# Equational reasoning in cocktail

Coppens, David L.

*Award date:*
2007

EINDHOVEN UNIVERSITY OF TECHNOLOGY
Department of Mathematics and Computer Science

# Equational Reasoning in Cocktail

by
D.L. Coppens

Supervisor:
dr.ir. M.G.J. Franssen

August 14, 2007

# Abstract

Cocktail is a software tool intended to offer assistance to the process of deriving a program from its specification, using the Hoare/Dijkstra method. It contains an interactive theorem prover, which allows users to interactively construct proofs for theorems, as well as an automated theorem prover to automatically construct proofs. The automated theorem prover of Cocktail is based on the tableau method. One of its shortcomings is the lack of support for equational reasoning.

This thesis describes an extension of the tableau method to allow reasoning with equalities. Rigid E-unification problems are introduced, along with an algorithm to solve rigid E-unification problems. Calculus $\mathcal{BSE}$ is introduced to rewrite rigid E-unification problems into more simple ones.

In order to use this calculus we introduce sets of constraints. We need to determine satisfiability of such constraint sets. This is done by calculating their solved form. From this solved form several simple systems are derived, using a graph structure. If at least one of these simple systems is satisfiable, the constraint set is satisfiable.

# Contents

# Chapter 1

# Introduction

When using an algorithm that was specified to solve some (difficult) problem, it is important that this algorithm is *correct*. Here, correctness means that the algorithm adheres to its specification (i.e. it solves the problem it is supposed to solve).

An approach to proving correctness of an algorithm is to formalize the specification of the algorithm using first-order logic. Then the algorithm can be derived from this specification step by step, using Hoare triples defining pre- and postconditions of each step. Each step gives rise to a set of proof obligations that need to be fulfilled in order for the derivation to be correct. This approach to programming was introduced by E.W.D. Dijkstra and is taught to computer science students at Eindhoven University of Technology. For more information about this method we refer to [Kal90].

## 1.1 Problem statement

Cocktail is a tool that supports the approach described above. It allows the designer of an algorithm to give a formal specification and derive a program from this specification. The tool keeps track of the proof obligations and allows users to fulfill these obligations both in an interactive and automatic way. A short introduction to Cocktail is given in chapter 3.

In [Fra00] a listing of the shortcomings of Cocktail is presented. Amongst these shortcomings is the lack of support for equational reasoning in the Automated Theorem Prover (ATP). Since most proofs in this context require equational reasoning, many seemingly easy proofs can not be constructed

automatically. Fortunately, the shortcoming is presented in [Fra00] along with a possible solution: rigid E-unification.

The main goal of the project that is described in this thesis can be stated as follows:

> *"Study rigid E-unification in the literature and create an implementation of an algorithm that can solve rigid E-unification problems, which can be used in the Cocktail architecture."*

## 1.2 Structure of this thesis

First, preliminary definitions are given that are used throughout the thesis, as well as some explanation about the notation used.

The rest of the thesis is structured as follows:

#### Context and background

We will discuss the context and background of this project in chapter 3. The tool Cocktail is introduced briefly. Its ATP is discussed in somewhat more detail, since it is the focus of this project. During this discussion it will become clear why the currently used ATP is unable to reason with equations.

#### Problem definition

The formal definition of (rigid) E-unification is given in chapter 4. Several different versions of E-unification are discussed, along with their properties. It will become clear that we will need to solve *rigid* E-unification problems in order to reason with equality using the ATP of Cocktail.

#### Solution to the problem

After the formal definition of the problem, chapter 5 describes the steps needed to find a solution to a rigid e-unification problem. First a calculus, named $\mathcal{BSE}$, is described that can be used to rewrite the problem. Furthermore, an algorithm is discussed to determine the satisfiability of *ordering constraints*. Such constraints are introduced along with the calculus $\mathcal{BSE}$ and are the key to solving the rigid e-unification problem.

#### Implementation

The algorithm described in chapter 5 is implemented. Implementation specific details regarding this solving method are described in chapter 6.

