UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Extending the E-Hyper Tableau Calculus for Reasoning with the Unique Name Assumption

Diplomarbeit

zur Erlangung des Grades eines Diplom-Informatikers
im Studiengang Informatik

vorgelegt von
Markus Bender

Erstgutachter:	Prof. Dr. Ulrich Furbach (Institut für Informatik, AG Künstliche Intelligenz)
Zweitgutachter:	Dipl. Inform. Björn Pelzer (Institut für Informatik, AG Künstliche Intelligenz)

Koblenz, im Januar 2012

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☒	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☒	☐

. . . . . .
(Ort, Datum) (Unterschrift)

Deutsche Zusammenfassung

In einigen Bereichen des automatischen Theorembeweisens benötigt man das Wissen, dass Konstanten paarweise ungleich sind. Um dieses zu erreichen, fügt man Fakten, die dieses Wissen explizit angeben, zu den Wissensbasen hinzu. Wenn man diese Eigenschaft für viele Konstanten definieren muss, wird die Klauselmenge der Wissensbasen schnell sehr umfangreich und wegen der vielen — eigentlich irrelevanten — Ungleichheiten kann man den Blick auf das eigentlich formalisierte Problem verlieren. Da die Größe der Wissensbasis in vielen Fällen Einfluss auf die Geschwindigkeit hat, ist es auch aus diesem Grund sinnvoll, die Anzahl dieser Fakten gering zu halten.

Die *unique name assumption* erlaubt auf die Einführung der Ungleichheits-Fakten zu verzichten, da sie festlegt, dass zwei Konstanten genau dann gleich sind, wenn ihre Interpretationen identisch sind. Auf diesem Wege lässt sich das Aufblähen von Wissensbasen mit Ungleichheits-Fakten verhindern.

In dieser Arbeit wird der E-Hyper-Tableau-Kalkül erweitert um die unique name assumption nutzen zu können. Der in dieser Arbeit entwickelte Kalkül ist vollständig und korrekt, was durch formale Beweise in dieser Arbeit belegt wird. Um zu zeigen, dass die native Behandlung von Ungleichheiten dem Einführen von Ungleichheits-Fakten überlegen ist, wird der Kalkül in den Theorembeweiser E-KRHyper implementieren. Der Theorembeweiser E-KRHyper ist ein etabliertes System und basiert in seiner ursprünglichen Version auf dem E-Hyper-Tableau.

Mit systematischen Tests wird dann gezeigt, dass die entwickelte Implementierung des erweiterten Kalküls nie schlechter ist, als der original E-KRHyper, diesen aber in einigen Fällen in der Ausführungsgeschwindigkeit deutlich übertrifft.

Contents

Notations

If not stated otherwise, the following conventions on notations are used throughout this thesis:

- $\mathbb{T}$ is the set of terms
- $\mathbb{F}$ is the set of functions
- $\mathbb{C}$ is the set of constants
- $\mathbb{D}$ is the set of distinct object identifiers
- $\mathbb{V}$ is the set of variables
- $s, t, u \in \mathbb{T}$
- $f, g, h \in \mathbb{F}$
- $a, b, c \in \mathbb{C}$
- $i, j \in \mathbb{D}$
- $x, y, z \in \mathbb{V}$
- A, B, C are atoms
- K, L, M are literals
- $\mathcal{A}, \mathcal{B}, C$ are clauses
- $\mathcal{S}, \mathcal{T}$ are clause sets
- $\mathbf{t}$ is a tree
- $\mathbf{N}$ is a node
- $\mathbf{B}$ is a branch and an abbreviation for $\lambda(\mathbf{B})$
- $\mathbf{B}^j$ is an initial segment of $\mathbf{B}$
- $\mathbf{T}$ is an E-hyper tableau
- $\mathbf{t}_\infty$ is the limit tree
- $\mathbf{B}_\infty$ is the set of persistent clauses of $\mathbf{B}$

With exception of the special constant `t`, words written in `typewriter` fonts are concepts taken from the implementation.

Chapter 1

Introduction

In automated theorem proving, there are some problems that need information on the inequality of certain constants[2]. In most cases this information is provided by adding facts of form $false \leftarrow c_0 = c_1$ to the knowledge base. This fact explicitly states that those two constants are unequal. Depending on the number of constants, a huge amount of this facts can clutter the knowledge base and distract the author and readers of the problem from its actual proposition.

For most cases it is save to assume that a larger knowledge base reduces the performance of a theorem prover, which is another drawback of explicit inequality facts.

Using the *unique name assumption* in those reasoning tasks renders the introduction of inequality facts obsolete as the unique name assumptions states that two constants are identical iff their interpretation is identical. Implicit handling of non-identical constants makes the problems easier to comprehend and reduces the execution time of reasoning.

Both set of clauses given in Figure 1.1 formalize the same problem. In Figure 1.1a the inequalities are explicitly stated and in Figure 1.1b implicit knowledge of inequalities is used.

In this thesis we will show how to integrate the unique name assumption into the E-hyper tableau calculus and that the modified calculus is sound and complete. The calculus will be implemented into the E-KRHyper theorem prover and we will show, by empiric evaluation, that the changed implementation, which is able to use the unique name assumption, is superior to the traditional version of E-KRHyper.

We start by introducing some theoretical preliminaries in Chapter 2 which includes needed logical concepts (see Section 2.1) and the original version of the E-hyper tableau calculus (see Section 2.2). In Chapter 3, we introduce the unique name assumption (see 3.1), show how the calculus is extended (see 3.2), and prove that the modified version of the calculus is sound and complete (see Section 3.3).

In the practical part of this thesis (see Chapter 4) we introduce the E-KRHyper theorem prover (see Section 4.1.2) and draft how we extended it with our developed calculus (see Section 4.1.3).

```
sel(sto(A,I,E),I)=E.
sel(sto(A,I,E),J)=sel(A,J):- not(X=Y).
A=B:-sel(A,sk(A,B))=sel(B,sk(A,B)).
sto(sto(sto(sto(sto(a,i0,e0),i1,e1),i3,e3),i4,e4),i2,e2)=
sto(sto(sto(sto(sto(a,i0,e0),i1,e1),i2,e2),i3,e3),i4,e0).
false:-i0=i1.   false:-e0=e1.
false:-i0=i2.   false:-e0=e2.
false:-i0=i3.   false:-e0=e3.
false:-i0=i4.   false:-e0=e4.
false:-i1=i2.   false:-e1=e2.
false:-i1=i3.   false:-e1=e3.
false:-i1=i4.   false:-e1=e4.
false:-i2=i3.   false:-e2=e3.
false:-i2=i4.   false:-e2=e4.
false:-i3=i4.   false:-e3=e4.
```

(a)

```
sel(sto(A,I,E),I)=E.
sel(sto(A,I,E),J)=sel(A,J):- not(X=Y).
A=B:-sel(A,sk(A,B))=sel(B,sk(A,B)).
sto(sto(sto(sto(sto(a,i0,e0),i1,e1),i3,e3),i4,e4),i2,e2)=
sto(sto(sto(sto(sto(a,i0,e0),i1,e1),i2,e2),i3,e3),i4,e0).
```

(b)

Figure 1.1: A problem with explicit inequality (a) and implicit inequality (b) of constants.

This implementation is then used to evaluate the impact of our extension, which is described in Section 4.2.

We introduce projects that are related to our approach in Chapter 5.

In Chapter 6, we conclude our work and point out some future work.

Chapter 2

Theoretical Preliminaries

2.1 Formal Preliminaries

2.1.1 Introduction

In the following section several concepts that are needed in this thesis are introduced. As most preliminaries are shared with [18] and [12], parts of the following explanations are taken from or inspired by it.

In section 2.1.2 the first order language and some language related concepts used in this thesis is introduced. Instead of dealing with all possible elements of this language, a subset with certain properties, called clauses, is defined thereafter in section 2.1.3. After the syntax is defined, the semantics are introduced in section 2.1.4 by defining the concept of interpretations. For efficient reasoning with equalities, an order of terms with certain properties is required, which are discussed in section 2.1.5. Section 2.1.6 defines the concept of redundancy and negligible clauses, which are other important points for efficiency. Thereafter section 2.1.7 introduces trees and tableaux, which is the basis for the decision procedure being used in this thesis and introduced in section 2.2.

2.1.2 The First Order Logic Language

Familiarity with first order logic is assumed, therefore a brief overview is given instead of detailed formal definitions.

The language $\mathfrak{L}$ used throughout this thesis consists of the logical symbols $\forall, \exists, \vee, \wedge, \neg$, the elements of the set of *variables* $\mathbb{V}$, the set of *function* symbols $\mathbb{F}$ and the set of *predicate* symbols $\mathbb{P}$. $\mathfrak{L}$ is determined by the fixed signature $\Sigma = (\mathbb{F}, \mathbb{P})$.

$\mathbb{V}$, $\mathbb{F}$, $\mathbb{P}$, are possibly infinite, non-empty and mutually disjoint. The elements of $\mathbb{F}$ and $\mathbb{P}$ have a fixed arity. Function symbols with arity zero are called *constants* where $\mathbb{C} \subseteq \mathbb{F}$ is the set of constants.

$\mathbb{T}$ is the set of *terms* of $\mathfrak{L}$ and is inductively defined as follows:

1. All constants and variables are terms.

2. If $f \in \mathbb{F}$ has arity n and $t_1 \ldots t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

The set of *subterms* of a term u are defined as follows:

1. u is a subterm of u.

2. if $u = f(t_1, \ldots, t_n)$ then all subterms of $t_1, \ldots, t_n$ are subterms of u.

A subterm of t that is not t itself is called *proper subterm*.

A special binary predicate symbol is the *equality* $\simeq$. If $s, t \in \mathbb{T}$, $s \simeq t$ is called an *equation* and states, that s and t are semantically equal. Equations are symmetric, i.e. $s \simeq t$, iff $t \simeq s$. The negation of an equation is called *negative equation* and written as $s \not\simeq t$ instead of $\neg s \simeq t$.

To simplify the introduction of the calculus later on, it is assumed that $\simeq$ is the only predicate symbol in $\mathbb{P}$ and therefore all atoms are equations. This does not lead to a loss of generality as all non-equation atoms A can be transformed to an equation $A \simeq \mathtt{t}$ where $\mathtt{t}$ is a newly introduced, distinct constant. For sake of simplicity the notation $p(x)$ is used instead of the formal notation $p(x) \simeq \mathtt{t}$.

Formally this entails that predicates are functions with the signature $\mathbb{T}^* \rightarrow \{true, false\}$ but the notions of typed functions and typed logic are not introduced in this thesis..

The concept of *positions* allows to refer a specific subterm by a sequence of natural numbers as follows: If t is a term and p is a position then $t|_p$ denotes the subterm of t at position p. In particular, if ϵ is the empty sequence and $t = f(t_1, \ldots, t_n)$, then $t|_\epsilon = t$ and $t|_{i.p} = t_i|_p$ for $1 \leq i \leq n$. If p is a position in t, then the notation $t[s]_p$ will be used for $t|_p = s$, and $t[p/s']$ represents the term obtained by replacing $t|_p$ with s' at position p in t. If p is obvious or unimportant within the context, then it can be omitted, so that $t[s]$ denotes the term t with the subterm s, and $t[s']$ denotes the same term t except for its subterm s having been replaced by s'.

The *set of variables* of a term t is denoted by $vars(t)$. A term t is called *ground* iff $vars(t) = \emptyset$.

A *substitution* σ is a mapping from $\mathbb{F}$ to $\mathbb{T}$, with finite *domain* $dom(\sigma) = \{x \mid x \neq \sigma x\}$ and a finite *range* $ran(\sigma) = \{x\sigma \mid x \neq \sigma x\}$, $x \in \mathbb{V}$. A ground substitution γ is a substitution with $vars(ran(\gamma)) = \emptyset$.

A *renaming* ρ is a substitution which is a bijection of $\mathbb{V}$ onto itself.

Given two terms s and t, a substitution σ is a *unifier* for s and t if $s\sigma = t\sigma$. σ is a *most general unifier* (mgu), if for any other unifier τ for s and t there is a substitution ψ with $\sigma\psi = \tau$.

A term s is an *instance* of a term t (written as $s \gtrsim t$) if there is a substitution σ such that $s\sigma = t$.

A term s is a *variant* of t (written as $s \sim t$) if there is a renaming ρ such that $s\rho = t$.

The set of *formulæ* of $\mathfrak{L}$ and is inductively defined as follows:

1. If $p \in \mathbb{P}$ has arity n and $t_1 \ldots t_n$ are terms, then $p(t_1, \ldots, t_n)$ is a formula.

2. If A, B are formulæ then $\neg A, A \wedge B, A \vee B$ are formulæ.

3. If A is a formula and $x \in \mathbb{V}$ then $\forall xA, \exists xA$ are formulæ.

A formula of form $p(t_1, \ldots, t_n)$ with $p \in \mathbb{P}$ and $t_1 \ldots t_n$ is called *atomic formula* or *atom*.

A *literal* is an atom or the negation of an atom.

A literal $K = \overline{L}$ is called the *complement* of L.

A literal is *ground* if its component terms are ground.

The notions of the set of variables, substitutions, renamings, unifiers, instances and variants are extended to literals in the obvious way.

2.1.3 Clauses

A *clause* $C = A_1 \vee \ldots \vee A_m \vee \neg B_1 \vee \ldots \vee \neg B_n$ is a set of literals, usually written as a as an implication $A_1, \ldots, A_m \leftarrow B_1, \ldots, B_n$ with $m, n \geq 0$.

The set $\mathcal{A} = \{A_1, \ldots, A_m\}$ is called the *head* of the clause C while $\mathcal{B} = \{B_1, \ldots., B_n\}$ is called the *body* of C. Accordingly, $A_1, \ldots, A_m$ denote both the head atoms and the head literals of C, while $B_1, \ldots, B_n$ refer to the body atoms and $\neg B_1, \ldots, \neg B_n$ to the body literals. The notation $A, \mathcal{A} \leftarrow B, \mathcal{B}$ refers to the clause with the head atoms $\{A\} \cup \mathcal{A}$ and the body atoms $\{B\} \cup \mathcal{B}$.

A *unit* is a clause consisting of exactly one literal.

A clause is *empty* if both its head and its body are empty. The empty clause is denoted by $\square$.

The notions of the set of variables, substitutions, renamings, unifiers, instances and variants are extended to clauses in the obvious way.

A clause is *ground* iff its literals are ground.

A clause is *pure* if none of its distinct head literals share variables, that is, iff $vars(A_i) \cap vars(A_j) = \emptyset$ for $i, j \in \{1, \ldots, m\}$ and $i \neq j$. A substitution π is a *purifying substitution* for C iff $C\pi$ is pure.

All variables in a clause are taken to be universally quantified.

A *clause set* is a conjunction of clauses and is sometimes called a *knowledge base*.

Ω_Σ is the *universal set* of clauses, which contains all possible clauses for a given signature Σ.

2.1.4 Interpretation

As we assume that $\simeq$ is the only predicate symbol of the language $\mathfrak{L}$ with signature Σ, the *Herbrand interpretation* I is a set of ground Σ-equations that are considered true in I. The notions of satisfiability and validity are defined as usual. $I \models F$ denotes that I satisfies F, with F being a ground Σ-literal, a ground Σ-clause or a set thereof.

An *E-Interpretation* is an interpretation which is also a congruence relation on the Σ-terms. I^E denotes the smallest congruence relation on the Σ-terms that includes I. $I^E \models F$ denotes that I^E-satisfies F; this will be written as $I \models_E F$, though. Iff every E-interpretation satisfying F also satisfies F', then this means that F E-entails F', written as $F \models_E F'$.

Creating equivalence classes by using Herbrand models and the properties of the equality (symmetry, transitivity, reflexivity) leads to an easy understandable representation of a model. This is illustrated in Example 1

Example 1 (E-Interpretation)**.** Assume the set of literals

$$\{a \simeq b, c \simeq d, a \simeq d, f \simeq g, a \simeq g, h \simeq i, i \simeq j\}$$

is a Herbrand model for some formula.

Then with the corresponding E-Interpretation

$$\{\{a, b, c, d, f, g\}, \{i, j, h\}\}$$

it is easier to see that the formula is satisfied if the constants in the two equivalence classes are equal.

□

2.1.5 Term Ordering

For efficient equality reasoning with the E-hyper tableau calculus a term ordering $\succ$ with certain properties is needed. $\succ$ must be:

1. a strict partial ordering (irreflexive, antisymmetric and transitive),
2. well-founded,
3. closed under context - if $s \succ s'$ for $s, s' \in \mathbb{T}$, then $t[p/s] \succ t[p/s']$ for any $t \in \mathbb{T}$ and any position p in t,
4. liftable - if $s \succ t$ for $s, t \in \mathbb{T}$, then $s\sigma \succ t\sigma$ for any substitution σ, and finally
5. total on ground terms.

$\succ$ is lifted to atoms, literals and clauses in the obvious way.

$\succ$ induces the non-strict ordering $\succeq$. The converse is denoted by $\prec$ and $\preceq$ respectively.

Such an ordering is called *reduction ordering*. As the specific ordering is not of interest in the context of this work, no more details are given but it is referred to [18] for further information.

2.1.6 Negligible Clauses

One important task in an efficient equality treatment is to generate the fewest possible clauses. For this, it is necessary to define criteria by which clauses can be identified as not useful for the reasoning process.

The first criterion is that of redundancy which means that a clause is not helpful if it follows from a set clauses that are smaller w.r.t. $\succ$. Before this informal description is formalized in Definition 2.1.1 following [3] the notation $\mathcal{S}_{\mathcal{D}}$ is introduced.

Let $\mathcal{S}'$ be the set of all ground instances of all clauses in a clause set $\mathcal{S}$ then $\mathcal{S}_{\mathcal{D}} = \{C \in \mathcal{S}' \mid \mathcal{D} \succ C\}$ is the set of all ground instances of all clauses of $\mathcal{S}$ that are smaller w.r.t. $\succ$ than $\mathcal{D}$

Definition 2.1.1 (Redundancy)
Let $\mathcal{D}$ be a clause, $\mathcal{S}$ a set of clauses.

A ground clause $\mathcal{D}$ is *redundant w.r.t. a clause set* $\mathcal{S}$ iff $\mathcal{S}_{\mathcal{D}} \models_E \mathcal{D}$.

A non-ground clause $\mathcal{D}$ is redundant w.r.t. a clause set $\mathcal{S}$ iff every ground instance of $\mathcal{D}$ is redundant w.r.t. $\mathcal{S}$.

■

The second criterion is that of non-proper subsumption. A clause is non-properly subsumed by another clause if it is an instance of that clause. A clause is non-properly subsumed by a set of clauses if it is an instance of a clause of this particular clause set.

Definition 2.1.2 (Non-proper Subsumption)
Let $\mathcal{D}$ be a clause and $\mathcal{S}$ a set of clauses.

A clause $\mathcal{D}$ is *non-properly subsumed* by a clause C iff there is a substitution σ such that $\mathcal{D} = C\sigma$.

A clause $\mathcal{D}$ is non-properly subsumed w.r.t. a clause set $\mathcal{S}$ iff there is a $C \in \mathcal{S}$ that non-properly subsumes $\mathcal{D}$.

■

A clause is negligible w.r.t. to a set of clauses if it is redundant w.r.t. to this set or non-properly subsumed w.r.t. to it.

Definition 2.1.3 (Negligible Clauses (prelim.))
Let $\mathcal{D}$ be a clause and $\mathcal{S}$ a set of clauses.

A clause $\mathcal{D}$ is *negligible w.r.t. a clause set* $\mathcal{S}$ iff at least one of the following holds:

- $\mathcal{D}$ is redundant w.r.t. $\mathcal{S}$.

- $\mathcal{D}$ is non-properly subsumed w.r.t. $\mathcal{S}$.

■

To avoid confusion with the definitions' names, *(prelim.)* is appended to the names of those definitions that are extended later on to indicate that these are preliminary definitions.

In most cases it is obvious what set of clauses $\mathcal{S}$ is meant and therefore the addition *w.r.t. to a clause set* $\mathcal{S}$ might be omitted for the sake of brevity.

2.1.7 Trees and Tableaux

A *tree* is a directed, acyclic graph denoted by a pair $(\mathbb{N}, \mathbb{E})$ consisting of a set of nodes $\mathbb{N}$ and a set of edges $\mathbb{E} \subset (\mathbb{N} \times \mathbb{N})$. A node that has no incoming edges is called *root* a node with not outgoing edges is called *leaf*.

A *branch* $\mathbf{B}$ in $\mathbf{T}$ is a sequence $\mathbf{N}_0, \ldots, \mathbf{N}_n$ of nodes in $\mathbf{T}$, with $\mathbf{N}_0$ being the root node of $\mathbf{T}$, each $\mathbf{N}_i$ being the immediate predecessor of $\mathbf{N}_{i+1} (0 \leq i < n)$, and $\mathbf{N}_n$ being a leaf of $\mathbf{T}$.

A *initial segment* $\mathbf{B}^j$ of a branch $\mathbf{B} = (\mathbf{N}_i)_{0 \leq i < v}$ is defined as a sequence of nodes of $\mathbf{B}$ starting with $\mathbf{N}_0$ and ending with $\mathbf{N}_j$, i.e. $\mathbf{B}^j = (\mathbf{N}_i)_{0 \leq i \leq < j}$.

An *E-hyper tableau* $\mathbf{T}$ *over a signature* Σ is a pair $(\mathbf{t}, \lambda)$, where $\mathbf{t}$ is a finite, ordered tree and λ is a labelling function assigning an Σ-clause to each node of $\mathbf{t}$.

$\lambda(\mathbf{B}) = \{\lambda(\mathbf{N}_0), \ldots, \lambda(\mathbf{N}_n)\}$ is the set of clauses in $\mathbf{B}$, called the *tableau clauses*. For the sake of convenience, the notation $C \in \mathbf{B}$ is used iff $C \in \lambda(\mathbf{B})$.

The notation $\mathbf{B} \cdot C$ represents the tableau branch obtained from attaching a node labelled with C to the leaf of $\mathbf{B}$, while $\mathbf{B} \cdot \mathbf{B}'$ represents the tableau obtained from concatenating $\mathbf{B}$ and the node sequence $\mathbf{B}'$.

A branch in an E-hyper tableau is *closed* iff it contains the empty clause $\square$ otherwise it is *open*. An E-hyper tableau is closed iff all of its branches are closed, and it is open otherwise.

2.2 E-Hyper Tableau Calculus

2.2.1 Introduction

The *E-hyper tableau calculus* was developed in 2007 by Baumgartner, Furbach, and Pelzer at the National ICT Australia and the University of Koblenz-Landau as an extension of the *hyper tableau calculus*[12].

The latter was developed by Baumgartner, Furbach, and Niemelä at the University of Koblenz-Landau in 1996 as an efficient model generation and proof-procedure for first-order theories. Its main idea is to combine the benefits of tableau calculi, i.e. rich structure for the derivation process, partial models as by-product, and the advantages of the hyper resolution, i.e. the hyper property for resolving

negative literals of a clause in a single inference step, universally quantified variables and subsumption as pruning technique[10].

As the hyper tableau lacked the treatment of equalities, which is mandatory in most proving applications, the calculus was changed to treat equalities, which lead to the E-hyper tableau calculus, which is introduced in this section.

The first step is the introduction of inference rules in section 2.2.2. Section 2.2.3 shows how the inference rules and additional rules are used to extend an E-hyper tableau and modify its nodes. The next step is the definition of a derivation which is done in section 2.2.4.

As in section 2.1 most concepts are shared with [18] and [12], therefore parts of the following explanations are taken from or inspired by it.

2.2.2 Inference Rules

The E-hyper tableau calculus makes use of the four inference rules sup-left, unit-sup-right, ref, split to work on clauses. The first three are adapted from the superposition calculus to deal with equality and will be referred to as the *equality* rules. If they are applicable, i.e. certain conditions are met, each of the equality rules takes a set of clauses as input and derives a new clause. The fourth so-called *split* rule is used to to create a branch split from a disjunctive clause head. As these four inference rules operate on clauses, they will then be lifted to tableaux in matching extension rules. Additionally, two rules for the handling of redundant and non-properly subsumed clauses will be introduced.

In Figure 2.1 the four inference rules of the E-hyper tableau calculus and the conditions that are needed to apply them are shown.

The sup-left rule (*superposition left*) applies a positive unit equation to a body literal of another clause. If the sup-left rule is applied with clause C as left premise, $\mathcal{D}$ as right premise, the mgu σ and the conclusion $\mathcal{E}$, then this inference instance will be denoted by $C, \mathcal{D} \Rightarrow_{\text{sup-left}(\sigma)} \mathcal{E}$.

