(EQ)}} \qquad (\Gamma''; N; U \cup \{C\}; \beta; k'; \top)$$

Assume there are clauses in $N' \subseteq (gnd(N \cup U)^{\preceq_{\Gamma^*} C\sigma})$ such that $N' \models C\sigma$. Since $N' \preceq_{\Gamma^*} C\sigma$ and $C\sigma$ is β-*defined* in Γ, there is no β-*undefined* literal in N', as

all β-*undefined* literals are greater than all β-*defined* literals. If $\Gamma \models N'$ then $\Gamma \models C\sigma$, a contradiction. Thus there is a $C' \in N'$ with $C' \preceq_{\Gamma^*} C\sigma$ such that C' is β-*false* in Γ. Now we have two cases:

1. $\Gamma' \neq \Gamma$. Then $\Gamma = \Gamma', \Delta$. Thus at least one *Skip* was applied, so $C\sigma$ does not contain a literal that is β-*undefined* without the rightmost literal of Γ, therefore $C\sigma \neq D\sigma$. Suppose that this is not the case, so $C\sigma = D\sigma$. Then $D\sigma$ is β-*false* in Γ'. But since *Backtrack* does not produce any β-*false* clauses by Lemma 2.4.4, *Conflict* could have been applied earlier on $D\sigma$ contradicting a regular run. Since $C' \preceq_{\Gamma^*} C\sigma$ we have that $C' \neq D\sigma$ as well. Thus, again since *Backtrack* does not produce any β-*false* clauses by Lemma 2.4.4, at a previous point in the derivation there must have been a state such that C' was β-*false* under the current trail and *Conflict* was applicable but not applied, a contradiction to the definition of a regular run.

2. $\Gamma' = \Gamma$, then conflict was applied immediately after an application of *Decide* by Corollary 2.4.5 and the definition of a regular run. Thus $\Gamma = \Delta, K^{(k-1):D'\cdot\delta}, L^{k:D''\cdot\tau}$. C' does not have any β-*undefined* literals. Suppose that C' has no literals of level k. Then all literals in C' are of level $i < k$. Since C' is β-*false* in Γ, C' is β-*false* in Δ, K as well, since it does not have any literals of level k. Thus, again since *Backtrack* does not produce any β-*false* clauses by Lemma 2.4.4, at a previous point in the derivation there must have been a state such that C' was β-*false* under the current trail and *Conflict* was applicable but not applied, a contradiction to the definition of a regular run.
   Since $C' \preceq_{\Gamma^*} C\sigma$, it may have at most one literal of level k, namely $comp(L)$, since $comp(L) \in C\sigma$ by definition of a regular run, since *Skip* was not applied, and there exists only L such that $L \prec_{\Gamma^*} comp(L)$ and L is of level k. But L is β-*true* in Γ. Thus $L \notin C'$ has to hold.
   Now suppose that C' has one literal of level k. Thus $C' = C'' \vee comp(L)$, where C'' is β-*false* in Δ, K. But by Lemma 2.4.6 there does not exist such a clause. Contradiction.

□

Auxiliary Lemma for the Proof of Lemma 2.4.8

Lemma 2.4.7. Assume a clause $L_1 \vee ... \vee L_m$, a trail Γ resulting from a regular run starting from the initial state, and a reducible (by $conv(\Gamma)$) grounding substitution σ, such that $L_i\sigma$ is β-*false* (β-*true* or β-*undefined*) in Γ and $L_i\sigma \prec_T \beta$ for $1 \leq i \leq m$. Then there exists a substitution σ' that is irreducible by $conv(\Gamma)$ such that $L_i\sigma'$ is β-*false* (β-*true* or β-*undefined*) in Γ, $L_i\sigma' \prec_T \beta$ and $L_i\sigma{\downarrow_{conv(\Gamma)}} = L_i\sigma'{\downarrow_{conv(\Gamma)}}$.

Proof. Let $L_1 \vee ... \vee L_m$ be a clause, Γ a trail resulting from a regular run. Let $\sigma := \{x_1 \to t_1, ..., x_n \to t_n\}$. Now set $\sigma' := \{x_1 \to (t_1{\downarrow_{conv(\Gamma)}}), ..., x_n \to (t_n{\downarrow_{conv(\Gamma)}})\}$. Obviously σ' is irreducible by $conv(\Gamma)$ and $L_i\sigma' \prec_T \beta$ for all $1 \leq i \leq m$. By definition, $conv(\Gamma)$ is a confluent and terminating rewrite system. Since Γ is consistent, $t_j{\downarrow_{conv(\Gamma)}} \approx t_j$ is β-*true* in Γ for $1 \leq j \leq n$. Thus there exists a chain such that $L_i\sigma \to_{conv(\Gamma)} ... \to_{conv(\Gamma)} L_i\sigma'$ and $L_i\sigma'$ is β-*false* (β-*true*

or β-*undefined*) in Γ. Now there also exists a chain $L_i\sigma \rightarrow_{conv(\Gamma)} ... \rightarrow_{conv(\Gamma)} L_i\sigma{\downarrow}_{conv(\Gamma)}$. By definition of convergence there must exist a chain $L_i\sigma' \rightarrow_{conv(\Gamma)} ... \rightarrow_{conv(\Gamma)} L_i\sigma{\downarrow}_{conv(\Gamma)}$. Thus $L_i\sigma{\downarrow}_{conv(\Gamma)} = L_i\sigma'{\downarrow}_{conv(\Gamma)}$. □

Auxiliary Lemma for the Proof of Lemmas 2.3.11 and 2.3.17

Lemma 2.4.8. Suppose a sound state $(\Gamma; N; U; \beta; k; \top)$ resulting from a regular run. If there exists a $C \in N \cup U$ and a grounding substitution σ such that $C\sigma$ is β-*false* in Γ, then *Conflict* is applicable. Otherwise, If there exists a $C \in N \cup U$ and a grounding substitution σ such that $C\sigma \prec_T \beta$ and there exists at least one $L \in C$ such that $L\sigma$ is β-*undefined*, then one of the rules *Propagate* or *Decide* is applicable and a β-*undefined* literal $K \in D$, where $D \in gnd_{\prec_T}\beta(N \cup U)$ is β-*defined* after application.

Proof. Let $(\Gamma; N; U; \beta; k; \top)$ be a state resulting from a regular run. Suppose there exists a $C \in N \cup U$ and a grounding σ such that $C\sigma$ is β-*false* in Γ, then by Lemma 2.4.7 there exists an irreducible substitution σ' such that $C\sigma'$ is β-*false*. Thus *Conflict* is applicable. Now suppose there exists a $C \in N \cup U$ and a grounding substitution σ such that $C\sigma \prec_T \beta$ and there exists at least one $L \in C$ such that $L\sigma$ is β-*undefined*. By Lemma 2.4.7 there exists a irreducible substitution σ' such that $L\sigma'$ is β-*undefined*. Now assume that $C = C_0 \vee C_1 \vee L$ such that $C_1\sigma' = L\sigma' \vee ... \vee L\sigma'$ and $C_0\sigma'$ is β-*false* in Γ. Then *Propagate* is applicable. Let $C_1 = L_1, ..., L_n$ and $\mu = mgu(L_1, ..., L_n, L)$. Now let $[I_1, ..., I_m]$ be the reduction chain application from Γ to $L\sigma'^{k:(L \vee C_0)\mu\cdot\sigma'}$. Let $I_m = (s_m\#t_m\cdot\sigma_m, (s_m\#t_m \vee C_m)\cdot\sigma_m, I_j, I_k, p_m)$. Then $L\sigma'{\downarrow}_{conv(\Gamma)} = s_m\#t_m\sigma_m$ by definition of a reduction chain application. Thus $L\sigma'$ is β-*true* in $\Gamma, s_m\#t_m\sigma_m$. Since $L\sigma{\downarrow}_{conv(\Gamma)} = L\sigma'{\downarrow}_{conv(\Gamma)}$ by Lemma 2.4.7, $L\sigma$ is β-*true* in $\Gamma, s_m\#t_m\sigma_m$ as well. If $C_0\sigma$ is β-*undefined* or β-*true* in Γ then *Propagate* is not applicable to $C\sigma'$. If *Decide* is not applicable by definition of a regular run, then there exists a clause $C' \in (N \cup U)$ and a substitution δ such that *Propagate* is applicable. Then we can apply *Propagate* by definition of a regular run and a previously undefined literal gets defined after application as seen above and we are done. Now suppose that there exists no such clause. Then let $[I'_1, ..., I'_l]$ be the reduction chain application from Γ to $L\sigma'^{k+1:C\cdot\sigma'}$ and $I'_l = (s_l\#t_l\cdot\sigma_l, (s_l\#t_l \vee C_l)\cdot\sigma_l, I'_j, I'_k, p_l)$. Then $L\sigma'{\downarrow}_{conv(\Gamma)} = (s_l\#t_l)\sigma_l$ by definition of a reduction chain application. Thus $L\sigma'$ is β-*true* in $\Gamma, (s_l\#t_l)\sigma_l$. Since $L\sigma{\downarrow}_{conv(\Gamma)} = L\sigma'{\downarrow}_{conv(\Gamma)}$ by Lemma 2.4.7, $L\sigma$ is β-*true* in $\Gamma, (s_l\#t_l)\sigma_l$ as well. $(s_l\#t_l)\sigma_l^{k+1:(s_l\#t_l \vee comp(s_l\#t_l))\cdot\sigma_l}$ can be added to Γ by definition of a regular run and also by definition of *Decide* since $C \in N \cup U$, σ' is grounding for C and irreducible in $conv(\Gamma)$, $L\sigma'$ is β-*undefined* in Γ and $C\sigma' \prec_T \beta$. □

Auxiliary Lemma for the Proof of Lemmas 2.3.11

Lemma 2.4.9. Suppose a sound state $(\Gamma; N; U; \beta; k; D\cdot\sigma)$ resulting from a regular run. Then $D\sigma$ is of level 1 or higher.

Proof. Let $(\Gamma; N; U; \beta; k; D\cdot\sigma)$ be a state resulting from a regular run. Suppose that $D\sigma$ is not of level 1 or higher, thus $D\sigma$ is of level 0. Then *Conflict* was applied earlier to a clause that was of level 1 or higher. Thus there must have been an application of *Explore-Refutation* on a state $(\Gamma, \Gamma', L; N; U; \beta; l; D'\cdot\sigma')$ between the state after the application of *Conflict* and the current state resulting in a

state $(\Gamma, \Gamma', L^{l:(L \vee C)\cdot\delta}; N; U; \beta; l; D''\cdot\sigma'')$ such that $D'\sigma'$ is of level l and $D''\sigma''$ is of level 0, since no other rule can reduce the level of $D'\sigma'$. Then there exists a $K \in D'\sigma'$ such that L is the defining literal of K. Let $[I_1, ..., I_m]$ be the refutation of K and $I_j = (s_j \# t_j \cdot \sigma_j, (s_j \# t_j \vee C_j)\cdot\sigma_j, I_i, I_k, p_j)$ be the step that was chosen by *Explore-Refutation*. Then $D''\sigma'' = (s_j \# t_j \vee C_j)\sigma_j$. $C\delta \subset C_j\sigma_j$ has to hold since L is the defining literal of K. Then $C\delta$ must be of level 0 or empty. Note that $C\delta$ is of level l if L is a decision literal. But then, by the definition of a regular run, $L^{l:(L\vee C)\cdot\delta}$ must have been propagated before the first decision, since propagation is exhaustive at level 0. Contradiction. □

Lemma 2.3.11 If a regular run (without rule Grow) ends in a stuck state $(\Gamma; N; U; \beta; k; D)$, then $D = \top$ and all ground literals $L\sigma \prec_T \beta$, where $L \vee C \in N \cup U$ are β-*defined* in Γ.

Proof. First we prove that stuck states never appear during conflict resolution. Assume a sound state $(\Gamma; N; U; \beta; k; D \cdot \sigma)$ resulting from a regular run. Now we show that we can always apply a rule. Suppose that $D\sigma = (D' \vee L \vee L')\sigma$ such that $L\sigma = L'\sigma$. Then we must apply *Factorize* by the definition of a regular run. Now suppose that *Factorize* is not applicable and $\Gamma := \Gamma', L$ and $D\sigma$ is false in Γ'. If $D\sigma = (D' \vee s \not\approx s')\sigma$ such that $s\sigma = s'\sigma$, we can apply *Equality-Resolution*. So suppose that *Equality-Resolution* is not applicable. Then we can apply *Skip*. Now suppose that $\Gamma := \Gamma', L^{k:(L\vee C)\delta}$ and L is the defining literal of at least one literal in $D\sigma$, so *Skip* is not applicable. If $D\sigma = (D' \vee L')\sigma$ where $D'\sigma$ is of level $i < k$ and $L'\sigma$ is of level k and *Skip* was applied at least once during this conflict resolution, then *Backtrack* is applicable. If *Skip* was not applied and $L = comp(L'\sigma)$ and L is a decision literal, then *Backtrack* is also applicable. Otherwise, let $(s \# t)\sigma \in D\sigma$ such that $K \prec_{\Gamma^*} (s \# t)\sigma$ for all $K \in D\sigma$. $(s \# t)\sigma$ exists since *Factorize* is not applicable. By Lemma 2.4.9, $(s \# t)\sigma$ must be of level 1 or higher. By the definition of $\prec_{\Gamma^*}$, L must be the defining literal of $(s \# t)\sigma$ since L is of level 1 or higher and any literal in $D\sigma$ that has another defining literal is smaller than $(s \# t)\sigma$. Now suppose that L is a decision literal and $(s \# t)\sigma = comp(L)$. Then $(s \# t)\sigma$ is of level k and all other literals $K \in D\sigma$ are of level $i < k$, since $(s \# t)\sigma$ is the smallest β-*false* literal of level k and *Factorize* is not applicable. In this case *Explore-Refutation* is not applicable since a paramodulation step with the decision literal does not make the conflict clause smaller. But *Backtrack* is applicable in this case even if *Skip* was not applied earlier by the definition of a regular run. Thus $(s \# t)\sigma \neq comp(L)$ or L is a propagated literal has to hold. We show that in this case *Explore-Refutation* is applicable. Let $[I_1, ..., I_m]$ be a refutation of $(s \# t)\sigma$ from Γ, $I_m = (s_m \# t_m \cdot \sigma_m, (s_m \# t_m \vee C_m)\cdot\sigma_m, I_j, I_k, p_m)$. Since $[I_1, ..., I_m]$ is a refutation $s_m \# t_m\sigma_m = s' \not\approx s'$. Furthermore any I_i either contains a clause annotation from $\Gamma, (s \# t)\sigma^{k:D\cdot\sigma}$ or it is a rewrite inference from $I_{j'}, I_{k'}$ with $j', k' < i$. Thus by Lemma 2.2.15 it inductively follows that $C_m\sigma_m = D'\sigma_m \vee ... \vee D'\sigma_m \vee C'_1\sigma_m \vee ... \vee C'_n\sigma_m$, where $C'_1\sigma_m, ..., C'_n\sigma_m$ are clauses from Γ without the leading trail literal and $D\sigma = D'\sigma_m \vee (s \# t)\sigma$. Since L is the defining literal of $(s \# t)\sigma$ there must exist at least one C'_i such that $C'_i\sigma_m = C\delta$. If L is a propagated literal, then any literal in $C'_i\sigma_m$ is smaller than $(s \# t)\sigma$, since they are already *false* in Γ'. If L is a decision literal, then $C'_i\sigma_m = comp(L)$. Then $comp(L)$ is smaller, since $(s \# t)\sigma \neq comp(L)$ and $(s \# t)\sigma \neq L$. Thus $comp(L) \prec_{\Gamma^*} (s \# t)\sigma$. Any other literal in $C_1\sigma_m, ..., C'_n\sigma_m$ is smaller in $\prec_{\Gamma^*}$, since they are already defined in Γ'. Since *Factorize* is not applicable $(s \# t)\sigma$

is also strictly maximal in $D'\sigma_m$. Thus $(s_m \# t_m \vee C_m)\sigma_m \prec_{\Gamma^*} D\sigma$ which makes *Explore-Refutation* applicable.
Now by Lemma 2.4.8 it holds that if there exists an β-*undefined* literal in $gnd_{\prec_T \beta}(N \cup U)$, we can always apply at least one of the rules *Propagate* or *Decide* which makes a previously β-*undefined* literal in $gnd_{\prec_T \beta}(N \cup U)$ β-*defined*. □

Lemma 2.3.12 Suppose a sound state $(\Gamma; N; U; \beta; k; D)$ resulting from a regular run where $D \notin \{\top, \bot\}$. If *Backtrack* is not applicable then any set of applications of *Explore-Refutation*, *Skip*, *Factorize*, *Equality-Resolution* will finally result in a sound state $(\Gamma'; N; U; \beta; k; D')$, where $D' \prec_{\Gamma^*} D$. Then *Backtrack* will be finally applicable.