#### Conclusion

Finally, the outcome of the project is discussed in chapter 7, along with a description of future work that might be considered. This thesis is concluded with an appendix in which an example of a proof using the tableau method extended with rigid e-unification is described in detail.

# Chapter 2

# Preliminaries

This chapter introduces some general notation and definitions used throughout this thesis.

## 2.1 Notation

$P[x]$ means that first-order logic formula $P$ contains subterm $x$. Substituting a variable $x$ by some term $t$ in term $P[x]$ is denoted as $P[x := t]$, following the convention used in [Fra00].

An equality is denoted by $s = t$. In this thesis no difference is made between $s = t$ and $t = s$, since the $=$ operator is symmetric. Formulas of the form $\neg(s = t)$ are written as $s \neq t$ and are called inequalities.

The following notation is used to indicate that some equality $s = t$ holds in a given context: $\{s_1 = t_1, \ldots, s_n = t_n\} \vdash_\forall s = t$.

## 2.2 Definitions

We define first-order logic. Let $\mathcal{P}$ be a set of predicate symbols, $\mathcal{F}$ a set of function symbols and let $\mathcal{V}$ be a set of variables. First-order logic is built up using terms and formulas. In [HR00] these are defined as follows in Backus Naur form:

**Definition 2.2.1 (Terms)**
*Let $\mathcal{T}$ be the set of terms. $\mathcal{T}$ is defined as follows:*

$$\mathcal{T} ::= \quad x \mid f(t_1, \ldots, t_n)$$

*where* $x \in V, f \in \mathcal{F}$ *and* $f$ *has arity* $n$, $(\forall i : 1 \leq i \leq n : t_i \in \mathcal{T})$ *and* $t, s \in \mathcal{T}$. *Functions* $f$ *with arity zero are regarded as constants.*

**Definition 2.2.2 (Formulas)**
*Let Prop be the set of formulas. Prop is defined inductively as follows:*

$$\begin{aligned} Prop ::= \quad & P(t_1, \ldots, t_n) \mid (\neg Q) \mid (Q \wedge R) \mid (Q \vee R) \mid \\ & (Q \Rightarrow R) \mid (\forall x :: Q) \mid (\exists x :: Q) \end{aligned}$$

*where* $P \in \mathcal{P}$ *with arity* $n$, $(\forall i : 1 \leq i \leq n : t_i \in \mathcal{T})$, $x \in \mathcal{V}$, $Q \in Prop$ *and* $R \in Prop$.

Variables that occur in a term can be free or can be bound by some quantifier. We will formally define these concepts now.

**Definition 2.2.3 (Free variables)**
*Function FV that calculates all free variables in some first-order logic formula or term is defined recursively as follows:*

- $FV(x) = x$, *for* $x \in \mathcal{V}$
- $FV(f(t_1, \ldots, t_n)) = FV(t_1) \cup \ldots \cup FV(t_n)$
- $FV(s = t) = FV(s) \cup FV(t)$
- $FV(P(t_1, \ldots, t_n)) = FV(t_1) \cup \ldots \cup FV(t_n)$
- $FV(\neg Q) = FV(Q)$
- $FV(Q \wedge R) = FV(Q \vee R) = FV(Q \Rightarrow R) = FV(Q) \cup FV(R)$
- $FV(\forall x : x \in U : Q) = FV(Q) \setminus \{x\}$
- $FV(\exists x : x \in U : Q) = FV(Q) \setminus \{x\}$

In a similar way we define bound variables:

**Definition 2.2.4 (Bound variables)**
*Function BV that calculates all bound variables in some first-order logic formula is defined recursively as follows:*

- $BV(t) = \emptyset$

- $BV(\neg Q) = BV(Q)$
- $BV(Q \wedge R) = BV(Q \vee R) = BV(Q \Rightarrow R) = BV(Q) \cup BV(R)$
- $BV(\forall x :: Q) = BV(Q) \cup \{x\}$
- $BV(\exists x :: Q) = BV(Q) \cup \{x\}$

Now we can introduce ground terms.