The unit-sup-right rule (*unit superposition right*) applies a positive unit equation to another positive unit equation. If the unit-sup-right rule is applied with clause C as left premise, $\mathcal{D}$ as right premise, the mgu σ and the conclusion $\mathcal{E}$, then this inference instance will be denoted by $C, \mathcal{D} \Rightarrow_{\text{unit-sup-right}(\sigma)} \mathcal{E}$.

The ref rule (*reflexivity*) works on a single clause and removes an equational body literal whose both sides can be unified. If the ref rule is applied to clause C with the mgu σ and the conclusion $\mathcal{E}$, then this inference instance will be denoted by $C \Rightarrow_{\text{ref}(\sigma)} \mathcal{E}$.

The split rule (*split*) works on a positive disjunction. In combination with a purifying substitution, it creates a new unit clause for each head literal. If the split rule is applied to clause C with the purifying substitution π and the conclusions $A_1\pi \leftarrow, \ldots, A_m\pi \leftarrow$, then this inference instance will be denoted by $C \Rightarrow_{\text{split}(\pi)} A_1\pi \leftarrow, \ldots, A_m\pi \leftarrow$.

Several notions are now extended to the inference rules sup-left, unit-sup-right, ref and split, as well as to their instances:

$$\textsf{sup-left}(\sigma)\ \frac{\mathcal{A} \leftarrow s[u'] \simeq t, \mathcal{B} \qquad u \simeq r \leftarrow}{(\mathcal{A} \leftarrow s[r] \simeq t, \mathcal{B})\sigma} \quad \text{if (1-4) holds}$$

$$\textsf{unit-sup-right}(\sigma)\ \frac{s[u'] \simeq t \leftarrow \qquad u \simeq r \leftarrow}{(s[r] \simeq t \leftarrow)\sigma} \quad \text{if (1-5) holds}$$

$$\textsf{ref}(\sigma)\ \frac{\mathcal{A} \leftarrow s \simeq t, \mathcal{B}}{(\mathcal{A} \leftarrow \mathcal{B})\sigma} \quad \text{if (6) holds}$$

$$\textsf{split}(\pi)\ \frac{A_1, \ldots, A_n \leftarrow}{A_1\pi \leftarrow \quad \ldots \quad A_n\pi \leftarrow} \quad \text{if (7,8) holds}$$

Conditions:

1. u' is not a variable
2. σ is a mgu of u and u'
3. $u\sigma \not\leq r\sigma$
4. $s\sigma \not\leq t\sigma$
5. $(s \simeq \mathtt{t})\sigma \not\leq (u \simeq r)\sigma$
6. σ is a mgu of s and t
7. $n \geq 2$
8. π is a purifying substitution for $A_1, \ldots, A_n \leftarrow$

Figure 2.1: Rules for the E-hyper tableau calculus.

- An inference is *ground* iff its constituent clauses (premises and conclusions) are ground. For ground inferences the substitution σ and the purifying substitution π can both be assumed to be the empty substitution ϵ.
- If $C, \mathcal{D} \Rightarrow_{\textsf{sup-left}(\sigma)} \mathcal{E}$ is a sup-left inference and γ is a substitution such that $C\sigma\gamma, \mathcal{D}\sigma\gamma \Rightarrow_{\textsf{sup-left}(\epsilon)} \mathcal{E}\gamma$ is ground, then the latter is called a ground instance of $C, \mathcal{D} \Rightarrow_{\textsf{sup-left}(\sigma)} \mathcal{E}$.
- If $C, \mathcal{D} \Rightarrow_{\textsf{unit-sup-right}(\sigma)} \mathcal{E}$ is a unit-sup-right inference and γ is a substitution such that $C\sigma\gamma, \mathcal{D}\sigma\gamma \Rightarrow_{\textsf{unit-sup-right}(\epsilon)} \mathcal{E}\gamma$ is ground, then the latter is called a ground instance of $C, \mathcal{D} \Rightarrow_{\textsf{unit-sup-right}(\sigma)} \mathcal{E}$.
- If $C \Rightarrow_{\textsf{ref}(\sigma)} \mathcal{E}$ is a ref inference and γ is a substitution such that $C\sigma\gamma \Rightarrow_{\textsf{ref}(\epsilon)} \mathcal{E}\gamma$ is ground, then the latter is called a ground instance of $C \Rightarrow_{\textsf{ref}(\sigma)} \mathcal{E}$.
- If $C \Rightarrow_{\textsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$ is a split inference and γ is a substitution such that $C\pi \Rightarrow_{\textsf{split}(\epsilon)} A_1\gamma \leftarrow, \ldots, A_m\gamma \leftarrow$ is ground, then the latter inference is a ground instance of the former.
- Let $\mathcal{S}$ be a set of not necessarily ground clauses:
  - A ground inference $C, \mathcal{D} \Rightarrow_{\textsf{sup-left}(\epsilon)} \mathcal{E}$ is *redundant w.r.t.* $\mathcal{S}$ iff $\mathcal{E}$ is redundant w.r.t. $\mathcal{S}_C \cup \{\mathcal{D}\}$.
  - A ground inference $C, \mathcal{D} \Rightarrow_{\textsf{unit-sup-right}(\epsilon)} \mathcal{E}$ is *redundant w.r.t.* $\mathcal{S}$ iff $\mathcal{E}$ is redundant w.r.t. $\mathcal{S}_C \cup \{\mathcal{D}\}$.

  - A ground inference $C \Rightarrow_{\mathsf{ref}(\epsilon)} \mathcal{E}$ is *redundant w.r.t.* $\mathcal{S}$ iff $\mathcal{E}$ is redundant w.r.t. $\mathcal{S}_C$.
  - A ground inference $C \Rightarrow_{\mathsf{split}(\epsilon)} A_1 \leftarrow, \ldots, A_m \leftarrow$ is *redundant w.r.t.* $\mathcal{S}$ iff there is an i with $1 \leq i \leq m$ such that $A_i \leftarrow$ is redundant w.r.t. $\mathcal{S}_C$.

- For all inference rules sup-left, unit-sup-right, ref and split, a (possibly non-ground) inference is redundant with respect to $\mathcal{S}$ iff each of its ground instances is redundant w.r.t. $\mathcal{S}$.

Now it is possible to define the notion of a clause set being *saturated up to redundancy*, which is an important part for to the completeness of the calculus.

Definition 2.2.1 (Saturation up to Redundancy (prelim.))
A clause set $\mathcal{S}$ is *saturated up to redundancy* iff for all clauses $C \in \mathcal{S}$ such that C is not redundant with respect to $\mathcal{S}$ all of the following hold:

1. Every inference $C, \mathcal{D} \Rightarrow_{\mathsf{sup\text{-}left}(\sigma)} \mathcal{E}$,
   where $\mathcal{D}$ is a fresh variant of a positive unit clause from $\mathcal{S}$,
   such that neither $C\sigma$ nor $\mathcal{D}\sigma$ is redundant w.r.t. $\mathcal{S}$, is redundant w.r.t. $\mathcal{S}$.

2. Every inference $C, \mathcal{D} \Rightarrow_{\mathsf{unit\text{-}sup\text{-}right}(\sigma)} \mathcal{E}$,
   where $\mathcal{D}$ is a fresh variant of a positive unit clause from $\mathcal{S}$,
   such that neither $C\sigma$ nor $\mathcal{D}\sigma$ is redundant w.r.t. $\mathcal{S}$, is redundant w.r.t. $\mathcal{S}$.

3. Every inference $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$,
   such that $C\sigma$ is not redundant w.r.t. $\mathcal{S}$, is redundant w.r.t. $\mathcal{S}$.

4. Every inference $C \Rightarrow_{\mathsf{split}(\sigma)} A_1 \leftarrow, \ldots, A_m \leftarrow$,
   such that $C\pi$ is not redundant w.r.t. $\mathcal{S}$, is redundant w.r.t. $\mathcal{S}$.

■

2.2.3 Rules on Tableaux

This section shows how the inference rules, introduced in section 2.2.2, can be used to extend an E-hyper tableau. Additionally, two rules to modify the labelling of existing node are introduced.

If **T** is an E-hyper tableau with a branch **B**, then **T** can be extended by application of the extension rules shown in Figure 2.2. The Equality extension consolidates the three equality inferences and the Split extension applies the split rule to a branch.

The annotation ^{d} marks the derived clauses as *decision clauses*. A node labelled with a decision clause will be referred to as a *decision node*.

In addition to rules that extend an E-hyper tableau, the calculus must provide methods that destructively modify it in order to remove unwanted clauses. Those

$$\textsf{Equality}\ \frac{\mathbf{B}}{\mathbf{B}\cdot\mathcal{E}} \quad \text{if (1-4) holds}$$

$$\textsf{Split}\ \frac{\mathbf{B}}{\mathbf{B}\cdot A_1 \leftarrow^d \quad \ldots \quad \mathbf{B}\cdot A_n \leftarrow^d} \quad \text{if (1,5,6) holds}$$

Conditions:

1. there is a $C \in \mathbf{B}$
2. there is a fresh variant $\mathcal{D}$ of a positive unit clause in $\mathbf{B}$
3. there is a σ such that $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ with $\mathsf{R} \in \{\textsf{sup-left}, \textsf{unit-sup-right}\}$ or $C \Rightarrow_{\textsf{ref}(\sigma)} \mathcal{E}$
4. $\mathbf{B}$ contains no variant of $\mathcal{E}$
5. there is a π such that $C \Rightarrow_{\textsf{split}(\pi)} A_1 \leftarrow, \ldots, A_n \leftarrow$
6. $\mathbf{B}$ contains no variant of $A_i, i \in \{1, \ldots, n\}$

Figure 2.2: Extension rules for E-hyper tableaux.

$$\textsf{Del}\ \frac{\mathbf{B}\cdot C^{(d)}\cdot\mathbf{B}_1\cdot\mathbf{B}_2}{\mathbf{B}\cdot \mathtt{t}\simeq\mathtt{t}^{(d)}\cdot\mathbf{B}_1\cdot\mathbf{B}_2} \quad \text{if (1,2) holds}$$

$$\textsf{Simp}\ \frac{\mathbf{B}\cdot C^{(d)}\cdot\mathbf{B}_1\cdot\mathbf{B}_2}{\mathbf{B}\cdot \mathcal{D}^{(d)}\cdot\mathbf{B}_1\cdot\mathbf{B}_2} \quad \text{if (2-4) holds}$$

Conditions:

1. C is negligible w.r.t. $\mathbf{B}\cdot\mathbf{B}_1$
2. $\mathbf{B}_1$ does not contain a decision clause
3. $\mathbf{B}\cdot C\cdot\mathbf{B}_1 \models_E \mathcal{D}$
4. C is redundant w.r.t. $\mathbf{B}\cdot\mathcal{D}\cdot\mathbf{B}_1$

Figure 2.3: Deletion and simplification rules for E-hyper tableaux.

rules are shown in Figure 2.3. The Del rule (*deletion*) eliminates redundant or non-properly subsumed clauses (or more specifically, it overwrites such a clause with a trivially true unit clause, thus preserving the node while changing its label). The Simp rule (*simplification*) overwrites a clause with one that is smaller according to the term ordering.

The notation $^{(d)}$ indicates that if C is a decision clause, then the resulting new label remains a decision clause.

In both rules, the scope of clauses that may subsume the premise clause C or make it redundant is limited to those above C in the branch $\mathbf{B}$ and those below until the first decision clause. This preserves the soundness of the calculus. A decision clause only occurs after a branch split. If there is a decision clause below C, then C is a member of at least two branches resulting from concatenation of nodes to $\mathbf{B}$ in a Split extension. Only those clauses occurring above the first decision clause

below C are guaranteed to be members of all branches resulting from splits below C, and if any of these clauses make C negligible, then C must be negligible in all branches below C. On the other hand, if a clause exclusively belonging to one branch resulting from a split below C were to be used to overwrite C, then C would have been destructively modified or removed from all the other branches resulting from that split as well, even though C may not have been negligible in any of these.

2.2.4 Derivation

With the notion of an E-hyper tableau and rules to modify E-hyper tableaux, the concept of a *derivation* can now be introduced. A derivation is a series of tableaux that starts with the initial tableau. For all tableaux in this series holds, that they are either the initial tableau or they have been derived by applying one of the introduced rules to a tableau of the series.

A more formal definition is now given in Definition 2.2.2 and then exemplified in Example 2.

Definition 2.2.2 (E-hyper Tableau Derivation (prelim.))
An *E-hyper tableau derivation* of a set $\{C_1, \ldots, C_n\}$ of Σ-clauses is a possibly infinite sequence of tableaux

$$\mathbf{D} = (\mathbf{T}_i)_{0 \leq i < \kappa}$$

such that

1. $\mathbf{T}_0$ is the clausal tableau over Σ that consists of a single branch of length n with the tableau clauses $C_1, \ldots, C_n$, and
2. for all $i > 0$, $\mathbf{T}_i$ is obtained from $\mathbf{T}_{i-1}$ by a single application of the Equality, Split, Del or Simp rule to an open branch in $\mathbf{T}_i$.

■

Example 2 (E-Hyper Tableau Derivation)**.** Figure 2.4 shows the tableaux $\mathbf{T}_0$ and $\mathbf{T}_6$ of an E-hyper tableau derivation. The intermediate tableaux $\mathbf{T}_1$ to $\mathbf{T}_5$ are not shown in the figure but are introduced textual.

$\mathbf{T}_0$ is the initial tableau of the derivation and contains four clauses. As it is easy to see there is an σ such that $(f(x) \simeq x \leftarrow)\sigma = f(c) \simeq c \leftarrow$ holds, namely $\sigma = \{x/c\}$ and therefore $f(c) \simeq c \leftarrow$ is non-properly subsumed by $f(x) \simeq x \leftarrow$ an can be rewritten to $\mathtt{t} \simeq \mathtt{t}$ by an application of the Del rule. This leads to the tableau $\mathbf{T}_1$.

By applying the Equality rule with underlying inference $p(f(a)) \simeq \mathtt{t} \leftarrow, f(x) \simeq x \leftarrow\!\Rightarrow_{\text{unit-sup-right}(\sigma)} p(a) \simeq \mathtt{t} \leftarrow$ and $\sigma = \{x/a\}$ on $\mathbf{T}_1$, $\mathbf{T}_2$ is derived.

For the sake of simplicity, the following steps are given as list of underlying inference rules.

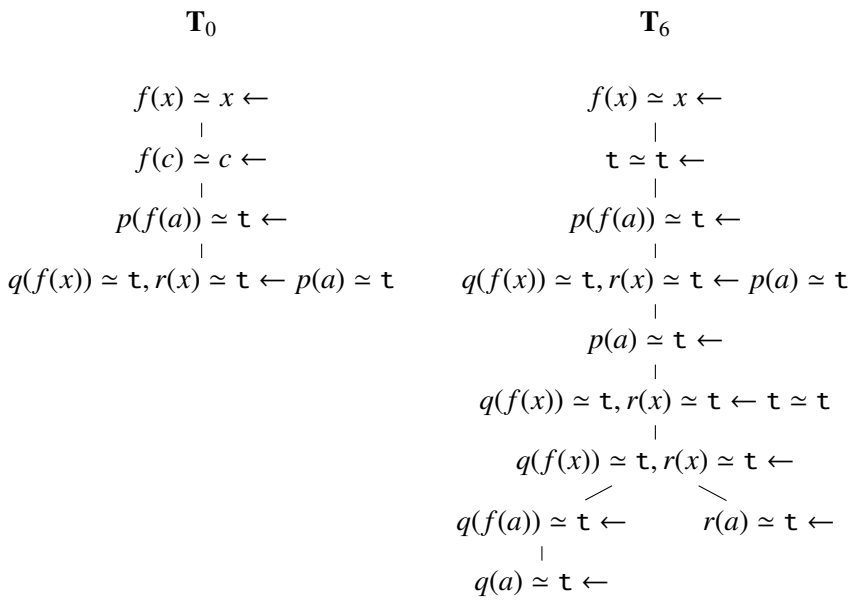


Figure 2.4: The tableaux $\mathbf{T}_0$ and $\mathbf{T}_6$ of an E-hyper tableau derivation.

1. $C, \mathcal{D} \Rightarrow_{\text{sup-left}(\sigma)} \mathcal{E}$, with
   $C = q(f(x)) \simeq \mathtt{t}, r(x) \simeq \mathtt{t} \leftarrow p(a) \simeq \mathtt{t}$
   $\mathcal{D} = p(a) \simeq \mathtt{t} \leftarrow$
   $\sigma = \{\}$
   $\mathcal{E} = q(f(x)) \simeq \mathtt{t}, r(x) \simeq \mathtt{t} \leftarrow \mathtt{t} \simeq \mathtt{t}$
2. $C \Rightarrow_{\text{ref}(\sigma)} \mathcal{E}$, with
   $C = q(f(x)) \simeq \mathtt{t}, r(x) \simeq \mathtt{t} \leftarrow \mathtt{t} \simeq \mathtt{t}$
   $\sigma = \{\}$
   $\mathcal{E} = q(f(x)) \simeq \mathtt{t}, r(x) \simeq \mathtt{t} \leftarrow$
3. $C \Rightarrow_{\text{split}(\pi)} A_1\pi \leftarrow, A_2\pi \leftarrow$, with
   $C = q(f(x)) \simeq \mathtt{t}, r(x) \simeq \mathtt{t} \leftarrow$
   $\pi = \{x/a\}$
   $A_1\pi = q(f(a)) \simeq \mathtt{t}$
   $A_2\pi = r(a) \simeq \mathtt{t}$
4. $C, \mathcal{D} \Rightarrow_{\text{unit-sup-right}(\sigma)} \mathcal{E}$, with
   $C = q(f(a)) \simeq \mathtt{t} \leftarrow$
   $\mathcal{D} = f(x) \simeq x \leftarrow$
   $\sigma = \{\}$
   $\mathcal{E} = q(a) \simeq \mathtt{t} \leftarrow$

These four steps lead from $\mathbf{T}_2$ to $\mathbf{T}_6$. The application of the Del rule is independent and must not have happened as the first step but could have been somewhere between $\mathbf{T}_1$ and $\mathbf{T}_6$. The ordering of the other extension steps is fixed as they need to be applied consecutive.

□

For using the E-hyper tableaux calculus to calculate a model for a formula or to show that a formula is unsatisfiable, we need to define some criterion that states how many derivation steps are necessary to make such a statement. To do so we need some notations and concepts that are introduced now.

The first one is the concept of a *limit tree*. It represents the overall tree structure of the derivation. Its set of nodes is the union of all nodes of all of the derivations tableaux and its set of edges is the union of all edges of all of the derivations

tableaux. If a derivation is finite, the limit tree matches the tree of the last tableau in the derivation. A limit tree is not a tableau as it has no associated labelling function. A more formal definition of limit tree is given now in Definition 2.2.3.

Definition 2.2.3 (Limit Tree)
Let $\mathbf{T} = (\mathbf{t}, \lambda)$ be an E-hyper tableau with $\mathbf{t} = (\mathbb{N}, \mathbb{E})$ a tree consisting of the set of nodes $\mathbb{N}$ and the set of edges $\mathbb{E}$. Furthermore let the derivation $\mathbf{D} = ((\mathbb{N}_i, \mathbb{E}_i), \lambda_i)_{0 \leq i < \kappa}$.

Then the *limit tree* $\mathbf{t}_\infty$ of the derivation $\mathbf{D}$ is defined as:

$$\mathbf{t}_\infty = \left(\bigcup_{0 \leq i < \kappa} \mathbb{N}_i, \bigcup_{0 \leq i < \kappa} \mathbb{E}_i \right)$$

■

The limit tree can now be used to define the set of *persistent clauses*. It is a set of clauses that contains the labels of all the nodes that have not been rewritten by the Del or Simp rule. These clauses are used to generate a model for a given set of formulæ or to show that the set is unsatisfiable. To construct the set of persistent clauses the λ'-function is needed that is defined as follows:

If $\mathbf{N}$ is a node and λ is the labelling function then

$$\lambda'(\mathbf{N}) := \begin{cases} \{\lambda(\mathbf{N})\} & \text{if } \mathbf{N} \in dom(\lambda) \\ \Omega_\Sigma & \text{otherwise} \end{cases}$$

Informally, it can be seen as a wrapper for λ, which creates a set containing the result of λ. If λ is undefined for an input the universal set for the language is returned. This behaviour is needed as we want to build the intersection of all labellings for a node throughout the whole derivation process and it might happen that a λ for a node is not yet defined in a certain derivation steps. The set of persistent clauses is then created by joining those intersections for all nodes of a branch.

The definition of persistent clauses is given in Definition 2.2.4 and illustrated in Example 3.

Definition 2.2.4 (Persistent Clauses)
Let $\mathbf{t}_\infty = (\bigcup_{0 \leq i < \kappa} \mathbb{N}_i, \bigcup_{0 \leq i < \kappa} \mathbb{E}_i)$ be the limit tree of the derivation $\mathbf{D} = ((\mathbb{N}_i, \mathbb{E}_i), \lambda_i)_{0 \leq i < \kappa}$ and $\mathbf{B} = (\mathbf{N}_j)_{0 \leq j < \nu}$ be a (possibly infinite) branch in $\mathbf{t}_\infty$.

Then the set of *persistent clauses (of* $\mathbf{B}$*)* is defined as

$$\mathbf{B}_\infty = \bigcup_{0 \leq i < \nu} \left(\bigcap_{0 \leq j < \kappa} \lambda'_j(\mathbf{N}_i) \right)$$

■

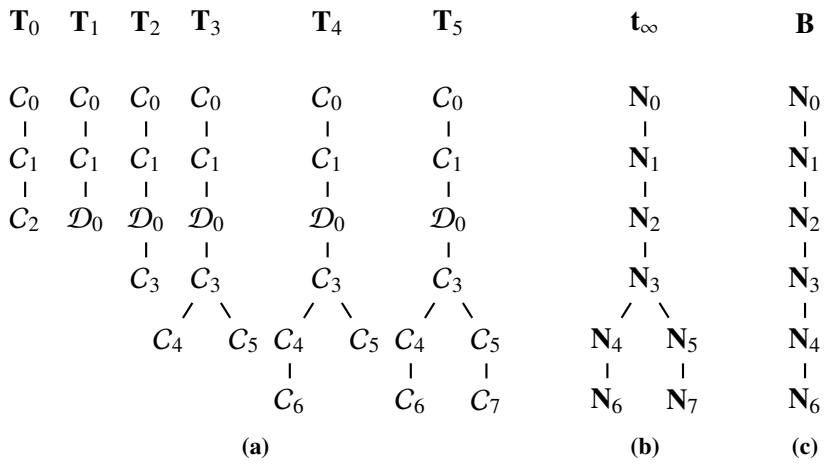


Figure 2.5: The derivation **D**, starting with the initial tableau $\mathbf{T}_0$ (a), its limit tree $\mathbf{t}_\infty$ (b) and a single branch of $\mathbf{t}_\infty$(c).

Example 3 (Limit Trees and Persitent Clauses)**.** In Figure 2.5a a derivation with five tableaux is given. For the sake of brevity neither concrete clauses nor concrete extension rules are given, but it is easy to see that such an derivation is possible.