Proof. Assume a sound state $(\Gamma; N; U; \beta; k; D \cdot \sigma)$ resulting from a regular run. Let $(s \# t)\sigma \in D\sigma$ such that $L \preceq_{\Gamma^*} (s \# t)\sigma$ for all $L \in D\sigma$. If $(s \# t)\sigma$ occurs twice in $D\sigma$, then *Factorize* is applicable. Suppose that it is applied. Then $D\sigma = (D' \vee (s \# t) \vee L)\sigma$, where $L\sigma = (s \# t)\sigma$. Then $\mu = mgu(s \# t, L)$ and the new conflict clause is $(D' \vee s \# t)\mu\sigma \prec_{\Gamma^*} D\sigma$. Thus in this case we are done. If *Factorize* is not applicable, then the only remaining applicable rules are *Skip*, *Explore-Refutation* and *Equality-Resolution*. If $\Gamma = \Gamma', L, \Gamma''$ where L is the defining literal of $(s \# t)\sigma$, then *Skip* is applicable $|\Gamma''|$ times, since otherwise $(s \# t)\sigma$ would not be maximal in $D\sigma$. So at some point it is no longer applicable. Since $D\sigma$ is finite, *Equality-Resolution* can be applied only finitely often. Thus we finally have to apply *Explore-Refutation*. Then $[I_1, ..., I_m]$ is a refutation of $(s \# t)\sigma$ from Γ, and there exists an $1 \leq j \leq m$, such that $I_j = (s_j \# t_j \cdot \sigma_j, (s_j \# t_j \vee C_j) \cdot \sigma_j, I_l, I_k, p_j)$, $(C_j \vee s_j \# t_j)\sigma_j \prec_{\Gamma^*} (D' \vee s \# t)\sigma$. Otherwise *Explore-Refutation* would not be applicable, contradicting Lemma 2.3.11. Thus in this case we are done.
Now we show that *Backtrack* is finally applicable. Since $\prec_{\Gamma^*}$ is well-founded and Γ is finite there must be a state where *Explore-Refutation*, *Skip*, *Factorize*, *Equality-Resolution* are no longer applicable. By Lemma 2.4.9 the conflict clause in this state must be of level 1 or higher, thus $\bot$ cannot be inferred. Suppose that it is always of level $i \geq l$ for some l. The smallest literal of level l that is *false* in Γ is $comp(L)$, where L is the decision literal of level l. Since we can always reduce if *Backtrack* is not applicable and since we can always apply a rule by Lemma 2.3.11, we must finally reach a conflict clause $comp(L) \vee C$, where C is of level $j < l$. Thus *Backtrack* is applicable. □

Lemma 2.3.15 Let N be a set of clauses and β be a ground term. Then any regular run that never uses Grow terminates.

Proof. Assume a new ground clause $D\sigma$ is learned. By Lemma 2.3.7 all learned clauses are non-redundant. Thus $D\sigma$ is non-redundant. By the definition of a regular run *Factorize* has precedence over all other rules. Thus $D\sigma$ does not contain any duplicate literals. By Theorem 2.3.4, $D\sigma \prec_T \beta$ has to hold. There are only finitely many clauses $C\sigma \prec_T \beta$, where $C\sigma$ is neither a tautology nor does it contain any duplicate literals. Thus there are only finitely many clauses $D\sigma$ that can be learned. Thus there are only finitely many literals that can be decided or propagated. □

Lemma 2.3.16 If a regular run reaches the state $(\Gamma; N; U; \beta; k; \bot)$ then N is unsatisfiable.

Proof. By definition of soundness, all learned clauses are consequences of $N \cup U$, Definition 2.3.1.5, and Γ is satisfiable, Definition 2.3.1.1. □

Theorem 2.3.17 Let N be an unsatisfiable clause set, and $\prec_T$ a desired term ordering. For any ground term β where $gnd_{\prec_T \beta}(N)$ is unsatisfiable, any regular SCL(EQ) run without rule Grow will terminate by deriving $\bot$.

Proof. Since regular runs of SCL(EQ) terminate we just need to prove that it terminates in a failure state. Assume by contradiction that we terminate in a state $(\Gamma; N; U; \beta; k; \top)$. If no rule can be applied in Γ then for all $s \# t \in C$ for some arbitrary $C \in gnd_{\prec_T \beta}(N)$ it holds that $s \# t$ is *β-defined* in Γ (otherwise *Propagate* or *Decide* woud be applicable, see Lemma 2.4.8) and there aren't any clauses in $gnd_{\prec_T \beta}(N)$ *β-false* under Γ (otherwise *Conflict* would be applicable, see again Lemma 2.4.8). Thus, for each $C \in gnd_{\prec_T \beta}(N)$ it holds that C is *β-true* in Γ. So we have $\Gamma \models gnd_{\prec_T \beta}(N)$, but by hypothesis there is a Superposition refutation of N that only uses ground literals from $gnd_{\prec_T \beta}(N)$, so also $gnd_{\prec_T \beta}(N)$ is unsatisfiable, a contradiction. □

Lemma 2.4.10 (Only Non-Redundant Clauses Building the Trail)**.** Let $\Gamma = [L_1^{i_1 : C_1 \cdot \sigma_1}, ..., L_n^{i_n : C_n \cdot \sigma_n}]$ be a trail. If $L_j^{i_j : C_j \cdot \sigma_j}$ is a propagated literal and there exist clauses $\{D_1 \vee K_1, ..., D_m \vee K_m\}$ with grounding substitutions $\delta_1, ..., \delta_m$ such that $N := \{(D_1 \vee K_1)\delta_1, ..., (D_m \vee K_m)\delta_m\} \prec_{\Gamma^*} C_j\sigma_j$ and $\{(D_1 \vee K_1)\delta_1, ..., (D_m \vee K_m)\delta_m\} \models C_j\sigma_j$, then there exists a $(D_k \vee K_k)\delta_k \in N$ such that

$$[L_1^{i_1 : C_1 \cdot \sigma_1}, ..., L_{j-1}^{i_{j-1} : C_{j-1} \cdot \sigma_{j-1}}, K_k^{i_j : (D_k \vee K_k) \cdot \delta_k}, ..., L_n^{i_n : C_n \cdot \sigma_n}]$$

is a trail.

Proof. Let $N = \{(D_1 \vee K_1)\delta_1, ..., (D_m \vee K_m)\delta_m\}$ and $L_j^{i_j : C_j \cdot \sigma_j}$ be as above. Let $\Gamma' = [L_1^{i_1 : C_1 \cdot \sigma_1}, ..., L_{j-1}^{i_{j-1} : C_{j-1} \cdot \sigma_{j-1}}]$. Now suppose that for every literal $L \in N$ it holds $L \prec_{\Gamma^*} L_j$. Then every literal in N is defined in Γ' and $\Gamma' \models N$, otherwise *Conflict* would have been applied to a clause in N. Thus $\Gamma' \models C_j\sigma_j$ would have to hold as well. But by definition of a trail L_j is undefined in Γ'. Thus there must be at least one clause $(D_k \vee K_k)\delta_k \in N$ with $K_k = L_j$ and $D_k\delta_k \prec_{\Gamma^*} L_j$ (otherwise $(D_k \vee K_k)\delta_k \not\prec_{\Gamma^*} C_j\sigma_j$), such that $\Gamma' \not\models D_k$. Suppose that $\Gamma' \models D_k$. Then $N \not\models C_j\sigma_j$, since there exists an allocation, namely $\Gamma', \neg L_k$ such that $\Gamma', \neg L_k \models N$ but $\Gamma', \neg L_k \not\models C_j\sigma_j$. Thus we can replace $L_j^{i_j : C_j \cdot \sigma_j}$ by $K_k^{i_j : (D_k \vee K_k) \cdot \delta_k}$ in Γ. □

2.5 Discussion of SCL(EQ)

I presented SCL(EQ), a new sound and complete calculus for reasoning in first-order logic with equality. I will now discuss some of its aspects.

The SCL(EQ) calculus can be viewed as a generalization of the first-order without equality SCL calculus [BSW23], where syntactic equality with respect to trail literals is replaced with equality modulo the presented equational theory. If standard first-order literals like $R(x, y)$ are represented by equations like $f_R(x, y) \approx true$ then performing SCL(EQ) on the latter simplifies to classical SCL reasoning on the first-order literals with a slightly different strategy.

The trail induced ordering, Definition 2.2.9, is the result of letting the calculus follow the logical structure of the clause set on the literal level and at the same time supporting rewriting at the term level. It can already be seen by examples on ground clauses over (in)equations over constants that this combination requires a layered approach as suggested by Definition 2.2.9, see Example 2.5.1.

Example 2.5.1 (Propagate Smaller Equation)**.** Assume a term ordering $\prec_{kbo}$, unique weight 1 and with precedence $d \prec c \prec b \prec a$. Further assume β to be large enough. Assume the ground clause set N solely built out of constants

$$C_1 := c \approx d \qquad C_2 := c \not\approx d \vee a \approx b$$
$$C_3 := a \not\approx b \vee a \approx c$$

and the trail $\Gamma := [c \approx d^{0:C_1}, a \approx b^{0:C_2}, b \approx d^{0:C_3}]$. Now, although the first two steps propagated equations that are strictly maximal in the ordering in their respective clauses, the finally propagated equation $b \approx d$ is smaller in the term ordering $\prec_{kbo}$ than $a \approx b$. Thus the structure of the clause set forces propagation of a smaller equation in the term ordering. So the more complicated trail ordering is a result of following the structure of the clause set rather than employing an a priori fixed ordering.

In contrast to Superposition, SCL(EQ) does also inferences below variable level. In general, single Superposition inferences below variables are redundant [BG94]. Inferences in SCL(EQ) are guided by a false clause with respect to a partial model assumption represented by the trail. They are typically not single Superposition steps, but a sequence of Superposition inferences eventually resulting in a non-redundant clause changing the partial model assumption. Therefore, compared to the syntactic style of Superposition-based theorem proving, in SCL(EQ) reasoning below variables does not result in an explosion in the number of possibly inferred clauses but also rather in the derivation of more general clauses, see Example 2.5.2.

Example 2.5.2 (Rewriting below variable level)**.** Assume a term ordering $\prec_{kbo}$, unique weight 1 and with precedence $d \prec c \prec b \prec a \prec g \prec h \prec f$. Further assume β to be large enough. Assume the clause set N:

$$C_1 := f(x) \approx h(b) \vee x \not\approx g(a) \qquad C_2 := c \approx d \vee f(g(b)) \not\approx h(b)$$
$$C_3 := a \approx b \vee f(g(b)) \approx h(b)$$

Let $\sigma = \{x \to g(a)\}$ be a substitution. $C_1\sigma$ must be propagated: $\Gamma = [f(g(a)) \approx h(b)^{0:C_1\sigma}]$. Now suppose that we decide $f(g(b)) \not\approx h(b)$. Then $\Gamma = [f(g(a)) \approx h(b)^{0:C_1\sigma}, f(g(b)) \not\approx h(b)^{1:f(g(b))\not\approx h(b)\vee f(g(b))\approx h(b)}]$ and C_3 is a conflict clause. *Explore-Refutation* now creates the following refutation for $a \approx b$:

$$\begin{array}{ll}
I_1 := & (f(x) \approx h(b) \cdot \sigma, C_1 \cdot \sigma, \epsilon, \epsilon, \epsilon) \\
I_2 := & (f(g(b)) \not\approx h(b), C_2, \epsilon, \epsilon, \epsilon) \\
I_3 := & (a \approx b, C_3, \epsilon, \epsilon, \epsilon) \\
I_4 := & (f(g(b)) \approx h(b), f(g(b)) \approx h(b) \vee g(a) \not\approx g(a) \vee f(g(b)) \approx h(b), I_3, I_1, \epsilon) \\
I_5 := & (h(b) \not\approx h(b), h(b) \not\approx h(b) \vee g(a) \not\approx g(a) \vee f(g(b)) \approx h(b) \\
& \vee f(g(b)) \approx h(b), I_4, I_2, \epsilon)
\end{array}$$

Multiple applications of *Equality-Resolution* and *Factorize* result in the final conflict clause $C_4 := f(g(b)) \approx h(b)$ with which we can backtrack. The clause set resulting from this new clause is:

$$C_1 = f(x) \approx h(b) \vee x \not\approx g(a) \qquad C_2' = c \approx d$$
$$C_4 = f(g(b)) \approx h(b)$$

where C_2' is the result of a unit reduction between C_4 and C_2. Note that the refutation required rewriting below variable level in step I_4. Superposition would create the following clauses (Equality-Resolution and Factorization steps are implicitly done):

$$\begin{aligned} N \quad &\Rightarrow_{Sup(C_2,C_3)} N_1 \cup \{C_4 := c \approx d \vee a \approx b\} \\ &\Rightarrow_{Sup(C_1,C_2)} N_2 \cup \{C_5 := c \approx d \vee g(a) \not\approx g(b)\} \\ &\Rightarrow_{Sup(C_4,C_5)} N_3 \cup \{C_6 := c \approx d\} \end{aligned}$$

For Superposition the resulting clause set is thus:

$$C_1 = f(x) \approx h(b) \vee x \not\approx g(a) \qquad C_2 = a \approx b \vee f(g(b)) \approx h(b)$$
$$C_6 = c \approx d$$

Currently, the reasoning with solely positive equations is done on and with respect to the trail. It is well-known that also inferences from this type of reasoning can be used to speed up the overall reasoning process. The SCL(EQ) calculus already provides all information for such a type of reasoning, because it computes the justification clauses for trail reasoning via rewriting inferences. By an assessment of the quality of these clauses, e.g., their reduction potential with respect to trail literals, they could also be added, independently from resolving a conflict.

Chapter 3

$CC(\mathcal{X})$: Non-Ground Congruence Closure

As we have seen in the previous Chapter, SCL(EQ) relies heavily on equational models to find true and false literals. Congruence closure could be used for this task. However, for completeness, we require exhaustive propagation of units which would result in the instantiation of all possible ground instances smaller than β. In this Chapter, I therefore present CC($\mathcal{X}$) [LW24], which is a generalized congruence closure algorithm for terms with variables. This Chapter is now organized as follows. In Section 3.1, I briefly discuss related work. Section 3.2 presents my calculus in detail and in Section 3.3, I prove its correctness. Section 3.4 describes the details and adjustments I have made for the implementation. In Section 3.5, I provide the results of my evaluation and in Section 3.6, I show how KBO can be included as a constraint solver. Finally, in Section 3.7, I conclude. CC($\mathcal{X}$) was mainly developed by me. Christoph Weidenbach was involved in the exchange of ideas and the final polishing of the paper.

3.1 Related Work

To the best of my knowledge, the only algorithm that is similar to mine is Joe Hurd's Congruence Classes with Logic Variables [Hur01]. The algorithm creates a set of classes, where each class consists of multiple, possibly non-ground terms. It incrementally finds all matchers between all pairs of classes and applies these matchers to extend these classes. Therefore, in order to test equality of two terms they need to be added and the algorithm restarted. The size of terms is not constrained by the algorithm so it may diverge. The author does not introduce a notion of redundancy. The semantics of the classes is with respect to an infinite signature. In contrast, I generate a complete classification of all considered ground terms, i.e., testing equality of two terms means testing membership in the same class. Due to a notion of redundancy, my algorithm always terminates as seen in Lemma 3.3.5. Furthermore we created an implementation (Section 3.4).

Satisfiability modulo theory (SMT) [GHN$^+$04, NOT06] solvers (e.g., [dMB08, BBB$^+$22, BCBdODF09, Dut14, CGSS13]) make use of congruence closure [NO80, DST80, Sho84]. For SMT solvers many techniques have been invented to instantiate non-ground input equations [BFR17, RBF18, RTdM14] in order to make

them applicable to CC. All these techniques do not consider CC on equations with variables, instead the equations are grounded first. This applies in particular to [BFR17] where equations are assumed to be ground, but the equality to be tested may contain variables and the procedure aims at finding further potentially useful ground instances to the equations.

3.2 The Calculus

I now present my calculus in full detail. I begin with the definition of the ordering.

Definition 3.2.1 (Term Ordering)**.** Let $\preceq$ be a total quasi-ordering on ground terms where the strict part is well-founded. The ordering is lifted to the non-ground case via instantiation: I define $t \preceq s$ if for all grounding substitutions σ it holds $t\sigma \preceq s\sigma$. Given a ground term β then $\text{gnd}_{\preceq\beta}$ computes the set of all ground instances of a term, equation, or sets thereof where all ground terms are smaller or equal to β with respect to $\preceq$. By $T_{\preceq\beta}(\Omega, \emptyset)$ or just $T_{\preceq\beta}$ I denote the set of all ground terms $\preceq \beta$.

Definition 3.2.2. A *constrained term* $\Gamma \parallel s$ is a term s with a constraint Γ. The *constraint* Γ is a conjunction of atoms $t \preceq \beta$. A substitution σ is *grounding* for a constraint term $\Gamma \parallel s$ if $\Gamma\sigma$ and $s\sigma$ are ground. A ground constraint Γ is *true* if for all $t \preceq \beta \in \Gamma$ it indeed holds $t \preceq \beta$, and *false* otherwise. A constraint Γ is satisfiable if there exists a grounding σ such that $\Gamma\sigma$ is *true*.

The constraint Γ restricts the possible ground instances of the term s to those instances $s\sigma$ such that $\Gamma\sigma$ evaluates to true. If it is clear from the context, I omit the $\preceq \beta$ and just write the left-hand side of the inequation. A constraint class is a set of constraint terms. I distinguish between separating and free variables, where a separating variable occurs in all terms within the class whereas a free variable does not.

Definition 3.2.3 (Congruence Class)**.** A *congruence class* or simply class is a finite set of constraint terms $\Gamma \parallel s$. Let $A = \{\Gamma_1 \parallel s_1, ..., \Gamma_n \parallel s_n\}$ be a class. The set of *separating* variables X of A is defined as $X = vars(s_1) \cap ... \cap vars(s_n)$. The set of *free* variables Y of A is defined as $Y = (vars(s_1) \cup ... \cup vars(s_n)) \setminus X$. A substitution is *grounding* for A if it is grounding for all constraint terms $\Gamma_i \parallel s_i$.

If the terms in a congruence class all have the same constraint then I use $\{\Gamma \parallel s_1, ..., s_n\}$ as a shorthand for $\{\Gamma \parallel s_1, ..., \Gamma \parallel s_n\}$. For example, with all shorthands we can now write $\{g(x), h(x) \parallel g(x), h(x)\}$ instead of $\{g(x) \preceq \beta, h(x) \preceq \beta \parallel g(x), g(x) \preceq \beta, h(x) \preceq \beta \parallel h(x)\}$. In the calculus later on the constraints of each term within a class are always the same. Variables in a class can always be renamed.