**Definition 2.2.5 (Ground term)**
*A term $t$ is* ground *iff $FV(t) = \emptyset$.*

In chapter 5 ordered constraints are introduced as a means to solve rigid E-unification problems. For this purpose an ordering $\succeq$ on terms is introduced. This ordering has to have the following property to ensure completeness of the solving procedure, as stated in [Com90]:

The ordering is total on ground terms, i.e. for all ground terms $s, t$ and $u$ ordering $\succeq$ has the following properties:

- $s \succeq t \wedge t \succeq s \Rightarrow s = t$ (antisymmetry)
- $s \succeq t \wedge t \succeq u \Rightarrow s \succeq u$ (transitivity)
- $s \succeq t \vee t \succeq s$ (totality)

An ordering that adheres to this condition is the lexicographical path ordering (LPO). Let $\succ_{\mathcal{F}}$ denote some *precedence ordering* on $\mathcal{F}$.
The operator $\succ_{lpo}^{\mathcal{F}}$ denotes the lexicographical path ordering (LPO) generated by $\succ_{\mathcal{F}}$, which is used to solve rigid E-unification problems. Since the same ordering and precedence function is used throughout this thesis, the LPO operator is simplified to $\succ$. LPO is defined in [Com90] as follows, where $s \succeq t \Leftrightarrow s \succ t \vee s = t$.

**Definition 2.2.6 (Lexicographic path ordering)**
*Let $s \equiv f(s_1, \ldots, s_n)$ and $t \equiv g(t_1, \ldots, t_m)$ be two terms. Then $s \succ t$, if and only if one of the following holds:*

1. $(\exists i : 1 \leq i \leq m : s_i \succeq t)$
2. $f \succ_{\mathcal{F}} g$ *and* $(\forall j : 1 \leq j \leq m : s \succ t_j)$
3. $f \equiv g$, $(s_1, \ldots, s_n) \gg (t_1, \ldots, t_m)$ *and* $(\forall j : 1 \leq j \leq m : s \succ t_j)$

*where* $(s_1, \ldots, s_n) \gg (t_1, \ldots, t_m)$ *if and only if* $(\exists j : 1 \leq j \leq n : (\forall i : 1 \leq i \leq j : s_i \equiv t_i \wedge s_j \succ t_j))$.

If two terms $s$ and $t$ are not comparable using LPO we denote this as $s \nsim t$. For instance assume $s \equiv f(x)$ and $t \equiv f(y)$, then we do not know if $s \succ t$, $t \succ s$ or $t = s$, since $s$ and $t$ are not ground and $\succ$ is only total on ground terms.

### 2.2.1 Skolem Negation Normal Form

In the method used to solve rigid E-unification problems that is described in chapter 5, formulas need to be in *skolem negation normal form*. Since Cocktail allows arbitrary formulas in first-order logic, an algorithm is needed to translate such an arbitrary first-order logic formula into skolem negation normal form. This section defines skolem negation normal form and describes how to perform translations from arbitrary first-order logic formulas to an equivalent first-order logic formula in skolem negation normal form.

**Definition 2.2.7 (negation normal form)**
*A formula $P$ is in negation normal form iff negations occur only in front of atomic subformulas of $P$, and $P$ contains only the connectives $\vee$ and $\wedge$ and predicates $\forall$, and exists.*

A set of rules to translate a formula in first-order logic to an equivalent formula in negation normal form is listed in [NW01] and presented in table 2.1.

$$\begin{array}{rcc}
\neg(A \vee B) & \rightarrow & \neg A \wedge \neg B \\
\neg(A \wedge B) & \rightarrow & \neg A \vee \neg B \\
\neg\neg A & \rightarrow & A \\
A \Rightarrow B & \rightarrow & (\neg A \vee B) \\
\neg(\forall x : x \in U : A) & \rightarrow & (\exists x : x \in U : \neg A) \\
\neg(\exists x : x \in U : A) & \rightarrow & (\forall x : x \in U : \neg A)
\end{array}$$

Table 2.1: Translation to negation normal form

After the original formula $P$ is translated to an equivalent formula $P'$ in negation normal form, skolemization is used to translate $P'$ to skolem negation normal form.