This derivation has the limit tree $\mathbf{t}_\infty$ shown in Figure 2.5b. As the shown derivation is finite, the limit tree equals the tree of the last tableau of the derivation $\mathbf{T}_5$. In Figure 2.5c the leftmost branch of $\mathbf{t}_\infty$ is given as the branch **B**.

$$
\begin{array}{lllllllllllll}
\lambda'_0(\mathbf{N}_0) & \cap & \lambda'_1(\mathbf{N}_0) & \cap & \lambda'_2(\mathbf{N}_0) & \cap & \lambda'_3(\mathbf{N}_0) & \cap & \lambda'_4(\mathbf{N}_0) & \cap & \lambda'_5(\mathbf{N}_0) & \bigcup \\
\lambda'_0(\mathbf{N}_1) & \cap & \lambda'_1(\mathbf{N}_1) & \cap & \lambda'_2(\mathbf{N}_1) & \cap & \lambda'_3(\mathbf{N}_1) & \cap & \lambda'_4(\mathbf{N}_1) & \cap & \lambda'_5(\mathbf{N}_1) & \bigcup \\
\lambda'_0(\mathbf{N}_2) & \cap & \lambda'_1(\mathbf{N}_2) & \cap & \lambda'_2(\mathbf{N}_2) & \cap & \lambda'_3(\mathbf{N}_2) & \cap & \lambda'_4(\mathbf{N}_2) & \cap & \lambda'_5(\mathbf{N}_2) & \bigcup \\
\lambda'_0(\mathbf{N}_3) & \cap & \lambda'_1(\mathbf{N}_3) & \cap & \lambda'_2(\mathbf{N}_3) & \cap & \lambda'_3(\mathbf{N}_3) & \cap & \lambda'_4(\mathbf{N}_3) & \cap & \lambda'_5(\mathbf{N}_3) & \bigcup \\
\lambda'_0(\mathbf{N}_4) & \cap & \lambda'_1(\mathbf{N}_4) & \cap & \lambda'_2(\mathbf{N}_4) & \cap & \lambda'_3(\mathbf{N}_4) & \cap & \lambda'_4(\mathbf{N}_4) & \cap & \lambda'_5(\mathbf{N}_4) & \bigcup \\
\lambda'_0(\mathbf{N}_6) & \cap & \lambda'_1(\mathbf{N}_6) & \cap & \lambda'_2(\mathbf{N}_6) & \cap & \lambda'_3(\mathbf{N}_6) & \cap & \lambda'_4(\mathbf{N}_6) & \cap & \lambda'_5(\mathbf{N}_6) &
\end{array}
$$

(a)

$$
\begin{array}{lllllllllll}
\{C_0\} & \cap & \{C_0\} & \cap & \{C_0\} & \cap & \{C_0\} & \cap & \{C_0\} & \cap & \{C_0\} & \bigcup \\
\{C_1\} & \cap & \{C_1\} & \cap & \{C_1\} & \cap & \{C_1\} & \cap & \{C_1\} & \cap & \{C_1\} & \bigcup \\
\{C_2\} & \cap & \{\mathcal{D}_0\} & \cap & \{\mathcal{D}_0\} & \cap & \{\mathcal{D}_0\} & \cap & \{\mathcal{D}_0\} & \cap & \{\mathcal{D}_0\} & \bigcup \\
\Omega_\Sigma & \cap & \Omega_\Sigma & \cap & \{C_3\} & \cap & \{C_3\} & \cap & \{C_3\} & \cap & \{C_3\} & \bigcup \\
\Omega_\Sigma & \cap & \Omega_\Sigma & \cap & \Omega_\Sigma & \cap & \{C_4\} & \cap & \{C_4\} & \cap & \{C_4\} & \bigcup \\
\Omega_\Sigma & \cap & \Omega_\Sigma & \cap & \Omega_\Sigma & \cap & \Omega_\Sigma & \cap & \{C_6\} & \cap & \{C_6\} &
\end{array}
$$

(b)

Table 2.1: Abstract (a) and concrete (b) calculation of the set of persistent clauses for the derivation shown in Figure 2.5.

Table 2.1 shows how the set of the persistent clauses is constructed for the branch **B**. In Table 2.1a the definition of the set of persistent clauses was written down, without evaluating the labelling functions. In Table 2.1b λ has been evaluated and the appropriate clauses are used in the formula.

This formula shows that it is possible that some nodes might not have a labelling at a certain derivation step as the tree is not yet enough extended. For example $\mathbf{N}_3$ is introduced in $\mathbf{T}_2$ and therefore $\lambda'_0(\mathbf{N}_3) = \lambda'_0(\mathbf{N}_3) = \Omega_\Sigma$.

The set of persistent clauses for this example is then:

$$\mathbf{B}_\infty = \{C_0\} \cup \{C_1\} \cup \emptyset \cup \{C_3\} \cup \{C_4\} \cup \{C_6\} = \{C_0, C_1, C_3, C_4, C_6\}$$

Neither C_2 nor $\mathcal{D}_0$ are members of $\mathbf{t}_\infty$ which conforms with our intention. C_2 has be rewritten, and therefore it was either redundant or non-properly subsumed and thus was not necessarily needed to construct a model or show the clause sets unsatisfiability. If C_2 was rewritten by the Del rule $\mathcal{D}_0 = \mathtt{t} \simeq \mathtt{t} \leftarrow$, it is easy to see that this clause is not of relevance. If the Simp rule caused the rewrite of C_2, it has been rewritten to a clause $\mathcal{D}_0$ that is smaller w.r.t. $\succ$ and already in the branch and thus $\mathcal{D}_0$ is already in $\mathbf{t}_\infty$.

□

With the concepts of limit trees and persistent clauses defined, the notion of an *exhausted branch* can now be introduced. When a branch is exhausted, all the useful rule applications are done and all following rule applications would not contribute in creating a model. The formal definition is given in Definition 2.2.5.

Definition 2.2.5 (Exhausted Branch (prelim.))
Let $\mathbf{t}_\infty$ be a limit tree, $\mathbf{B} = (\mathbf{N}_k)_{0 \le k < \nu}$ be a branch in $\mathbf{t}_\infty$ and $\mathbf{B}^i$ and $\mathbf{B}^j$ initial segments of $\mathbf{B}$.

The branch $\mathbf{B}$ is *exhausted* iff it does not contain the empty clause, and for every clause $C \in \mathbf{B}_\infty$ and every fresh variant $\mathcal{D}$ of every positive unit clause in $\mathbf{B}_\infty$ such that neither C nor $\mathcal{D}$ is redundant with respect to $\mathbf{B}_\infty$ all of the following hold, for all $i < \nu$ such that $C \in \mathbf{B}^i$ and $\mathcal{D}$ is a variant of a clause in $\mathbf{B}^i$:

1. if Equality is applicable to $\mathbf{B}^i$ with underlying inference
   $C, \mathcal{D} \Rightarrow_{\text{sup-left}(\sigma)} \mathcal{E}$,
   and neither $C\sigma$ nor $\mathcal{D}\sigma$ is redundant w.r.t. $\mathbf{B}^i$,
   then there is a $j < \nu$ such that the inference
   $C, \mathcal{D} \Rightarrow_{\text{sup-left}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}^j$.

2. if Equality is applicable to $\mathbf{B}^i$ with underlying inference
   $C, \mathcal{D} \Rightarrow_{\text{unit-sup-right}(\sigma)} \mathcal{E}$,
   and neither $C\sigma$ nor $\mathcal{D}\sigma$ is redundant w.r.t. $\mathbf{B}^i$,
   then there is a $j < \nu$ such that the inference
   $C, \mathcal{D} \Rightarrow_{\text{unit-sup-right}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}^j$.

3. if Equality is applicable to $\mathbf{B}^i$ with underlying inference
   $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$
   and $C\sigma$ is not redundant w.r.t. $\mathbf{B}^i$,
   then there is a $j < \nu$ such that the inference
   $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}^j$.

4. if Split is applicable to $\mathbf{B}^i$ with underlying inference
   $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$
   and $C\pi$ is not redundant w.r.t. $\mathbf{B}^i$,
   then there is a $j < \nu$ such that the inference
   $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$ is redundant w.r.t. $\mathbf{B}^j$.

■

We are now able to state which kind of derivation is of interest for deriving a model for a set of clauses $\mathcal{S}$ or showing that $\mathcal{S}$ is unsatisfiable, namely a *fair* derivation. A derivation is fair, if it contains a closed tableau or its limit tree has an exhausted branch.

A finite E-hyper tableau derivation of a clause set $\mathcal{S}$ which contains an closed tableau is called *E-hyper tableau refutation of* $\mathcal{S}$.

The concrete connection between an exhausted branch and a model for a clause set is given in section 3.3.

The E-hyper tableau calculus shown in this section is *complete* and *sound*. Proofs for this properties can be found in [12].

This concludes the introduction of the original E-hyper tableau calculus.

Chapter 3

Handling the Unique Name Assumption

3.1 Introduction

The *unique name assumption* (UNA) is a convention on how to handle equality of objects in knowledge bases. It defines that two constants denote the same object iff the constants are identical [21].

To be more flexible and to conform the TPTP's way of dealing with the unique name assumption[27] (see [26] for the latest revision of), the unique name assumption is not assumed to hold for the whole set of constants, but for a subset of it, called the set of *distinct object identifiers*.

Definition 3.1.1 (Distinct Object Identifiers)
The set of *distinct object identifiers* $\mathbb{D} \subseteq \mathbb{C}$ is a subset of all constants.

■

Definition 3.1.2 (Unique Name Assumption)
Let I be an interpretation function.

The *unique name assumption* holds for $\mathbb{D}$ iff
$a \simeq b \Leftrightarrow I(a) = I(b)$ for all $a, b \in \mathbb{D}$.

■

With the introduction of $\mathbb{D}$ some small changes need to be made to some descriptions of the logical language introduced in section 2.1.2. The signature of the language is additionally defined by $\mathbb{D}$, i.e. $\Sigma = (\mathbb{D}, \mathbb{F}, \mathbb{P})$. From $\mathbb{D} \subseteq \mathbb{C}$ follows that $\mathbb{D}$ is disjoint to $\mathbb{V}$ and $\mathbb{P}$. To avoid confusion, the term *basic constants* is introduced to explicitly denote constants that are not distinct object identifiers.

Additionally, the term ordering must fulfil the requirement that all distinct object identifiers are smaller w.r.t. the term ordering than any other non-variable term, i.e. $\forall t \in (\mathbb{T} \setminus (\mathbb{V} \cup \mathbb{D})), \forall i \in \mathbb{D} : i \prec t$.

Instead of modelling the unique name assumption by introducing additional equality-/inequality axioms, the calculus is extended to take care of the needed changes. This needs some effort, but this effort is worthwhile nevertheless as a native handling of the unique name assumption prevents a blow-up of the knowledge base by axioms, and as less axioms entail fewer comparisons in the reasoning process, it leads to a performance increase when dealing with distinct object identifiers [24].

To determine how the calculus has to be changed, a small analysis of the distinct object identifiers' impact on the calculus follows. For most literals/clauses it is obvious that there is no impact on the calculus, therefore only the non obvious combinations shown below are considered.

1. $i \simeq i \leftarrow$
2. $f(i) \simeq f(i) \leftarrow$
3. $i \simeq j \leftarrow$
4. $f(i) \simeq f(j) \leftarrow$
5. $\leftarrow i \simeq i$
6. $\leftarrow f(i) \simeq f(i)$
7. $\leftarrow i \simeq j$
8. $\leftarrow f(i) \simeq f(j)$

Only positive and negative unit clauses are shown, as the non-unit clauses can be broken down to unit clauses by using the ordinary calculus rules. f is a function symbol and i and j are two not identical distinct object identifiers, i.e $i, j \in \mathbb{D}, i \neq j$ and $f \in \mathbb{F}$.

For 1.) $i \simeq i \leftarrow$ and 2.) $f(i) \simeq f(i) \leftarrow$ the calculus needs not to be changed as these positive units are redundant and therefore rewritten to $\mathtt{t} \simeq \mathtt{t}$. 3.) $i \simeq j \leftarrow$ is obviously a contradiction and clauses of this form will be called *unit contradiction*. To cope with unit contradictions the calculus needs to be extended. Although 4.) $f(i) \simeq f(j) \leftarrow$ looks like a unit contradiction as well, it is not as the interpretation of f might make the equation true. Therefore this clause is handled like any positive unit.

For 5.) $\leftarrow i \simeq i$ and 6.) $\leftarrow f(i) \simeq f(i)$ the calculus needs not to be changed as these clauses can be dealt with by the ref rule. 7.) $\leftarrow i \simeq j$ is obviously a tautology which can never contribute to show an inconsistency of the clause set. As unit, this does not have an impact on the reasoning, but if a clause looks like $\mathcal{A} \leftarrow i \simeq j, \mathcal{B}$ with $\mathcal{A}$ and $\mathcal{B}$ not empty, literals from $\mathcal{A}$ or $\mathcal{B}$ might be used in the reasoning which would be useless as those literal cannot contribute to show an inconsistency. For the sake of efficiency the calculus needs to be changed to ignore clauses of the form $\mathcal{A} \leftarrow i \simeq j, \mathcal{B}$ which are called *object tautology clauses*. Although 8.) $\leftarrow f(i) \simeq f(j)$ looks like a object tautology clause as well, it is not as the interpretation of f might make the equation true. Therefore no special treatment is needed.

To summarize, the unique name assumption leads to two different kind of clauses that need an adoption of the calculus. The first one is the unit contradiction $i \simeq j \leftarrow$ and the second one is the object tautology clause $\mathcal{A} \leftarrow i \simeq j, \mathcal{B}$.

This adoption is shown in the following section and then illustrated by a derivation example. In section 3.3 it is shown that the modified version of the calculus is still sound and complete.

3.2 Extending the Calculus

The extension of the E-hyper tableau calculus being introduced in this section is an adoption of the work of Schulz and Bonacina who introduced a way of handling distinct object identifiers in the superposition calculus in [24].

In the first step a way of dealing with object tautology clauses, i.e. clauses of form $\mathcal{A} \leftarrow i \simeq j, \mathcal{B}$ with $i \neq j$ is introduced.

Definition 3.2.1 (Object Tautology Clause)
Let $\mathcal{A}$ be a possibly empty set of head literals, $\mathcal{B}$ be a possibly empty set of body literals and $i, j \in \mathbb{D}$ with $i \neq j$.

A clause $\mathcal{D}$ is an *object tautology clause* iff is is of form $\mathcal{A} \leftarrow i \simeq j, \mathcal{B}$.

■

As mentioned in section 3.1 an object tautology clause cannot contribute in closing a branch. Therefore, it is useless for the refutation attempt and should not be used as a constituent clause for any of the calculus' extension rules. This behaviour can be achieved by extending Definition 2.1.3 to Definition 3.2.2 and including object tautology clauses into the set of of negligible clauses.

Definition 3.2.2 (Negligible Clauses)
Let $\mathcal{D}$ be a clause and $\mathcal{S}$ a set of clauses.

A clause $\mathcal{D}$ is negligible (w.r.t. a clause set $\mathcal{S}$) iff at least one of the following holds:

- $\mathcal{D}$ is redundant (w.r.t. $\mathcal{S}$).
- $\mathcal{D}$ is non-properly subsumed (w.r.t. $\mathcal{S}$).
- $\mathcal{D}$ is an object tautology clause.

■

With the new definition of negligible clauses it is properly taken care of the object tautology clauses as they are rewritten by the Del rule and therefore prevent inefficient reasoning steps.

The second change concerns the handling of unit contradiction clauses, i.e. clauses that look like $i \simeq j \leftarrow$ where i and j are two non-identical distinct object identifiers.

$$\textsf{unit-cont-right}(\sigma)\ \frac{X \simeq Y \leftarrow}{\square} \quad \text{if (1,2) holds}$$

Conditions:

1. $X, Y \in \mathbb{V} \cup \mathbb{D}$
2. $(X \simeq Y \leftarrow)\sigma = i \simeq j \leftarrow$ where $i, j \in \mathbb{D}$ and $i \neq j$

$$\textsf{Inc}\ \frac{\mathbf{B}}{\mathbf{B} \cdot \square} \quad \text{if (1,2) holds}$$

Conditions:

1. there is a $C \in \mathbf{B}$
2. there is a σ such that $C \Rightarrow_{\textsf{unit-cont-right}(\sigma)} \square$

Figure 3.1: The unit-cont-right rule and the corresponding Inc extension rule for handling unit contradictions.

Definition 3.2.3 (Unit Contradiction)
Let $i, j \in \mathbb{D}$ with $i \neq j$.

A clause $\mathcal{D}$ is an *unit contradiction* iff it is of form $i \simeq j \leftarrow$.

■

As mentioned in section 3.1 a unit contradiction is a sign of an inconsistency in the clause set. Therefore a new inference and a new extension rule is needed to close a branch if it contains a unit contradiction. Figure 3.1 shows the newly introduced unit-cont-right (*unit contradiction right*) inference rule and the corresponding extension rule Inc (*Inconsistency*).

With the introduced changes some of the definitions in sections 2.2.2 and 2.2.4 need to be extended.

We start by extending Definition 2.2.1 on page 11 (saturation up to redundancy) to Definition 3.2.4 by adding that a set $\mathcal{S}$ is not saturated up to redundancy if the unit-cont-right is applicable to any clause in $\mathcal{S}$. Additionally it is amended that object tautology clauses are not considered as constituent clauses for inference rules.

Definition 3.2.4 (Saturation up to Redundancy)
A clause set $\mathcal{S}$ is *saturated up to redundancy* iff for all clauses $C \in \mathcal{S}$ such that C is neither an object tautology nor redundant with respect to $\mathcal{S}$ all of the following hold:

1. Every inference $C, \mathcal{D} \Rightarrow_{\textsf{sup-left}(\sigma)} \mathcal{E}$,
   where $\mathcal{D}$ is a fresh variant of a positive unit clause from $\mathcal{S}$,
   such that neither $C\sigma$ nor $\mathcal{D}\sigma$ is an object tautology or redundant w.r.t. $\mathcal{S}$, is redundant w.r.t. $\mathcal{S}$.

2. Every inference $C, \mathcal{D} \Rightarrow_{\text{unit-sup-right}(\sigma)} \mathcal{E}$,
   where $\mathcal{D}$ is a fresh variant of a positive unit clause from $\mathcal{S}$,
   such that neither $C\sigma$ nor $\mathcal{D}\sigma$ is an object tautology or redundant w.r.t. $\mathcal{S}$, is redundant w.r.t. $\mathcal{S}$.

3. Every inference $C \Rightarrow_{\text{ref}(\sigma)} \mathcal{E}$,
   such that $C\sigma$ is neither an object tautology nor redundant w.r.t. $\mathcal{S}$, is redundant w.r.t. $\mathcal{S}$.

4. Every inference $C \Rightarrow_{\text{split}(\sigma)} A_1 \leftarrow, \ldots, A_m \leftarrow$,
   such that $C\pi$ is neither an object tautology nor redundant w.r.t. $\mathcal{S}$, is redundant w.r.t. $\mathcal{S}$.

5. No inference $C \Rightarrow_{\text{unit-cont-right}(\sigma)} \square$ is applicable.

■

In the next step Definition 2.2.2 on page 13 E-hyper tableau derivation is extended to Definition 3.2.5 by including that the Inc rule can be used on an open branch of a tableau $\mathbf{T}$ to extend it.

Definition 3.2.5 (E-Hyper Tableau Derivation)
An *E-hyper tableau derivation* of a set $\{C_1, \ldots, C_n\}$ of Σ-clauses is a possibly infinite sequence of tableaux $\mathbf{D} = (\mathbf{T}_i)_{0 \leq i < \kappa}$ such that

1. $\mathbf{T}_0$ is the clausal tableau over Σ that consists of a single branch of length n with the tableau clauses $C_1, \ldots, C_n$, and

2. for all $i > 0$, $\mathbf{T}_i$ is obtained from $\mathbf{T}_{i-1}$ by a single application of the Equality, Split, Del, Simp or Inc rule to an open branch in $\mathbf{T}_i$.

■

The last definition that needs to be extended is the Definition 2.2.5 on page 17 (exhausted branch) that is extended to Definition 3.2.6 by adding that a branch is not exhausted if the Inc can be applied to a clause in this branch. Additionally, it is amended that object tautology clauses are not considered as constituent clauses for inference rules.

Definition 3.2.6 (Exhausted Branch)
Let $\mathbf{t}_\infty$ be a limit tree, $\mathbf{B} = (\mathbf{N}_k)_{0 \leq k < \nu}$ be a branch in $\mathbf{t}_\infty$ and $\mathbf{B}^i$ and $\mathbf{B}^j$ initial segments of $\mathbf{B}$.

The branch $\mathbf{B}$ is *exhausted* iff it does not contain the empty clause, and for every clause $C \in \mathbf{B}_\infty$ and every fresh variant $\mathcal{D}$ of every positive unit clause in $\mathbf{B}_\infty$ such that neither C nor $\mathcal{D}$ is an object tautology or redundant with respect to $\mathbf{B}_\infty$ all of the following hold, for all $i < \nu$ such that $C \in \mathbf{B}^i$ and $\mathcal{D}$ is a variant of a clause in $\mathbf{B}^i$:

1. if Equality is applicable to $\mathbf{B}^i$ with underlying inference
   $C, \mathcal{D} \Rightarrow_{\text{sup-left}(\sigma)} \mathcal{E}$,
   and neither $C\sigma$ nor $\mathcal{D}\sigma$ is an object tautology or redundant w.r.t. $\mathbf{B}^i$,
   then there is a $j < \nu$ such that the inference
   $C, \mathcal{D} \Rightarrow_{\text{sup-left}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}^j$.

2. if Equality is applicable to $\mathbf{B}^i$ with underlying inference
   $C, \mathcal{D} \Rightarrow_{\text{unit-sup-right}(\sigma)} \mathcal{E}$,
   and neither $C\sigma$ nor $\mathcal{D}\sigma$ is an object tautology or redundant w.r.t. $\mathbf{B}^i$,
   then there is a $j < \nu$ such that the inference
   $C, \mathcal{D} \Rightarrow_{\text{unit-sup-right}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}^j$.

3. if Equality is applicable to $\mathbf{B}^i$ with underlying inference
   $C \Rightarrow_{\text{ref}(\sigma)} \mathcal{E}$
   and $C\sigma$ is neither an object tautology nor redundant w.r.t. $\mathbf{B}^i$,
   then there is a $j < \nu$ such that the inference
   $C \Rightarrow_{\text{ref}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}^j$.

4. if Split is applicable to $\mathbf{B}^i$ with underlying inference
   $C \Rightarrow_{\text{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$
   and $C\pi$ is neither an object tautology nor redundant w.r.t. $\mathbf{B}^i$,
   then there is a $j < \nu$ such that the inference
   $C \Rightarrow_{\text{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$ is redundant w.r.t. $\mathbf{B}^j$.

5. Inc is not applicable to $\mathbf{B}^i$ with underlying inference
   $C \Rightarrow_{\text{unit-cont-right}(\sigma)} \square$.

■

Those introduced changes allow the E-hyper tableau calculus to deal with distinct object identifiers in a complete and sound way. Before the completeness and soundness is formally proven in section 3.3 a derivation example is given to illustrate the introduced changes and their function.

Example 4 (Derivation with the modified E-hyper tableau calculus)**.** Figure 3.2 shows the tableaux $\mathbf{T}_0$ and $\mathbf{T}_5$ of an E-hyper tableau derivation with distinct object identifiers. The intermediate tableaux $\mathbf{T}_1$ to $\mathbf{T}_4$ are not shown in the figure but are introduced textual.

$\mathbf{T}_0$ is the initial tableau of the derivation and contains three clauses. As there are two distinct objects identifiers i, j as terms in the set of clauses, it makes sense to use the modified version of the calculus. It is easy to see that $p(x) \simeq \mathtt{t}, q(f(x)) \simeq \mathtt{t} \leftarrow i \simeq j, r(g(a)) \simeq \mathtt{t}$ is an object tautology clause as it contains $i \simeq j$ on the right side and thus the clause can be rewritten to $\mathtt{t} \simeq \mathtt{t}$ by an application of the Del rule. This leads to the tableau $\mathbf{T}_1$.

For the sake of simplicity the following steps are given as list of underlying inference rules instead of showing all the tableaux.

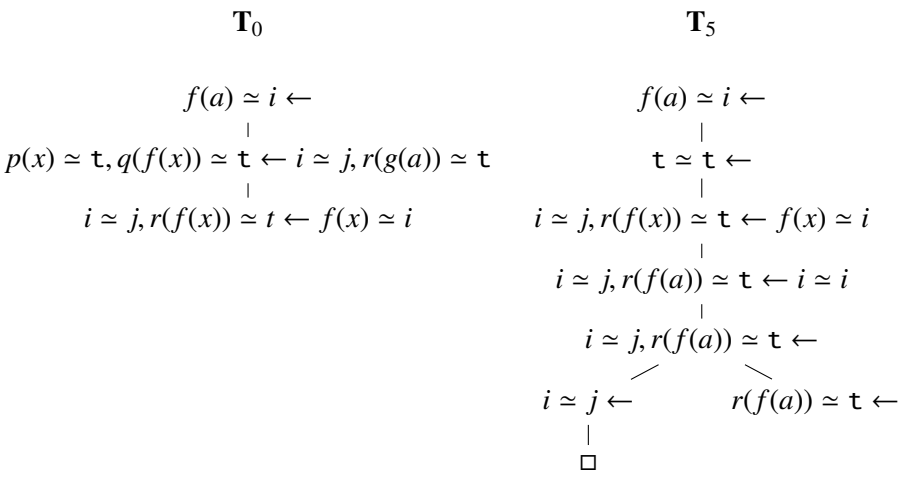


Figure 3.2: The tableaux $\mathbf{T}_0$ and $\mathbf{T}_5$ of an E-hyper tableau derivation with DOI.