Definition 3.2.4. Let $A = \{\Gamma_1 \parallel s_1, ..., \Gamma_n \parallel s_n\}$ be a class and μ a substitution. I define $A\mu$ as $\{\Gamma_1\mu \parallel s_1\mu, ..., \Gamma_n\mu \parallel s_n\mu\}$. In particular, if μ is grounding for A I overload its application by $A\mu = \{s\mu \mid (\Gamma \parallel s \in A \text{ and } \Gamma\mu \text{ true})\}$.

The semantics of a congruence class with variables is defined by creating a mapping to the corresponding ground classes. It is important to distinguish between separating and free variables here. Separating variables divide the non-ground class into several ground classes.

Definition 3.2.5 (Congruence Class Semantics)**.** Let A be a congruence class. Let X be the separating variables of A. Then the set $\mathrm{gnd}'(A)$ is defined as:

$$\bigcup_{\substack{\sigma \text{ ground},\\ \mathsf{dom}(\sigma)=X}} \{\{(\Gamma\sigma \parallel s\sigma) \mid (\Gamma \parallel s) \in A \text{ and } \Gamma\sigma \text{ satisfiable}\}\}$$

and the set $\mathrm{gnd}(A)$ is defined as:

$$\bigcup_{B \in \mathrm{gnd}'(A)} \Big\{ \bigcup_{\sigma \text{ grounding for } B} \{s\sigma \mid (\Gamma \parallel s) \in B \text{ and } \Gamma\sigma \text{ true}\} \Big\}$$

Example 3.2.6. Assume $\Omega = \{g, h, a, b\}$, a term $\beta = g(a)$, an ordering such that only $a, b, g(a), g(b), h(a), h(b) \preceq g(a)$ and classes

$$A = \{g(x), h(x) \parallel g(x), h(x)\},\ B = \{g(x), h(y) \parallel g(x), h(y)\}$$

Then

$$\mathrm{gnd}(A) = \{\{g(a), h(a)\}, \{g(b), h(b)\}\} \text{ and } \mathrm{gnd}(B) = \{\{g(a), h(a), g(b), h(b)\}\}$$

Definition 3.2.7 (Normal Class)**.** Let A be a class. The *normal* class $norm(A)$ is defined as $\{(\Gamma \wedge \Gamma\sigma \parallel s) \mid (\Gamma \parallel s) \in A\} \cup \{(\Gamma \wedge \Gamma\sigma \parallel s\sigma) \mid (\Gamma \parallel s) \in A$ and s contains free variables$\}$ for a renaming σ on the free variables which are introducing only fresh variables.

A class can be turned into a normal class by generating exactly one renamed copy for all constrained terms containing free variables. The motivation for normal classes is of technical nature. $\mathrm{CC}(\mathcal{X})$ rules always operate on two terms out of a class. In case of terms with variables the two terms may be actually instances of the same term from the class. This can only happen for terms with free variables. By introducing one renamed copy for such terms, the style of $\mathrm{CC}(\mathcal{X})$ rules is preserved and the rules do not need to distinguish between free and separated variables. For example, the class $A = \{g(x), h(y) \parallel g(x), h(y)\}$ contains all ground terms build with top-symbols g and h. So for a $\mathrm{CC}(\mathcal{X})$ step picking $g(a)$ and $g(b)$ out of the class the term $g(x)$ needs to be instantiated with two different constants. By using renamed copies $norm(A) = \{g(x), g(x'), h(y), h(y') \parallel g(x), g(x'), h(y), h(y')\}$ this technical issue is removed. Obviously, $\mathrm{gnd}(A) = \mathrm{gnd}(norm(A))$, holding for all classes and their normal counterparts.

Definition 3.2.8 (Subsumption)**.** A class B subsumes another class A if for all $A' \in \mathrm{gnd}(A)$ there exists a $B' \in \mathrm{gnd}(B)$ such that $A' \subseteq B'$.

The following Definitions and Lemmas prepare termination, Lemma 3.3.5. I show that I can not create infinitely many classes restricted by β such that each new class is not subsumed by any existing class, guaranteeing termination.

Definition 3.2.9. Let $A = \{\Gamma_1 \parallel s_1, ..., \Gamma_n \parallel s_n\}$ be a class and β a ground term. A is constrained by β (or β-constrained) iff $s_i \preceq \beta \in \Gamma_i$ for all $1 \leq i \leq n$.

Lemma 3.2.10. Let β be a ground term and A a β-constrained class. Then $\mathrm{gnd}(A) \in \mathcal{P}(T_{\preceq\beta})$, the powerset of $T_{\preceq\beta}$.

Proof. For any $B \in \text{gnd}'(A)$ there exists a σ such that $B = \{(\Gamma\sigma \parallel s\sigma) \mid (\Gamma \parallel s) \in A \text{ and } \Gamma\sigma \text{ satisfiable}\}$. Thus for any $\Gamma\sigma \parallel s\sigma \in B$ we have $s\sigma \preceq \beta \in \Gamma\sigma$. Thus B is β-constrained. For any $A' \in \text{gnd}(A)$ we have $A' = \bigcup_{\sigma \text{ grounding for } B}\{s\sigma \mid (\Gamma \parallel s) \in B \text{ and } \Gamma\sigma \text{ true}\}$ for some $B \in \text{gnd}'(A)$. Thus for any $s\sigma \in A'$ we have $s\sigma \preceq \beta$, since B is β-constrained and $s\sigma$ is ground. Thus $A' \subseteq T_{\preceq\beta}$. Thus $\text{gnd}(A) \subseteq \mathcal{P}(T_{\preceq\beta})$. □

Lemma 3.2.11. Let β be a ground term. There exists no infinite chain of (possibly non-ground) β-constrained classes $A_0, A_1, ...$ such that for all $i \geq 0$, A_i is not subsumed by any A_j $(0 \leq j < i)$.

Proof. Assume there exists such a chain. Let $\mathcal{P}(T_{\preceq\beta})$ be the powerset of $T_{\preceq\beta}$. Since $T_{\preceq\beta}$ is finite, $\mathcal{P}(T_{\preceq\beta})$ is finite as well. There are only $2^{|T_{\preceq\beta}|}$ different subsets in $\mathcal{P}(T_{\preceq\beta})$. For any A_i in the chain it holds $\text{gnd}(A_i) \in \mathcal{P}(T_{\preceq\beta})$ by Lemma 3.2.10. If there exist indices $i \neq j$ such that $\text{gnd}(A_i) = \text{gnd}(A_j)$ then A_j subsumes A_i and vice versa contradicting the assumption. Thus, for any pair of indices $i \neq j$ in the chain it has to hold $gnd(A_i) \neq gnd(A_j)$ which contradicts the fact that there are only finitely many different subsets. □

Definition 3.2.12. Let $A = \{\Gamma_1 \parallel s_1, ..., \Gamma_n \parallel s_n\}$ be a class. The set $vars(A)$ is defined as $\bigcup_{1\leq i\leq n}(\bigcup_{t\in\Gamma_i} vars(t)) \cup vars(s_i)$.

A *state* of $\text{CC}(\mathcal{X})$ is a finite set of congruence classes. For a state $\Pi = \{A_1, ..., A_n\}$ I define $\text{gnd}(\Pi) = \text{gnd}(A_1) \cup ... \cup \text{gnd}(A_n)$. Let Π be a state. For any classes $\{A, B\} \subseteq \Pi$ with $A \neq B$, I assume that $vars(A) \cap vars(B) = \emptyset$ at any time during execution. Now given a set of equations E, where $\text{gnd}_{\preceq\beta}(s \approx t) \neq \emptyset$ for all $s \approx t \in E$ the initial state of $\text{CC}(\mathcal{X})$ is

$$\begin{aligned}\Pi = &\{\{s \preceq \beta \wedge t \preceq \beta \parallel s, s \preceq \beta \wedge t \preceq \beta \parallel t\} \mid s \approx t \in E\} \cup \\ &\{\{f_i(x_{1_i}, \ldots, x_{k_i}) \preceq \beta \parallel f_i(x_{1_i}, \ldots, x_{k_i})\} \mid f_i \in \Omega\}\end{aligned}$$

In particular, the linear single term classes $f_i(x_{1_i}, \ldots, x_{k_i})$ are needed for rule *Deduction* to build terms that are not contained in E as a subterm but $\preceq \beta$. I present my algorithm in the form of two abstract rewrite rules:

Merge $\Pi \cup \{A, B\} \Rightarrow_{\text{CC}(\mathcal{X})} \Pi \cup \{A, B, (A' \cup B')\mu\}$
provided $norm(A) = \{\Gamma_1 \parallel s_1, ..., \Gamma_n \parallel s_n\}$, $norm(B) = \{\Delta_1 \parallel t_1, ..., \Delta_n \parallel t_n\}$, there exist $(\Gamma \parallel s) \in A, (\Delta \parallel t) \in B$ and μ such that $\mu = mgu(s, t)$, $A' = \{\Gamma_1 \wedge \Gamma \wedge \Delta \parallel s_1, ..., \Gamma_n \wedge \Gamma \wedge \Delta \parallel s_n\}$, $B' = \{\Delta_1 \wedge \Gamma \wedge \Delta \parallel t_1, ..., \Delta_n \wedge \Gamma \wedge \Delta \parallel t_n\}$, there exists no $A'' \in \Pi \cup \{A, B\}$ such that $(A' \cup B')\mu$ is subsumed by A''.

The rule *Merge* takes as input two classes where a term in the first class is unifiable with a term in the second class. The result is the union of their normal classes with the unifier applied. For termination it is crucial to check if there exists a class that subsumes the newly generated class. Note that the *Merge* rule can be seen as a generalization of the *Merge* rule in $\Rightarrow_{\text{CC}}$.

Deduction $\Pi \cup \{A, B\} \Rightarrow_{\text{CC}(\mathcal{X})} \Pi \cup \{A, B, \{\Gamma' \parallel f(s'_1, ..., s'_n), \Gamma' \parallel f(t'_1, ..., t'_n)\}\mu\}$
provided $\Gamma \parallel f(s_1, ..., s_n) \in A$, $\Delta \parallel f(t_1, ..., t_n) \in B$, and for each $0 < i \leq n$, there exists a $D_i \in \Pi$ such that $\Gamma_i \parallel s'_i \in norm(D_i), \Delta_i \parallel t'_i \in norm(D_i)$, μ is a simultaneous mgu of $f(s'_1, ..., s'_n) = f(s_1, ..., s_n)$ and $f(t'_1, ..., t'_n) = f(t_1, ..., t_n)$, $\Gamma' = \Gamma \wedge \Delta \wedge \Gamma_1 \wedge ... \wedge \Gamma_n \wedge \Delta_1 \wedge ... \wedge \Delta_n$, there exists no $A' \in \Pi \cup \{A, B\}$ such that $\{\Gamma' \parallel f(s'_1, ..., s'_n), \Gamma' \parallel f(t'_1, ..., t'_n)\}\mu$ is subsumed by A'.

The rule *Deduction* creates a new class if there exist terms with the same top symbol in two different (copies of) classes such that their arguments are unifiable with terms that are in the same class. Again *Deduction* is a generalization of the *Deduction* rule from $\Rightarrow_{\mathrm{CC}}$. Note, that in *Merge* and *Deduction* A and B can be identical. In this case I assume the consideration of a renamed copy. Furthermore, in both rules I inherit all parent constraints for the new class. Using this invariant, I could also represent each class by a single constraint that is not dedicated to a term. I do not do so in order to get a nicer representation, the way constraints are composed depending on the term they belong to. My way of constraint composition can also result in constraints containing variables that do not occur in any term of the class anymore. I'll take care of these constraints in Section 3.4.

Note that during the creation of new classes, a lot of classes may become redundant. While these redundant classes affect neither soundness nor completeness getting rid of redundant classes is essential for an efficient implementation. To this end I introduce a Subsunption rule that has precedence over all other rules.

Subsumption $\Pi \cup \{A, B\} \;\Rightarrow_{\mathrm{CC}(\mathcal{X})}\; \Pi \cup \{B\}$
provided B subsumes A.

Example 3.2.13. Suppose we have the following equations: $g(x) \approx a$, $h(y) \approx a$, $g(h(z)) \approx h(h(z))$. Initially, without single term classes, we get

$$\Pi = \{\{g(x), a \parallel g(x), a\}, \{h(y), a \parallel h(y), a\}, \{g(h(z)), h(h(z)) \parallel g(h(z)), h(h(z))\}\}$$

Merging the last class with the second class we get

$$\{g(h(z)), h(h(z)), h(y), a \parallel g(h(z)), h(h(z)), h(y), a\}$$

We can now merge this new class with the first class to get:

$$\{g(h(z)), h(h(z)), a, h(y), g(x) \parallel g(h(z)), h(h(z)), a, h(y), g(x)\}$$

This class subsumes all other classes, for example in our ordering defined in Section 3.4. So this would be the final result. Note that this result is independent of the chosen β. No matter how large β is the result of the calculus is always the same (assuming that β allows for the initial classes).

Another example where the number of classes is dependant on β is shown below.

Example 3.2.14. Suppose we have the single equation $f(x) \approx g(x)$. Initially we have $\Pi = \{\{f(x), g(x) \parallel f(x), g(x)\}\}$. Depending on the size of β we get more and more classes with the *Deduction* rule, like $\{f(f(x)), f(g(x)), f(x), g(x) \parallel f(f(x)), f(g(x))\}$ and $\{g(f(x)), g(g(x)), f(x), g(x) \parallel g(f(x)), g(g(x))\}$. Which we can again merge with the first class to get

$$\{f(f(x)), f(g(x)), g(f(x)), g(g(x)), f(x), g(x) \parallel$$

$$f(f(x)), f(g(x)), g(f(x)), g(g(x))\}$$

Increasing β further we get even larger classes. This is an example where we gain quite little compared to congruence closure.

The following example shows that *Merge* also has to be applied to two instances of the same class.

Example 3.2.15. Assume initial classes (without single term classes):

$$\Pi = \{\{f(x,y), g(x,y) \parallel f(x,y), g(x,y)\},$$
$$\{f(x,y), g(y,x) \parallel f(x,y), g(y,x)\}\}$$

Now we can merge the classes by unifying $f(x,y)$:

$$\{f(x,y), g(x,y), g(y,x) \parallel f(x,y), g(x,y), g(y,x)\}$$

The new class subsumes both initial classes. To get the final result we have to merge this new class with itself by unifying $g(x,y)$ and $g(y,x)$ to get:

$$A = \{f(x,y), g(x,y), g(y,x), f(y,x) \parallel$$
$$f(x,y), g(x,y), g(y,x), f(y,x)\}$$

otherwise, e.g. $\{f(a,b), f(b,a)\} \not\subseteq A'$ for all $A' \in gnd(A)$ for an appropriate set of function symbols in Ω.

Note, that $\mathrm{CC}(\mathcal{X})$ does not always guarantee less or equally many Congruence Classes than CC. Consider the following example

Example 3.2.16. Let $f(a) \approx h(a), g(a) \approx h(a), f(b) \approx h(b), g(b) \approx h(b), f(x) \approx g(x), a \approx f(a), b \approx f(b)$ be some input equations. Further assume that the ground terms occuring in the input equations are the only ground terms smaller than a given β. Initially $\mathrm{CC}(\mathcal{X})$ contains a class for the terms in each equation. Note, that Subsumption is not applicable in this state. Now, multiple *Merge* operations are possible, but no matter in which sequence they are applied the result is always

$$\{\{f(a), h(a), g(a), a \parallel f(a), h(a), g(a), a\}, \{f(b), h(b), g(b), b \parallel f(b), h(b), g(b), b\},$$
$$\{f(x), g(x) \parallel f(x), g(x)\}\}$$

Subsumption is not applicable to this set of classes. In ground congruence closure, however, we only get two classes:

$$\{\{f(a), h(a), g(a), a\}, \{f(b), h(b), g(b), b\}\}$$

In the special case of equations with flat terms, where the left-hand side is variable disjoint to the right-hand side and every variable occurs only once, $\mathrm{CC}(\mathcal{X})$ terminates even without constraints as I will show in the following Lemma.

Lemma 3.2.17. Let E be a set of equations of the form $f(x_1, ..., x_n) \approx g(y_1, ..., y_m)$. Then $\mathrm{CC}(\mathcal{X})$ terminates on E with empty contraints.

Proof. We prove that

1. the terms stay flat, i.e. *Deduction* is never applicable and if *Merge* is applicable, then μ is a renaming and

2. the number of *Merge* operations is at most $|E| - 1$

Since there are no constraints and no separating variables, any subsumption check reduces to a check if there exists a term in the subsuming class and a matcher for the free variables for every term in the subsumed class.

Assume that *Deduction* is applicable. Then there exists a $f(x_1, ..., x_n)$ in A and $f(y_1, ..., y_n)$ in B. *Deduction* would now create a new class $\{f(s_1, ..., s_n), f(t_1, ..., t_n)\}$. But there already exists a class $\{f(x_1, ..., x_n), g(y_1, ..., y_m), ...\}$ with no constraints. Thus there exists two matchers δ, δ' such that $f(x_1, ..., x_n)\delta = f(s_1, ..., s_n)$ and $f(x_1, ..., x_n)\delta' = f(t_1, ..., t_n)$. Thus $\{f(s_1, ..., s_n), f(t_1, ..., t_n)\}$ is subsumed.

Assume that *Merge* is applicable to classes A and B. Then there exists a $f(x_1, ..., x_n)$ in A and $f(y_1, ..., y_n)$ in B. Then μ is a renaming. If *Merge* is applied to the same class, i.e. if B is a renaming of A, then the resulting class is obviously subsumed by A. So A and B must be different. Let C be the resulting class. C subsumes A and B, since for every $t \in A$ there exists a $t' \in C$ such that there exists a matcher δ such that $t'\delta = t$ and the constraints are empty, and analogously for B.