*Skolemization* is the process of removing existential quantifiers in a formula and replacing the variables bound by the removed quantifiers with functions

that represent the originally bound variables. Such functions are called *skolem functions*. It is important that such a function preserves the context in which the variable it represents exists. For example, after removing the existential quantifier in the formula $(\forall x : x \in U : (\exists y : y \in U : P(x) \vee P(y)))$ the resulting formula would be $(\forall x : x \in U : P(x) \vee P(f(x)))$, in which $f$ is the skolem function representing bound variable $y$. The argument of $f$ indicates that $y$ depends on $x$ in the original formula.

A function to skolemize a first-order logic formula is suggested in [Fra00] and is defined recursively in table 2.2. Note that since the input formula is in negation normal form, no rules for translating implications and negated terms are needed in the definition.

| | | | |
|---|---|---|---|
| $skol(A)$ | $=$ | $A$ | for atomic $A$ |
| $skol(\neg A)$ | $=$ | $\neg A$ | for atomic $A$ |
| $skol(A \wedge B)$ | $=$ | $skol(A) \wedge skol(B)$ | |
| $skol(A \vee B)$ | $=$ | $skol(A) \vee skol(B)$ | |
| $skol(\forall x : x \in U : A)$ | $=$ | $(\forall x : x \in U : skol(A))$ | |
| $skol(\exists x : x \in U : A)$ | $=$ | $skol(A[x := f(x_1, \ldots, x_n)])$ | where skolem function $f$ is fresh with arity $n$, and $FV(\exists x : x \in U : A) = \{x_1, \ldots, x_n\}$. |

Table 2.2: Skolemization

**Definition 2.2.8 (Skolem negation normal form)**
*A formula $P$ is in skolem negation normal form, if $P$ is in negation normal form and $P$ only contains universal quantifiers (i.e. existential quantifiers are replaced by skolem functions).*

# Chapter 3

# Project context

This chapter discusses the context in which the project is placed. First we briefly introduce Cocktail. Then its automated theorem prover is discussed.

## 3.1 Cocktail

This section provides some general information about *Cocktail*. Cocktail was designed and developed by Michael Franssen as a PhD project at Eindhoven University of Technology. Documentation of the tool and its design can be found in [Fra00].

Cocktail is a software tool intended to offer assistance to the process of deriving a program from its specification, using the Hoare/Dijkstra method described in chapter 1. It contains an interactive theorem prover, which allows users to interactively construct proofs for theorems, as well as an automated theorem prover to automatically construct proofs. Cocktail uses first-order logic to express theorems, program specifications, etc.

The tool is based upon three important concepts: a pure type system (PTS) that corresponds to first-order logic, an automated theorem prover (ATP) and a Hoare logic. For this project, only the ATP of Cocktail is of interest. The ATP currently used in Cocktail is described in the next section.

## 3.2 Tableau method

The ATP integrated in Cocktail is based on the *semantic tableaux method*. This section describes semantic tableaux and how they can be used to con-

| | | | |
|---|---|---|---|
| special rule | $\frac{\neg\neg P}{P}$ | | |
| $\alpha$ rule | $\frac{P \wedge Q}{P, Q}$ | $\frac{\neg(P \Rightarrow Q)}{P, \neg Q}$ | $\frac{\neg(P \vee Q)}{\neg P, \neg Q}$ |
| $\beta$ rule | $\frac{\neg(P \wedge Q)}{\neg P \mid \neg Q}$ | $\frac{P \Rightarrow Q)}{\neg P \mid Q}$ | $\frac{P \vee Q}{P \mid Q}$ |
| $\gamma$ rule | $\frac{\neg\exists x : x \in U : P}{\neg\exists x : x \in U : P, \neg P[x := t]}$ | $\frac{\forall x : x \in U : P}{\forall x : x \in U : P, P[x := t]}$ | |
| $\delta$ rule | $\frac{\exists x : x \in U : P}{P[x := \alpha]}$ | $\frac{\neg\forall x : x \in U : P}{\neg P[x := \alpha]}$ | |

Table 3.1: Tableau-expansion rules, taken from [Fra00]. $\alpha$ represents a fresh variable of type $U$. $t$ represents an arbitrary term of type $U$.

struct proofs.