1. $C, \mathcal{D} \Rightarrow_{\text{sup-left}(\sigma)} \mathcal{E}$, with
   $C = i \simeq j, r(f(x)) \simeq t \leftarrow f(x) \simeq i$
   $\mathcal{D} = f(a) \simeq i \leftarrow$
   $\sigma = \{x/a\}$
   $\mathcal{E} = i \simeq j, r(f(a)) \simeq \mathtt{t} \leftarrow i \simeq i$
2. $C \Rightarrow_{\text{ref}(\sigma)} \mathcal{E}$, with
   $C = i \simeq j, r(f(a)) \simeq \mathtt{t} \leftarrow i \simeq i$
   $\sigma = \{\}$
   $\mathcal{E} = i \simeq j, r(f(a)) \simeq \mathtt{t} \leftarrow$
3. $C \Rightarrow_{\text{split}(\pi)} A_1\pi \leftarrow, A_2\pi \leftarrow$, with
   $C = i \simeq j, r(f(a)) \simeq \mathtt{t} \leftarrow$
   $\pi = \{\}$
   $A_1\pi = i \simeq j \leftarrow$
   $A_2\pi = r(f(a)) \simeq \mathtt{t} \leftarrow$
4. $C \Rightarrow_{\text{unit-cont-right}(\sigma)} \square$, with
   $C = i \simeq j \leftarrow$
   $\sigma = \{\}$

These for steps lead from $\mathbf{T}_1$ to $\mathbf{T}_5$. The application of the Del rule is independent and must not have happened as the first step but could have been somewhere between $\mathbf{T}_1$ and $\mathbf{T}_5$. The ordering of the other extension steps is fixed as they need to be applied consecutive.

$\square$

3.3 Properties

3.3.1 Overview

This section shows that the modified E-hyper tableau calculus is still sound and complete. The basis for the proofs is taken from [12] and adapted accordingly.

There are four main parts in this section. In the first part the concept of term rewriting systems is introduced, which are used as a means to construct a model for a set of clauses.

The details how to construct a model for a given set of clauses are given in the second part (Proposition 3). It leads to the result that the shown procedure is complete, i.e. if a clause set has a model the method can derive it (Theorem 4).

So far, the actual E-hyper tableau calculus has not been involved as only sets of clauses were considered. In part three, we show that if the set of persistent clauses has a model, the set of clauses of the initial tableau is satisfiable, i.e. the E-hyper tableau calculus is complete (Theorem 13). This is done by showing that the set of persistent clauses for an exhausted branch is saturated up to redundancy (Proposition 11). This enables us to use the result of the previous part, i.e. a rewriting system can be a model for a set of clauses.

The fourth and last part is a straight forward proof for the soundness of the calculus (Theorem 16).

3.3.2 Rewrite Systems

This section gives a rough introduction on rewrite systems as only concepts needed for the following proofs are mentioned. More details on term rewriting can be found in [3, 13, 17].

Given is a logic $\mathfrak{L}$ with the signature Σ. A *rewrite system* is a possibly infinite set of rewrite rules where a *rewrite rule* is an expression of the form $l \rightsquigarrow r$ with l and r Σ-terms. If a rewrite system only works on ground terms it's called a *ground rewrite system*. A ground rewrite system $\mathtt{R}$ with $l \succ r$, for every rule $l \rightsquigarrow r \in \mathtt{R}$ is called *ordered rewrite system*. We assume $\succ$ to be the term ordering introduced in section 2.1.5.

If the rewrite system does not contain two different rules of the forms $l \rightsquigarrow r$ and $s[l] \rightsquigarrow t$, i.e. no left hand side of a rule can be rewritten by another rule, it is *lhs-irreducible*.

A term rewriting system is called *confluent* iff all of its terms are confluent. A term s is confluent iff the following holds: If there are rewriting rules $s \rightsquigarrow^* u$ and $s \rightsquigarrow^* v$ then there are rules such that $u \rightsquigarrow^* t$ and $v \rightsquigarrow^* t$, i.e. different ways of rewriting s finally yield the same result. This property is also called *Church-Rosser*.

If a ordered ground rewrite system is lhs-irreducible it is a *convergent* ground rewrite system, i.e. it is confluent and does terminate. For two given Σ-terms s and t and a convergent rewrite system $\mathtt{R}$, $\mathtt{R} \models_E s \simeq t$ holds iff there is exactly one Σ-term u such that $s \rightsquigarrow_{\mathtt{R}}^* u$ and $t \rightsquigarrow_{\mathtt{R}}^* u$, where $\mathtt{R} \models_E \mathcal{S}$ denotes that the interpretation $\{l \simeq r \mid l \rightsquigarrow r \in \mathtt{R}\}$ satisfies $\mathcal{S}$.

If $s \rightsquigarrow_{\mathtt{R}}^* u$ and $t \rightsquigarrow_{\mathtt{R}}^* u$, s and t are *joinable* by $\mathtt{R}$.

If $s \rightsquigarrow_{\mathtt{R}}^* t$ and there is no rewrite rule with left hand side t, t is called the *(*$\mathtt{R}$*-)normalform* of s.

In the following section $\mathtt{R}$ and its variations will always denote a ground lhs-irreducible rewrite system.

One way of proving that is often used in this section is well-founded induction on the order of the terms. As the base cases are obvious in the most cases, they are not shown.

3.3.3 Model Construction

This section shows how a given set of clauses $\mathcal{S}$ induces a ground lhs-irreducible rewrite system $\mathrm{R}_{\mathcal{S}}$ and how the rewrite system can be used to show the satisfiability of $\mathcal{S}$. We assume that $\mathcal{S}$ does not contain unit contradictions. It is safe to make this assumption for our purpose as later on the set of persistent clauses will be the set we are examining and by definition of an exhausted branch, this set must not contain unit contradictions. The main idea is to interpret a ground positive unit clause as a rewrite rule, i.e. $l \simeq r$ and $r \simeq l$ can be seen as the rewrite rule $l \rightsquigarrow r$ if $l \succ r$ or as $r \rightsquigarrow l$ if $r \succ l$.

The first step is to define how the rewrite system and thus the potential model is constructed (see Definition 3.3.1). Then we introduce and prove Lemma 1 to show that if a rewrite system is a model for a clause then the extension of that rewrite system is still a model for that clause. Lemma 2 then claims the fact that the rewrite system for a clause is an extension of the rewrite system for a smaller (w.r.t. $\succ$) clause. Proposition 3 claims the model construction abilities of the introduced method. It is needed in the proof of Theorem 4 that states, that if a clause set is saturated up to redundancy and does not contain the empty clause, this clause set is satisfiable.

The rewrite system $\mathrm{R}_{\mathcal{S}}$ for $\mathcal{S}$ can be developed by using induction on the term ordering to create the rewrite rules for all clauses of $\mathcal{S}$ starting with the smallest one. Each clause $C \in \mathcal{S}$ has two sets of rewrite rules associated with it where R_C contains the rewrite rules of all clauses that are smaller than C and ϵ_C contains the rewrite rule for C itself if it is a positive unit or is empty otherwise. Therefore $\mathrm{R}_{\mathcal{S}}$ can be derived by joining all R_C for all ground Σ-clauses C of $\mathcal{S}$.

Before a more formal definition is given in Definition 3.3.1, the notion $\bigcup_{C \succ \mathcal{D}} \epsilon_{\mathcal{D}}$ is introduced that denotes the union of $\epsilon_{\mathcal{D}}$ for all clauses $\mathcal{D}$ such that $C \succ \mathcal{D}$ holds.

Definition 3.3.1 (Rewrite System Construction)
Let $\mathcal{S}$ be a set of clauses and $C \in \mathcal{S}$ a clause.

- $\epsilon_C = \begin{cases} \{l \rightsquigarrow r\} & \text{if } C = l \simeq r \leftarrow \text{ is a ground instance of a positive unit in } \mathcal{S}, \\ & l \succ r, \text{ and l is irreducible w.r.t. } \mathrm{R}_C \\ \emptyset & \text{otherwise} \end{cases}$
- $\mathrm{R}_C = \bigcup_{C \succ \mathcal{D}} \epsilon_{\mathcal{D}}$.
- $\mathrm{R}_{\mathcal{S}} = \bigcup_{C \in \mathcal{S}} \epsilon_C$

■

For future use, it is useful to know that a superset of a rewrite systems satisfies at least all the clauses that are satisfied by the original rewrite system. This property is formalized in Lemma 1 and its validity is shown by the following proof.

Lemma 1
Let $\mathcal{S}$ be a clause set, $C = \mathcal{A} \leftarrow \mathcal{B} \in \mathcal{S}$ a ground clause and $\mathtt{R}$ and $\mathtt{R}'$ rewrite systems such that $\mathtt{R}_C \subseteq \mathtt{R} \subseteq \mathtt{R}' \subseteq \mathtt{R}_\mathcal{S}$ holds.

If $\mathtt{R} \models_E C$ then $\mathtt{R}' \models_E C$.

■

Proof. Suppose $\mathtt{R} \models_E C$ holds. Let A be a head literal of $\mathcal{A}$.

We now prove that if $\mathtt{R} \models_E \mathcal{A} \leftarrow \mathcal{B}$ holds $\mathtt{R}' \models_E \mathcal{A} \leftarrow \mathcal{B}$ holds as well. The main idea is to examine two different cases where in the first we assume $\mathtt{R} \models_E \mathcal{B}$ and in the second $\mathtt{R} \not\models_E \mathcal{B}$.

Case 1 ($\mathtt{R} \models_E \mathcal{B}$)

Suppose $\mathtt{R} \models_E \mathcal{B}$. With $\mathtt{R} \models_E \mathcal{B}$ and $\mathtt{R} \models_E \mathcal{A} \leftarrow \mathcal{B}$, $\mathtt{R} \models_E A$ must hold. As first order logic with equality is monotonous and $\mathtt{R} \subseteq \mathtt{R}'$, $\mathtt{R}' \models_E A$ holds and thus $\mathtt{R}' \models_E \mathcal{A} \leftarrow \mathcal{B}$.

Case 2 ($\mathtt{R} \not\models_E \mathcal{B}$)

Suppose $\mathtt{R} \not\models_E \mathcal{B}$. This case is proven by contradiction therefore we assume $\mathtt{R}' \models_E \mathcal{B}$ and $\mathtt{R}' \not\models_E A$ for any A in $\mathcal{A}$ which leads to $\mathtt{R}' \not\models_E \mathcal{A} \leftarrow \mathcal{B}$. For $\mathtt{R}' \models_E \mathcal{B}$ and $\mathtt{R} \not\models_E \mathcal{B}$ there must be a $B = s \simeq t$ in $\mathcal{B}$ such that $\mathtt{R}' \models_E B$ but $\mathtt{R} \not\models_E B$. In other words $s \simeq t$ is joinable by $\mathtt{R}'$ but not by $\mathtt{R}$, i.e. there are rewrite rules in $(\mathtt{R}' \setminus \mathtt{R})$ to rewrite s and t to a common normalform u.

Every rule $l \rightsquigarrow r \in \mathtt{R}_\mathcal{S}$ is obtained from a ground instance of $l \simeq r \leftarrow\in \mathcal{S}$. As we assume $l \rightsquigarrow r \in (\mathtt{R}' \setminus \mathtt{R})$ and $\mathtt{R}_C \subseteq \mathtt{R}$ it follows $l \rightsquigarrow r \notin \mathtt{R}_C$. By definition of $\mathtt{R}_C$ (see Definition 3.3.1) for all clauses $\mathcal{D}$ that are not in $\mathtt{R}_\mathcal{S}$, $\mathcal{D} \geq C$ holds. As C looks like $A_1, \ldots, A_n \leftarrow s \simeq t, B_1, \ldots, B_m$, it entails $C \neq l \simeq r \leftarrow$ and thus $l \simeq r \leftarrow\!\!> C$ holds.

By the definition of the reduction ordering (see Definition 2.1.5) it follows that $l \rightsquigarrow r$ cannot be used for rewriting s or t. As no rule in $(\mathtt{R}' \setminus \mathtt{R})$ can be used to rewrite s and t the $\mathtt{R}'$- and $\mathtt{R}$-normalforms of s and t are the same and therefore $\mathtt{R}' \models_E s \simeq t$ and $\mathtt{R} \models_E s \simeq t$ holds.

As this is a contradiction to our assumption that $\mathtt{R}' \models_E \mathcal{B}$ holds but $\mathtt{R}' \models_E A$ does not, this entails $\mathtt{R}' \not\models_E \mathcal{B}$ or $\mathtt{R}' \models_E A$. Thus $\mathtt{R}' \models_E \mathcal{A} \leftarrow \mathcal{B}$.

□

At this point it is useful to state that the rewrite rules of a clause $\mathcal{D}$ are a subset of the rewrite rules for clause C that is larger than $\mathcal{D}$ according to the term ordering. This property is formalized in Lemma 2 and its validity is shown by the following proof which is basically the application of the rewrite system construction rules (see Definition 3.3.1).

Lemma 2
Let $\mathcal{S}$ be a clause set and $C, \mathcal{D} \in \mathcal{S}$ be ground.

If $C \succ \mathcal{D}$ then $\mathrm{R}_{\mathcal{D}} \cup \epsilon_{\mathcal{D}} \subseteq \mathrm{R}_C$

■

Proof. Suppose $C, \mathcal{D} \in \mathcal{S}$ be ground and $C \succ \mathcal{D}$. By definition of the construction procedure for the rewrite system (see Definition 3.3.1) $\mathrm{R}_C = \bigcup_{C \succ \mathcal{E}} \epsilon_{\mathcal{E}}$ and $\mathrm{R}_{\mathcal{D}} = \bigcup_{\mathcal{D} \succ \mathcal{E}} \epsilon_{\mathcal{E}}$.

With $C \succ \mathcal{D}$ it follows $\epsilon_{\mathcal{D}} \subseteq \mathrm{R}_C$ and $\mathrm{R}_{\mathcal{D}} \subseteq \mathrm{R}_C$ which in combination entails $\mathrm{R}_{\mathcal{D}} \cup \epsilon_{\mathcal{D}} \subseteq \mathrm{R}_C$.

□

The following proposition (see Proposition 3) is the core element of this part. It states that $\mathrm{R}_C \cup \epsilon_C \models_E C$. Additionally, it states that a clause C is either redundant to a set of clauses $\mathcal{S}$ and R_C already satisfies C or else when C is not redundant w.r.t. $\mathcal{S}$, extension of R_C by ϵ_C will satisfy C.

Before we can introduce the proposition we need to introduce the notion $\mathcal{S}_C$ that is defined as follows. For a clause set $\mathcal{S}$, a clause $C \in \mathcal{S}$ and a grounding substitution γ, $\mathcal{S}_C$ is the set of all ground clauses that are smaller than C, i.e. $\mathcal{S}_C = \{\mathcal{D} \in \mathcal{S} \mid C \succ \mathcal{D}\gamma\}$.

As the proposition has two cases, the proof is done in two cases. The first one is just a straight forward approach by using induction on the ordering of the clauses. The second case takes different forms of clauses into account and either shows by contradiction that this kind clause can not appear, or that the property holds.

Proposition 3 (Model Construction)
Let $\mathcal{S}$ be a clause set that is saturated up to redundancy and such that $\square \notin \mathcal{S}$.

Then for every ground instance C of every clause from $\mathcal{S}$ the following holds:

1. If $\mathcal{S}_C \models_E C$ then $\epsilon_C = \emptyset$ and $\mathrm{R}_C \models_E C$
2. If $\mathcal{S}_C \not\models_E C$ then $\mathrm{R}_C \cup \epsilon_C \models_E C$

■

Proof. The propositions' two cases are proven separately by combining well-founded induction on the ground instances of $\mathcal{S}$ and contradiction. For the induction we choose a ground clause $C \in \mathcal{S}$ and assume the proposition holds for all ground instances $\mathcal{D} \in \mathcal{S}$ with $C \succ \mathcal{D}$.

Case 1 ($\mathcal{S}_C \models_E C$)

Suppose $\mathcal{S}_C \models_E C$. By combining the conclusions of both of the propositions cases and using well-founded induction on the term ordering $\mathrm{R}_{\mathcal{D}} \cup \epsilon_{\mathcal{D}} \models_E \mathcal{D}$ can

be concluded for every clause $\mathcal{D} \in \mathcal{S}_C$ with $C \succ \mathcal{D}$. With Lemma 2 $\mathrm{R}_{\mathcal{D}} \cup \epsilon_{\mathcal{D}} \subseteq \mathrm{R}_C$ holds which leads to $\mathrm{R}_C \models_E \mathcal{D}$ by using Lemma 1. As $\mathrm{R}_C \models_E \mathcal{D}$ holds for all $\mathcal{D} \in \mathcal{S}_C$, $\mathrm{R}_C \models_E \mathcal{S}_C$ holds and with the premisses of this case $\mathcal{S}_C \models_E C$, $\mathrm{R}_C \models_E C$ holds as desired.

As its proven that $\mathcal{S}_C \models_E C$ holds, it remains to be shown that $\epsilon_C = \emptyset$. This is done by trying to derive a contradiction. Therefore we assume $\epsilon_C = \{l \rightsquigarrow r\}$ with $C = l \simeq r \leftarrow$. For l and r to be equal in the E-interpretation induced by the convergent rewrite system R_C both terms must be joinable i.e. they must have the same normalform w.r.t. R_C. Therefore there must be a rewrite rule in R_C that rewrites l. If l can be rewritten it is not irreducible which is a contradiction to the definition for $\epsilon_C \neq \emptyset$ (see Definition 3.3.1).

Case 2 ($\mathcal{S}_C \not\models_E C$)

Suppose $\mathcal{S}_C \not\models_E C$. This entails that C is not redundant w.r.t. $\mathcal{S}_C$ and therefore not redundant w.r.t. $\mathcal{S}$, and C a ground instance of a clause $\mathcal{E} \in \mathcal{S}$, i.e. $\mathcal{E}\gamma = C$ for a grounding substitution γ, $\mathcal{E}$ cannot be redundant w.r.t. $\mathcal{S}$. The proof of the propositions second case is now done by analysing different structures for the clause C and then try to show that this kind of clause cannot appear or to show that $\mathrm{R}_C \cup \epsilon_C \models_E C$ holds. For trying to derive a contradiction we use $\mathcal{S} \not\models_E \mathcal{E}$ and the definition of saturated up to redundancy (see Definition 3.2.4)

1. $C = (\mathcal{E}[x])\gamma$ and $x\gamma$ is reducible w.r.t.. R_C.

   Suppose $C = \mathcal{E}\gamma$, for some clause $\mathcal{E} \in \mathcal{S}$ and some (grounding) substitution γ and $\mathcal{E}$ contains a variable x, i.e. $\mathcal{E}[x]$. Suppose as well that $x\gamma$ is reducible w.r.t. R_C. We show by contradiction that this case cannot appear.

   If $x\gamma$ is reducible, there must be a rule $l \rightsquigarrow r \in \mathrm{R}_C$ and l must occur in $x\gamma$, i.e. $x\gamma[l]$.

   We now assume a (grounding) substitution γ' that is similar to γ in such a way, that both substitution are identical with the exception that where γ has l, γ' has r, i.e. the rewrite rule $l \rightsquigarrow r$ has been applied. Thus $x\gamma' = x\gamma[r]$. As only larger terms are rewritten, $l \succ r$ holds and therefore $\mathcal{E}\gamma \succ \mathcal{E}\gamma'$ which in combination with the induction hypothesis leads to $\mathrm{R}_{\mathcal{E}\gamma'} \cup \epsilon_{\mathcal{E}\gamma'} \models_E \mathrm{R}_{\mathcal{E}\gamma}$. From $\mathcal{E}\gamma \succ \mathcal{E}\gamma'$ and Lemma 2 follows $\mathrm{R}_{\mathcal{E}\gamma'} \cup \epsilon_{\mathcal{E}\gamma'} \subseteq \mathrm{R}_{\mathcal{E}\gamma}$ which with Lemma 1 leads to $\mathrm{R}_{\mathcal{E}\gamma} \models_E \mathcal{E}\gamma'$.

   Because of $l \rightsquigarrow r \in \mathrm{R}_C$, $\mathcal{E}\gamma = C$ and by definition of γ' conclude with congruence $\mathrm{R}_C \models_E C$ which is a contradiction to $\mathcal{S}_C \not\models_E C$.

2. $C = (\mathcal{A} \leftarrow s \simeq t, \mathcal{B})\gamma$ and $s\gamma = t\gamma$.

   Suppose $C = (\mathcal{A} \leftarrow s \simeq t, \mathcal{B})\gamma$ for some clause $\mathcal{A} \leftarrow s \simeq t, \mathcal{B} \in \mathcal{S}$ and some grounding substitution γ and $s\gamma = t\gamma$ holds. We show by contradiction that this case cannot appear.

If $s\gamma = t\gamma$ there is an inference $\mathcal{A} \leftarrow s \simeq t, \mathcal{B} \Rightarrow_{\mathsf{ref}(\sigma)} (\mathcal{A} \leftarrow \mathcal{B})\sigma$ with σ being the mgu of s and t and part of the grounding substitution γ, i.e. $\gamma = \sigma\delta$ for a substitution δ. By the definition of saturation up to redundancy (see Definition 3.2.4) and the propositions premiss that $\mathcal{S}$ is saturated up to redundancy such an inference is redundant. Therefore the clause $(\mathcal{A} \leftarrow \mathcal{B})\sigma$ is redundant w.r.t. $\mathcal{S}_C$, i.e. $\mathcal{S}_C \models_E (\mathcal{A} \leftarrow \mathcal{B})\sigma$ which trivially entails $\mathcal{S}_C \models_E \mathcal{A} \leftarrow s \simeq t, \mathcal{B}$. As $C = \mathcal{A} \leftarrow s \simeq t, \mathcal{B}$ this is a contradiction to $\mathcal{S}_C \not\models_E C$.

3. $C = (\mathcal{A} \leftarrow s \simeq t, \mathcal{B})\gamma$ with $(s \simeq t)\gamma = i \simeq j$, with $i, j \in \mathbb{D}$ and $i \neq j$.

   Suppose $C = (\mathcal{A} \leftarrow s \simeq t, \mathcal{B})\gamma$ for some clause $\mathcal{A} \leftarrow s \simeq t, \mathcal{B} \in \mathcal{S}$ and some grounding substitution γ and $(s \simeq t)\gamma = i \simeq j$ holds with i and j are two non-identical distinct object identifiers. We show by contradiction that this case cannot appear.

   As i and j are two non-identical members of $\mathbb{D}$ and the unique name assumption applies to $\mathbb{D}$ the equation $i \simeq j$ can never be true. With the false literal $i \simeq j$ in the body of the clause the whole clause becomes true. From this follows that C can be written as $\mathtt{t} \simeq \mathtt{t} \leftarrow$, i.e. $C = \mathtt{t} \simeq \mathtt{t} \leftarrow$.

   As $\mathcal{S}_C \models_E \mathtt{t} \simeq \mathtt{t} \leftarrow$ holds trivially it is a contradiction to $\mathcal{S}_C \not\models_E C$.

4. $C = (\mathcal{A} \leftarrow s \simeq t, \mathcal{B})\gamma$ and $s\gamma \succ t\gamma$ and $s\gamma$ is irreducible w.r.t. $\mathtt{R}_C$.

   Suppose $C = (\mathcal{A} \leftarrow s \simeq t, \mathcal{B})\gamma$ for some clause $\mathcal{A} \leftarrow s \simeq t, \mathcal{B} \in \mathcal{S}$ and some grounding substitution γ and $s\gamma \neq t\gamma$ holds. Furthermore assume that $s\gamma$ is the larger side of the equation $(s \simeq t)\gamma$ i.e. $s\gamma \succ t\gamma$ and that $s\gamma$ is irreducible w.r.t. $\mathtt{R}_C$.

   As $s\gamma$ is irreducible w.r.t. $\mathtt{R}_C$ and therefore $s\gamma$ and $t\gamma$ are not joinable w.r.t. $\mathtt{R}_C$, $\mathtt{R}_C \not\models_E s\gamma \simeq t\gamma$ holds. With $s\gamma \simeq t\gamma$ in the body of the clause C this trivially entails $\mathtt{R}_C \models_E C$ which is one part that was to be shown.

   The second part of the first cases' conclusion holds trivial, as C is not a positive unit and by definition of the rewrite system (see Definition 3.3.1) $\epsilon_C = \emptyset$ holds.

5. $C = (s \simeq t \leftarrow)\gamma$ with $(s \simeq t \leftarrow)\gamma = i \simeq j \leftarrow$, $i, j \in \mathbb{D}$ and $i \neq j$.