Since *Deduction* is never applicable and *Merge* always creates a class that subsumes the merged classes, the number of *Merge* operations is at most $|E| - 1$, since every *Merge* operation reduces the total number of classes by 1. Thus $\mathrm{CC}(\mathcal{X})$ terminates with the correct result for empty constraints. □

3.3 Correctness

I will now prove termination, soundness and completeness. I start with soundness.

Lemma 3.3.1. Let A be a class. There exists a $A' \in gnd(A)$ s.t. $\{s, t\} \subseteq A'$ iff there exists a substitution σ' such that $\{s, t\} \subseteq norm(A)\sigma'$.

Proof. Let X be the separating and Y be the free variables of A. Let τ be the renaming that maps the free variables to fresh variables to create the normal class.

Assume $\{s, t\} \subseteq A'$ for some $A' \in gnd(A)$. There must exist terms $\{\Gamma \parallel s', \Delta \parallel t'\} \subseteq A$ and substitutions $\sigma : X \to \mathcal{T}(\Omega, \emptyset), \delta : Y \to \mathcal{T}(\Omega, \emptyset), \delta' : Y \to \mathcal{T}(\Omega, \emptyset)$ such that $s'\sigma\delta = s$ and $t'\sigma\delta' = t$ by Definition 3.2.5. Let $\delta' = \{x_1 \to s_1, ..., x_n \to s_n\}$. Construct $\delta'' = \{x_1\tau \to s_1, ..., x_n\tau \to s_n\}$. Now we can construct σ' to be $\sigma\delta\delta''$ and we are done, since $\{\Gamma' \parallel s', \Delta' \parallel t'\tau\} \subseteq norm(A)$ and $s'\sigma' = s$ and $t'\tau\sigma' = t$.

Now assume there exists a substitution σ' such that $\{\Gamma\sigma' \parallel s'\sigma', \Delta\sigma' \parallel t'\sigma'\} \subseteq norm(A)\sigma'$ and $s'\sigma' = s$ and $t'\sigma' = t$. Define $\sigma : X \to \mathcal{T}(\Omega, \emptyset)$ and $\delta : (Y \cup Y\tau) \to \mathcal{T}(\Omega, \emptyset)$ such that $\sigma' = \sigma\delta$. Let τ' be the reverse renaming of τ. Let $\delta = \{x_1 \to s_1, ..., x_n \to s_n, y_1 \to t_1, ..., y_m \to t_m\}$, where the y_i are the fresh variables introduced by τ. Construct $\delta' = \{x_1 \to s_1, ..., x_n \to s_n\}$ and $\delta'' = \{y_1\tau' \to t_1, ..., y_m\tau' \to t_m\}$. Now there must exist $\{\Gamma\tau' \parallel s'\tau', \Delta\tau' \parallel t'\tau'\} \subseteq A$ such that ($s'\tau'\sigma'\delta' = s$ or $s'\tau'\sigma'\delta'' = s$) and ($t'\tau'\sigma'\delta'' = t$ or $t'\tau'\sigma'\delta' = t$). □

Lemma 3.3.2. Let E be a finite set of non-ground equations and β a ground term. Let A be a class where all constraints are the same, X sep. variables, Y free variables of A and $gnd_{\preceq\beta}(E) \models s\sigma \approx t\sigma$ for all $\{s\sigma, t\sigma\} \subseteq A\sigma$ and grounding σ. Then for all $\Gamma \parallel s \in A$ we have $gnd_{\preceq\beta}(E) \models s\sigma\delta \approx s\sigma\delta'$ for all $\sigma : X \to \mathcal{T}(\Omega, \emptyset)$,

$\delta : Y \to \mathcal{T}(\Omega, \emptyset)$ and $\delta' = \{y \mapsto \alpha \mid \alpha$ a smallest term acc. to $\preceq, y \in Y\}$, $\Gamma\sigma\delta$ satisfiable.

Proof. Let Y' be the free variables of s. Choose $y' \in Y'$. There must exist a $\Gamma \parallel t \in A$ s.t. $y' \notin vars(t)$. By assumption $gnd_{\preceq\beta}(E) \models s\sigma\delta \approx t\sigma\delta$. Now let δ'' be δ but y' maps to α. Then $gnd_{\preceq\beta}(E) \models s\sigma\delta \approx t\sigma\delta''$ since $t\sigma\delta = t\sigma\delta''$, and by hypothesis $gnd_{\preceq\beta}(E) \models t\sigma\delta'' \approx s\sigma\delta''$. Obviously the constraints are still satisfiable since we replace by smallest term α. Now we can continue analogously for $s\sigma\delta''$ until we reach $s\sigma\delta'$ which shows the Lemma. □

Corollary 3.3.3. Let E be a finite set of non-ground equations and β a ground term. Let A be a class where all constraints are the same, and $gnd_{\preceq\beta}(E) \models s\sigma \approx t\sigma$ for all $\{s\sigma, t\sigma\} \subseteq A\sigma$ and grounding σ. Then $gnd_{\preceq\beta}(E) \models s\sigma \approx t\sigma$ for all $\{s\sigma, t\sigma\} \subseteq norm(A)\sigma$ and grounding σ.

Proof. Follows from Lemma 3.3.1 and 3.3.2. □

Lemma 3.3.4 ($\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ is sound)**.** Let E be a finite set of non-ground equations and β a ground term. For any run of $\Rightarrow_{\mathrm{CC}(\mathcal{X})}$, any state Π in this run and for all terms s, t and grounding substitution σ such that there exists a class $A \in \Pi$ such that $\{\Gamma \parallel s, \Delta \parallel t\} \subseteq norm(A)$, $\Gamma\sigma$ and $\Delta\sigma$ satisfiable, it holds that $\mathrm{gnd}_{\preceq\beta}(E) \models s\sigma \approx t\sigma$.

Proof. We show this for all $\{\Gamma \parallel s, \Delta \parallel t\} \subseteq A$. Since constraints are always the same for each class in a run it follows for all $\{\Gamma \parallel s, \Delta \parallel t\} \subseteq norm(A)$ by Corollary 3.3.3. Proof by induction. Initially Π is such that $\{s, t \parallel s, t\} \subseteq \Pi$ for all $s \approx t \in E$. Now assume a grounding substitution σ such that $s\sigma \preceq \beta$ and $t\sigma \preceq \beta$. Then $(s \approx t)\sigma \in \mathrm{gnd}_{\preceq\beta}(E)$. The initial $\{f_i(x_1, \ldots, x_{k_i}) \preceq \beta \parallel f_i(x_1, \ldots, x_{k_i})\}$ are single term classes. So the assumption holds. Now assume that the assumption holds for Π and we apply a rule:
1) assume that Subsumption is applied. Then B subsumes A. Thus for all $A' \in gnd(A)$ there exists a $B' \in gnd(B)$ such that $A' \subseteq B'$. Thus we only remove redundant classes, so the assumption holds by i.h.
2) assume that *Merge* is applied. Then there exists $\Gamma \parallel s \in A, \Delta \parallel t \in B$ and $\mu = mgu(s, t)$. We show that $\mathrm{gnd}_{\preceq\beta}(E) \models s'\mu\sigma \approx t'\mu\sigma$ for all $\{\Gamma'\mu \parallel s'\mu, \Delta'\mu \parallel t'\mu\} \subseteq (A' \cup B')\mu$ and grounding σ such that $\Gamma'\mu\sigma$ and $\Delta'\mu\sigma$ satisfiable. Let $\sigma' = \mu\sigma$. By i.h. $\mathrm{gnd}_{\preceq\beta}(E) \models s'\sigma' \approx s\sigma'$ for $\Gamma'' \parallel s' \in norm(A)$, since $\Gamma'' \subseteq \Gamma'$ and $\Gamma \subseteq \Gamma'$, and analogously for $t'\sigma' \approx t\sigma'$. Now we have $s'\sigma' \approx s\mu\sigma = t\mu\sigma \approx t'\sigma'$. Thus by transitivity of equality it holds $\mathrm{gnd}_{\preceq\beta}(E) \models s'\mu\sigma \approx t'\mu\sigma$.
3) assume that *Deduction* is applied. By i.h. $\mathrm{gnd}_{\preceq\beta}(E) \models s'_i\sigma \approx t'_i\sigma$ for all grounding σ such that $\Gamma_i\sigma$ and $\Delta_i\sigma$ are satisfiable and $1 \leq i \leq n$. Thus, by congruence of equality, $\mathrm{gnd}_{\preceq\beta}(E) \models f(s'_1, ..., s'_n)\sigma \approx f(t'_1, ..., t'_n)\sigma$ for all σ such that $\Gamma'\sigma$ satisfiable, since $\Gamma' = \Gamma \wedge \Delta \wedge \Gamma_1 \wedge ... \wedge \Gamma_n \wedge \Delta_1 \wedge ... \wedge \Delta_n$. Thus $\mathrm{gnd}_{\preceq\beta}(E) \models f(s'_1, ..., s'_n)\mu\sigma \approx f(t'_1, ..., t'_n)\mu\sigma$ for all σ such that $\Gamma'\mu\sigma$ satisfiable, since $f(s'_1, ..., s'_n)\mu$ and $f(t'_1, ..., t'_n)\mu$ are instances of $f(s'_1, ..., s'_n)$ and $f(t'_1, ..., t'_n)$. □

Lemma 3.3.5 ($\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ is Terminating)**.** Let E be a finite set of non-ground equations and β a ground term. For any run of $\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ we reach a state, where no rule of $\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ is applicable anymore.

Proof. By Lemma 3.2.11 there are only finitely many possible β-constrained classes that are not subsumed. Since all terms are constraint by β in $\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ and *Merge* and *Deduction* check if the new class is subsumed by another class, they can be applied only finitely often. A class removed by *Subsumption* cannot be added again by *Merge* or *Deduction* since it is subsumed by another class in Π. Thus $\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ is terminating. □

Lemma 3.3.6 ($\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ is Complete)**.** Let E be a finite set of non-ground equations and β a ground term. Let Π be the result of a run of $\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ such that no rule of $\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ is applicable anymore. Then for all $\{s,t\} \subseteq T_{\preceq\beta}$ such that $\mathrm{gnd}_{\preceq\beta}(E) \models s \approx t$ there exists a class $A \in \Pi$, $\{\Gamma \parallel s', \Delta \parallel t'\} \subseteq norm(A)$ and a grounding substitution σ such that $s'\sigma = s$ and $t'\sigma = t$ and $\Gamma\sigma, \Delta\sigma$ satisfiable.

Proof. We show by induction for any sequence $E_1 \Rightarrow_{EQ} ... \Rightarrow_{EQ} E_n$ of applications of $\Rightarrow_{EQ}$ with $E_1 = \mathrm{gnd}_{\preceq\beta}(E)$ there exists a sequence $\Pi_0 \Rightarrow_{\mathrm{CC}(\mathcal{X})} ... \Rightarrow_{\mathrm{CC}(\mathcal{X})} \Pi_m$ of applications of $\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ rules such that for all $s \approx t \in E_n$, where $s \preceq \beta$ and $t \preceq \beta$, there exists a class $A \in \Pi_m, \Gamma \parallel l \in norm(A), \Delta \parallel r \in norm(A)$ and a substitution σ such that $l\sigma \approx r\sigma = s \approx t$, $\Gamma\sigma, \Delta\sigma$ satisfiable. Initially, this is true, since $\{s, t \parallel s, t\} \in \Pi_0$ for all $s \approx t \in E$. Since reasoning is based on ground terms only, we can ignore the *Instance* rule of $\Rightarrow_{EQ}$. Now assume we are in step i.
1) Reflexivity. $E_i \Rightarrow_{EQ} E_i \cup \{t \approx t\}$, $t = f(s_1, \ldots, s_n) \preceq \beta$. Then $t \in (\{f_i(x_{1_i}, \ldots, x_{k_i}) \preceq \beta \parallel f_i(x_{1_i}, \ldots, x_{k_i})\}\sigma)$ for $\sigma = \{x_{j_i} \mapsto s_j \mid 1 \leq j \leq n\}$.
2) Symmetry. $E_i \cup \{t \approx t'\} \Rightarrow_{EQ} E_i \cup \{t \approx t', t' \approx t\}$, $t, t' \preceq \beta$. Then by i.h. there exists a class $A \in \Pi$ such that $\{l\sigma, r\sigma\} \subseteq norm(A)\sigma$ and $(l \approx r)\sigma = t \approx t'$ for some substitution σ because $norm(A)$ is normal. But then also $(r \approx l)\sigma = t' \approx t$.
3) Transitivity. $E_i = E_i' \cup \{s \approx t \wedge t \approx s'\} \Rightarrow_{EQ} E_i \cup \{s \approx s'\} = E_{i+1}$, $s, s', t \preceq \beta$. By hypothesis there must exist $\{A, B\} \subseteq \Pi$, $\{\Gamma \parallel l, \Delta \parallel r\} \subseteq norm(A)$, $\{\Gamma' \parallel l', \Delta' \parallel r'\} \subseteq norm(B)$, a grounding substitution σ, such that $(l \approx r)\sigma = s \approx t$ and $(l' \approx r')\sigma = t \approx s'$ and $\Gamma\sigma, \Delta\sigma, \Gamma'\sigma, \Delta'\sigma$ all satisfiable. If $(A' \cup B')\mu$ is subsumed by some $C \in \Pi$, then we are already done, since there exists a $C' \in gnd(C)$ such that $\{s, s'\} \subseteq C'$. Otherwise *Merge* is applicable and $(A' \cup B')\mu$ added to the state. Then $\{\Gamma \wedge \Delta \wedge \Gamma' \parallel l, \Delta \wedge \Gamma' \parallel r, \Delta \wedge \Gamma' \parallel l', \Delta' \wedge \Delta \wedge \Gamma' \parallel r'\}\mu \subseteq (A' \cup B')\mu$ by the definition of *Merge*. Obviously, $(\Gamma \wedge \Delta \wedge \Gamma' \wedge \Delta')\mu$ is satisfiable since $(\Gamma \wedge \Delta \wedge \Gamma' \wedge \Delta')\sigma$ is satisfiable. Thus there exists a σ' such that $\{s, s'\} \subseteq (A' \cup B')\mu\sigma'$.
4) Congruence. $E_i = E_i' \cup \{s_1 \approx t_1, ..., s_m \approx t_m\} \Rightarrow_{EQ} E_i \cup \{f(s_1, ..., s_m) \approx f(t_1, ..., t_m)\} = E_{i+1}$, where $f(s_1, ..., s_m), f(t_1, ..., t_m) \preceq \beta$. By hypothesis there must exist $\{A_1, ..., A_m\} \subseteq \Pi$, $\{\Gamma_i \parallel l_i, \Delta_i \parallel r_i\} \subseteq norm(A_i)$ because $norm(A_i)$ is normal and a grounding substitution σ such that $(l_i \approx r_i)\sigma = s_i \approx t_i$ and $\Gamma_i\sigma, \Delta_i\sigma$ satisfiable. Now take two renamed copies of $\{f_i(x_{1_i}, \ldots, x_{k_i}) \preceq \beta \parallel f_i(x_{1_i}, \ldots, x_{k_i})\}$ and grounding substitutions σ_1, σ_2 such that $f(x_1', ..., x_m')\sigma_1 = f(s_1, ..., s_m)$ and $f(y_1', ..., y_m')\sigma_2 = f(t_1, ..., t_m)$ where the respective constraints are obviously satisfied. Thus there must exist a simultaneous most general unifier μ as defined in the *Deduction* rule. If $\{\Gamma'\mu \parallel f(l_1, ..., l_m)\mu, \Gamma'\mu \parallel f(r_1, ..., r_m)\mu\}$ is subsumed by some $C \in \Pi$, then we are already done, since there exists a $C' \in gnd(C)$ such that $\{f(s_1, ..., s_m), f(t_1, ..., t_m)\} \subseteq C'$. Otherwise *Deduction* is applicable and $\{\Gamma'\mu \parallel f(l_1, ..., l_m)\mu, \Gamma'\mu \parallel f(r_1, ..., r_m)\mu\}$ added to the state. Since $\Gamma'\sigma$ is satisfiable $\{f(s_1, ..., s_m), f(t_1, ..., t_m)\} \subseteq \{\Gamma'\mu \parallel f(l_1, ..., l_m)\mu, \Gamma'\mu \parallel f(r_1, ..., r_m)\mu\}\sigma$ has to hold. □

3.4 Implementation

I have implemented $\Rightarrow_{CC(\mathcal{X})}$ on the basis of the SPASS workbench infrastructure. I provide pseudo code of the implementation. The pseudo code is intended to give an idea of the implementation and not to present the concrete details of the implementation. For example, the use of path index and discrimination tree is missing.

I make use of the standard procedure to work through the classes, namely a *worked-off* and *usable* queue. Initially the *worked-off* queue contains all single-term classes and *usable* contains the initial classes that are created from the input equations. In each loop I select a class from the *usable* queue, perform all possible *Merge* and *Deduction* steps on it and add the class to the *worked-off* queue afterwards. Newly created classes are added to the usable queue. Algorithm 1 shows the initial state and main loop of my implementation. A CLASS has two sets $terms$ and $cstrs$, containing the terms and constraints. The number of separating variables is stored in the field $sepvars$. I assume that this value is updated automatically in the pseudo code. There is a first optimisation option here, namely which classes should be picked from the usable queue first. My heuristic selects the classes with the fewest terms and the most variables from the usable queue, or if the number of terms and variables are equal, then the class with the fewest separating variables. Various benchmarks have shown that this produces the best results.

Algorithm 2 sketches the implementation of the merge function. Here the implementation follows the rules, except that subsumption is checked directly after the new class is created and the code performs all possible merge operations of the input class. For the subsumption function, an order must be defined first, hence it will be given later.