Let $\Gamma$ be a first-order logic proposition (e.g. $\Gamma \equiv \gamma_0 \wedge \ldots \wedge \gamma_{n-1}$). $\Gamma$ is called the context. Now, suppose we want to prove that some first-order logic formula $\mathcal{P}$ is true in a given context $\Gamma$, i.e. we want to prove that $\Gamma \Rightarrow \mathcal{P}$ is a tautology.

The basic idea of the tableau method is to try to show that the negation of the proposition that is to be proved (e.g. $\neg(\Gamma \Rightarrow \mathcal{P})$) never holds, by constructing a *tableau*.

A tableau is a tree structure in which each node is labelled. Each label is a set of formulas. This tableau is expanded using *tableau-expansion rules* listed in table 3.1.
Construction of such a tableau is described in [Fra00] as follows:

1. Start with a single node, labelled with $\{\neg P\}$, where $P$ is the formula to be proved.

2. Select a leaf from the partially constructed tree. Select from the corresponding label $L$ a formula $X$ to which one of the rules from figure 3.1

can be applied. Extend the leaf with a number of nodes equal to the number of conclusions of the rule. Conclusions are separated by a ' | ' and can contain several formulas separated by a ' , '. A successor node is labelled with $(L \setminus \{X\}) \cup Y$, where $Y$ is the conclusion for which the successor was created. There exists a model for at least one of the successor nodes if and only if there exists a model for the parent node.

3. Repeat step 2 until:

   (a) Every leaf either contains the special predicate symbol $\perp$ or it contains both an $X$ and $\neg X$ for some formula $X$. $X$ may be different for every leaf. Such a leaf is called closed. If all leaves of a tableau are closed, the tableau itself is also called closed.

   (b) There exists a non-closed leaf which contains only literals different from $\perp$. Such a leaf actually provides an interpretation, and hence a model, in which the original formula $P$ does not hold, hence the tableau provides a counterexample.

If all leafs of the tableau are closed, we can conclude that the formula stored in the root can never hold, and thus, that the formula that was to be proved is a tautology. It can also happen that no more rules can be applied to any leaf in the tableau, while not all leafs are closed. In that case, it is shown that the formula in the root is true for at least one instantiation. This means that the original formula is not a tautology, and the formulas on the non-closed leaf(s) provide(s) a counter-example.

### 3.2.1 Closing a leaf of the tableau

As discussed above, leafs of the tableau can be closed if its label contains two contradicting formulas. So a very important aspect of the tableau method is to identify such contradictions.

If the label of a leaf contains some formula $P$ and its negation $\neg P$, it is evident that there is a contradiction and the leaf can be closed. However, first-order logic terms can contain free variables, which can be instantiated. This means that there might be terms in the label that could close the leaf if some suitable instantiation of the free variables is found. For example, it might be the case that the label contains terms $Q(x)$ and, for instance, $\neg Q(y)$. In this case, the leaf can be closed if $x = y$. Finding such a suitable instantiation is called *unification*. Unification is defined in [Kni89] as follows:

**Definition 3.2.1 (Unification)**
*Two terms $s$ and $t$ are unifiable if there exists a substitution $\sigma$ such that $\sigma(s) = \sigma(t)$. In such a case, $u$ is called a unifier of $s$ and $t$, and $\sigma(s)$ is called a unification of $s$ and $t$.*

Several unifiers may exist for two terms $t$ and $s$. However, there is always one unifier $\sigma$ that is more general than any other unifier. This means that $\sigma$ can be transformed to any other unifier of $t$ and $s$ by applying some substitutions. $\sigma$ is called the *most general unifier* of $s$ and $t$. It is defined in [Kni89] as follows:

**Definition 3.2.2 (Most general unifier)**
*A unifier $\sigma$ of terms $s$ and $t$ is called a most general unifier (MGU) of $s$ and $t$, if for any other unifier $\theta$, there is a substitution $\tau$ such that $\tau \circ \sigma = \theta$.*

When applying a $\gamma$ rule on a formula of the form $(\forall x : x \in U : P)$ or $\neg(\exists x : x \in U : P)$, some arbitrary term $t$ is introduced and substituted for $x$. Which term $t$ leads to a contradiction in as little steps as possible is not always known at the moment of choosing such term $t$. Therefore, as described in [Fra00], Cocktail postpones this choice and substitutes a fresh variable $X$ for $x$.

Such variables $X$ are called tableau variables. They represent appliances of the $\gamma$ rule, for which the actual substitutions have not yet been executed. Like in the case of free variables, unification can be used to find a substitution for these tableau variables that closes a leaf. For example, if the label of a leaf contains formulas $P(X)$ and $\neg P(c)$, substituting $c$ for $X$ leads to a contradiction.

However, there are restrictions on the terms that are allowed to be substituted for tableau variables. If some variable $\alpha$ was introduced by a $\delta$ rule, this variable may not be substituted for a tableau variable $X$ that was introduced before $\alpha$. Variable $X$ was introduced instead of an arbitrary term $t$. At that moment, variable $\alpha$ did not exist and could not have been selected as term $t$. We introduced variable $X$ to postpone the choice of a suitable term $t$. Since that choice is made as soon as a term $t$ is substituted for $X$ in one leaf, $t$ has to be substituted for $X$ in all leafs. This indicates that we need to create one unifier that closes all leafs.

To ensure that these restrictions are met, a context containing all variables introduced by $\gamma$ and $\delta$ rules in the order in which they are introduced is

maintained. So when trying to find a suitable term $t$ to substitute for a variable $X$, it may not contain variables introduced by $\delta$ rules that occur after $X$ in the context.

### 3.2.2 Properties of the tableau method

The tableau method is complete for first-order logic: for arbitrary tautology $P$, a closed tableau exists starting with root node $\neg P$. This is proved in [Smu68]. Another property of semantic tableaux proved in the same work by Smullyan is soundness: if a closed tableau exists for $\neg P$, where $P$ is an arbitrary first-order logic formula, then $P$ is a tautology.

Unfortunately, completeness of the tableau method does not guarantee that a tableau can indeed be constructed for each tautology $P$ using the tableau method, within a reasonable amount of time. Completeness only guarantees that for a tautology $P$ a tableau exists. It still could take a very long time to construct the tableau.

The tableau method is (positively) semi-decidable. A *fair* algorithm that checks whether a tableau exists for a tautology $P$ will end in a finite amount of time with a positive answer. However, if $P$ is not a tautology, the algorithm might run forever. For instance, assume that the label of a leaf contains a universally quantified formula. Applying a $\gamma$ rule to this leaf will lead to a new leaf, of which the label still contains the universally quantified formula. So, the $\gamma$ rule can be applied again. This can continue forever. Note that fairness of the algorithm is an important requirement here: if the algorithm is not fair, it can apply $\gamma$ rules on the same formula without ever applying another rule. So, even if $P$ is a tautology, an unfair algorithm might not give a positive result within a finite amount of time.

Cocktail has a number of options to control the tableau method, which prevent the algorithm to run too long or to run out of memory. The running time of the algorithm can be bounded. Also, bounds can be set on the size of the tableau, i.e. its depth and the number of leafs. So, summarizing the results discussed above, if some formula $P$ is being proved using the tableau method, there are three possible outcomes:

1. The algorithm finishes after finding a closed tableau. In this case it can be concluded that $P$ is a tautology.

2. No more tableau expansion rules are applicable to the tableau con-

structed so far and the tableau is not closed. At least one leaf of the tableau is not closed in this case. This means that $P$ is not a tautology, and such a non-closed leaf provides a counter-example.