   Suppose $C = (s \simeq t \leftarrow)\gamma$ for some positive unit clause $(s \simeq t \leftarrow) \in \mathcal{S}$ and some grounding substitution γ and $(s \simeq t)\gamma = i \simeq j$ holds with i and j are two non-identical distinct object identifiers. We show by contradiction that this case cannot appear.

   If such a C exists the inference $C \Rightarrow_{\mathsf{unit\text{-}cont\text{-}right}(\gamma)} \square$ is applicable. This is clearly a contradiction to the preconditions of the model construction in Proposition 3 as $\mathcal{S}$ is required to be saturated up to redundancy and by case 4 of the definition of saturation up to redundancy 3.2.4 a set where unit-cont-right is applicable is not saturated up to redundancy.

6. $C = (s \simeq t \leftarrow)\gamma$ and $s\gamma \succ t\gamma$ and $s\gamma$ is irreducible w.r.t. R_C.

   Suppose $C = (s \simeq t \leftarrow)\gamma$ for some positive unit clause $(s \simeq t \leftarrow) \in \mathcal{S}$ and some grounding substitution γ and $s\gamma \neq t\gamma$. Furthermore assume that $s\gamma$ is the larger side of the equation $(s \simeq t)\gamma$ i.e. $s\gamma \succ t\gamma$ and that $s\gamma$ is irreducible w.r.t. R_C.

   Thus, by definition of the rewrite system (see Definition 3.3.1) $\epsilon_C = \{s \rightsquigarrow t\}$ which trivially entails $\mathrm{R}_C \cup \epsilon_C \models_E C$

7. $C = (A_1, \ldots, A_m \leftarrow)\gamma$ and $m \geq 2$.

   Suppose $C = \mathcal{E}\gamma$ for some positive non-unit clause $\mathcal{E} = (A_1, \ldots, A_m \leftarrow) \in \mathcal{S}$ and some grounding substitution γ and $m \geq 2$. Furthermore assume $\gamma = \pi\delta$ for a purifying substitution π and some (possibly empty) substitution δ. We show by contradiction that this case cannot appear.

   From the assumption of case 2 follows that C is not redundant w.r.t. $\mathcal{S}$ which entails that $\mathcal{E}$ is not redundant w.r.t. $\mathcal{S}$. By the definition of saturation up to redundancy (see Definition 3.2.4) and the requirement, that $\mathcal{S}$ is saturated up to redundancy conclude that $\mathcal{E} \Rightarrow_{\mathsf{split}(\pi)} A_1\pi \leftarrow, \ldots, A_m\pi \leftarrow$ is redundant w.r.t. $\mathcal{S}$. This entails that in particular its ground instance $C \Rightarrow_{\mathsf{split}(\epsilon)} A_1\gamma \leftarrow, \ldots, A_m\gamma \leftarrow$ is redundant w.r.t. $\mathcal{S}$. By definition of redundancy (see Definition 2.1.1) $\mathcal{S}_C \models_E A_i\gamma$ holds for some i with $1 \leq i \leq m$ which entails $\mathcal{S}_C \models_E C$.

8. $C = (\mathcal{E}[s])\gamma$ and $s\gamma$ is reducible at an non-variable position.

   In this case we need to consider the clauses that do not fall into any of the previous cases. Therefore we analyse which kind of formulæ are not yet treated which lead to two cases:

   (a) $\mathcal{E} = \mathcal{A} \leftarrow s \simeq t, \mathcal{B}$ with $s\gamma \succ t\gamma$, $s\gamma$ is reducible w.r.t. R_C and γ is a grounding substitution.

   (b) $\mathcal{E} = s \simeq t \leftarrow$ with $s\gamma \succ t\gamma$, $s\gamma$ is reducible w.r.t. R_C and γ is a grounding substitution.

   As both cases can be treated in a similar way, we use a shared approach to show by contradiction that both kind of clauses cannot appear.

   As $s\gamma$ is reducible by R_C there must be a rule $l \rightsquigarrow r \in \mathrm{R}_C$ that rewrites $s\gamma$. As case 2.1. already deals with the application of rewrite rules at a variable position we now assume that $s\gamma$ is not rewritten at or below a variable position. More formal $s\gamma[l]_p$ is a non-variable position of s for any p.

   By construction of the rewrite rules (see Definition 3.3.1), the rule $l \rightsquigarrow r$ is obtained from the ground instance of a positive unit clause in $\mathcal{S}$. Let $\mathcal{F} = l' \simeq r' \leftarrow$ be a fresh variant of the appropriate unit clause and assume that γ is extended in such way that $l'\gamma = l$ and $r'\gamma = r$.

As $l \rightsquigarrow r \in \mathtt{R}_C$ and thus $l'\gamma \rightsquigarrow r'\gamma \in \mathtt{R}_C\, C \succ \mathcal{F}\gamma$ must hold. As $C \succ \mathcal{F}\gamma$ and by the induction hypothesis the proposition holds for all clauses smaller (w.r.t. $\succ$) than C, it has to hold for $\mathcal{F}$.

Considering the first case of the proposition i.e. $\mathcal{S}_{\mathcal{F}\gamma} \models_E \mathcal{F}\gamma$ which requires $\epsilon_{\mathcal{F}\gamma} = \emptyset$ and thus $l\gamma \rightsquigarrow r\gamma \notin \mathtt{R}_{\mathcal{S}}$. Therefore this case is not possible and the second case of the proposition i.e. $\mathcal{S}_{\mathcal{F}\gamma} \not\models_E \mathcal{F}\gamma$, stating that $\mathcal{F}\gamma$ is redundant w.r.t. $\mathcal{S}$, must hold.

We now need to treat the two different kind of formulæ separately for one step as follows:

(a) For $\mathcal{E} = \mathcal{A} \leftarrow s \simeq t, \mathcal{B}$ consider the ground sup-left rule

$$(\mathcal{A}\gamma \leftarrow s\gamma[l'\gamma]_p \simeq t\gamma, \mathcal{B}\gamma), \mathcal{F}\gamma \Rightarrow_{\text{sup-left}(\epsilon)} \mathcal{A}\gamma \leftarrow s\gamma[r'\gamma]_p \simeq t\gamma, \mathcal{B}\gamma \tag{3.1}$$

Because p is a position of a non-variable term in s, say, l'', the sup-left inference

$$(\mathcal{A} \leftarrow s[l'']_p \simeq t, \mathcal{B}), \mathcal{F} \Rightarrow_{\text{sup-left}(\sigma)} (\mathcal{A} \leftarrow s[r']_p \simeq t, \mathcal{B})\sigma \tag{3.2}$$

exists, where σ is a mgu of l' and l'' and $\gamma = \sigma\delta$ for some substitution δ. The ground sup-left inference (1) then is a ground inference of the sup-left inference (2).

(b) For $\mathcal{E} = s \simeq t \leftarrow$ consider the ground unit-sup-right rule

$$(s\gamma[l'\gamma]_p \simeq t\gamma \leftarrow), \mathcal{F}\gamma \Rightarrow_{\text{unit-sup-right}(\epsilon)} s\gamma[r'\gamma]_p \simeq t\gamma \leftarrow \tag{3.3}$$

Because p is a position of a non-variable term in s, say, l'', the unit-sup-right inference

$$(s[l'']_p \simeq t \leftarrow), \mathcal{F} \Rightarrow_{\text{unit-sup-right}(\sigma)} (s[r']_p \simeq t \leftarrow)\sigma \tag{3.4}$$

exists, where σ is a mgu of l' and l'' and $\gamma = \sigma\delta$ for some substitution δ. The ground unit-sup-right inference (3) then is a ground inference of the unit-sup-right inference (4).

As we concluded, that $\mathcal{F}\gamma$ is not redundant w.r.t. $\mathcal{S}$, the more general clause $\mathcal{F}\sigma$ can neither be redundant w.r.t. $\mathcal{S}$.

By case 2 of the definition of saturation up to redundancy (see Definition 3.2.4) the inferences (2) and (4) are redundant w.r.t. $\mathcal{S}$ and thus their ground instances (1) and (3) are redundant w.r.t. $\mathcal{S}$ as well.

For the sake of brevity and to treat both cases a) and b) at once we introduce the new clause $\mathcal{G}$ where we assume that it is the conclusion of the inference

(1) or the inference (2). For the following it makes no difference if $\mathcal{G} = \mathcal{A}\gamma \leftarrow s\gamma[r'\gamma]_p \simeq t\gamma, \mathcal{B}\gamma$ or $\mathcal{G} = s\gamma[r'\gamma]_p \simeq t\gamma \leftarrow$ as it holds for both cases.

By definition of redundancy $\mathcal{S}_C \cup \{\mathcal{F}\}\gamma \models_E \mathcal{G}$. By induction over the ordering of the clauses and the combination of both conclusions of the proposition we derive $\mathrm{R}_{\mathcal{H}} \cup re_{\mathcal{H}} \models_E \mathcal{H}$ for every $\mathcal{H} \in \mathcal{S}_C$. In combination with lemma 2 this leads to $\mathrm{R}_{\mathcal{H}} \cup \epsilon_{\mathcal{H}} \subseteq \mathrm{R}_C$ which with lemma 1 leads to $\mathrm{R}_C \models_E \mathcal{H}$, for every clause $\mathcal{H} \in \mathcal{S}_C$. This is equivalent with $\mathrm{R}_C \models_E \mathcal{S}_C$.

As $\mathcal{F} = (l' \simeq r')\gamma$ is present as a rewrite rule $l'\gamma \rightsquigarrow r'\gamma \in \mathrm{R}_C$ thus $l \rightsquigarrow r \in \mathrm{R}_C$, it follows trivially that $\mathrm{R}_C \models_E \mathcal{F}\gamma$. In combination with $\mathrm{R}_C \models_E \mathcal{S}_C$ and $\mathcal{S}_C \cup \{\mathcal{F}\}\gamma \models_E \mathcal{G}$ conclude $\mathrm{R}_C \models_E \mathcal{G}$.

From $l \rightsquigarrow r \in \mathrm{R}_C$ conclude by congruence $\mathrm{R}_C \models_E C$ which in combination with $\mathrm{R}_C \models_E \mathcal{S}_C$ is a contradiction to $\mathcal{S}_C \not\models_E C$.

□

The last step of this part concerns the static completeness (see Theorem 4) that claims, that a clause set, that is saturated up to redundancy and does not contain the empty clause is E-satisfiable. To show that the set is E-satisfiable it suffices to prove that there is a model for this set. This is basically done straight forward by using Proposition 3 to show that there is a model for such a set.

Theorem 4 (Static Completeness)
Let $\mathcal{S}$ be a clause set saturated up to redundancy.

If $\square \notin \mathcal{S}$ then $\mathcal{S}$ is E-satisfiable.

■

Proof. Suppose $\square \notin \mathcal{S}$. For $\mathcal{S}$ to be E-satisfiable there must be an E-Model. We therefore show that $\mathrm{R}_{\mathcal{S}}$ is an E-model for $\mathcal{S}$ by showing that $\mathrm{R}_{\mathcal{S}} \models_E C\gamma$ for an arbitrary chosen clause $C \in \mathcal{S}$ and an arbitrary chosen grounding substitution γ. Proposition 3 leads to $\mathrm{R}_{C\gamma} \cup \epsilon_{C\gamma} \models_E C\gamma$ which with Lemma 1 leads to $\mathrm{R}_{\mathcal{S}} \models_E C\gamma$ what was to be shown.

□

This concludes the second part of this section. So far we have only talked of set of clauses and the E-hyper tableau calculus was not involved. In the following chapter we use the properties gathered so far in relation with the E-hyper tableau calculus rules and properties to prove its completeness.

3.3.4 Completeness

We can now use the results about static completeness to prove Theorem 13 that is the main contribution of this part and states that the modified version of the E-hyper tableau calculus is complete, i.e. if a fair derivation of a set of clauses is not a refutation, then the set of clauses is E-satisfiable.

To prove Theorem 13 by using Theorem 4 we need have a set of clauses that is saturated up to redundancy. Thus Proposition 11 is introduced and proven that states, that the set of persistent clauses of an exhausted branch of an fair derivation is saturated up to redundancy.

Additionally, a couple of lemmas, which are introduced now, are needed for the proves. The first one is Lemma 5 which states that if a clause C is satisfied by the union of the set of clauses of an initial segment of a branch and the set of clauses $\mathcal{S}$, then C is satisfied by the union of $\mathcal{S}$ and the set of persistent clauses of that branch, as well.

Lemma 5
Let C_1 and C_2 be ground clauses, $\mathcal{S}$ a set of ground clauses, $\mathbf{D}$ a derivation, $\mathbf{t}_\infty$ the limit tree of $\mathbf{D}$ and $\mathbf{B}$ a branch of $\mathbf{t}_\infty$. Furthermore let $\mathbf{B}^j$ be the initial segment of $\mathbf{B}$ and $\mathbf{B}_\infty$ the set of persistent clauses for $\mathbf{B}$.

If $(\mathbf{B}^j)_{C_1} \cup \mathcal{S} \models_E C_2$ for some $j < \nu$ then $(\mathbf{B}_\infty)_{C_1} \cup \mathcal{S} \models_E C_2$.

■

Proof. To proof that $(\mathbf{B}_\infty)_{C_1} \cup \mathcal{S} \models_E C_2$ holds we use well-founded induction and assume that the lemma holds for all clauses C_1' with $C_1 \succ C_1'$.

If $(\mathbf{B}^j)_{C_1} \subseteq (\mathbf{B}_\infty)_{C_1}$ then the results follows from the monotonicity of first-order logic with equality. If $(\mathbf{B}^j)_{C_1} \nsubseteq (\mathbf{B}_\infty)_{C_1}$ we can use the compactness of first order-logic with equality to remove the clauses that are in $(\mathbf{B}_\infty)_{C_1}$ but not in $(\mathbf{B}^j)_{C_1}$. Thus we define $(\mathbf{B}^j)_{C_1}$ to be a finite subset of $(\mathbf{B}^j)_{C_1}$ for which the entailment in the lemmas premiss ($(\mathbf{B}^j)_{C_1} \cup \mathcal{S} \models_E C_2$) holds.

Let $\mathbf{B}' = (\mathbf{B}^j)_{C_1} \setminus (\mathbf{B}_\infty)_{C_1}$ be those clauses from $(\mathbf{B}^j)_{C_1}$ that are not an instance of any persisting clause in $(\mathbf{B}_\infty)_{C_1}$. We now choose a $C' \in \mathbf{B}'$ which by construction is a ground clause of $(\mathbf{B}^j)_{C_1}$ that is not in $(\mathbf{B}_\infty)_{C_1}$, i.e. $C' \in (\mathbf{B}^j)_{C_1}$ and $C' \notin (\mathbf{B}_\infty)_{C_1}$.

If $C' \in (\mathbf{B}^j)_{C_1}$ but $C' \notin (\mathbf{B}_\infty)_{C_1}$ the clause C' was removed from the clause set $\mathbf{B}_k$ by the application of the Del or Simp rule at a certain step $k < \kappa$. Therefore C'must be an object tautology clause, or non-properly subsumed or redundant. We now consider each possibility.

1. C is an object tautology clause

   Suppose C was removed from $\mathbf{B}_k$ because it was an object tautology clause, i.e. $C\sigma$ is like $\mathcal{A}, \leftarrow i_1 \simeq i_2, \mathcal{B}$ with $i_1, i_2 \in \mathbb{D}$ and $i_1 \neq i_2$.

   As i_1 and i_2 are two non-identical members of $\mathbb{D}$ and the unique name assumption applies to $\mathbb{D}$ the equation $i_1 \simeq i_2$ can never be true. With the false literal $i_1 \simeq i_2$ in the body of the clause the whole clause becomes true.

   As there can be no non-tautological clauses $\mathcal{D}$ such that $\mathit{true} \models_E \mathcal{D}$ the lemma holds trivially.

2. C is non-properly subsumed

Suppose C was removed from $\mathbf{B}_k$ because it was non- properly subsumed by a clause $\mathcal{D} \in \mathbf{B}_k$. C must be a proper instance of $\mathcal{D}$ as by the derivation rules Equality and Split no derived clause set $\mathbf{B}_i$ can contain a clause and a variant of it. The converse relation to proper instantiation, called proper generalisation, is well founded. Thus, by induction on this ordering, there is a clause $\mathcal{D}'$ in $\mathbf{B}_\infty$ that non- properly subsumes C. As C' is an instance of C and C is an instance of $\mathcal{D}'$, C' is an instance of $\mathcal{D}'$. With $\mathcal{D}' \in \mathbf{B}_\infty$, C' is an instance of a persisting clause in $\mathbf{B}_\infty$ which is a contradiction to the construction of $\mathbf{B}'$ as $\mathbf{B}'$ contains those clauses of $(\mathbf{B}^j)_{C_1}$ that are not an instance of any persisting clause in $\mathbf{B}_\infty$. Therefore this case is impossible.

3. C is redundant

   Suppose C was removed from $\mathbf{B}_k$ because it and its instance C' was redundant w.r.t. a specific subset $\mathbf{B}''$ of the derived branch $\mathbf{B}_{k+1}$ where $\mathbf{B}''$ is the branch specified in the definition of the Del and Simp derivation rules. Because $\mathbf{B}'' \subseteq \mathbf{B}_{k+1}$ it follows, that C' is redundant w.r.t. $\mathbf{B}_{k+1}$ i.e. $(\mathbf{B}_{k+1})_{C'} \models_E C'$. By monotonicity of first order logic with equality $(\mathbf{B}_{k+1})_{C'} \cup \mathcal{S} \models_E C'$ holds.

   As $C' \in \mathbf{B}'$ and $\mathbf{B}' \subseteq (\mathbf{B}^j)_{C_1}$ it follows that $C' \prec C_1$. By induction then

   $$(\mathbf{B}_\infty)_{C'} \cup \mathcal{S} \models_E C'. \tag{3.5}$$

   $C' \prec C_1$ leads to $(\mathbf{B}_\infty)_{C'} \subseteq (\mathbf{B}_\infty)_{C_1}$ which in combination with (5) and the monotonicity of first order logic with equality entails

   $$(\mathbf{B}_\infty)_{C_1} \cup \mathcal{S} \models_E C'. \tag{3.6}$$

   This entailment allows us to replace the clause C' in the premise $(\mathbf{B}^j)_{C_1}$ by the stronger set $(\mathbf{B}_\infty)_{C_1} \cup \mathcal{S}$. That is from $(\mathbf{B}^j)_{C_1} \cup \mathcal{S} \models_E C_2$ and 6 follows

   $$((\mathbf{B}_\infty)_{C_1} \cup \mathcal{S}) \cup \left((\mathbf{B}^j)_{C_1} \setminus \{C'\}\right) \cup \mathcal{S} \models_E C_2. \tag{3.7}$$

   As this has to hold for all members of $\mathbf{B}'$, (7) can be extended to

   $$((\mathbf{B}_\infty)_{C_1} \cup \mathcal{S}) \cup \left((\mathbf{B}^j)_{C_1} \setminus \mathbf{B}'\right) \cup \mathcal{S} \models_E C_2. \tag{3.8}$$

   With the definition of $\mathbf{B}' = (\mathbf{B}^j)_{C_1} \setminus (\mathbf{B}_\infty)_{C_1}$ which implies $(\mathbf{B}^j)_{C_1} \setminus \mathbf{B}' \subseteq (\mathbf{B}_\infty)_{C_1}$ and (8) $(\mathbf{B}_\infty)_{C_1} \cup \mathcal{S} \models_E C_2$ follows immediately.

□

The result of Lemma 5 allows a straight forward proof of Lemma 6 which states, that if a clause is redundant to the set of clauses for an initial segment of a branch it is redundant to the set of persistent clauses of this branch as well.

Lemma 6
Let C be a clause, $\mathbf{D}$ a derivation, $\mathbf{t}_\infty$ the limit tree of $\mathbf{D}$ and $\mathbf{B}$ a branch of $\mathbf{t}_\infty$. Furthermore let $\mathbf{B}^j$ be the initial segment of $\mathbf{B}$ and $\mathbf{B}_\infty$ the set of persistent clauses for $\mathbf{B}$.

If C is redundant w.r.t. $\mathbf{B}^j$ for some $j < \nu$ then C is redundant w.r.t. $\mathbf{B}_\infty$.

■

Proof. Suppose C is redundant w.r.t. $\mathbf{B}^j$ for some $j < \nu$. To show that C is redundant w.r.t. $\mathbf{B}_\infty$ it suffices to show that an arbitrary ground clause of C is redundant w.r.t. $\mathbf{B}_\infty$. Therefore let $\mathcal{D} = C\gamma$ for a grounding substitution γ. As C is redundant w.r.t. $\mathbf{B}^j$ its instance $\mathcal{D}$ is redundant w.r.t. $\mathbf{B}^j$, i.e. $(\mathbf{B}^j)_\mathcal{D} \models_E \mathcal{D}$. Lemma 5 leads to the conclusion $(\mathbf{B}_\infty)_\mathcal{D} \models_E \mathcal{D}$ i.e. $\mathcal{D}$ is redundant w.r.t. $\mathbf{B}_\infty$.

□

For the proof of Proposition 11 four more lemmas are needed. The first three claim, that if an application of the sup-left, unit-sup-right, ref and split rule is redundant with respect to the set of clauses of an initial segment of a branch, the application of those rules is redundant with respect to the set of persistent clauses for this branch.

The fourth lemma claims that if the unit-cont-right is not applicable to the set of clauses of an initial segment of a branch, it is not applicable to the set of persistent clauses.

Lemma 7 formalizes this statement for the sup-left and unit-sup-right inference rules. The according proof is straight forward by applying definitions and Lemma 5.

Lemma 7
Let C be a clause, $\mathcal{D}$ be a positive unit clause, $\mathbf{D}$ a derivation, $\mathbf{t}_\infty$ the limit tree of $\mathbf{D}$ and $\mathbf{B}$ a branch of $\mathbf{t}_\infty$. Furthermore let $\mathbf{B}^j$ be the initial segment of $\mathbf{B}$ and $\mathbf{B}_\infty$ the set of persistent clauses for $\mathbf{B}$.

Any inference $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$, where $\mathsf{R} \in \{\text{sup-left}, \text{unit-sup-right}\}$ that is redundant w.r.t. $\mathbf{B}^j$, for some $j < \nu$, is redundant w.r.t. $\mathbf{B}_\infty$.

■

Proof. Suppose an inference $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$, where $\mathsf{R} \in \{\text{sup-left}, \text{unit-sup-right}\}$ is redundant w.r.t. $\mathbf{B}^j$, for some $j < \nu$. To show that $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ is redundant w.r.t. to $\mathbf{B}_\infty$ it suffices to show that an arbitrary ground instance of the inference is redundant w.r.t. $\mathbf{B}_\infty$. Let γ be an arbitrary grounding substitution such that $\gamma = \sigma\delta$ and $C\gamma, \mathcal{D}\gamma \Rightarrow_{\mathsf{R}(\epsilon)} \mathcal{E}\delta$ is a ground instance of $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ we show that $C\gamma, \mathcal{D}\gamma \Rightarrow_{\mathsf{R}(\epsilon)} \mathcal{E}\delta$ is redundant w.r.t. $\mathbf{B}_\infty$.

As $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}^j$ it follows trivially that $C\gamma, \mathcal{D}\gamma \Rightarrow_{\mathsf{R}(\epsilon)} \mathcal{E}\delta$ is redundant w.r.t. $\mathbf{B}^j$ i.e. $(\mathbf{B}^j)_{C\gamma} \cup \{\mathcal{D}\gamma\} \models_E \mathcal{E}\gamma$ which with Lemma 5 leads to

$(\mathbf{B}^i)_{C\gamma} \cup \{\mathcal{D}\gamma\} \models_E \mathcal{E}\gamma$, i.e. $C\gamma, \mathcal{D}\gamma \Rightarrow_{\mathsf{R}(\epsilon)} \mathcal{E}\delta$ is redundant w.r.t. $\mathbf{B}_\infty$, which was to be shown.

□

Lemma 8 formalizes this statement for the ref inference rules. The according proof is similar to that of Lemma 7.

Lemma 8
Let C be a clause, $\mathcal{D}$ be a positive unit clause, $\mathbf{D}$ a derivation, $\mathbf{t}_\infty$ the limit tree of $\mathbf{D}$ and $\mathbf{B}$ a branch of $\mathbf{t}_\infty$. Furthermore let $\mathbf{B}^j$ be the initial segment of $\mathbf{B}$ and $\mathbf{B}_\infty$ the set of persistent clauses for $\mathbf{B}$.

Any inference $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ that is redundant w.r.t. $\mathbf{B}^j$, for some $j < \nu$, is redundant w.r.t. $\mathbf{B}_\infty$.