Algorithm 1 Main function of the algorithm

```
function MAIN(E)
    for all s ≈ t ∈ E do
        C_new = new CLASS
        C_new→terms = {s, t}
        C_new→cstrs = {s, t}
        us = Push(us, C_new)
    end for
    for all f ∈ Ω, arity(f) = n do
        C_new = new CLASS
        C_new→terms = {f(x_1, ..., x_n)}
        C_new→cstrs = {f(x_1, ..., x_n)}
        wo = Push(wo, C)
    end for
    while us ≠ ∅ do
        C = Pop(us)
        if Merge(C) && Deduct(C) then
            wo = Push(wo, C)
        end if
    end while
end function
```

Algorithm 2 Merge function

```
function MERGE(C_0)
    for C_1 in wo do
        for each (t_0, t_1) ∈ {(t_0, t_1) | t_0 ∈ C_0→terms ∧ t_1 ∈ C_1→terms} do
            if unifiable(t_0,t_1) then
                μ = mgu(t_0, t_1)
                C_new = new CLASS
                C_new→terms = (C_0→terms ∪ C_1→terms)μ
                C_new→cstrs  = (C_0→cstrs ∪ C_1→cstrs)μ
                if SAT(C_new→cstrs) then
                    subsumed = FALSE
                    for C in wo ∪ us ∪ {C_0} do
                        if CheckSubsumption(C, C_new) then
                            subsumed = TRUE
                            break
                        end if
                    end for
                    if subsumed == FALSE then
                        for C in wo ∪ us ∪ {C_0} do
                            if CheckSubsumption(C_new, C) then
                                wo = wo \ {C}
                                us = us \ {C}
                                if C == C_0 then subsumed = TRUE
                                end if
                            end if
                        end for
                        us = us ∪ {C_new}
                        if subsumed == TRUE then
                            return FALSE
                        end if
                    end if
                end if
            end if
        end for
    end for
    return TRUE
end function
```

Although my implementation is not completely naive, it is not at the level of actual $\Rightarrow_{\mathrm{CC}}$ implementations as existing in state-of-the-art SMT solvers [dMB08, CGSS13, BBB+22] to which I will eventually compare the performance of my implementation in Section 3.5. In the following, I describe the assumptions and optimizations that were made for the implementation. I start with a simplification of the *Deduction* rule. In the implementation it suffices to use only the single term classes for A and B in the *Deduction* rule.

Lemma 3.4.1. Assume a state Π such that *Deduction* is applicable for some $\{A, B\} \subseteq \Pi$ and $\Gamma \parallel f(s_1, ..., s_n) \in A, \Delta \parallel f(t_1, ..., t_n) \in B$. Let μ be the resulting simultaneous mgu and $\{\Gamma_i \parallel s'_i, \Delta_i \parallel t'_i\} \subseteq norm(D_i)$ for all $1 \leq i \leq n$ such that $\{\Gamma' \parallel f(s'_1, ..., s'_n), f(t'_1, ..., t'_n)\}\mu$ is the resulting new class. Then *Deduction* is applicable for two variable disjoint copies of the single term class $\{f(x_1, ..., x_n) \parallel f(x_1, ..., x_n)\}$.

Proof. We build a unifier $\mu' = \{x_1 \to s'_1, ..., x_n \to s'_n, y_1 \to t'_1, ..., y_n \to t'_n\}$ for the single term class and the renamed single term class $\{f(y_1, ..., y_n) \parallel f(y_1, ..., y_n)\}$. Then $\Gamma'' = f(x_1, ..., x_n) \preceq \beta \wedge f(y_1, ..., y_n) \preceq \beta \wedge \Gamma_1 \wedge ... \wedge \Gamma_n \wedge \Delta_1 \wedge ... \wedge \Delta_n$. The resulting class is thus $\{\Gamma'' \parallel f(s'_1, ..., s'_n), f(t'_1, ..., t'_n)\}\mu'$. □

It is easy to see that the class created by the single term class is always more general than the other classes that can be created by *Deduction*. Algorithms 3 and 4 sketch the implementation of my deduction function. I assume that any function call creates a copy of the input to that function. Note that the constraints are always verified as soon as possible in the actual implementation. To keep the pseudocode simple, I only check the satisfiability of the constraints at the end. There are more optimizations possible here, e.g. we can delete constraints that do not share a variable with the used terms in the class. I also take care of this in the actual implementation.

Algorithm 3 Deduction function

```
function DEDUCT(C_0)
    wo = wo ∪ {C_0}
    for each f ∈ Ω with arity(f) = n and n > 0 do
        t_0 = f(x_1, ..., x_n)
        t_1 = f(y_1, ..., y_n)
        if DeductIntern(t_0, t_1, 0, arity(f), C_0, new CLASS, false) == FALSE
then
            wo = wo \ {C_0}
            return FALSE
        end if
    end for
end function
```

A naive implementation of Subsumption, Definition 3.2.8, by ground instantiation results in a practically intractable procedure. Therefore, I lift subsumption rule to the non-ground level.

Definition 3.4.2 (Subsumption by Matching)**.** Let A, B be classes. Let X be the separating variables of B and Y the free variables of B. B subsumes A by matching iff there exists a substitution $\sigma : X \to \mathcal{T}(\Omega, vars(A))$ that maps every

Algorithm 4 Internal Deduction function

```
function DeductIntern(t_0, t_1, i, n, C_0, C_new, used)
    if i ≠ n then
        for C ∈ wo do
            C_new→cstrs = C_new→cstrs ∪ C→cstrs
            for {s_0, s_1} ⊆ C→terms do
                t_0 = t_0[s_0]_i
                t_1 = t_1[s_1]_i
                if not DeductIntern(t_0, t_1, i + 1, n, C_0, C_new, (C == C_0 || used))
then
                    return FALSE
                end if
            end for
            C_new→cstrs = C_new→cstrs \ C→cstrs
        end for
    else if used then
        C_new→terms = {t_0, t_1}
        C_new→cstrs = C_new→cstrs ∪ {t_0, t_1}
        if SAT(C_new→cstrs) then
            for C in wo ∪ us do
                if CheckSubsumption(C, C_new) then
                    return TRUE
                end if
            end for
            subsumed = FALSE
            for C in wo ∪ us do
                if CheckSubsumption(C_new, C) then
                    wo = wo \ {C}
                    us = us \ {C}
                    if C == C_0 then subsumed = TRUE
                    end if
                end if
            end for
            us = us ∪ {C_new}
            if subsumed == TRUE then
                return FALSE
            end if
        end if
    end if
    return TRUE
end function
```

variable in X, such that for every $\Gamma \parallel t \in A$ there is a $\tau : Y \to \mathcal{T}(\Omega, vars(A))$ and $(\Delta \parallel s)\sigma \in B\sigma$ such that $t = s\sigma\tau$ and $\forall\delta.(\Gamma\delta \to \exists\delta'.\Delta\sigma\tau\delta\delta')$.

Lemma 3.4.3. Let A, B be classes. If B subsumes A by matching then B subsumes A' by matching for all $A' \in gnd(A)$.

Proof. Assume B subsumes A by matching. By assumption there exists a substitution σ such that for all $\Gamma \parallel t \in A$ there exists a $(\Delta \parallel s)\sigma \in B\sigma$ and τ such that $t = s\sigma\tau$ and $\forall\delta.(\Gamma\delta \to \exists\delta'.\Delta\sigma\tau\delta\delta')$. σ matches all separating variables of B to terms containing only separating variables of A. If not, then every $t \in A$ contains a free variable in $cdom(\sigma)$. But then these are separating variables. Contradiction.

Let $A' \in gnd'(A)$ and $\mu : X \to \mathcal{T}(\Omega)$ be the substitution such that $A\mu = A'$, where X are the separating variables of A. Now construct substitution $\sigma' = \sigma\mu$. Then for all $(\Gamma \parallel t)\mu \in A'$ there exists a $(\Delta \parallel s)\sigma' \in B\sigma'$ and $\tau' = \tau\mu$ such that $t\mu = s\sigma'\tau'$ and $\forall\delta.(\Gamma\mu\delta \to \exists\delta'.\Delta\sigma'\tau'\delta\delta')$, since $s\sigma'\tau' = s\sigma\mu\tau\mu = s\sigma\tau\mu$.

Now let $A'' \in gnd(A')$. We have $gnd(A') = \{A''\}$. Then there exist grounding substitutions $\sigma_1, ..., \sigma_n$ such that $A'' = A'\sigma_1 \cup ... \cup A'\sigma_n$. Now, construct $\sigma'' = \sigma'$. Then for all $(\Gamma \parallel t)\mu\sigma_i \in A''$ there exists a $(\Delta \parallel s)\sigma'' \in B\sigma''$ and $\tau'' = \tau'\sigma_i$ such that $t\mu\sigma_i = s\sigma''\tau''$ and $\forall\delta.(\Gamma\mu\sigma_i\delta \to \exists\delta'.\Delta\sigma''\tau''\delta\delta')$, since $s\sigma''\tau'' = s\sigma\mu\tau\mu\sigma_i = s\sigma\tau\mu\sigma_i$.

□

Lemma 3.4.4. Let A, B be classes. If B subsumes A by matching then B subsumes A.

Proof. Assume that B does not subsume A. Then there exists an $A' \in gnd'(A)$ and an $A'' \in gnd(A')$ such that there exists no $B' \in gnd(B)$ such that $A'' \subseteq B'$. Now assume that there exists a σ such that for any $(\Gamma \parallel t)\tau' \in A'', \Gamma \parallel t \in A'$ there is a $(\Delta \parallel s)\sigma \in B\sigma$ and τ with $s\sigma\tau = t\tau'$ and $\forall\delta.(\Gamma\tau'\delta \to \exists\delta'.\Delta\sigma\tau\delta\delta')$. So there exist $\tau_1, ..., \tau_n$ such that $A'' \subseteq B\sigma\tau_1 \cup ... \cup B\sigma\tau_n$. Obviously, the free variables of B are also free variables of $B\sigma$. $B\sigma$ has no separating variables, otherwise $B\sigma\tau_i$ would not be ground for all $1 \leq i \leq n$. Thus $gnd(B\sigma) = \{B''\}$ and $B\sigma\tau_1 \cup ... \cup B\sigma\tau_n \subseteq B''$. Thus, $A'' \subseteq B''$, contradicting assumption. Thus, by Lemma 3.4.3 B cannot subsume A by matching.

□

Note that Subsumption by matching does also not guarantee less or equal classes than ground congruence closure. It allows for even more examples where this is not the case. For example, consider symbols f, g, a, b, β such that only $a, b, f(a), f(b), g(a), g(b) \preceq \beta$ and classes $A = \{f(x), g(a), g(b) \parallel f(x), g(a), g(b)\}$ and $B = \{f(a), f(b), g(x) \parallel f(a), f(b), g(x)\}$. Then neither A subsumes B nor B subsumes A by subsumption by matching, although $gnd(A) = gnd(B)$. However, in Section 3.5 I show that this rarely happens in practice.

It remains to find an appropriate solver for the constraints. Performance is highly dependent on the underlying order that was chosen for the algorithm. In my implementation I decided for a simple ordering that counts the number of symbols of a term.

Definition 3.4.5 (Symbol Count Order)**.** Let s, t be two terms. The Symbol Count Order $\preceq$ is defined as $s \preceq t$ if $\text{size}(s) \leq \text{size}(t)$.

For example, $f(x, g(a)) \preceq g(h(a), h(b))$ or $x \preceq a$. From now on I consider $\preceq$ to be the above symbol counting order. Recall that the signature is finite, so the symbol counting order is well-founded. There always exist only finitely many smaller or syntactically equal ground terms for a given maximum term t and a finite signature. Solving $\preceq$-constraints is equivalent to solving linear integer arithmetic constraints. Note that a constraint Γ is satisfiable iff it is satisfiable without applying any substitution, because variables and constants are the smallest terms. With respect to the implication $\forall\delta.(\Gamma\delta \to \exists\delta'.\Delta\sigma\tau\delta\delta')$, see Definition 3.4.2, the quantifier alternation can be removed, so $\forall\delta.(\Gamma\delta \to \exists\delta'.\Delta\sigma\tau\delta\delta')$ holds iff $\forall\delta.(\Gamma\delta \to \Delta\sigma\tau\delta)$ holds.

Definition 3.4.6 (LIA Constraint Abstraction)**.** Let $t \preceq \beta$ be a constraint. Let $vars(t) = \{x_1, ..., x_n\}$. Then $\text{lic}(t \preceq \beta) = x_1 \geq 1 \wedge ... \wedge x_n \geq 1 \wedge \#(x_1, t) * x_1 + ... + \#(x_n, t) * x_n \leq \text{size}(\beta) - (\text{size}(t) - \sum_{1 \leq i \leq n} \#(x_i, t))$ is the linear arithmetic constraint of $t \preceq \beta$.

Lemma 3.4.7 (Correctness LIA Constraint Abstraction)**.** Let $t \preceq \beta$ be a constraint, $vars(t) = \{x_1, ..., x_n\}$.

1. For any ground substitution $\sigma = \{x_i \mapsto s_i \mid 1 \leq i \leq n\}$: if $t\sigma \preceq \beta$ is true then $\text{lic}(t \preceq \beta)\{x_i \mapsto \text{size}(s_i) \mid 1 \leq i \leq n\}$ is true.

2. For any substitution $\sigma = \{x_i \mapsto k_i \mid 1 \leq i \leq n, k_i \in \mathbb{N}\}$: if $\text{lic}(t \preceq \beta)\sigma$ is true then $t\delta \preceq \beta$ is true for all $\delta = \{x_i \mapsto s_i \mid 1 \leq i \leq n\}$ where all s_i are ground, and $\text{size}(s_i) = k_i$.

Proof. by applying the definitions □

Lemma 3.4.8. Let β be a ground term and $\Gamma_1 = \{s_1 \preceq \beta \wedge ... \wedge s_n \preceq \beta\}$ and $\Gamma_2 = \{t_1 \preceq \beta \wedge ... \wedge t_m \preceq \beta\}$ be two constraints. Then $\forall\sigma.(\Gamma_1\sigma \to \exists\sigma'.\Gamma_2\sigma\sigma')$ iff $\text{lic}(s_1 \preceq \beta) \wedge ... \wedge \text{lic}(s_n \preceq \beta) \to \text{lic}(t_1 \preceq \beta) \wedge ... \wedge \text{lic}(t_m \preceq \beta)$.

Proof. Follows from Lemma 3.4.7 and the above observation that the quantifier alternation can be removed. □

Thus checking if a $\preceq$-constraint Γ models a $\preceq$-constraint Γ' reduces to a linear integer arithmetic implication test. I make use of the linear arithmetic solver implemented in SPASS-SATT [BFSW19]. Algorithm 5 shows my implementation of subsumption. The function $BuildLAC(constraints)$ creates a linear arithmetic constraint as defined in 3.4.6 and $LAImplicationTest(LAC1, LAC0)$ is the implemenation of the implication test of the linear arithmetic solver. The implementation first checks if there is a matcher for the separating variables. Then it checks if there are matchers for the free variables under the assumption of the matcher for the separating variables.

Before creating a new class I rename all involved classes and then apply *Merge* or *Deduct*. Especially after application of *Deduct* the new class may contain variables in a constraint that do not occur in any of the class terms. Subsequent merges then continuously increases the number of constraints and variables. Fortunately, only one extra variable is needed.

Lemma 3.4.9. Let $A = \{\Gamma \parallel s_1, ..., s_n\}$ be a class. Let $Y = vars(\Gamma) \setminus (vars(s_1) \cup ... \cup vars(s_n))$ be the variables occurring in Γ but not in $s_1, ..., s_n$. Let $\sigma : Y \to \{y'\}$ for some fresh variable $y' \in \mathcal{X}$. Then A subsumes $A\sigma$ and $A\sigma$ subsumes A.

Algorithm 5 Subsumption function

```
function CheckSubsumption(C_0, C_1)
    if C_0→sepvars == 0 then
        return CheckSubsumptionFreeVars(C_0, C_1, {})
    else
        for each (t_0, t_1) ∈ {(t_0, t_1) | t_0 ∈ C_0→terms ∧ t_1 ∈ C_1→terms} do
            if exists σ s.t. t_0σ = t_1 then
                σ = σ \ {x → t | x → t ∈ σ and x a free variable}
                if CheckSubsumptionFreeVars(C_0, C_1, σ) then
                    return TRUE
                end if
            end if
        end for
    end if
    return FALSE
end function
function CheckSubsumptionFreeVars(C_0, C_1, σ)
    for each t_1 ∈ C_1→terms do
        result = FALSE
        for each t_0 ∈ C_0→terms do
            if exists δ s.t. t_0σδ = t_1 then
                LAC_0 = BuildLAC({tσδ | t ∈ C_0→cstrs})
                LAC_1 = BuildLAC(C_1→cstrs)
                if LAImplicationTest(LAC_1, LAC_0) then
                    result = TRUE
                    break
                end if
            end if
        end for
        if not result then
            return FALSE
        end if
    end for
    return TRUE
end function
```

Proof. A constraint is satisfiable with respect to the symbol counting order, if it is satisfiable by substituting a constant for all variables. Concerning the semantics of classes, variables only occurring in the constraint do not play a role as long as the constraint is satisfied. Thus $\mathrm{gnd}(A) = \mathrm{gnd}(A\sigma)$. □

I can further reduce the number of constraints by removing terms that have the same variable occurrences and are smaller or equal to some other term in the constraint according to $\preceq$, e.g. $f(x)$ can be removed if $f(g(x))$ already exists.