3. The algorithm finishes without finding a closed tableau, nor finding a counter-example. This happens if one of the bounds is exceeded. If this happens, nothing can be concluded about $P$. It might be the case that the tableau can be closed after performing some more steps, but since the method is semi-decidable it can also be the case that $P$ is not a tautology and that the algorithm can keep running forever, without ever finding a counter-example.

### 3.2.3 What about equalities?

An equality $e = t$ gives information about terms $e$ and $t$ that could be useful (or even essential) when constructing a proof, namely that they are equal and thus that they can be used interchangeably. For example, if $f(e) = 0$, we can also conclude that $f(t) = 0$. However, this information is discarded in the standard tableau method. Different from the definition of first-order logic given in chapter 2, equalities are not part of first-order logic in Cocktail and are treated as atomic predicates.

Assume that the formula that needs to be proved, or the context, contains one or more equations. Then such equations are considered to be one atomic entity in the tableau method. The actual fact that two terms are defined to be equal is not taken into account.

For example, assume that we need to prove that the following formula is a tautology, for arbitrary constant terms (i.e. functions with zero arguments) $e$ and $t$: $e = t \wedge t = 0 \Rightarrow e = 0$. We would start a tableau with a root node labelled with $e = t \wedge t = 0 \wedge \neg(e = 0)$. Using the tableau expansion rules would lead to the (simplified) tableau depicted in figure 3.1.

Now, this leaf would have to be closed by unifying $e$ and $t$, leading to $e = 0$ and $\neg(e = 0)$. However, since $e$ and $t$ are constants, this is impossible. Unification algorithms identify the fact that $t$ and $e$ are functions with different names, which are impossible to unify. So, in order to close the leaf it is necessary to use the fact that $t = 0$, because that would allow the transformation of $e = t$ to $e = 0$.

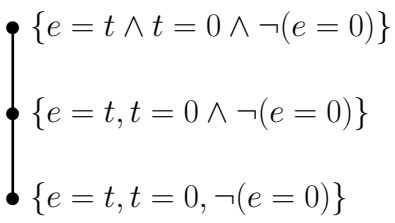



Figure 3.1: Example of a tableau that can not be closed using standard unification

This example indicates that in order to allow reasoning about equalities, the standard tableau method is not sufficient: the tableau method is not complete for first-order logic with equalities.

There are two possible directions to take from here, both of which are described in [Bec97]. The first possibility would be to introduce extra tableau-extension rules to handle equations. Secondly, one could try to use a different mechanism to close leafs. This mechanism, unlike the standard unification algorithm, should take the equations into account. We discuss both possibilities next.

#### Introducing additional tableau-expansion rules

An additional tableau-expansion rule is described in [Bec97]. This rule was first defined by Fitting and is stated as follows:

$$\text{eq rule} \quad \frac{t = s, P(t')}{\mu(P[t' := s])}$$

When applying this rule, the MGU $\mu$ of $t$ and $t'$ is calculated. If terms $t$ and $t'$ are not unifiable, the rule can not be applied.

Since $t$ is equal to $s$, $\mu(P(t'))$ may then be replaced by $\mu(P(s))$. It is important that $\mu$ is applied to all formulas in the branch of $t = s$ and $P(t')$, since the application of the rule depends on the instantiation of variables defined in $\mu$. However, it is not necessary to fully apply $\mu$, since universally quantified variables can be instantiated multiple times, using arbitrary terms (of the right type). Let $U$ be the set of variables in either $t = s$ or $P[t']$ that are universally quantified, or introduced by instantiating a universally quantified formula.

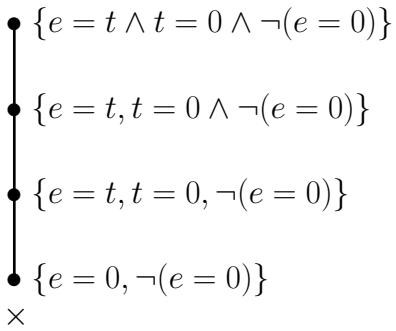



Figure 3.2: Closed tableau using the eq rule.