■

Proof. Suppose an inference $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}^j$, for some $j < \nu$. To show that $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ is redundant w.r.t. to $\mathbf{B}_\infty$ it suffices to show that an arbitrary ground instance of the inference is redundant w.r.t. $\mathbf{B}_\infty$. Let γ be an arbitrary grounding substitution such that $\gamma = \sigma\delta$ and $C\gamma \Rightarrow_{\mathsf{ref}(\epsilon)} \mathcal{E}\delta$ is a ground instance of $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ we show that $C\gamma \Rightarrow_{\mathsf{ref}(\epsilon)} \mathcal{E}\delta$ is redundant w.r.t. $\mathbf{B}_\infty$.

As $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}^j$ it follows trivially that $C\gamma \Rightarrow_{\mathsf{ref}(\epsilon)} \mathcal{E}\delta$ is redundant w.r.t. $\mathbf{B}^j$ i.e. $(\mathbf{B}^j)_{C\gamma} \cup \emptyset \models_E \mathcal{E}\gamma$ which with Lemma 5 leads to $(\mathbf{B}^i)_{C\gamma} \cup \emptyset \models_E \mathcal{E}\gamma$, i.e. $C\gamma \Rightarrow_{\mathsf{ref}(\epsilon)} \mathcal{E}\delta$ is redundant w.r.t. $\mathbf{B}_\infty$, which was to be shown.

□

Lemma 9 takes care of the remaining rule of the original calculus, namely split. Again the prove is a straight forward application of definitions but this time in combination with Lemma 6.

Lemma 9
Let C be a positive clause, π a purifying substitution for C, $\mathbf{D}$ a derivation, $\mathbf{t}_\infty$ the limit tree of $\mathbf{D}$ and $\mathbf{B}$ a branch of $\mathbf{t}_\infty$. Furthermore let $\mathbf{B}^j$ be the initial segment of $\mathbf{B}$ and $\mathbf{B}_\infty$ the set of persistent clauses for $\mathbf{B}$.

If the inference $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$ is redundant w.r.t. $\mathbf{B}^j$, for some $j < \nu$, then it is redundant w.r.t. $\mathbf{B}_\infty$.

■

Proof. Suppose an inference $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$ is redundant w.r.t. $\mathbf{B}^j$, for some $j < \nu$. To show that $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$ is redundant w.r.t. to $\mathbf{B}_\infty$ it suffices to show that an arbitrary ground instance of the inference is redundant w.r.t. $\mathbf{B}_\infty$. Let γ be an arbitrary grounding substitution such that $\gamma = \pi\delta$ and $C\gamma \Rightarrow_{\mathsf{split}(\epsilon)} A_1\delta \leftarrow, \ldots, A_m\delta \leftarrow$ is a ground instance of $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$ we show that $C\gamma \Rightarrow_{\mathsf{split}(\epsilon)} A_1\delta \leftarrow, \ldots, A_m\delta \leftarrow \delta$ is redundant w.r.t. $\mathbf{B}_\infty$.

As $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$ is redundant w.r.t. $\mathbf{B}^j$ it follows trivially that $C\gamma \Rightarrow_{\mathsf{split}(\epsilon)} A_1\delta \leftarrow, \ldots, A_m\delta \leftarrow$ is redundant w.r.t. $\mathbf{B}^j$, i.e. $A_i\delta \leftarrow$ for some i with $1 \leq i \leq m$ is redundant w.r.t. $\mathbf{B}^j$. With Lemma 6 $A_i\delta \leftarrow$ is redundant w.r.t. $\mathbf{B}_\infty$ which entails that $C\gamma \Rightarrow_{\mathsf{split}(\epsilon)} A_1\delta \leftarrow, \ldots, A_m\delta \leftarrow \delta$ is redundant w.r.t. to $\mathbf{B}_\infty$.

□

The last lemma needed to prove Proposition 11 is Lemma 10. The prove is done by contradiction and application of the appropriate definitions.

Lemma 10
Let C be a positive unit, $\mathbf{D}$ a derivation, $\mathbf{t}_\infty$ the limit tree of $\mathbf{D}$ and $\mathbf{B}$ a branch of $\mathbf{t}_\infty$. Furthermore let $\mathbf{B}^j$ be the initial segment of $\mathbf{B}$ and $\mathbf{B}_\infty$ the set of persistent clauses for $\mathbf{B}$.

If an inference $C \Rightarrow_{\mathsf{unit\text{-}cont\text{-}right}(\sigma)} \square$ is not applicable to $\mathbf{B}^j$, for some $j < \nu$ and some substitution σ, then it is not applicable to $\mathbf{B}_\infty$.

■

Proof. Suppose an inference $C \Rightarrow_{\mathsf{unit\text{-}cont\text{-}right}(\sigma)} \square$ is not applicable to $\mathbf{B}^j$, for some $j < \nu$. To show that $C \Rightarrow_{\mathsf{unit\text{-}cont\text{-}right}(\sigma)} \square$ is not applicable to $\mathbf{B}_\infty$ if it is not applicable to $\mathbf{B}^j$, we show by way of contradiction that it is not possible for $C \Rightarrow_{\mathsf{unit\text{-}cont\text{-}right}(\sigma)} \square$ to be applicable to $\mathbf{B}_\infty$ but to be not applicable to $\mathbf{B}^j$.

For $C \Rightarrow_{\mathsf{unit\text{-}cont\text{-}right}(\sigma)} \square$ to be applicable in $\mathbf{B}_\infty$ there must be a clause $C \in \mathbf{B}_\infty$ and an grounding substitution σ such that $C\sigma = i_1 \simeq i_2 \leftarrow$ with $i_1, i_2 \in \mathbb{D}$ and $i_1 \neq i_2$.

From the definition of the set of persistent clauses (see Definition 2.2.4) follows that $\mathbf{B}^j = \mathbf{B}_\infty \uplus \mathbf{B}$ where $\mathbf{B}$ is the set of clauses that have been rewritten in the derivation. By contradiction assume that $C \in \mathbf{B}$. That is C has been rewritten by the Del or Simp rule. It is easy to see that there is no possibility that C can be rewritten and thus $C \notin \mathbf{B}$. Thus $C \in \mathbf{B}^j$ must hold.

But if $C \in \mathbf{B}^j$, then $C \Rightarrow_{\mathsf{unit\text{-}cont\text{-}right}(\sigma)} \square$ is applicable which clearly is a contradiction.

□

Now Proposition 11, that states, that with an fair derivation the set of persistent clauses of an exhausted branch is saturated up to redundancy, is introduced and proven. The main idea of the proof is to show that all the requirements of the definition of saturation up to redundancy (see Definition 3.2.4) are fulfilled.

Proposition 11 (Exhausted branches are saturated up to redundancy)
If $\mathbf{B}$ is an exhausted branch of a limit tree of some fair derivation then $\mathbf{B}_\infty$ is saturated up to redundancy.

■

Proof. Suppose $\mathbf{B}$ is an exhausted branch of a limit tree of some fair derivation. To show that $\mathbf{B}_\infty$ is saturated up to redundancy it suffices to chose an arbitrary clause $C \in \mathbf{B}_\infty$ that is not redundant w.r.t. and prove that the properties for saturation up to redundancy (see Definition 3.2.4) hold for C.

Before we take care of the four properties of saturation up to redundancy, notice that if there is branch $\mathbf{B}^j$ with $j < \nu$ and C is redundant w.r.t. $\mathbf{B}^j$ it follows from lemma 6 that C is redundant w.r.t. $\mathbf{B}_\infty$ and nothing remains to be shown.

Therefore suppose that C is not redundant w.r.t. $\mathbf{B}^j$, for all $j < \nu$.

1. $C \Rightarrow_{\mathsf{unit\text{-}cont\text{-}right}(\sigma)} \square$

   By Definition 3.2.6 a branch $\mathbf{B}$ where Inc is applicable with underlying inference $C \Rightarrow_{\mathsf{unit\text{-}cont\text{-}right}(\sigma)} \square$ is not exhausted. Therefore there must be no $C \in \mathbf{B}$ such that $C\sigma$ is like $i_1 \simeq i_2 \leftarrow$ with $i_1, i_2 \in \mathbb{D}$ and $i_1 \neq i_2$.

   From Lemma 10 it follows, that if unit-cont-right is not applicable to $\mathbf{B}$ it is neither applicable to $\mathbf{B}_\infty$.

   Thus there is no clause in $\mathbf{B}_\infty$ such that the inference rule unit-cont-right is applicable which concludes the fourth case of the definition of saturation up to redundancy (see Definition 3.2.4).

2. $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ where $\mathsf{R} \in \{\mathsf{sup\text{-}left}, \mathsf{unit\text{-}sup\text{-}right}\}$

   Suppose there is an inference $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ where $\mathsf{R} \in \{\mathsf{sup\text{-}left}, \mathsf{unit\text{-}sup\text{-}right}\}$, σ is a substitution and $\mathcal{D}$ is a fresh variant of a positive unit clause from $\mathbf{B}_\infty$.

   To show that $\mathbf{B}_\infty$ is saturated up to redundancy it suffices to show one of the following: $C\sigma$ is redundant w.r.t. $\mathbf{B}_\infty$, $\mathcal{D}\sigma$ is redundant w.r.t., $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}_\infty$.

   If there is a $j < \nu$ such that $C\sigma$ is redundant w.r.t. $\mathbf{B}^j$, then by Lemma 6 $C\sigma$ is redundant w.r.t. $\mathbf{B}_\infty$, which concludes this case. The same holds for $\mathcal{D}\sigma$.

   Hence we assume that neither $C\sigma$ nor $\mathcal{D}\sigma$ is redundant w.r.t. $\mathbf{B}^j$ for all $j < \nu$.

   To show that $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}_\infty$ it suffices to show that an arbitrary ground instance $C\gamma, \mathcal{D}\gamma \Rightarrow_{\mathsf{R}(\epsilon)} \mathcal{E}\delta$ of the inference $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ with the grounding substitution $\gamma = \sigma\delta$ and some substitution δ is redundant w.r.t. to $\mathbf{B}_\infty$.

   As $C \in \mathbf{B}_\infty$ there must be an $i < \nu$ such that for all j with $i \leq j < \nu$, $C \in \mathbf{B}^j$. And as $\mathcal{D}$ is an variant of a clause in $\mathbf{B}_\infty$ there must be an i' such that for all j' with $i' \leq j' < \nu$, $\mathcal{D}$ is a variant of a clause in $\mathbf{B}_{j'}$. Without loss of generality assume that $i \geq i'$ and thus $\mathcal{D}$ is a variant of a clause in $\mathbf{B}^j$ for all $i \leq j < \nu$.

   Under these conditions, the derivation rule Equality is applicable to $\mathbf{B}_i$ with underlying inference $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ unless $\mathcal{E}$ is a variant of a clause in $\mathbf{B}_i$ which would entail that the inference is redundant w.r.t. $\mathbf{B}_i$ and conclude this proof.

By assumption $C\sigma$ and $\mathcal{D}\sigma$ are not redundant w.r.t. $\mathbf{B}^j$ for every $j < \nu$. As $\mathbf{B}$ is an exhausted branch and the definition of exhausted branches (see Definition 3.2.6) states there is a $k < \nu$ such that the inference $C, \mathcal{D} \Rightarrow_{\mathsf{R}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}_k$ by Lemma 7 follows that this inference is redundant w.r.t. $\mathbf{B}_\infty$.

This holds for its ground inference $C\gamma, \mathcal{D}\gamma \Rightarrow_{\mathsf{R}(\epsilon)} \mathcal{E}\delta$ as well.

3. $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$

   Suppose there is an inference $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ where σ is a substitution.

   To show that $\mathbf{B}_\infty$ is saturated up to redundancy it suffices to show that $C\sigma$ is redundant w.r.t. $\mathbf{B}_\infty$ or $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}_\infty$.

   If there is a $j < \nu$ such that $C\sigma$ is redundant w.r.t. $\mathbf{B}^j$, then by Lemma 6 $C\sigma$ is redundant w.r.t. $\mathbf{B}_\infty$, which concludes this case.

   Hence we assume that $C\sigma$ is not redundant w.r.t. $\mathbf{B}^j$ for all $j < \nu$.

   To show that $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}_\infty$ it suffices to show that an arbitrary ground instance $C\gamma \Rightarrow_{\mathsf{ref}(\epsilon)} \mathcal{E}\delta$ of the inference $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ with the grounding substitution $\gamma = \sigma\delta$ and some substitution δ is redundant w.r.t. to $\mathbf{B}_\infty$.

   As $C \in \mathbf{B}_\infty$ there must be an $i < \nu$ such that for all j with $i \leq j < \nu$, $C \in \mathbf{B}^j$.

   Under these conditions, the derivation rule Equality is applicable to $\mathbf{B}_i$ with underlying inference $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ unless $\mathcal{E}$ is a variant of a clause in $\mathbf{B}_i$ which would entail that the inference is redundant w.r.t. $\mathbf{B}_i$ and conclude this proof.

   By assumption $C\sigma$ is not redundant w.r.t. $\mathbf{B}^j$ for every $j < \nu$. As $\mathbf{B}$ is an exhausted branch and the definition of exhausted branches (see Definition 3.2.6) states there is a $k < \nu$ such that the inference $C \Rightarrow_{\mathsf{ref}(\sigma)} \mathcal{E}$ is redundant w.r.t. $\mathbf{B}_k$ by Lemma 8 follows that this inference is redundant w.r.t. $\mathbf{B}_\infty$.

   This holds for its ground inference $C\gamma \Rightarrow_{\mathsf{ref}(\epsilon)} \mathcal{E}\delta$ as well.

4. $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow \ldots, A_m \leftarrow$

   Suppose there is an inference $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow, \ldots, A_m \leftarrow$ where π is a purifying substitution.

   To show that $\mathbf{B}_\infty$ is saturated up to redundancy it suffices to show that $C\pi$ is redundant w.r.t. $\mathbf{B}_\infty$ or $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow \ldots, A_m \leftarrow$ is redundant w.r.t. $\mathbf{B}_\infty$.

   If there is a $j < \nu$ such that $C\sigma$ is redundant w.r.t. $\mathbf{B}^j$ then by Lemma 6 $C\sigma$ is redundant w.r.t. $\mathbf{B}_\infty$ which concludes this case.

   Hence we assume that $C\sigma$ is not redundant w.r.t. $\mathbf{B}^j$ for all $j < \nu$.

   To show that $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow \ldots, A_m \leftarrow$ is redundant w.r.t. $\mathbf{B}_\infty$ it suffices to show that an arbitrary ground instance $C\gamma \Rightarrow_{\mathsf{split}(\epsilon)} A_1\delta \leftarrow \ldots, A_m\delta \leftarrow$ of the inference $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow \ldots, A_m \leftarrow$ with the grounding substitution $\gamma = \pi\delta$ and some substitution δ is redundant w.r.t. to $\mathbf{B}_\infty$.

As $C \in \mathbf{B}_\infty$ there must be an $i < \nu$ such that for all j with $i \leq j < \nu$, $C \in \mathbf{B}^j$.

Under these conditions, the derivation rule Split is applicable to $\mathbf{B}_i$ with underlying inference $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow \ldots, A_m \leftarrow$ unless $A_h \leftarrow$ for some $1 \leq h \leq m$ is a variant of a clause in $\mathbf{B}_i$ which would entail that the inference is redundant w.r.t. $\mathbf{B}_i$ and conclude this proof.

By assumption $C\sigma$ is not redundant w.r.t. $\mathbf{B}^j$ for every $j < \nu$. As $\mathbf{B}$ is an exhausted branch and the definition of exhausted branches (see Definition 3.2.6) states there is a $k < \nu$ such that the inference $C \Rightarrow_{\mathsf{split}(\pi)} A_1 \leftarrow \ldots, A_m \leftarrow$ is redundant w.r.t. $\mathbf{B}_k$ by Lemma 9 follows that this inference is redundant w.r.t. $\mathbf{B}_\infty$.

This holds for its ground inference $C\gamma \Rightarrow_{\mathsf{split}(\epsilon)} A_1\delta \leftarrow \ldots, A_m\delta \leftarrow$ as well.

□

For the proof of the following theorem an additional lemma is needed. Lemma 12 states that if a rewrite system entails an clause set, and a clause is redundant with respect to this clause set, the rewrite system entails the clause. The proof is done straight forward.

Lemma 12
If $\mathtt{R} \models_E \mathcal{S}$ and C is redundant w.r.t. $\mathcal{S}$ then $\mathtt{R} \models_E C$.

■

Proof. Suppose $\mathtt{R} \models_E \mathcal{S}$ and C is redundant w.r.t. $\mathcal{S}$. To show $\mathtt{R} \models_E C$ it suffices to show that for an arbitrary ground clause $\mathcal{D}$ of C, i.e. $\mathcal{D} = C\gamma$ for a grounding substitution γ, $\mathtt{R} \models_E \mathcal{D}$ holds. As C is redundant w.r.t. $\mathcal{S}$ it follows trivially that $\mathcal{D}$ is redundant w.r.t. $\mathcal{S}$, i.e. $\mathcal{S}_\mathcal{D} \models_E \mathcal{D}$. With the assumption $\mathtt{R} \models_E \mathcal{S}$ and $\mathcal{S}_\mathcal{D} \subset \mathcal{S}$ we can conclude $\mathtt{R} \models_E \mathcal{D}$.

□

Theorem 13 states the completeness of the modified E-hyper tableau calculus, i.e. if a fair derivation for a clause set is not a refutation, this clause set is satisfiable.

The proof is straight forward and mainly relies on the theorem of static completeness (see Theorem 4) and the proposition that exhausted branches are saturated up to redundancy.

Theorem 13 (Completeness of E-Hyper Tableaux)
Let $\mathcal{S}$ be a clause set and $\mathbf{D}$ a fair derivation of $\mathcal{S}$.

If $\mathbf{D}$ is not a refutation the $\mathcal{S}$ is satisfiable.

■

Proof. By definition $\mathbf{D}$'s limit tree has an exhausted branch $\mathbf{B}$.

As $\mathbf{B}$ is an exhausted branch, by definition it does not contain the empty clause. Furthermore by proposition 11 $\mathbf{B}_\infty$ is saturated up to redundancy and because of $\square \notin \mathbf{B}$ it follows $\square \notin \mathbf{B}_\infty$.

By theorem 4 $\mathbf{B}_\infty$ is satisfiable as there is a rewrite system $\mathtt{R}_{\mathbf{B}\infty}$ that is an E-model for $\mathbf{B}_\infty$, i.e. $\mathtt{R}_{\mathbf{B}\infty} \models_E \mathbf{B}_\infty$.

To prove the theorem it suffices to show that $\mathtt{R}_{\mathbf{B}\infty} \models_E \mathcal{S}$ which can be done by showing that for any clause $C \in \mathcal{S}$ $\mathtt{R}_{\mathbf{B}\infty} \models_E C$ holds.

By definition of derivation $\mathbf{B}_0$ is the single branch of the initial tableau of the derivation $\mathbf{D}$ and thus equivalent with $\mathcal{S}$. Therefore assume $C \in \mathbf{B}_0$.

If $C \in \mathbf{B}_\infty$, $\mathtt{R}_{\mathbf{B}\infty} \models_E C$ follows immediately from $\mathtt{R}_{\mathbf{B}\infty} \models_E \mathbf{B}_\infty$. Therefore suppose $C \notin \mathbf{B}_\infty$.

If $C \in \mathbf{B}_0$ but $C \notin \mathbf{B}_\infty$ the clause C was removed from the clause set $\mathbf{B}_k$ by the application of the Del or Simp rule at a certain step. Therefore C must be an object tautology clause, or non-properly subsumed or redundant. We now consider each possibility.

1. C is an object tautology clause

   Suppose C was removed from $\mathbf{B}_k$ because it was an object tautology clause, i.e. $C\sigma$ is like $\mathcal{A}, \leftarrow i_1 \simeq i_2, \mathcal{B}$ with $i_1, i_2 \in \mathbb{D}$ and $i_1 \neq i_2$.

   As i_1 and i_2 are two non-identical members of $\mathbb{D}$ and the unique name assumption applies to $\mathbb{D}$ the equation $i_1 \simeq i_2$ can never be true. With the false literal $i_1 \simeq i_2$ in the body of the clause the whole clause becomes true. Thus $\mathtt{R}_{\mathbf{B}\infty} \models_E C$ follows trivially.

2. C is non-properly subsumed

   Suppose C was removed from $\mathbf{B}_k$ because it was non- properly subsumed by a clause $\mathcal{D} \in \mathbf{B}_k$. C must be a proper instance of $\mathcal{D}$ as by the derivation rules Equality and Split no derived clause set $\mathbf{B}_i$ can contain a clause and a variant of it. The converse relation to non-proper subsumption, called proper generalisation, is well founded. Thus, by induction on this ordering, there is a clause $\mathcal{D}'$ in $\mathbf{B}_\infty$ that non- properly subsumes C. As $\mathcal{D}' \in \mathbf{B}_\infty$ applies and $\mathtt{R}_b bi \models_E \mathcal{D}'$ holds, $\mathtt{R}_{\mathbf{B}\infty} \models_E C$ holds as well.

3. C is redundant

   Suppose C was removed from $\mathbf{B}_k$ because it was redundant w.r.t. a specific subset $\mathbf{B}'$ of the derived branch $\mathbf{B}_{k+1}$ where $\mathbf{B}'$ is specified in the definition of the Del and Simp derivation rules. Because $\mathbf{B}' \subseteq \mathbf{B}_{k+1}$ it follows trivially, that C is redundant w.r.t. $\mathbf{B}_{k+1}$. By Lemma 6 C is redundant w.r.t. $\mathbf{B}_\infty$ which with Lemma 12 leads to $\mathtt{R}_{\mathbf{B}\infty} \models_E C$.

□

With the completeness of the modified E-hyper tableau calculus shown and proven, the third part of the properties is concluded. The following and final part now takes care of the soundness.

3.3.5 Soundness

Now the soundness of the extended E-hyper tableau calculus is introduced and proven. In contrast to the completeness this part is rather straight forward and compact. It starts with Lemma 14 that states that if a premise of a derivation rule Equality, Split, Del or Simp is E-satisfiable, one conclusion is E-satisfiable as well. The proof is done straight forward by showing this statement for each of the four derivation rule.

Lemma 14
For each of the derivation rules Equality, Split, Del and Simp holds, if the premise of the rule is E-satisfiable, then one of its conclusions is E-satisfiable as well.

■

Proof. The claim is proven be examining each rule on its own.

For the Equality rule we take a look at the sup-left and unit-sup-right rule. If the premiss of such a rule is E-satisfiable then there is an E-model I. With the axioms of congruence it follows immediately that I is an E-model for the conclusion as well. For ref the claims follows immediately from the reflexivity. With sup-left, unit-sup-right and ref being the underlying inference rules for the Equality rule, it can be concluded that the claim holds for Equality.

For Split, assume an E-model I for the premiss $\mathbf{B}$. With $A_1, \ldots, A_m \leftarrow \in \mathbf{B}$ and the purifying substitution π, I is an E-model for $(A_1, \ldots, A_m \leftarrow)\pi$. As all variables are implicitly universally quantified I has to be an E-model for $\forall(A_1, \ldots, A_m \leftarrow)$ Due to purification the set of variables of each A_i is disjunct which allows to write $\forall A_1\pi \vee \ldots \vee \forall A_m\pi$ instead of $\forall(A_1\pi \vee \ldots \vee A_m\pi)$. Thus I is an E-model for one of $B \cdot A_1\pi \leftarrow^d, \ldots, B \cdot A_m\pi \leftarrow^d$.

For Del the claim holds directly from its definition.

For Simp assume an E-model I for the premiss. From the definition of the Simp rule(see Fig. 2.3) $(\mathbf{B} \cdot C \cdot \mathbf{B}_1) \models_E \mathcal{D}$ for clauses C, $\mathcal{D}$ and branches $\mathbf{B}$, $\mathbf{B}_1$ follows that $\mathcal{D}$ holds in I.

□

The next step is Lemma 15 that states, that if the derivation rule Inc is applicable, its premise is E-unsatisfiable. The prove is straight forward and uses the definition of the Inc derivation rule.

Lemma 15
If the derivation rule Inc is applicable, its premise is E-unsatisfiable.

■

Proof. For Inc to be applicable, there must be a $C \in \mathbf{B}$ and a substitution σ such that the inference $C \Rightarrow_{\text{unit-cont-right}(\sigma)} \square$ can be applied. Therefore C and σ must be of such a form that $C\sigma = i_1 \simeq i_2 \leftarrow$ with $i_1, i_2 \in \mathbb{D}$ and $i_1 \neq i_2$.