To keep the number of constrained terms within classes small, I also need a new *Condensation* rule:

Condensation $\Pi \cup \{\{\Gamma_1 \parallel s_1, ..., \Gamma_n \parallel s_n\}\} \Rightarrow_{\mathrm{CC}(\mathcal{X})} \Pi \cup \{\{\Gamma_1 \parallel s_1, ..., \Gamma_{j-1} \parallel s_{j-1}, \Gamma_{j+1} \parallel s_{j+1}, ..., \Gamma_n \parallel s_n\}\delta\}$
provided there exists indices i, j and a matcher δ such that $s_i\delta = s_j$, $\{\Gamma_1 \parallel s_1, ..., \Gamma_{j-1} \parallel s_{j-1}, \Gamma_{j+1} \parallel s_{j+1}, ..., \Gamma_n \parallel s_n\}\delta$ subsumes $\{\Gamma_1 \parallel s_1, ..., \Gamma_n \parallel s_n\}$.

For example, the *Condensation* rule would reduce the class $\{f(x), f(y), f(z) \parallel f(x), f(y), f(z)\}$ to $\{f(x), f(y) \parallel f(x), f(y)\}$. *Condensation* together with subsumption by matching ensures termination, because the number of separating variables is bounded. *Condensation* is an example where I could improve the performance of my implementation. It can be implemented without additional memory consumption, however, my current implementation copies the class, modifies it and then checks subsumption.

To keep the number of full subsumption checks to a minimum, I have also implemented fast pre-filtering techniques. It turns out that it is more efficient to first check whether for each term in the instance class there is a term in the general class that matches this term.

Corollary 3.4.10. Let A, B be two classes. If there exists a $\Gamma \parallel t \in A$ such that there exists no $\Delta \parallel s \in B$ and no matcher δ such that $s\delta = t$, then A is not subsumed by B.

I use bit vectors to track the number of occurrences of top symbols of terms within a class to check the above filter. For each symbol I store in one bit whether there are 0, 1 or more terms in the class that contain this symbol as a top symbol, where $[0]_{10} = [0]_2, [1]_{10} = [1]_2$. For two bit vectors $\mathcal{V}_0$ of a general and $\mathcal{V}_1$ of an instance class I compute $\mathrm{NOT}(\mathcal{V}_0)$ AND $\mathcal{V}_1$, where NOT and AND are bitwise operators. Note that classes containing a variable term must be excluded from this check.

Finally I store if merges or subsumption checks on classes have already been applied. To find candidates for the subsumption rule I maintain an index for each term in which classes it occurs. General terms are retrieved by a discrimination tree index, unifiables and instances of a term by a path index [McC92].

3.5 Evaluation

I evaluated my algorithm on all unit equality (UEQ) problems from TPTP-v8.2.0 [Sut17]. From each problem I created two benchmark problems: one with all inequations removed and the other by turning inequations into equations. I choose a fixed nesting depth of 6 and 8 in order to construct β. It is constructed by nesting all function symbols in one argument up to the chosen

Table 3.1: UEQ problem results for a nesting depth of 6

	Time (sec)		#Classes	
Problem	CC($\mathcal{X}$)	CC	CC($\mathcal{X}$)	CC
GRP478-1	762	1392	8903	197281
N-BOO057-10	0	timeout	6	timeout
ROB033-1	761	132	891	6383
N-GRP119-1	timeout	2	timeout	6908

depth and filling all other arguments with a constant. E.g., with function symbols $a/0$,$f/1$,$g/2$,$h/1$,$i/2$ and nesting depth 4 I create the term $\beta = i(h(g(f(a), a)), a)$. Then the size of all terms is limited to 7 symbols. I compared the performance of my algorithm to the performance of a CC implementation based on the implementation in the veriT solver [BCBdODF09]. The resulting benchmark problems turn out to be challenging for both algorithms.

CC($\mathcal{X}$) provides a solution to the entire ground input space smaller β. Therefore, for the comparison of the two algorithms, I feed all ground terms smaller β into CC. CC($\mathcal{X}$) also provides a solution to the entire ground input space. I skip all examples where no equation has ground instances $\preceq \beta$.

Experiments are performed on a Debian Linux server running AMD EPYC 7702 64-core CPUs with 3.35GHz and a total memory of 2TB. The time limit for each test is 30 minutes. The results of all runs as well as all input files and binaries can be found at https://nextcloud.mpi-klsb.mpg.de/index.php/s/RjcHAQYR97H6ZMy.

For a nesting depth of 6, CC($\mathcal{X}$) terminates on 519 and CC on 457 of the 2900 problems. CC($\mathcal{X}$) is faster on 474 problems and CC on 172. CC($\mathcal{X}$) terminated on 189 examples where CC timed out, and CC terminated on 127 examples where CC($\mathcal{X}$) timed out. Table 3.1 shows some examples of the results (with added N- for examples where the inequations are removed).

Figure 3.1 shows the results of all test cases for the runtime and figure 3.2 shows the results of all terminating examples for the number of classes. The performance of CC($\mathcal{X}$) currently drops if there are many different variables. This is mainly due to the current implementation of the redundancy checks. Concerning the number of classes, the number of classes generated by CC($\mathcal{X}$) is significantly smaller than the number in CC for almost all examples. Examples where this does not hold are border cases, i.e., they only contain few equations or contain only one constant.

For a nesting depth of 8, 299 of 2900 problems terminate in CC($\mathcal{X}$) and 102 in CC. CC($\mathcal{X}$) is faster in 294 terminating examples and CC in 24. CC($\mathcal{X}$) terminated on 216 examples where CC timed out and CC terminated on 19 examples where CC($\mathcal{X}$) timed out. Table 3.2 shows again some examples of the results (with added N- for examples where the inequations are removed).

Figure 3.3 shows the results of all test cases for the runtime and figure 3.4 shows the results of all terminating examples for the number of classes. CC($\mathcal{X}$) is particularly advantageous with a large β, where grounding is no longer feasible. The number of classes are again significantly smaller than in CC.

Table 3.3 shows the average and median time and number of classes of CC($\mathcal{X}$)

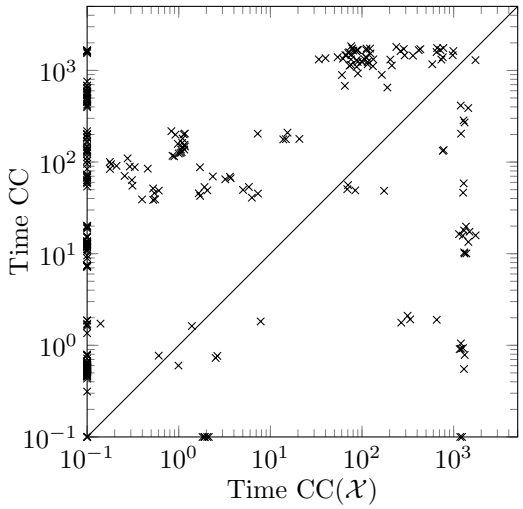

Figure 3.1: Comparison of the runtime of CC and CC($\mathcal{X}$) for a nesting depth of 6. Dots below the line indicate test cases where CC performs better (i.e. has less classes or took less time), and above indicate test cases where CC($\mathcal{X}$) performs better.

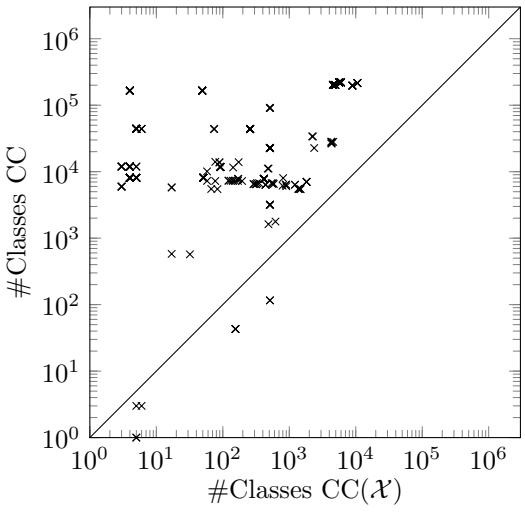

Figure 3.2: Comparison of the number of classes of CC and CC($\mathcal{X}$) for a nesting depth of 6.

Table 3.2: UEQ problem results for a nesting depth of 8

	Time (sec)		#Classes	
Problem	CC($\mathcal{X}$)	CC	CC($\mathcal{X}$)	CC
GRP404-1	4	1388	1273	164857
N-COL012-1	20	538	3077	229794
GRP420-1	28	timeout	8242	timeout
LAT080-1	timeout	248	timeout	18080

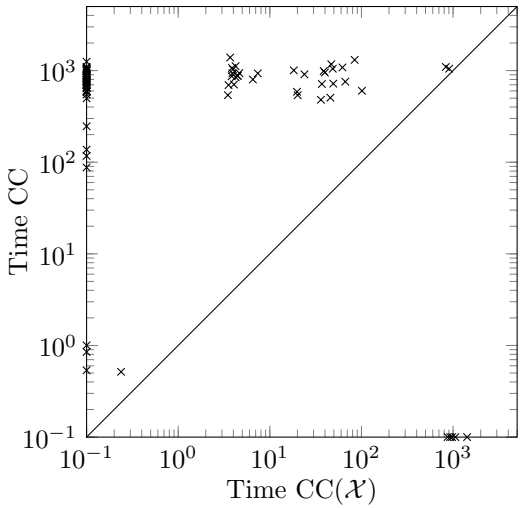

Figure 3.3: Comparison of the runtime of CC and CC($\mathcal{X}$) for a nesting depth of 8.

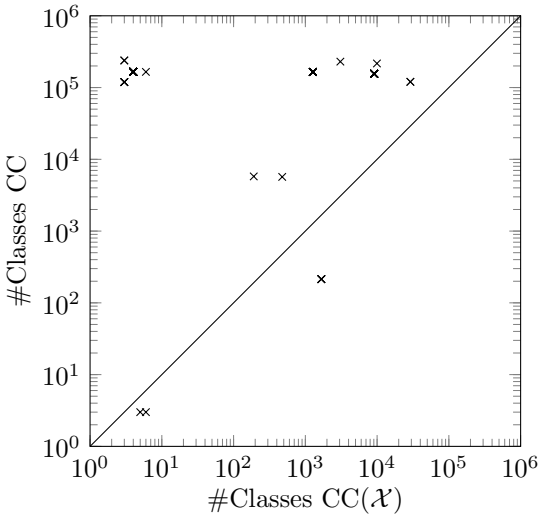

Figure 3.4: Comparison of the number of classes of CC and CC($\mathcal{X}$) for a nesting depth of 8.

and CC for nesting depth 6 and 8. One can see that CC($\mathcal{X}$) is significantly faster on average and produces only a fraction of the classes of the ground CC. The difference is even stronger when looking at the median. Here CC($\mathcal{X}$) only needs 1 second or less for half of all terminating examples whereas CC needs more than ten minutes.

3.6 KBO Constraint Solving

For the implementation of $SCL(EQ)$ I need a constraint solver for KBO Constraints. First steps towards this where made with joint work with Yasmine Briefs [BLW23]. The calculus and implementation provided below is work by Yasmine Briefs whereas the evaluation in a very early version of our congruence

Table 3.3: Average and Median of the test cases.

	Average			
	Time		#Classes	
Depth	CC($\mathcal{X}$)	CC	CC($\mathcal{X}$)	CC
6	201.2	394.6	2187	44248
8	171.5	659.0	2071	125407
	Median			
	Time		#Classes	
Depth	CC($\mathcal{X}$)	CC	CC($\mathcal{X}$)	CC
6	0.5	58.6	170	8126
8	0.8	736.8	20	164857

closure algorithm is done by me. Christoph Weidenbach was involved in the exchange of ideas and the final polishing of the paper.

Definition 3.6.1 ([BLW23]). A KBO constraint Γ is a finite set of atoms $t\#s$ where $t, s \in T(\Sigma, \mathcal{X})$ and $\# \in \{<, >, \neq, \leq, \geq, =\}$. We say that $\Gamma = \{t_1\#_1 s_1, \ldots, t_n\#_n s_n\}$ is satisfiable if there exists a substitution σ that is grounding for all t_j, s_j such that

$$\bigwedge_{j=1}^{n} t_j\sigma \ \#_j \ s_j\sigma.$$

Such a grounding substitution σ is called a solution.

Definition 3.6.2 ([BLW23]). A right-ground KBO constraint Γ is a KBO constraint where $s_1, \ldots, s_n \in T(\Sigma)$, i.e., only the t_j may contain variables.

Definition 3.6.3 ([BLW23]). A simple right-ground KBO constraint Γ is a right-ground KBO constraint where $\# \in \{<, \neq\}$.

For simple right-ground KBO constraints, we prefer more explicit notation: We now assume $t_1, \ldots, t_n, l_1, \ldots, l_m \in T(\Sigma, \mathcal{X})$, $s_1, \ldots, s_n, r_1, \ldots, r_m \in T(\Sigma)$ and call Γ satisfiable if there exists a substitution σ that is grounding for all t_j, l_j such that

$$\left(\bigwedge_{j=1}^{n} t_j\sigma < s_j\right) \wedge \left(\bigwedge_{j=1}^{m} l_j\sigma \neq r_j\right).$$

Proposition 3.6.4 ([BLW23]). Checking satisfiability for simple right-ground KBO constraints is NP-hard.

Proposition 3.6.5 ([BLW23]). Checking satisfiability for simple right-ground KBO constraints is in NP.

Next we propose an algorithm for testing satisfiability of simple right-ground KBO constraints. Let Γ be a simple right-ground KBO constraint with n inequations $t_j < s_j$ and m inequalities $l_j \neq r_j$.

Assume that $\mathit{vars}(\{t_j \mid 1 \leq j \leq n\} \cup \{l_j \mid 1 \leq j \leq m\}) = \{x_1, \ldots, x_k\}$. The proof of 3.6.5 shows that we only have to consider the $m + 1$ smallest terms for the grounding, so to begin, we generate an ordered list S of the $m + 1$ smallest terms. This way, a grounding substitution σ corresponds to a vector $\vec{v} \in \mathbb{N}^k$ where $v_i < m+1$ is the index of the term $\sigma(x_i)$ in S, i.e., $S[v_i] = \sigma(x_i)$. Let $\sigma(\vec{v})$ with $\sigma(\vec{v})(x_i) := S[v_i]$ denote the grounding corresponding to the vector $\vec{v}$. Later on, we give a dynamic programming algorithm to compute the k smallest terms for some number k. Actually, we do not directly generate the $m + 1$ smallest terms, but start with a constant number of terms and generate more terms as needed.

The algorithm is given by three inference rules that are represented by an abstract rewrite system. They operate on a state which is either $\bot$ or a four-tuple $(T; \vec{v}; F; \Gamma)$ where T is a sequence of variables, the *trace*; $\vec{v} \in \mathbb{N}^k$ is a grounding substitution in vector notation, the *current grounding*; F is a set of *forbidden* groundings; and Γ is a *simple right-ground KBO constraint*. The initial state

for a constraint Γ is $(\varepsilon; (0, \ldots, 0); \emptyset; \Gamma)$, i.e., the *trace* is empty, every variable is mapped to the smallest constant and there are no *forbidden* groundings.

We use the following partial ordering $\leq_F$ on groundings: $\vec{v} \leq_F \vec{u}$ iff for all $i \in \{1, \ldots, k\}$ we have $v_i \leq u_i$. By $inc(\vec{v}, i)$ we denote the grounding $\vec{v}'$ with $v'_i = v_i + 1$ and $v'_l = v_l$ for all $l \in \{1, \ldots, k\}$ with $l \neq i$, i.e., the grounding where we increase the term for the variable x_i by one. Analogously, we define $dec(\vec{v}, i)$, where we instead decrease the term for the variable x_i by one, i.e., $v'_i = v_i - 1$. The two operations *inc* and *dec* are only used when they are well-defined, i.e., they yield a grounding $\vec{v} \in \mathbb{N}^k$ where $v_i < m + 1$. The operation *inc* is only used when an inequality $l_j \neq r_j$ is not satisfied, and this can happen at most m times without intermediate Backtrack steps. The operation $dec(\vec{v}, i)$ is only used for Backtrack, and in this case $v_i > 0$.

The role of F is that we want to keep the algorithm from considering wrong groundings again. For all $\vec{u} \in F$, we do not visit states with grounding $\vec{v}$ if $\vec{v} \geq_F \vec{u}$. When we Backtrack, we insert the current grounding into F. The trace T records the last updated variables so Backtrack is able to undo the last Increase operation. As will be proven in 3.6.9, the algorithm terminates in $\perp$ iff there exists no solution, and if there exists a solution, then it terminates in a state where the current grounding $\vec{v}$ is a solution.

Increase $\quad (T; \vec{v}; F; \Gamma) \quad \Rightarrow \quad (Tx_i; \vec{v}'; F; \Gamma)$

provided $\vec{v}' = inc(\vec{v}, i)$, $l_j\sigma(\vec{v}) = r_j$ for some $l_j \neq r_j \in \Gamma$, $l_j\sigma(\vec{v}') \neq r_j$ and there is no $\vec{u} \in F$ with $\vec{v}' \geq_F \vec{u}$

Backtrack $\quad (Tx_i; \vec{v}; F; \Gamma) \quad \Rightarrow \quad (T; \vec{v}'; F \cup \{\vec{v}\}; \Gamma)$

provided $\vec{v}' = dec(\vec{v}, i)$ and either

1. $l_j\sigma(\vec{v}) = r_j$ for some $l_j \neq r_j \in \Gamma$, but for all $l \in \{1, \ldots, k\}$, we have that $l_j\sigma(inc(\vec{v}, l)) \neq r_j$ implies that there is a $\vec{u} \in F$ with $inc(\vec{v}, l) \geq_F \vec{u}$, or
2. $t_j\sigma(\vec{v}) \geq s_j$ for some $t_j < s_j \in \Gamma$

Fail $\quad (\varepsilon; \vec{v}; F; \Gamma) \quad \Rightarrow \quad \perp$

provided either

1. $l_j\sigma(\vec{v}) = r_j$ for some $l_j \neq r_j \in \Gamma$, but for all $l \in \{1, \ldots, k\}$, we have that $l_j\sigma(inc(\vec{v}, l)) \neq r_j$ implies that there is a $\vec{u} \in F$ with $inc(\vec{v}, l) \geq_F \vec{u}$, or
2. $t_j\sigma(\vec{v}) \geq s_j$ for some $t_j < s_j \in \Gamma$

Informally, Increase is applicable if some inequality $l_j\sigma(\vec{x}) \neq r_j$ is not fulfilled and we can fix this with the new grounding $inc(\vec{v}, i)$ which is not forbidden by F. Backtrack undoes an operation and is applicable if either some inequality $l_j\sigma(\vec{v}) \neq r_j$ is not fulfilled, but Increase is not applicable, or if some inequation $t_i\sigma(\vec{v}) < s_i$ is not fulfilled. Fail is applicable if Backtrack would be applicable on an empty *trace*, i.e., there is no operation to undo.