As i_1 and i_2 are two non-identical distinct object identifiers and the unique name assumption applies to the set of distinct object identifiers this equation can never be true. Therefore there is no E-model for $i_1 \simeq i_2 \leftarrow$ and as C is clause of $\mathbf{B}$, the branch $\mathbf{B}$ is E-unsatisfiable.

□

Finally both just introduced lemmas are used to prove Theorem 16, that states the soundness of the extended E-hyper tableau calculus, i.e. if a clause set has an E-hyper tableau refutation it is E-unsatisfiable.

Theorem 16 (Soundness of E-Hyper Tableaux)
Let $\mathcal{S}$ be a clause set that has a refutation. Then $\mathcal{S}$ is E-unsatisfiable.

■

Proof. Assume a refutation leads to the closed tableau $\mathbf{T}$. From Lemma 15 and the contrapositive of Lemma 14 we conclude that if a tableau $\mathbf{T}_i$ of a derivation contains only E-unsatisfiable branches, this holds for its predecessor $\mathbf{T}_{i-1}$ as well. Following the definition of a refutation, the final tableau $\mathbf{T}$ only consists of E-unsatisfiable branches.

By induction on the length of the refutation we can conclude that the initial tableau $\mathbf{T}_0$, consisting of one branch with the tableau clauses from $\mathcal{S}$, is E-unsatisfiable.

□

This concludes the theoretical part of this work.

Chapter 4

Implementation and Evaluation

4.1 Implementation

4.1.1 Introduction

Instead of starting from scratch with implementing the calculus, it was decided to extend the E-KRHyper prover. In section 4.1.2, we give a short introduction of the existing system before we explain some of the made adoptions in Section 4.1.3.

Subsequently, the set-up and results of the evaluation of the approach are shown in Section 4.2.

4.1.2 E-KRHyper

The E-KRHyper system[19, 20] is an automated theorem prover by Björn Pelzer at the Universität Koblenz-Landau, which implements the E-hyper tableau calculus. It is based on the KRHyper system[28, 29] by Christoph Wernhard at the Universität Koblenz-Landau, which is an implementation of the original hyper tableau calculus. The development of the KRHyper was stopped when it was superseded by the introduction of the E-KRHyper system. Both systems are well established and used in different areas. The areas of use include amongst others natural question answering[14], e-learning [8, 11] and ontology reasoning [9].

Both systems are written in the OCaml programming language[16]. It is a strong and static typed functional language, which allows to used other programming paradigms, as well, and offers the possibility to create high performance programs.

As the implementation of the E-KRHyper system involves about 29,000 lines of code and has a high complexity, a full-fledged explanation would exceed the scope of this thesis. Thus the introduction of the system is not a complete description of the code but only a glimpse on the parts that are of special importance for the general functionality of the prover or subject to change due to the extension of the calculus. Some more information on implementation details can be found in

[18]. The version of E-KRHyper that was used as foundation for our implementation was 1_2(05122011).

E-KRHyper can use two different kinds of input formats, namely the TPTP syntax and the protein[7] syntax. The latter is referred to as tme-syntax, as well, due to the `.tme`-file extension of protein files.

The input files are handled by a parser/lexer combination, where the lexer transforms the input into a stream of tokens. This stream of tokens is processed by the parser that uses `clausify.ml` and `term.ml` to create the appropriate `terms` and formulae out of these tokens.

Instead of writing the lexer and parser from scratch, the lexer is generated by `ocamllex` and the parser is generated by `ocamlyacc`. Thus one does not need to write the parser or lexer but just an appropriate lex (`.mll`) or yac (`.mly`) file.

A `term` is either the representation of a logical term or it is a so called `Meta`-statement which represents a command for E-KRHyper like `#(run).`, which starts the proving process.

If the `term` is the representation of a logical term or formula, methods from `env.ml` and `ic.ml` are used to create the appropriate clauses and add them to the initial node of the E-hyper tableau, which is represented by a variable of type `Tableau.node`. The implementation of an E-hyper tableau differs from the theoretical introduction as a node does not represent a single clause but a set of clauses. As long as no split occurs, the algorithm generates new clauses if possible, which are then added to the node. If the split rule is applied, new `Tableau.node`s are created and the newly introduced nodes inherit the clauses of their father node. With this behaviour a `Tableau.node` contains the clauses of all the nodes that lie between the root and the next split in the E-hyper tableau.

In the theory, it does not matter which rules to apply in which order, as long as it leads to a fair derivation. In the implementation however, the order of rule applications influences runtime performance. Thus E-KRHyper derives new clauses according to a certain principle, which we introduce now in a simplified version. All clauses have a certain weight assigned that either depends on the number of positions in the clause or on the depth of the clause. By parameter it can be chosen which of the two methods to calculate the weight is used. The derivation process starts with the set of clauses in the start node and a certain maximum weight. The weight limit is set and is used as boundary for a iterative deepening strategy to construct the tableau. Now sup-left, unit-sup-right and ref rules are applied to all possible clauses and the resulting clauses are added to the node as well. The rules are only applicable if the weight of the generated clause is smaller than the maximum weight. If no more applications of sup-left, unit-sup-right or ref are possible under this conditions, and the split rule is applicable, the algorithm applies the split rule and hence introduce new branches. Now the procedure is repeated for the newly generated branches. If there is no more way to generate clauses with a weight less than the weight limit, the algorithm evaluates if all clauses that can be generated and have a weight that is equal or greater than the weight limit, are redundant. In that case the tableau has at least one open exhausted branch and the derivation is

finished. If there are overweight clauses, which are not redundant, the weight limit is increased, a backtracking mechanism is used and the algorithm continues. The usage of the weight as criterion for splitting prevents early branch splitting which would result in a bad runtime behaviour. Delaying the splitting indefinitely would lead to an incomplete calculus, as early splits are sometimes needed to be able to close a tableau. [18] offers more details on this issue.

The term ordering that is needed for the calculus is provided by `lrpOrder.ml`, which implements methods that represent the $\succ$ relation.

To avoid confusion, the code of the unchanged version of the E-KRHyper is called *traditional* code.

4.1.3 Handling Distinct Object Identifiers

To avoid the introduction of errors and unwanted side effects and to benefit from the existing implementation as much as possible, we looked for a way to adapt the E-KRHyper to the presented calculus with as few changes as possible in the code. The chosen solution does not implement a new datatype for distinct object identifiers but introduces a new boolean attribute `doi` in the record `Term.symbol`, which is the datatype for representing logical constants in the implementation. As `doi` is false by default and thus has to be set explicitly and as there are no code segments in the traditional code that consider `doi`, this is a suitable method to introduce distinct object identifiers without side effects on the existing code. This behaviour follows the intention of the logic as distinct object identifiers are represented as constants with special properties. As the ability to store and process distinct object identifiers with the implementation of the traditional calculus does not suffice for an implementation of the extended calculus, the missing changes in the code are introduced now.

The first step in proper handling of distinct object identifiers is to enable E-KRHyper to load problem files that contain distinct object identifiers. The TPTP syntax specifies that distinct object identifiers are strings that are enclosed in double quotes — for example "a" — and we follow this specification. For an easier comprehension when printing terms, the double quotes are kept as part of the constant's name.

For the TPTP syntax, the lexer — `tptpLexer.ml` — has already been able to process distinct object identifiers and to produce `DISTINCT_OBJECT` tokens. For the tme-syntax, the generator file `tmeLexer.mll` was extended to mirror this behaviour.

In its original version, `tptpParser.ml`, was handling the `DISTINCT_OBJECT` token in a non-proper way, so we changed the behaviour that `Clausify.Doi` is called. This constructor creates an object of the type `Clausify.Doi` that is then saved as a prototype term. Prototype terms are needed due to some technicalities as intermediate step between parsing and the generation of the proper terms. `tmeParser.ml` had no support for the `DISTINCT_OBJECT` token and was extended in the same fashion as the TPTP version.

To create the proper term representation, the prototype stream is iterated and if an objects of type `Clausify.Doi` is processed an object of type `Term.Constant` is created. This behaviour is similar to the behaviour for prototype terms of type `Clausify.Constant` but in the first case, the boolean flag `doi` is set during the creation.

To simplify the handling of distinct objects, two additional methods are introduced in `term.ml`. Both are called with a parameter of type `term`. The first one returns whether this `term` is an equality of two non-equal distinct object identifiers and the second one returns if this term can be instantiated to be such an equality, i.e. if the parameter is of form $i \simeq j$ or $X \simeq j$ or $X \simeq Y$ with $i, j \in \mathbb{D}, X, Y \in \mathbb{V}$ and $i \neq j$. This distinction of the two methods is needed: If an equation appears as a positive unit it suffices to check if it is a possible contradiction, i.e. the latter method returns true, and if the signature of the loaded problem contains two non-identical distinct object identifiers to classify it as false. This behaviour is sound as all variables are implicitly universally quantified and thus it is enough to find one substitution that leads to a contradiction. This method cannot be used if an equation is a subterm of a larger clause, as one or both sides of the equation might appear in other literals of the same clause and thus it is not sound to just chose a substitution that leads to a contradiction.

To check if a problem contains at least two non-identical distinct object identifiers, the method `has_dois` was implemented in `ic.ml`. The `symbol_table` that contains all the symbols that appear in a problem is iterated and the method returns true if there are at least two distinct object identifiers.

In `ic.ml` a method is introduced to check if a clause is an object tautology clause. A boolean flag `is_object_tautology` is added to the record `Ic.clause` which is the datatype to represent clauses, to prevent repetitive calls to the method. When creating a new clause the method is called once and the flag is set accordingly. To avoid the addition of useless clauses, while creating the initial node, clauses, that are an object tautology are not added. This is done by extending a conditional statement in the method `make_and_add_clause` in `env.ml` .

Another modification that improves the performance by keeping the set of clauses small was introduced in `ic.ml`. The method `is_redundant` was extended so that it returns true for distinct object tautologies.

For the actual reasoning with unit contradictions, `ic.ml` was extended with the method `process_head_literals` in the following way. If the head literal is a unit contradiction a `BranchClosed` exception is raised to signal that this branch of the proof is closed. This behaviour is also applied if the head literal is a possible unit contradiction and the knowledge base contains at least two non-identical distinct object identifiers.

The adoption of the ordering in `lrpOrder.ml` is done straight forward. In the traditional code, the means of comparing two constants was to compare their names lexicographically. We introduce an additional step that checks if the two constants that are compared are distinct object identifiers. If both are distinct object identifiers, or both are not distinct object identifiers, the lexicographical comparison is

used to determine the greater term. In the case where exactly one of the constants has its `doi` flag set, compare defines this constant to be the smaller one, which conforms the requirement of the calculus.

At this point the implementation of the calculus is finished. For efficiency reason, we introduce another small change. In the traditional implementation, there is a check in `ic.ml` before creating a split in the proof. This check has two purposes. On the one hand it looks for complemented units of the head literals. If such a complement exist no new branch is created as it is already shown, that it is possible to close the branch. On the other hand it looks for clauses that are identical and then removes the duplicates. This prevents the creation of unnecessary branches as well. For our implementation we extended the method in such way, that unit contradictions are removed before splitting. This is sound, as this resembles a closed branch.

4.2 Evaluation

4.2.1 Introduction

With a suitable implementation at hand, we are now able to evaluate the impact of the changed calculus. Therefore we split the evaluation in two independent parts. In the first part we show that the modified implementation does not perform worse on problems that do not contain distinct object identifiers. In the second part we show that the use of distinct object identifiers has benefits for the execution time.

We start by introducing the machine that is used for running the benchmarks and we then introduce the test data that we use (see Section 4.2.2). In the second part of this section (see Section 4.2.3) we present and discuss the gathered data and give a short conclusion on the evaluation.

4.2.2 Test Conditions

All of the following benchmarks have been executed on the same machine with the following specifications:

CPU: Intel Core 2 Quad (Q9550) @ 2.83GHz

Memory: 4GB PC2-6400

Operating System: openSUSE 11.3

Kernel Version: 2.6.34.10-0.2-desktop

OCaml Version: 3.12.0

For showing that the introduced changes do not impair the reasoning if no distinct object identifiers are involved a sample of the TPTPv5.3.0 problems is used as basis for a comparison of traditional and modified implementation. Those samples

are chosen as follows. In the first step, we randomly choose 4000 TPTP problems out of the set of CNF and FOF problems, where we use 2000 problem of each problem form. [1]. These two types of problems can be processed by the E-KRHyper system. The 4000 problems are then randomly divided into four lists of equal length. Then four instances of the traditional version of E-KRHyper are started, where every instance processes one list of problems. This splitting was chosen to utilise the four CPU cores — four hardware cores, no hyperthreading — of the test machine, as a single instance of E-KRHyper does not benefit from multiple cores. The overall execution time for each of the four instances is measured and stored and the results of the proof attempt for each problem is saved. When all 4000 problems are processed, the procedure is repeated with the modified version of the E-KRHyper. In both cases a time limit of 300 seconds and a memory limit of 1024 MB are set. If the proof attempt of a problem breaks one limit, the proof attempt is interrupted.

For showing that the introduced changes improve E-KRHyper's behaviour if distinct object identifiers are used in problems, appropriate examples are needed. Unfortunately the TPTP contains only eight problems that utilise distinct object identifiers. Three of those problems are neither CNF nor FOF and thus not suitable for E-KRHyper. Two of the remaining problems cannot be used as they contain symbols that are not supported by E-KRHyper, which leaves three usable examples of the TPTP that contain distinct object identifiers for evaluating our approach. The synthetic benchmarks STORECOMM and STORECOMM-INVALID[1, 2], which are introduced now where thus be chosen as problems for evaluating our approach.

Both problem classes are situated in the theory of arrays, and as it is not natively supported by E-KRHyper, we need axioms to describe the theory of arrays. We start by introducing the function $sel : ARRAY \times INDEX \rightarrow ELEMENT$, which returns the element that is stored at the given index of the given array, and the function $sto : ARRAY \times INDEX \times ELEMENT \rightarrow ARRAY$, which returns an array that is constructed by storing a given element at the given index of a given array. Additionally we need the skolem function $sk : ARRAY \times ARRAY \rightarrow INDEX$ as a helper function, as neither the E-hyper tableau nor E-KRHyper is able to handle existential quantified variables. Those three operations are sufficient for our purpose and allow to introduce the following axioms of the theory of arrays to E-KRHyper.

$$sel(sto(A, I, E), I) = E \tag{4.1}$$

$$sel(sto(A, I, E), J) = sel(A, J) \Leftarrow I \neq J \tag{4.2}$$

$$A = B \Leftarrow sel(A, sk(A, B)) = sel(B, sk(A, B)) \tag{4.3}$$

Due to some technicalities, Axiom 4.2 cannot be used in this form. Thus we rewrite it to the semantically equivalent formula 4.4.

[1]The two problem sets are generated by using the TPTP's script `tptp2T` once with parameter `Form CNF` and once with parameter `Form FOF`

$$I = J, sel(sto(A, I, E), J) = sel(A, J) \tag{4.4}$$

This formula contains I and J in both subformulæ of the disjunctive head, i.e. it is not pure. Thus E-KRHyper looks for an appropriate purifying substitution when this clause is used in the proving process, which does not terminate for this case. Hence the formula must be adopted to prevent the infinite search for a purifying substitution which we achieve by adding the domain predicates $index(I)$ and $index(J)$ to the body of the clause. This guarantees a pure head, when the split-rule is applied and thus prevents the not regularly terminating search for a purifying substitution. This modification leads to the formula 4.5.

$$I = J, sel(sto(A, I, E), J) = sel(A, J) \Leftarrow index(I), index(J) \tag{4.5}$$

Thus equations 4.1, 4.3 and 4.5 form the axioms for the theory of arrays, that are used in the test cases.

A test case from STORECOMM is the task to show that given two permutations of unique store operations on an array they result in the same array.

A test case from STORECOMM-INVALID is the task to show that given two sequences of unique store operations that differ in at least one element they do not result in the same array.

In this context, the term *unique store operation* implies, that each index of an array is written to exactly one time.

For evaluation we, need four different kinds of test cases: STORECOMM without native handling of distinct object identifiers, STORECOMM with native handling of DOI, STORECOMM-INVALID without native handling of DOI, and STORECOMM-INVALID with native handling of DOI.

To create a test case, four parameters are needed: A list $p = 0, \ldots, n-1$, a permutation of that list, called q, a flag that indicates if we want to generate a test case for STORECOMM or STORECOMM-INVALID v and a flag that indicates if this test case uses distinct object identifiers or not d.

Independent of the chosen parameters every test case contains the three axioms which describe the theory of arrays. Additionally every test contains n unique predicates of form $index(i_x)$. with $0 \leq x < n$ introducing the constants, which represent the arrays indices. If distinct object identifiers are used, this predicates look like $index(\text{"}i_x\text{"})$.

If no distinct object identifiers are used, we need to express that all indices and are distinct, which is done by introducing $\binom{n}{2}$ unique predicates of form $false$:- $i_x = i_y$. with $(x, y) \in C_2^n$ where C_2^n is the set of 2-combinations over $\{0, \ldots, n-1\}$. Additionally we need to express that all elements are distinct, which is done by introducing $\binom{n}{2}$ unique predicates of form $false$:- $e_x = e_y$. with $(x, y) \in C_2^n$.

The actual property that is to be proven is then added by the equality predicate $T_{n,v,d}(q) = T_{n,v,d}(p)$. where $T_{k,v,d}(l)$ is defined as follows.

$$T_{k,v,d}(l) = \begin{cases} a & \text{if } k = 0 \\ sto(T_{k-1,v,d}(l), i_{l(k)}, e_0) & \text{if } k = 1 \text{ and } v = 0 \text{ and } d = 0 \\ sto(T_{k-1,v,d}(l), "i_{l(k)}", "e_0") & \text{if } k = 1 \text{ and } v = 0 \text{ and } d = 1 \\ sto(T_{k-1,v,d}(l), i_{l(k)}, e_{l(k)}) & \text{if } 0 \leq k < n \text{ and } v = 1 \text{ and } d = 0 \\ sto(T_{k-1,v,d}(l), "i_{l(k)}", "e_{l(k)}") & \text{if } 0 \leq k < n \text{ and } v = 1 \text{ and } d = 1 \end{cases}$$

For illustration, Figure 4.1 shows four files for the four different type of test cases for an array with length two. For a better overview, the axioms are shown once and then referred to by the meta symbol «AXIOMS» in the specific examples.

For evaluation, we created test cases for array length from 6 to 20 elements with 10 different samples for each length, which results in $15*10*4 = 600$ files. We split the 600 files into four lists, depending on the fact if they involve distinct objects or not and if they cover STORECOMM or STORECOMM-INVALID. Four instances of the E-KRHyper are started where each works on one list. The problems without distinct object identifiers are processed by the traditional version and the problems with distinct object identifiers are processed by the modifies version. The execution time and outcome of each problem is saved.

4.2.3 Analysis

Table 4.1 show the results for the evaluation of the 4000 chosen TPTP problem files. The first seven rows of the first column contain the different kind of results that E-KRHyper can produce for a problem. *MemoryOut* is returned, when the memory limit is exceeded in the derivation process and *Timeout* is returned, when the time limit is exceeded in the derivation process. E-KRHyper returns *ERROR* when a fatal error occurs in the reasoning process. The other four results *Satisfiable*, *Unsatisfiable*, *Theorem*, *CounterSatisfiable* are result types defined by TPTP. Execution time is the overall time needed to finish all problems and is computed by adding the execution times needed by each of the four instances of E-KRHyper.

The data gathered suppose that the introduced modifications do not impair E-KRHyper's behaviour for problems without distinct object identifiers. The differences of the number of problems that broke the memory or time limit is not of interest and the difference in the execution time is insignificant.

At the first glance, the fact that there is one problem that was classified as satisfiable by the traditional implementation but not the modified one seems like a grave impact, but it is not the case. The problem in question is YN513-1 and additional examinations were done to understand this behaviour. When running this single problem multiple times with both versions of E-KRHyper, the results were alternating between Satisfiable and Timeout. This leads to the assumption that the time needed to solve the problem is close to the timeout limit and external interruptions — like a redistribution of CPU time by the linux kernel — delay

```
sel(sto(A,I,E),I) = E.

I = J; sel(sto(A,I,E),J) = sel(A,J) :-
    index(I),
    index(J).

A = B :- sel(A,sk(A,B)) = sel(B,sk(A,B)).
```

(a) The AXIOMS.

```
<<AXIOMS>>

index(i0).
index(i1).

false :- i0 = i1.
false :- e0 = e1.

sto(sto(a,i1,e1),i0,e0) = sto(sto(a,i0,e0),i1,e1).
```

(b) STORECOMM without distinct object identifiers.

```
<<AXIOMS>>

index("i0").
index("i1").

sto(sto(a,"i1","e1"),"i0","e0") = sto(sto(a,"i0","e0"),"i1","e1").
```

(c) STORECOMM with distinct object identifiers.

```
<<AXIOMS>>

index(i0).
index(i1).

false :- i0 = i1.
false :- e0 = e1.

sto(sto(a,i1,e1),i0,e0) = sto(sto(a,i0,e0),i1,e0).
```

(d) STORECOMM-INVALID without distinct object identifiers.

```
<<AXIOMS>>

index("i0").
index("i1").

sto(sto(a,"i1","e1"),"i0","e0") = sto(sto(a,"i0","e0"),"i1","e0").
```

(e) STORECOMM-INVALID with distinct object identifiers.

Figure 4.1: Examples for STORECOMM and STORECOMM-INVALID files with size two.

Result	Traditional	Extended	Difference
Satisfiable	84	83	-1
Unsatisfiable	594	595	+1
Theorem	569	569	0
CounterSatisfiable	116	116	0
MemoryOut	1555	1548	-7
Timeout	1020	1027	+7
ERROR	62	62	0
Execution time	498741s	498592s	-149s
Average Execution time	124.69s	124.65s	-0.04s

Table 4.1: Results for 4000 TPTP problems with traditional and modified version.

the execution long enough to break the timeout limit. Therefore the difference in the outcome is not caused by the introduced changes. Running this example without time and memory limits always produces the result Satisfiable for both code versions.

The same holds for the problem LCL210-3, which was classified as MemoryOut in the traditional implementation and as Unsatisfiable in the modified version. For determining a MemoryOut, the OCaml garbage collector is needed and its behaviour is not fully predictable. Running this example multiple times without limits always produced the same result in both versions of the code.

For the second part of the evaluation the first result is that all of the 600 test cases where classified correctly, i.e. all STORECOMM cases lead to an fair derivation with an open tableau and all STORECOMM-INVALID cases lead to a closed tableau. These results were found for the DOI and non-DOI version.

Table 4.2 shows the execution times for the benchmark runs. Instead of printing the results for the 15 different array sizes, with ten samples each, we calculated the arithmetic mean for each array size and benchmark type in Table 4.2a and the variation coefficients in Table 4.2b. Figure 4.2 illustrates the average execution times, where Figure 4.2a shows the result for STORECOMM and Figure 4.2b shows the result for STORECOMM-INVALID.

The first observation one can make is, that the runtime of all four cases grows in an exponential way. This behaviour could already be seen in preliminary tests and lead to the decision to not use arrays with more then 20 elements for the sake of keeping the overall runtime of the test reasonable. Execution times of more than 20 minutes for a single problem make a real world use quite unlikely.

Another observation that is easy to make is the rather huge difference between the average execution times of STORECOMM and STORECOMM-INVALID cases with the same size. With knowledge of E-KRHyper's mode of operation and the E-hyper tableau calculus this gap is plausible. For an invalid case the derivation of a branch stops as soon as a contradiction can be found. For an valid case there is no contradiction to close the branch and thus the calculus must derive new clauses

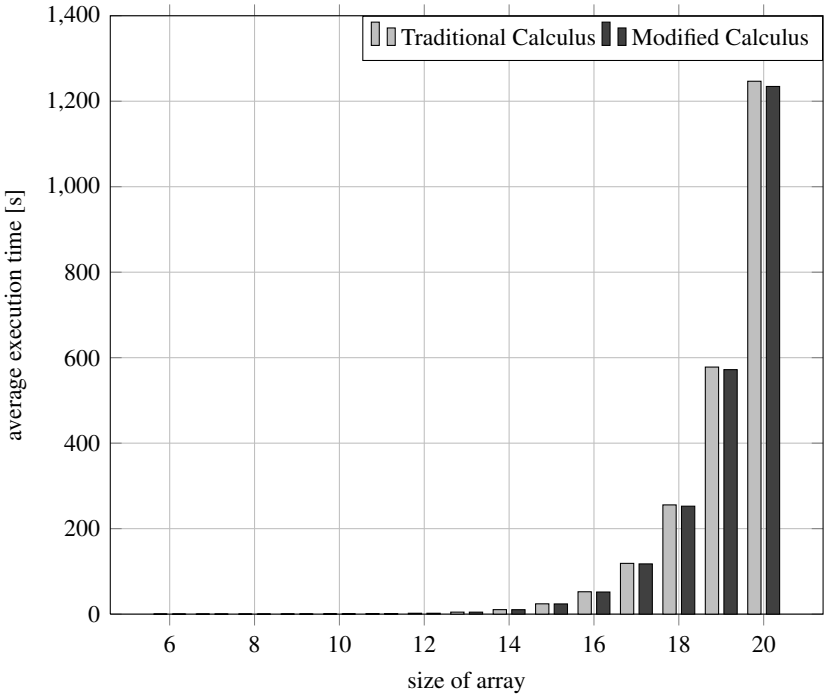
Array Size	STORECOMM		STORECOMM-INVALID	
	DOI[s]	Non-DOI[s]	DOI[s]	Non-DOI[s]
6	0.029	0.030	0.022	0.024
7	0.053	0.054	0.033	0.038
8	0.100	0.100	0.073	0.081
9	0.199	0.201	0.095	0.107
10	0.423	0.425	0.276	0.300
11	0.922	0.934	0.541	0.619
12	2.054	2.071	1.253	1.382
13	4.634	4.675	2.602	2.861
14	10.405	10.500	5.687	6.264
15	23.877	24.112	12.452	13.830
16	51.745	52.346	27.582	30.656
17	117.512	118.762	59.623	65.744
18	252.547	255.559	142.025	155.980
19	572.019	578.030	285.065	308.958
20	1234.563	1246.758	665.620	731.205

(a) Average execution times.

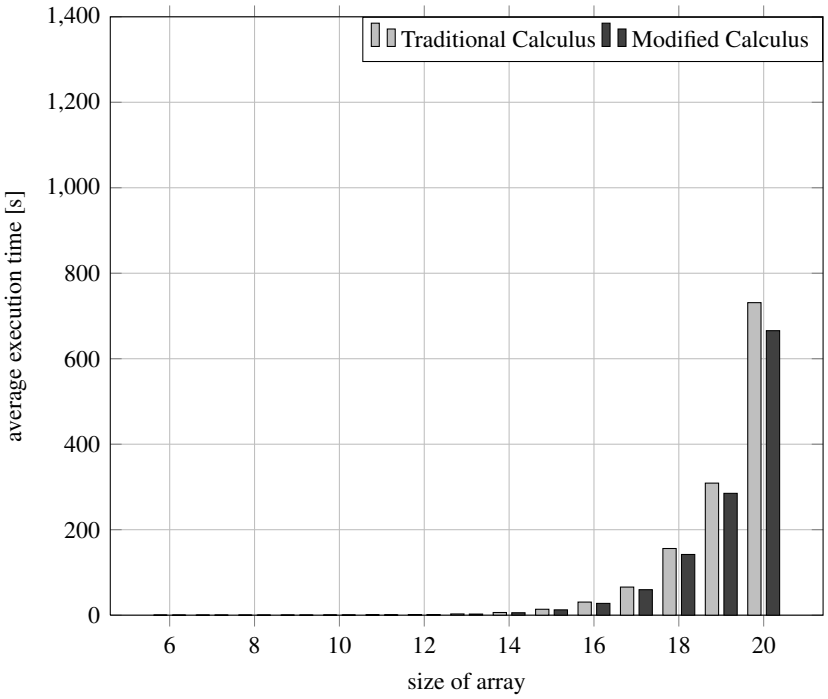
Array Size	STORECOMM		STORECOMM-INVALID	
	DOI	Non-DOI	DOI	Non-DOI
6	0.040	0.035	0.289	0.287
7	0.039	0.040	0.303	0.295
8	0.038	0.038	0.279	0.256
9	0.034	0.031	0.506	0.507
10	0.053	0.052	0.410	0.408
11	0.039	0.039	0.139	0.106
12	0.046	0.045	0.333	0.324
13	0.048	0.051	0.459	0.449
14	0.066	0.065	0.360	0.349
15	0.057	0.055	0.345	0.331
16	0.037	0.037	0.311	0.276
17	0.052	0.052	0.384	0.370
18	0.053	0.052	0.294	0.268
19	0.041	0.042	0.447	0.450
20	0.044	0.045	0.292	0.263

(b) Variation coefficients for the benchmark.

Table 4.2: Statistical data of the results for STORECOMM and STORECOMM-INVALID.



(a) Average execution time for STORECOMM.



(b) Average execution time for STORECOMM-INVALID.

Figure 4.2: Execution times for STORECOMM (a) and STORECOMM-INVALID (b).

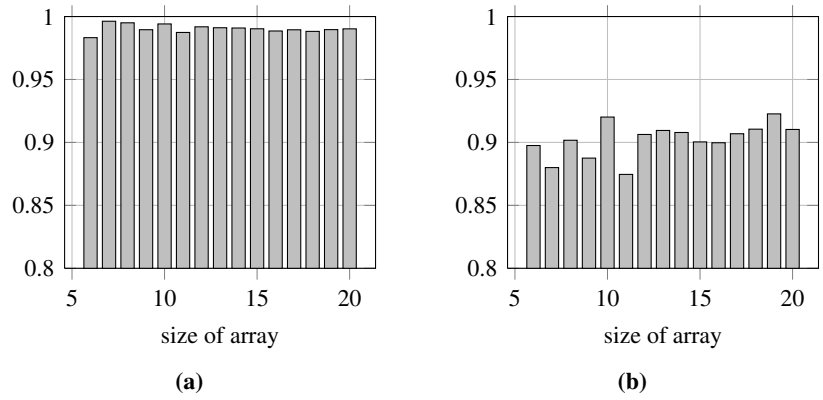


Figure 4.3: Relative execution times for STORECOMM (a) and STORECOMM-INVALID (b).

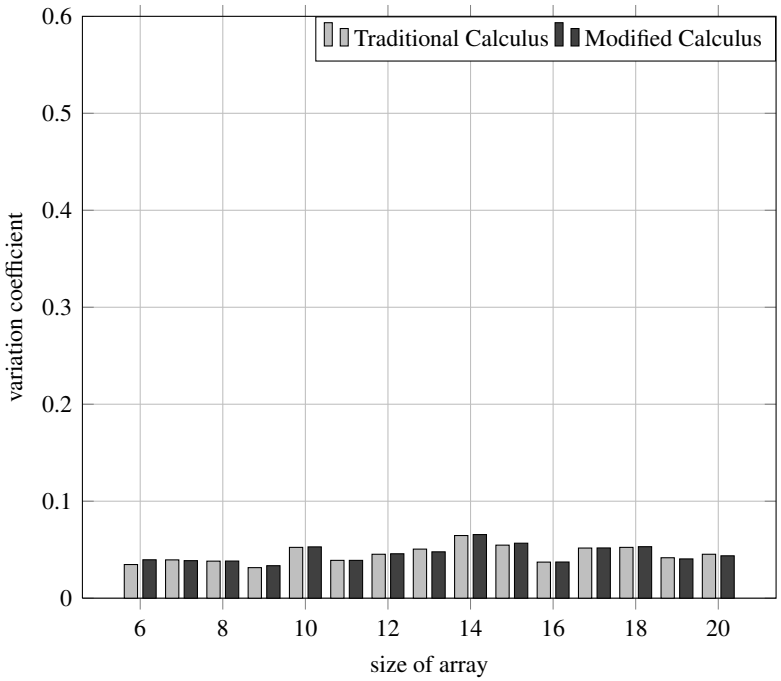
until the criterion for an exhausted branch is met, i.e. all newly generated clauses are redundant with respect to the set of already existing clauses. As E-KRHyper derives new clauses by applying all applicable rules to all suitable combinations of clauses, it is plausible that this needs more time.

Due to the exponential growth, a comparison of E-KRHyper's runtime performance in the traditional and modified version is not possible with the diagrams in Figure 4.2 as the bars for small array sizes are not visible at all.

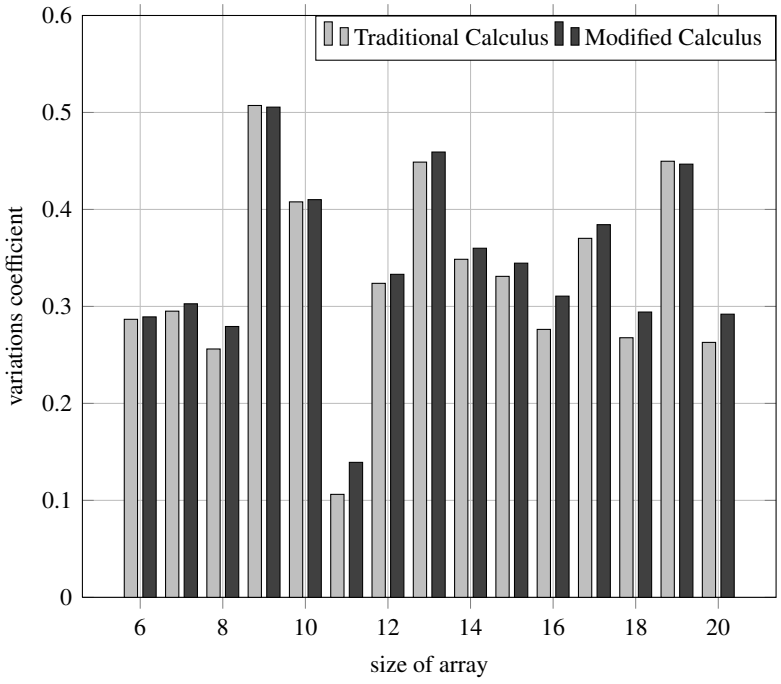
Therefore we have decided to normalize the execution times by choosing the value of the tradition version to be 1 and thus dividing the result for the DOI version by the according value of the non-DOI version. This leads to a relative comparison of both versions, which is shown in Figure 4.3, which is suitable for comparing the performance of the DOI and non-DOI version. For STORECOMM (see Figure 4.3a) the differences in the average execution times of traditional and modified version is insignificant. A possible explanation relies on the fact that an exhausted branch needs to be derived for showing a clause set to be satisfiable and thus many clauses need to generated. As neither the unique name assumption nor the inequality axioms for the non-DOI version do contribute in this process, it is plausible that the execution times do not change.

For STORECOMM-INVALID (see Figure 4.3b) the execution times for the modified version are about 7 to 13 percent faster than the ones of the traditional implementation. This supports the theoretical assumption that the use of distinct object identifiers can lead to an improvement of the reasoning process. With native handling of distinct object identifiers, the reasoning stops the instant a (possible) unit contradiction is found. In the traditional implementation E-KRHyper is exhaustively creating new clauses and is able to close a branch if it — by chance — was the right combination of constituent clauses that lead to a contradiction.

As no obvious tendency can be seen in Figure 4.3b, we calculated variations coefficients. The results are shown in Table 4.2 and illustrated in Figure 4.4.



(a) Variation coefficients for STORECOMM.



(b) Variation coefficients for STORECOMM-INVALID.

Figure 4.4: Variation coefficients for STORECOMM (a) and STORECOMM-INVALID (b).

For the STORECOMM cases (see Figure 4.4a) the variations coefficients' values are below 0.1 and uniformly distributed which proposes that the chosen samples have a high quality and thus the result of our benchmark for STORECOMM has a high validity.

For the STORECOMM-INVALID cases (see Figure 4.4b) on the other hand, the variations coefficients' value are rather high and heavily fluctuating. In combination with the unsteadiness of relative gain of the mean execution time this leads to the assumption that there is a factor involved that influences the benefit of distinct object identifiers.

A possible factor that might lead to such a behaviour is the structure of the nested store operation, i.e. the order of the permutation used. This is plausible as the order of the store operations determine how many proof steps are needed. This is a subject for further investigation.

For an overall comparison of the system's performance, we wanted to compare our results with CVC3[4] which is a satisfiability solve with native handling of the theories of arrays. As some preliminary tests with STORECOMM problems of size 20 from the SMT-LIB[5] resulted in execution times of less than a second, we decided to not do a full evaluation as CVC3 performance is clearly superior to the traditional and modified version of E-KRHyper.

We conclude this section by summarizing the results of the analysis. The first result is that the modification of the calculus does not impair the reasoning with problems that do not contain distinct object identifiers. The second result is that the execution time of E-KRHyper grows exponentially with respect to the array size for the benchmarks STORECOMM and STORECOMM-INVALID. We have further shown with a high confidence, that the use of distinct object identifiers does not lead to a better performance for test cases of the STORECOMM benchmark. This result is independent of the array size. For the test cases of the STORECOMM-INVALID benchmark the use of distinct object identifiers lead to a speed-up of 7 to 13 percent. The results allow to assume that the order of the store operations influences the improvement of the execution time. The last result of this sections states that neither the traditional nor the modified version of E-KRHyper can compete in the STORECOMM and STORECOMM-INVALID benchmarks with a satisfiability prover, which support native handling of arrays.

Due to the positive results of the evaluation, the introduced changes will be part of the next release of the E-KRHyper system.

Chapter 5

Related Work

As the topic of reasoning with the unique name assumption or distinct object identifiers has not yet gotten a lot of attention in the field of automated reasoning, there is only one paper known to us that is related with this thesis.

In [24], Schulz and Bonacina show a way to handle distinct object identifiers in the superposition calculus. The superposition calculus works on a set of clauses and uses derivation rules on members of the set to derive and add new clauses. If the empty clause can be derived the set is unsatisfiable.

Schulz and Bonacina identified two forms of clauses that require a change of the calculus. These two forms are semantically equivalent with what is called an *unit contradiction* and an *object tautology clause* in our approach and are thus handled in a similar way.

Four new rules have been introduced. If a clause contains a literal of the form $i \neq j$ with $i, j \in \mathbb{D}$ and $i \neq j$, the *object tautology deletion* rule is used to remove this clause from the clause set as it cannot contribute in deriving the empty clause. This behaviour is similar with declaring an object tautology clause as negligible and thus rewriting it in the modified E-hyper tableau calculus.

The counterpart to the unit-cont-right inference rule and Inc extension rule is the *object equality cutting* rule. It removes literals of the form $i \simeq j$ with $i, j \in \mathbb{D}$ and $i \neq j$ from clauses as they are invalid and the goal of the calculus is to derive the empty clause.

There are two additional rules, which can be seen as variations of the object equality cutting rule for not fully instantiated literals. If there is a clause that contains a literal of the form $X \simeq i$ or $X \simeq Y$ with $i \in \mathbb{D}, X, Y \in \mathbb{V}$, a substitution σ is applied to the whole clause to derive the literal $i \simeq j$ with $i, j \in \mathbb{D}$ and $i \neq j$, which is then deleted from the clause.

To evaluate their approach they extended a version of the E-prover[22, 23] and used instances of the *STORECOMM* and *STORECOMM-INVALID* benchmark classes[1, 2]. To rate the performance, they compared four different systems: CVC[25], which is a validity checker where the axioms for the theory of arrays is part of the actual system, CVC-Lite[6], which is the successor of CVC and has

native support for the theory of arrays, as well, a version of the E-prover with support for distinct objects identifiers and a version of the E-prover that does not support DOI and thus needs additional facts to define the inequality of distinct array indices.

The results of those tests are promising. The execution times of the version of E that supports DOI for the valid cases are identical with the runtime of CVC and with the invalid cases they are even lower. The version of E that does not support DOI and therefore relies on additional facts for the reasoning process is considerably slower, which supports the claim, that native handling of the unique name assumption can be beneficial for certain reasoning tasks.

Chapter 6

Conclusion and Outlook

Using the unique name assumption instead of facts to define inequalities of constants reduces the number of clauses in knowledge bases and thus allows the reader to focus on the parts that are of actual importance for a problem. A smaller set of clauses allows faster reasoning, as well.

This thesis shows a way to extend the E-hyper tableau calculus for reasoning with the unique name assumption. It was proven that the extended calculus is sound and complete, and we implemented the calculus in the E-KRHyper system.

This implementation was then used for evaluating whether the use of distinct object identifiers has an impact on the outcome or needed time for proving a problem. The evaluation shows that the implemented changes do not harm the reasoning without distinct object identifiers and significantly improves the execution time for unsatisfiable problems with distinct object identifiers. For proving problems that are satisfiable and contain distinct object identifiers there is an improvement as well, but it is hardly significant. This behaviour was explained with E-KRHyper's mode of operation.

Another observation made in the evaluation process was the large scattering of execution times for STORECOMM-INVALID samples with same array size. We suppose that the structure of the problem, i.e. the order of the store operations, has an impact on the execution time and the difference of the execution times between the traditional and modified version of the E-KRHyper. Thus future work is needed to perform a thorough study on the correlation of array size, order of store operations and execution time for a single sample to elaborate and support this assumption.

Independent of the actual execution time and the speed-up of the execution time by using distinct object identifiers, we learned that we cannot compete with a SMT-solver like CVC3 in STORECOMM and STORECOMM-INVALID which is not that exceptional as CVC3 is specifically tailored for solving such problems.

Additional tests with different benchmarks can be beneficial for judging the overall impact of distinct object identifiers in reasoning problems. An example of another benchmark can be found in [15]. Ganzinger and Sofronie-Stokkermans

introduce a way for reasoning in many valued logics that make heavy use of distinct truth values and thus might be suitable for using the unique name assumption.

Another point for future work is the implementation. For this thesis some parts were implemented in a straight forward manner and there is ample opportunity for optimisation.

By the results of Schulz and Bonacina and our approach, the use of distinct objects can be beneficial for the reasoning process. Therefore we encourage an extension of the TPTP by more problems that use distinct object identifiers to raise the interest for this topic.

While developing this thesis, certain ideas evolved, how this can be used in future extensions: If there is a formalism to mark a function as being bijective, the current calculus could be extended without great effort to treat those functions with the unique name assumption. Another idea is the introduction of a typed logic in such a case the calculus needs only small adoption to be able to handle equalities of two different types as an unit contradiction. The probably most useful idea is to introduce a mechanism into E-KRHyper which is able to determine whether a constant is better used as distinct object identifier or basic constant. This decision can be dependent on the occurrences of inequality facts, and it would provide a mean for a faster derivation process.

Bibliography

[1] Alessandro Armando, Maria Paola Bonacina, Aditya Kumar Sehgal, and Silvio Ranise. High-performance deduction for verification: A case study in the theory of arrays. In *Notes of the Workshop on Verification, Third Federated Logic Conference (FLoC02)*, pages 103–112, 2002.

[2] Alessandro Armando, Maria Paola Bonacina, Silvio Ranise, and Stephan Schulz. New results on rewrite-based satisfiability procedures. *ACM Transactions on Computational Logic (TOCL)*, 10(1), 2009.

[3] Leo Bachmair and Harald Ganzinger. Equational reasoning in saturation-based theorem proving. In Wolfgang Bibel and Peter H. Schmitt, editors, *Automated Deduction — A Basis for Applications*, volume I: Foundations. Kluwer, Dordrecht, 1998.

[4] Clark Barrett and Cesare Tinelli. CVC3. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 298–302. Springer, jul 2007. ISBN 978-3-540-73367-6. Berlin, Germany.

[5] Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.smt-lib.org`, 2004-2012. Accessed January, the 27th 2012.

[6] Clark W. Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker category B. In Rajeev Alur and Doron Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer, 2004. ISBN 3-540-22342-8.

[7] Peter Baumgartner and Ulrich Furbach. PROTEIN: A PROver with a Theory Extension INterface. In Alan Bundy, editor, *Automated Deduction - CADE-12, 12th International Conference on Automated Deduction, Nancy, France, June 26 - July 1, 1994, Proceedings*, volume 814 of *Lecture Notes in Computer Science*, pages 769–773. Springer, 1994. ISBN 3-540-58156-1.

[8] Peter Baumgartner and Ulrich Furbach. Living books, automated deduction and other strange things. In Dieter Hutter and Werner Stephan, editors, *Mechanizing Mathematical Reasoning, Essays in Honor of Jörg H. Siekmann on the Occasion of His 60th Birthday*, volume 2605 of *Lecture Notes in Computer Science*, pages 249–267. Springer, 2005. ISBN 3-540-25051-4.

[9] Peter Baumgartner and Renate A. Schmidt. Blocking and other enhancements for bottom-up model generation methods. In Ulrich Furbach and Natarajan Shankar, editors, *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4130 of *Lecture Notes in Computer Science*, pages 125–139. Springer, 2006. ISBN 3-540-37187-7.

[10] Peter Baumgartner, Ulrich Furbach, and Ilkka Niemelä. Hyper tableaux. In José Júlio Alferes, Luís Moniz Pereira, and Ewa Orlowska, editors, *Logics in Artificial Intelligence, European Workshop, JELIA '96, Évora, Portugal, September 30 - October 3, 1996, Proceedings*, volume 1126 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 1996. ISBN 3-540-61630-6.

[11] Peter Baumgartner, Ulrich Furbach, Margret Groß-Hardt, and Alex Sinner. Living book - deduction, slicing, and interaction. *Journal of Automated Reasoning*, 32(3):259–286, 2004.

[12] Peter Baumgartner, Ulrich Furbach, and Björn Pelzer. Hyper tableaux with equality. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 492–507. Springer, 2007. ISBN 978-3-540-73594-6.

[13] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, Cambridge, 2003. ISBN 0521391156.

[14] Ulrich Furbach, Ingo Glöckner, and Björn Pelzer. An application of automated reasoning in natural language question answering. *AI Communications*, 23(2-3):241–265, 2010.

[15] Harald Ganzinger and Viorica Sofronie-Stokkermans. Chaining techniques for automated theorem proving in many-valued logics. In *Multiple-Valued Logic, 30th IEEE International Symposium, ISMVL 2000, Portland, Oregon, USA, May 23-25, 2000, Proceedings.*, pages 337–344. IEEE Computer Society, 2000.

[16] INRIA. Website of OCaml. `http://caml.inria.fr/`, 2005-2012. Accessed January, the 27th 2012.

[17] Robert Nieuwenhuis and Albert Rubio. Paramodulation-based theorem proving. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. 2001.

[18] Björn Pelzer. E-KRHyper – extending the KRHyper theorem prover with equality reasoning. Diplomarbeit, University of Koblenz-Landau, March 2007.

[19] Björn Pelzer. Project website of E-KRHyper. `http://userpages.uni-koblenz.de/~bpelzer/ekrhyper/`, 2007-2012. Accessed January, the 27th 2012.

[20] Björn Pelzer and Christoph Wernhard. System description: E-KRHyper. In Frank Pfenning, editor, *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*, volume 4603 of *Lecture Notes in Computer Science*, pages 508–513. Springer, 2007. ISBN 978-3-540-73594-6.

[21] David Poole, Alan Mackworth, and Randy Goebel. *Computational Intelligence: A Logical Approach*. Oxford University Press, New York, January 1998.

[22] Stephan Schulz. E - a brainiac theorem prover. *AI Communications*, 15(2-3): 111–126, 2002.

[23] Stephan Schulz. System description: E 0.81. In David A. Basin and Michaël Rusinowitch, editors, *Automated Reasoning - Second International Joint Conference, IJCAR 2004, Cork, Ireland, July 4-8, 2004, Proceedings*, volume 3097 of *Lecture Notes in Computer Science*, pages 223–228. Springer, 2004. ISBN 3-540-22345-2.

[24] Stephan Schulz and Maria Paola Bonacina. On handling distinct objects in the superposition calculus. In Boris Konev and Stephan Schulz, editors, *Implementation of Logics, 5th International Workshop, IWIL 2005, Montevideo, Uruguay, March 13th, 2005, Proceedings*, pages 66–77, 2005.

[25] Aaron Stump, Clark W. Barrett, and David L. Dill. CVC: A cooperating validity checker. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 500–504. Springer, 2002. ISBN 3-540-43997-8.

[26] Geoff Sutcliffe. The TPTP website. `www.tptp.org`, 2004-2012. Accessed January, the 27th 2012.

[27] Geoff Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4): 337–362, 2009.

[28] Christoph Wernhard. System description: KRHyper. Fachberichte Informatik 14–2003, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Universitätsstr. 1, D-56070 Koblenz, 2003. URL http://www.uni-koblenz-landau.de/koblenz/fb4/publications/fachberichte/fb2003/rr-14-2003.pdf.

[29] Christoph Wernhard. Project website of KRHyper. http://userpages.uni-koblenz.de/~wernhard/krhyper/, 2003-2012. Accessed January, the 27th 2012.