Obviously, there is no state on which we can apply both Backtrack and Fail.

Definition 3.6.6 ([BLW23])**.** A reasonable strategy is a strategy that prefers Backtrack and Fail over Increase.

Example 3.6.7 ([BLW23])**.** Consider a signature with constants a, b, c and a binary function f. We set $w(a) = 1; w(b) = w(c) = 2; w(f) = 3$ and $a \prec b \prec c \prec f$. We consider the constraint

$$\Gamma = \{x_1 \neq a, f(x_1, x_2) < f(a, c)\}.$$

The $m+1$ smallest terms, where $m = 1$, are a, b. This is the unique execution of the algorithm. In order to increase readability, for $\vec{v}$, we write the terms instead of the indices.

$$\begin{array}{ll} & (\varepsilon; (a,a); \emptyset; \Gamma) \\ \Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_1; (b,a); \emptyset; \Gamma) \\ \Rightarrow^{\text{Backtrack}}_{\text{KCS}} & (\varepsilon; (a,a); \{(b,a)\}; \Gamma) \\ \Rightarrow^{\text{Fail}}_{\text{KCS}} & \bot \end{array}$$

The algorithm terminates in $\bot$, so there is no solution.

Example 3.6.8 ([BLW23])**.** Consider a signature with constants a, b, a binary function g and a ternary function f. Let $w(a) = 1, w(b) = w(f) = w(g) = 2$ and $a \prec b \prec g \prec f$. The constraint is

$$\Gamma = \{x_1 < b, g(x_2, a) < g(b, b), f(x_1, x_2, x_3) \neq f(a, a, a), g(x_1, x_2) \neq g(a, b)\}.$$

The $m + 1$ smallest terms, where $m = 2$, are $a, b, g(a, a)$.

$$\begin{array}{ll} & (\varepsilon; (a,a,a); \emptyset; \Gamma) \\ \Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_1; (b,a,a); \emptyset; \Gamma) \\ \Rightarrow^{\text{Backtrack}}_{\text{KCS}} & (\varepsilon; (a,a,a); \{(b,a,a)\}; \Gamma) \\ \Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_2; (a,b,a); \{(b,a,a)\}; \Gamma) \\ \Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_2x_2; (a, g(a,a), a); \{(b,a,a)\}; \Gamma) \\ \Rightarrow^{\text{Backtrack}}_{\text{KCS}} & (x_2; (a,b,a); \{(b,a,a), (a,g(a,a),a)\}; \Gamma) \\ \Rightarrow^{\text{Backtrack}}_{\text{KCS}} & (\varepsilon; (a,a,a); \{(b,a,a), (a,g(a,a),a), (a,b,a)\}; \Gamma) \\ \Rightarrow^{\text{Increase}}_{\text{KCS}} & (x_3; (a,a,b); \{(b,a,a), (a,g(a,a),a), (a,b,a)\}; \Gamma) \end{array}$$

The algorithm has found a solution, so no rule is applicable and it terminates. Note that after the third and fifth operation, we cannot increase x_1 because $(b, b, a) \geq_F (b, a, a) \in F$.

Theorem 3.6.9 ([BLW23])**.** The algorithm is correct: If there exists a solution, then starting from $(\varepsilon; (0, \ldots, 0); \emptyset; \Gamma)$, the algorithm terminates in a state $(T; \vec{v}; F; \Gamma)$ where $\vec{v}$ is a solution. If there is no solution, the algorithm terminates in $\bot$.

We have implemented the above algorithm in the context of the SPASS reasoning workbench. The efficiency of the algorithm depends on the respective variables we choose for Increase. If there exists a solution, then there exists an execution using only the rule Increase. The following criteria might be useful to select the best variable for Increase:

– We prefer variables that do not occur in "critical" inequations, or in a minimal number of inequations. A "critical" inequation is one where the weight difference is 0 or close to 0.

– We prefer variables x_i for which the next term is not restricted by any inequality $l_j \neq r_j$.

– We prefer variables x_i for which the next term does not have a larger weight, or for which the increase in weight is minimal.

– We prefer variables that fix multiple inequalities $l_j \neq r_j$ instead of just one.

It is possible to calculate and maintain some score for every variable here and decide based on this score. The exact selection criteria still need to be further explored.

A remaining problem from the presentation of the algorithm is how to compute the k smallest terms. If the occurring weights are rather small, the following dynamic programming algorithm might be useful in practice. The idea is to compute all terms of a specific weight for increasing weights until we generated at least k terms. Unfortunately, there may be exponentially many terms of a specific weight where the exponent is the maximal arity of a function and the base is the number of terms of smaller weights. However, k is bounded above by the number of inequalities m, the number of terms with smaller weights is bounded above by k and the maximal arity is probably small, so it is to be expected that this is not a big problem.

As it is probably hard to find the next possible weight, we simply always increase the weight by 1 starting by the weight of the smallest constant. Our DP array is two-dimensional, one dimension having the weight and the other dimension having the size of the tuple from 1 to *max_arity*. Actually, it is four-dimensional since every entry is a list of tuples of terms and every tuple is a list of its entries. A tuple of size 1 is just a term of the specific weight. The tuples of larger size are needed for the DP transitions where they serve as argument tuples for the functions. We maintain an array *smallest_terms* that will in the end contain the at least k smallest terms.

We iterate over the weights starting at the weight of the minimal constant. Let *curweight* denote the current weight. The idea is to compute all terms of weight *curweight*, sort them, add them to *smallest_terms*, and proceed with weight $curweight + 1$ if $|smallest\_terms|$ is still smaller than k. To do so, if *curweight* is not the smallest weight, we first compute the tuples of size 2 to *max_arity* for the previous weight. This is done via DP: For tuple size i we iterate over the terms $s \in smallest\_terms$. Then we iterate over the tuples t of size $i-1$ and weight $curweight - 1 - w(s)$ using the DP array and add (s,t) to the current DP entry. Afterwards, we calculate all terms of weight *curweight* by iterating over all symbols f and and all tuples t of size $\text{arity}(f)$ and weight $curweight - w(f)$ using the DP array. Then, the term $f(t)$ has weight *curweight*.

We finish this Section by a discussion of potential heuristics, sufficient conditions for a simple right-ground KBO constraint to have a solution. Every inequality $l_j \neq r_j$ rules out any assignment that satisfies τ_j, the matcher from l_j to r_j. Now assume we have m inequalities and know that there are more than m solutions for the inequation $t < s$, then one might think that there is a grounding that solves all inequalities $l_j \neq r_j$ and the inequation $t < s$. However, this is not true.

Example 3.6.10 ([BLW23])**.** Consider a signature with constants a, b and c and a binary function f. The weights are $w(a) = 1; w(b) = w(c) = 2; w(f) = 3$

and we use $a \prec b \prec c \prec f$ as a precedence. Now consider the constraint

$$\Gamma = \{x \neq a, f(x, y) < f(a, c)\}.$$

The inequation has two solutions, namely $\{x \mapsto a, y \mapsto a\}$ and $\{x \mapsto a, y \mapsto b\}$. However, it has no solution where x is not mapped to a, so for the overall problem, there is no solution.

So the above sufficient condition needs to refined in order to be correct. However, calculating the number of solutions is again NP-hard.

Proposition 3.6.11 ([BLW23])**.** Calculating the number of solutions σ for some right-ground inequation $t < s$ is NP-hard.

The problem with the aforementioned insufficient condition is that an inequality $l_j \neq r_j$ does not necessarily rule out only one grounding, but possibly infinitely many groundings. This happens if there are variables that are not restricted by the matcher τ_j of l_j and r_j. However, the criterion can be refined to a correct sufficient condition. If we restrict ourselves to the $m + 1$ smallest terms again, we would again at least have a finite number of groundings that $l_j \neq r_j$ rules out. If we now sum up these numbers over all inequalities, we have an upper bound on the total number of ruled out groundings. For the inequation $t < s$, the same problem with variables that do not occur arises (there may be infinitely many solutions), so here, we restrict ourselves to the $m + 1$ smallest terms again. If now, the number of solutions for $t < s$ is larger than the upper bound on the total number of ruled out groundings, we can actually be sure that there is a solution. However, this correct sufficient condition is hard to compute and therefore seems to be not very useful in practice.

For an extended version of CC($\mathcal{X}$) we need alternating KBO Constraints. The reason why we integrated inequalities as opposed to the CC($\mathcal{X}$) algorithm presented above is that we wanted to support the checking of non-ground equalities. Currently, one can only check them by examining all ground instances. By adding inequalities to the constraints, you can quickly check whether a more detailed equality check is even necessary for the terms to be checked. However, it turned out that the integration of inequalities is difficult to accomplish efficiently. We therefore decided not to integrate it for the first published version of CC($\mathcal{X}$). Note, that the early version of CC($\mathcal{X}$) also does not support subsumption checks.

Definition 3.6.12 (Alternating KBO Constraint)**.** An alternating KBO constraint Γ consists of terms $t, s_1, \ldots, s_n \in T(\Sigma, \mathcal{X})$ and $\beta \in T(\Sigma)$. We say that Γ is satisfiable if there exists a substitution σ that is grounding for t such that for all substitutions τ that are grounding for all s_j we have

$$\left(\bigwedge_{j=1}^{n} t\sigma \neq s_j\tau \right) \wedge t\sigma < \beta.$$

Proposition 3.6.13 ([BLW23])**.** Checking satisfiability for alternating KBO constraints is NP-hard.

If a reasonable strategy is used, satisfiability of alternating KBO constraints can be checked using the algorithm from this Section. Any solution σ must be

such that $t\sigma < \beta$, so we only have to consider instances of the $s_j\tau$ with $s_j\tau < \beta$. What we can now do is to calculate for all s_j all groundings τ with $s_j\tau < \beta$ and add the inequality $t \neq s_j\tau$ to the constraint. There are only finitely many such groundings because we did not allow unary functions f with $w(f) = 0$. This way, we obtain a simple right-ground KBO constraint, so we can apply the algorithm. A more efficient possibility to do this is to add the groundings of the s_j implicitly, i.e., to change the condition of Increase (and the first case of Backtrack and Fail) to whether there exists a matcher τ such that $l_j\sigma(\vec{v}) = r_j\tau$. Also, the condition for the next grounding for Increase changes: It is not that we fix the inequality anymore, but that we change a variable that occurs on the left side of the inequality.

Example 3.6.14 ([BLW23])**.** Consider the signature $\Sigma = \{f/2, g/1, a/0\}$, together with the following alternating KBO constraint Γ:

$$\begin{aligned} t &= f(x_1, x_2) & s_1 &= f(g(y_1), y_2) \\ \beta &= f(f(a, a), a) & s_2 &= f(a, a) \end{aligned}$$

We set $w(a) = w(g) = w(f) = 1$ and $a \prec g \prec f$. The few smallest terms are

$$a, g(a), g(g(a)), f(a, a).$$

Note that for alternating KBO constraints, it does not suffice anymore to consider the $n + 1$ smallest terms only since an inequality may rule out more than one term for a variable. However, as already mentioned, we calculate the smallest terms as needed, so this is not a problem. For shorter notation, for F, we omit groundings $\vec{u}$ if there is a grounding $\vec{v} \in F$ with $\vec{v} <_F \vec{u}$. A possible run of the algorithm looks as follows:

$$\begin{aligned} & (\varepsilon; (a, a); \emptyset; \Gamma) \\ \Rightarrow_{\text{KCS}}^{\text{Increase}} \quad & (x_1; (g(a), a); \emptyset; \Gamma) && s_2,\ \tau = \{\} \\ \Rightarrow_{\text{KCS}}^{\text{Increase}} \quad & (x_1x_1; (g(g(a)), a); \emptyset; \Gamma) && s_1,\ \tau = \{y_1 \mapsto a, y_2 \mapsto a\} \\ \Rightarrow_{\text{KCS}}^{\text{Increase}} \quad & (x_1x_1x_1; (f(a, a), a); \emptyset; \Gamma) && s_1,\ \tau = \{y_1 \mapsto g(a), y_2 \mapsto a\} \\ \Rightarrow_{\text{KCS}}^{\text{Backtrack}} \quad & (x_1x_1; (g(g(a)), a); \{(f(a, a), a)\}; \Gamma) && \beta \\ \Rightarrow_{\text{KCS}}^{\text{Backtrack}} \quad & (x_1; (g(a), a); \{(g(g(a)), a)\}; \Gamma) && s_1 \\ \Rightarrow_{\text{KCS}}^{\text{Backtrack}} \quad & (\varepsilon; (a, a); \{(g(a), a)\}; \Gamma) && s_1 \\ \Rightarrow_{\text{KCS}}^{\text{Increase}} \quad & (x_2; (a, g(a)); \{(g(a), a)\}; \Gamma) && s_2,\ \tau = \{\} \end{aligned}$$

3.6.1 Experiments

We implemented the algorithm of Section 3.6 and its extension to constraints with right hand side variables, Definition 3.6.12, and tested it in the context of a very early version of the CC($\mathcal{X}$) algorithm [Hur01, NO80, DST80, Sho84]. I implemented a rather naive variant of [Hur01] with the only goal to generate KBO constraints in order to test my new algorithm on KBO constraints. In contrast to [Hur01] my algorithm considers a finite signature, as usual for first-order logic

Table 3.4: UEQ problem results for a nesting depth of 4

Problem	$< \beta$	Time KBO Solver	#calls	# true	# false
GRP183-3	9969	3	8014	7946	68
LAT143-1	29720	8797	31033	29554	1479
GRP409-1	103565	0	6	6	0

problems. All experiments were carried out on a Debian Linux server equipped with AMD EPYC 7702 64-Core CPUs running at 3.35GHz and an overall memory of 2TB. The result of all runs as well as all input files and binaries can be found at https://nextcloud.mpi-klsb.mpg.de/index.php/s/BAwd99cxFpSJmSp.

As a first test case I considered all eligible UEQ problems from CASC-J11 [Sut16]. I consider equations and all inequalities for the congruence closure algorithm. The equations generate the congruence and for the inequalities I compute the congruence classes for the respective right and left side term of the inequality. For each example, the KBO function weight was always set to one and the precedence is generated with respect to the occurrence of symbols in the input file in ascending order. For β I chose a fixed nesting depth of 4 and build for each input file a nested term of exactly this depth using function symbols in the order of occurrence in the input, starting with a non-constant function symbol. Out of all eligible problems my CC algorithm terminated on 186 problems within a time limit of 30 minutes. Please note that although my CC implementation is rather naive, in contrast to the classical ground CC algorithm it does not need a complete grounding; for the examples where my naive algorithm runs out of time a complete grounding is not affordable. Table 3.4 shows some typical runs on the UEQ domain. All timings are presented in hundredths of a second and if they take less than one hundredth of a second I write zero. The table shows the problem name, the number of ground terms smaller than β indicating the solution space for the constraint, the summed up time of all calls to the KBO constraint solver during the CC run, the number of calls to the KBO constraint solver, and the results of these calls. The three selected examples are typical: most of the problems are satisfiable and the constraint solving algorithm needs almost no time. Note that for the first example all 8014 calls to the constraint solver needed in sum 3 hundreds of a second. The LAT143-1 is the example showing the worst constraint solving performance, i.e., still less than a hundreth of a second per call.

For the SMT-LIB examples of the UF domain [BFT16], I expanded let operators, removed the typing, coded predicates as equations, did a CNF transformation and then took the first literal of each clause as input for the CC algorithm. Nesting depth was set to 2, the rest done as for the UEQ examples. Removing types means that the number of smaller terms increases, i.e., the problems get potentially more difficult for the constraint solver, in particular for unsatisfiable constraints. Table 3.5 again shows some typical results. 1112 examples could be performed by the CC algorithm inside 30 min. The UF domain contains larger examples compared to the UEQ domain, but the characteristics remain. Constraint solving itself takes almost no time. Again all timings are presented in hundredths of a second.

Table 3.5: SMT-LIB problem results for a nesting depth of 2

Problem	$< \beta$	Time KBO Solver	#calls	# true	# false
00336	2120806	0	131	131	0
uf.926761	138397692	0	3882	1988	1894
uf.555113	254939	134	5120	3306	1814

Here uf.555113 is the worst example on constraint solving time with 1.34 seconds for 5120 calls. Although alternating KBO constraint solving is NP-hard, in practice there are typically only a few inequalities meaning that out of the overall number of terms smaller β, only a few need to be considered.

3.7 Discussion of CC($\mathcal{X}$)

I presented the new calculus Non-Ground Congruence Closure (CC($\mathcal{X}$)). It takes as input non-ground equations and computes the corresponding congruence classes for the overall set of ground terms smaller than a given maximum term β. The algorithm is sound, complete, and terminating due to a notion of redundancy and the finite ground input space. I developed and implemented a sophisticated redundancy concept, e.g., by introducing filters to expensive checks such as subsumption. I did first steps towards an implementation with KBO constraints needed for the SCL(EQ) calculus.

Still there is room for further improvement. From an implementation point of view, as already mentioned, *Condensation* modifies a copy of the class and checks for subsumption. In the *Merge* and *Deduction* rules, the number of copies of classes is also higher than necessary, especially when the generated class is subsumed. For some border cases CC outperforms CC($\mathcal{X}$). Extending $\Rightarrow_{\mathrm{CC}(\mathcal{X})}$ to cope with input equations with only a few constants or few equations is a further line of research. Already now CC($\mathcal{X}$) can decide shallow equational classes, where the only arguments to functions are variables. This is independently of β and due to the notion of redundancy.

Equality checking between ground terms amounts to instance finding in a particular class, once the congruence closure algorithm is finished, both for CC and CC($\mathcal{X}$) where for the latter this has to be done modulo matching. Checking the equality of non-ground terms is much more involved both for CC($\mathcal{X}$) and CC. This is mainly due to the fact that I consider finite signature. If two non-ground terms are not in the same class this does not actually mean that they are not equal, since it could be the case that all instances of these terms are in the same class. First steps towards a solution to this were made in the KBO part of this Chapter. I could extend the constraints to support inequalities in order to quickly find out if all instances of a term already exist in other classes.

In general, CC($\mathcal{X}$) outperforms CC if the ratio of different variables to considered ground terms is on the ground term side. The other way round, if there are many variables but only a few ground terms to consider, then running CC is beneficial. This situation can be easily checked in advance, so CC($\mathcal{X}$) can be selected on problems where CC will fail extending the overall scope of applicability.

From an application point of view CC($\mathcal{X}$) can be immediately applied for

detecting false or propagating clauses in SCL(EQ), even with respect to equations with variables. Unit equations need to be considered this way. In an SMT context it could immediately add to the ground CC in the following sense: Using the result of CC($\mathcal{X}$) with equations before grounding in the theory combination and this way detecting additional ground instances needed.

Chapter 4

Conclusion and Future Work

This thesis contributes SCL(EQ), the first approach to conflict driven clause learning in first-order logic with equality that only learns non-redundant clauses according to a dynamically changing ordering. I prove SCL(EQ) sound and complete. Moreover, the thesis contributes first steps towards an efficient implementation of SCL(EQ) by introducing and implementing the new calculus CC($\mathcal{X}$), a generalization of the congruence closure algorithm for terms with variables. I prove CC($\mathcal{X}$) to be sound, complete and terminating. The evaluation shows that CC($\mathcal{X}$) outperforms CC if grounding gets infeasible. In joint work with Yasmine Briefs we showed that KBO constraint solving can be effectively achieved, i.e. KBO fulfills the requirements of SCL(EQ) on effective algorithms for a term order. Thus, CC($\mathcal{X}$) might be an effective solver for SCL(EQ) to find propagating and conflicting instances.

It turns out that lifting from SCL calculus to first-order logic with equality involves an enormous increase in the complexity of the calculus. So far, none of the more complex CDCL-style calculi for first-order logic, like model evolution with lemma learning [BFT06] or SGGS [BP15], have ever been extended to first-order logic with equality. This hints at the complexity of the task, especially as extensions to equality have been promised but not delivered [BP15]. Apart from the calculus itself, an efficient implementation requires, in addition to the presented CC($\mathcal{X}$), many building blocks including an infrastructure for a Superposition based prover as well as one for CDCL/SMT based provers. I will now discuss future work especially towards an implementation of SCL(EQ).

The trail reasoning in SCL(EQ) is currently defined with respect to rewriting. Beyond finding propagating and conflicting instances with CC($\mathcal{X}$) it would be interesting to create a rewrite proof out of the congruence classes. Gallier et al [GNP$^+$93] already showed an algorithm which creates a ground rewrite system out of the congruence classes in polynomial time. Future work could extend this algorithm to CC($\mathcal{X}$) to effectively compute the actual propagations, decisions and refutations.

In the case that SCL(EQ) runs into a stuck state, i.e., the current trail is a model for the set of considered ground instances, then the trail information can be effectively used for a guided continuation [BKMW24]. For example, in order

to use the trail to certify a model, the trail literals can be used to guide the design of a lifted ordering for the clauses with variables such that propagated trail literals are maximal in respective clauses. Then, it could be checked by Superposition, if the current clause is saturated by such an ordering. If this is not the case, then there must be a Superposition inference larger than the current β, thus giving a hint on how to extend β. Another possibility is to try to extend the finite set of ground terms considered in a stuck state to the infinite set of all ground terms by building extended equivalence classes following patterns that ensure decidability of clause testing, similar to the ideas in [BFW21]. If this fails, then again this information can be used to find an appropriate extension term β for rule Grow.

Bromberger et al. [BGLW22] showed how to lift the two-watched literal scheme to SCL. I could make use of this in an implementation as well. The aspect of how to find interesting ground decision or propagation literals for the trail including the respective grounding substituion σ can be treated similar to CDCL [SS96, JS96, MMZ+01, BHvMW09]. A simple heuristic may be used from the start, like counting the number of instance relationships of some ground literal with respect to the clause set, but later on a bonus system can focus the search towards the structure of the clause sets. Ground literals involved in a conflict or the process of learning a new clause get a bonus or preference. However, since the number of ground literals is not fixed from the beginning with growing β, all these operations need to be done via hashing or indexing operations modulo matching/unification in contrast to simple look-ups in the CDCL case. The regular strategy requires the propagation of all ground unit clauses smaller than β. For an implementation a propagation of the (explicit and implicit) unit clauses with variables to the trail will be a better choice. This complicates the implementation of refutation proofs and rewriting (congruence closure), but because every reasoning is layered by a ground term β this can still be efficiently done.

Bibliography

[AW15] Gábor Alagi and Christoph Weidenbach. NRCL - A model building approach to the bernays-schönfinkel fragment. In Carsten Lutz and Silvio Ranise, editors, *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*, volume 9322 of *Lecture Notes in Computer Science*, pages 69–84. Springer, 2015.

[Bau98] Peter Baumgartner. Hyper tableau – the next generation. In Harrie C. M. de Swart, editor, *Automated Reasoning with Analytic Tableaux and Related Methods, International Conference, TABLEAUX '98, Oisterwijk, The Netherlands, May 5-8, 1998, Proceedings*, volume 1397 of *Lecture Notes in Computer Science*, pages 60–76. Springer, 1998.

[BBB+22] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022.

[BCBdODF09] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe, and Pascal Fontaine. verit: an open, trustable and efficient smt-solver. In *International Conference on Automated Deduction*, pages 151–156. Springer, 2009.

[BFLW18] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified sat solver framework with learn, forget, restart, and incrementality. *Journal of automated reasoning*, 61:333–365, 2018.

[BFP07] Peter Baumgartner, Ulrich Furbach, and Björn Pelzer. Hyper tableaux with equality. In Frank Pfenning, editor, *International Conference on Automated Deduction*, volume 4603 of *LNAI*, pages 492–507. Springer, 2007.

[BFR17] Haniel Barbosa, Pascal Fontaine, and Andrew Reynolds. Congruence closure with free variables. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017*, volume 10206 of *Lecture Notes in Computer Science*, pages 214–230, 2017.

[BFSS15] Maria Paola Bonacina, Ulrich Furbach, and Viorica Sofronie-Stokkermans. *On First-Order Model-Based Reasoning*, volume 9200 of *LNAI*, pages 181–204. Springer International Publishing, 2015.

[BFSW19] Martin Bromberger, Mathias Fleury, Simon Schwarz, and Christoph Weidenbach. SPASS-SATT - A CDCL(LA) solver. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 111–122. Springer, 2019.

[BFT06] Peter Baumgartner, Alexander Fuchs, and Cesare Tinelli. Lemma learning in the model evolution calculus. In Miki Hermann and Andrei Voronkov, editors, *13th International Conference, LPAR 2006*, volume 4246 of *LNAI*, pages 572–586. Springer, 2006.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BFW21] Martin Bromberger, Alberto Fiori, and Christoph Weidenbach. Deciding the Bernays-Schoenfinkel fragment over bounded difference constraints by simple clause learning over theories. In Fritz Henglein, Sharon Shoham, and Yakir Vizel, editors, *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, pages 511–533. Springer, 2021.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[BGLW22] Martin Bromberger, Tobias Gehl, Lorenz Leutgeb, and Christoph Weidenbach. A two-watched literal scheme for first-order logic. In Boris Konev, Claudia Schon, and Alexander Steen, editors, *Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), Haifa, Israel, August, 11 - 12, 2022*, volume 3201 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2022.

[BGV98] Leo Bachmair, Harald Ganzinger, and Andrei Voronkov. Elimination of equality via transformation with ordering constraints.

In C. Kirchner and H. Kirchner, editors, *International Conference on Automated Deduction*, volume 1421 of *Lecture Notes in Computer Science*, pages 175–190. Springer, 1998.

[BGW93] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Superposition with simplification as a decision procedure for the monadic class with equality. In Georg Gottlob, Alexander Leitsch, and Daniele Mundici, editors, *Computational Logic and Proof Theory, Third Kurt Gödel Colloquium*, volume 713 of *LNCS*, pages 83–96. Springer, August 1993.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[BKMW24] Martin Bromberger, Florent Krasnopol, Sibylle Möhle, and Christoph Weidenbach. First-order automatic literal model generation. In Christoph Benzmüller, Marijn J.H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning*, pages 133–153, Cham, 2024. Springer Nature Switzerland.

[BLW23] Yasmine Briefs, Hendrik Leidinger, and Christoph Weidenbach. KBO constraint solving revisited. In Uli Sattler and Martin Suda, editors, *Frontiers of Combining Systems*, volume 14279 of *Lecture Notes in Computer Science*, pages 81–98, Cham, 2023. Springer Nature Switzerland.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[BP15] Maria Paola Bonacina and David A Plaisted. Sggs theorem proving: an exposition. In Stephan Schulz, Leonardo De Moura, and Boris Konev, editors, *PAAR-2014. 4th Workshop on Practical Aspects of Automated Reasoning*, volume 31 of *EPiC Series in Computing*, pages 25–38. EasyChair, 2015.

[BPT12] Peter Baumgartner, Björn Pelzer, and Cesare Tinelli. Model evolution with equality—revised and implemented. *Journal of Symbolic Computation*, 47(9):1011–1045, 2012.

[BSW23] Martin Bromberger, Simon Schwarz, and Christoph Weidenbach. SCL(FOL) revisited, 2023.

[BT03] Peter Baumgartner and Cesare Tinelli. The model evolution calculus. In Franz Baader, editor, *Automated Deduction – CADE-19*, volume 2741 of *LNAI*, pages 350–364. Springer, 2003.

[BT05] Peter Baumgartner and Cesare Tinelli. The model evolution calculus with equality. In Robert Nieuwenhuis, editor, *20th International Conference on Automated Deduction*, volume 3632 of *LNAI*, pages 392–408. Springer, 2005.

[BW09] Peter Baumgartner and Uwe Waldmann. Superposition and model evolution combined. In Renate A. Schmidt, editor, *Automated Deduction – CADE-22*, volume 5663 of *LNAI*, pages 17–34. Springer, 2009.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795, pages 93–107. Springer, 2013.

[DLL62a] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DLL62b] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[DP01] Nachum Dershowitz and David A. Plaisted. Rewriting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier, 2001.

[DST80] Peter J. Downey, Ravi Sethi, and Robert Endre Tarjan. Variations on the common subexpression problem. *Journal of the ACM*, 27(4):758–771, 1980.

[Dut14] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014.

[Fon04] Pascal Fontaine. *Techniques for verification of concurrent systems with invariants*. PhD thesis, PhD thesis, Institut Montefiore, Université de Liege, Belgium, 2004.

[FW19] Alberto Fiori and Christoph Weidenbach. Scl clause learning from simple models. In Pascal Fontaine, editor, *27th International Conference on Automated Deduction, CADE-27*, volume 11716 of *LNAI*. Springer, 2019.

[GdN99] Harald Ganzinger and Hans de Nivelle. A superposition decision procedure for the guarded fragment with equality. In *LICS*, pages 295–304, 1999.

[GHN+04] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Dpll(/T): fast decision procedures. In Rajeev Alur and Doron A. Peled, editors, *16th International Conference, CAV 2004*, volume 3114 of *LNCS '04*, pages 176–188. Springer, 2004.

[GKR20] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption demodulation in first-order theorem proving. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 297–315. Springer, 2020.

[GNP+93] Jean Gallier, Paliath Narendran, David Plaisted, Stan Raatz, and Wayne Snyder. An algorithm for finding canonical sets of ground rewrite rules in polynomial time. *Journal of the ACM (JACM)*, 40(1):1–16, 1993.

[Hur01] Joe Hurd. Congruence classes with logic variables. *Log. J. IGPL*, 9(1):53–69, 2001.

[HW07] Thomas Hillenbrand and Christoph Weidenbach. Superposition for finite domains. Research Report MPI-I-2007-RG1-002, Max-Planck Institute for Informatics, Saarbruecken, Germany, April 2007.

[JS96] Roberto J. Bayardo Jr. and Robert Schrag. Using csp look-back techniques to solve exceptionally hard sat instances. In Eugene C. Freuder, editor, *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*, volume 1118 of *LNCS*, pages 46–60. Springer, 1996.

[KAM80] S. KAMIN. Attempts for generalizing the recursive path orderings. *Report, Dept. of Computer Science, Univ. of Illinois*, 1980.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In I. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297. Pergamon Press, 1970.

[Kor13] Konstantin Korovin. *Inst-Gen – A Modular Approach to Instantiation-Based Automated Reasoning*, pages 239–270. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[KS10] Konstantin Korovin and Christoph Sticksel. iprover-eq: An instantiation-based theorem prover with equality. In Jürgen Giesl and Reiner Hähnle, editors, *5th International Joint Conference, IJCAR 2010*, volume 6173 of *LNAI*, pages 196–202. Springer, 2010.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 1–35. Springer, 2013.

[LW22] Hendrik Leidinger and Christoph Weidenbach. SCL(EQ): SCL for first-order logic with equality. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning - 11th International Joint Conference, IJCAR 2022, Haifa, Israel, August 8-10, 2022, Proceedings*, volume 13385 of *Lecture Notes in Computer Science*, pages 228–247. Springer, 2022.

[LW23] Hendrik Leidinger and Christoph Weidenbach. SCL(EQ): SCL for first-order logic with equality. *Journal of Automated Reasoning*, 67(3):22, 2023.

[LW24] Hendrik Leidinger and Christoph Weidenbach. Non-ground congruence closure. *arXiv preprint arXiv:2412.10066*, 2024.

[McC92] William McCune. Experiments with discrimination-tree indexing and path indexing for term retrieval. *J. Autom. Reason.*, 9(2):147–167, 1992.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference, 2001. Proceedings*, pages 530–535. ACM, 2001.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27(2):356–364, 1980.

[NO07] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Information and Computation*, 205(4):557–580, 2007.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *Journal of the ACM*, 53:937–977, November 2006.

[PZ00] David A Plaisted and Yunshan Zhu. Ordered semantic hyperlinking. *Journal of Automated Reasoning*, 25(3):167–217, 2000.

[RBF18] Andrew Reynolds, Haniel Barbosa, and Pascal Fontaine. Revisiting enumerative instantiation. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 112–131. Springer, 2018.

[RTdM14] Andrew Reynolds, Cesare Tinelli, and Leonardo de Moura. Finding conflicting instances of quantified formulas in SMT. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 195–202, 2014.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence 4*, pages 135–150, 1969.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, 1984.

[SS96] João P. Marques Silva and Karem A. Sakallah. Grasp - a new search algorithm for satisfiability. In *International Conference on Computer Aided Design, ICCAD*, pages 220–227. IEEE Computer Society Press, 1996.

[Sti11] Christoph Sticksel. *Efficient Equational Reasoning for the Inst-Gen Framework*. PhD thesis, 2011. Copyright - Database copyright ProQuest LLC; ProQuest does not claim copyright in the individual underlying works; Analyte descriptor - N.A; Last updated - 2024-01-29.

[Sut16] Geoff Sutcliffe. The CADE ATP system competition - CASC. *AI Magazine*, 37(2):99–101, 2016.

[Sut17] Geoff Sutcliffe. The TPTP problem library and associated infrastructure - from CNF to th0, TPTP v6.4.0. *J. Autom. Reason.*, 59(4):483–502, 2017.

[Teu18] Andreas Teucke. *An Approximation and Refinement Approach to First-Order Automated Reasoning*. Doctoral thesis, Saarland University, 2018.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In Renate A. Schmidt, editor, *22nd International Conference on Automated Deduction (CADE-22)*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145, Montreal, Canada, August 2009. Springer.

[Wei01] Christoph Weidenbach. Combining superposition, sorts and splitting. In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, volume 2, chapter 27, pages 1965–2012. Elsevier, 2001.

[Wei15] Christoph Weidenbach. Automated reasoning building blocks. In Roland Meyer, André Platzer, and Heike Wehrheim, editors, *Correct System Design - Symposium in Honor of Ernst-Rüdiger Olderog on the Occasion of His 60th Birthday, Oldenburg, Germany, September 8-9, 2015. Proceedings*, volume 9360 of *Lecture Notes in Computer Science*, pages 172–188. Springer, 2015.

[Wis12] Patrick Wischnewski. *Effcient Reasoning Procedures for Complex First-Order Theories*. PhD thesis, Saarland University, November 2012.

[WTRB20] Uwe Waldmann, Sophie Tourret, Simon Robillard, and Jasmin Blanchette. A comprehensive framework for saturation theorem proving. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part I*, volume 12166 of *Lecture Notes in Computer Science*, pages 316–334. Springer, 2020.

[WW08] Christoph Weidenbach and Patrick Wischnewski. Contextual rewriting in spass. In *PAAR/ESHOL*, volume 373 of *CEUR Workshop Proceedings*, pages 115–124, Sydney,Australien, 2008.

[WW10] Christoph Weidenbach and Patrick Wischnewski. Subterm contextual rewriting. *AI Communications*, 23(2-3):97–109, 2010.