Now, lets apply this new eq rule to the tableau in figure 3.1. The eq rule can be applied on $e = t$ and $t = 0$. The MGU of $t$ and $t$ is empty, since $t = t$. So, we can add $e = 0$ to the leaf, resulting in the closed tableau depicted in figure 3.2. A closed leaf is indicated by $\times$ in the figure.

#### Reasoning about equalities in a tableau

Another way to use equalities in tableaux is to reason with the equalities in it, without making changes to the actual tableau construction method. This means that the example tableau sketched in figure 3.1 should be closed by using the fact that according to an equality in the label of the leaf $e$ is equal to $t$, and thus a contradiction is already present.

To do this, one needs to identify and solve so-called E-unification problems that are present in leafs of the tableau. By doing this, leafs can be closed.

Adding additional tableau expansion rules as described above will result in larger tableaux. [Bec97] states that it is impossible to solve even quite small problems using additional tableau-expansion rules in a somewhat timely fashion. According to [Bec97], the E-unification method is more efficient, since it does not increase the size of the tableau (and therefore, does not increase the search space for the problem). Based on these results, we choose to use E-unification to reason about equalities in the tableau prover of Cocktail. E-unification is discussed in more detail in the next chapter.

# Chapter 4

# Rigid E-Unification

This chapter introduces E-Unification problems and discusses their complexity. Then it is shown how *rigid* E-unification can be used in combination with the tableau method to reason about equalities. Actually solving a rigid E-unification problem is the subject of chapter 5.

## 4.1 Definition of Rigid E-Unification

Informally, a E-unification problem can be regarded as the question whether some equality can be logically deduced from a given set of equalities. Several classes of E-unification problems exist, following the classes of equalities defined in chapter 2: universal, rigid and mixed E-unification.

First we introduce the concept of a rigid equation. This definition is taken from [DV96].

**Definition 4.1.1 (Rigid equation)**
*Let $E$ be a set of equalities $\{e_1 \equiv (s_1 = t_1), \ldots, e_n \equiv (s_n = t_n)\}$, and let $s$ and $t$ be terms.*
*$E \vdash_\forall s = t$ is called a* rigid equation.

Now E-unification problems can be defined in general.

**Definition 4.1.2 (E-unification problem)**
*A* E-unification problem *is defined as follows: given some rigid equation $E \vdash_\forall s = t$, does a substitution $\theta$ exist, such that $\theta(E) \vdash_\forall \theta(s) = \theta(t)$?*

If a E-unification problem is solved by some substitution $\theta$, this $\theta$ is called a solution for the problem. The set of equalities $E$ in the rigid equation of some

E-unification problem determines the type of E-unification problem. If both bound and free variables occur in the problem, it is a mixed E-unification problem.

**Definition 4.1.3 (Mixed E-unification problem)**
*If the set $E$ in the rigid equation of some E-unification problem contains both bound and free variables, it is called a* mixed E-unification problem.
*E.g.* $E \equiv \{(\forall x : x \in U : f(x) = 7), g(y) = c\}$.

In a similar way, universal and rigid E-unification problems can be defined.

**Definition 4.1.4 (Universal E-unification problem)**
*If the set $E$ in the rigid equation of some E-unification problem contains no free variables, it is called a* universal E-unification problem.
*E.g.* $E \equiv \{(\forall x : x \in U : f(x) = 7)\}$.

**Definition 4.1.5 (Rigid E-unification problem)**
*If the set $E$ in the rigid equation of some E-unification problem contains no bound variables, it is called a* rigid E-unification problem.
*E.g.* $E \equiv \{g(y) = c\}$.

The difference between these three classes of E-unification problems is nicely stated in [Bec97] and quoted here:

> The different versions allow equalities to be used differently in an equational proof: in the universal case the equalities can be applied several times with different instantiations for the variables they contain; in the rigid case they can be applied more than once but with only one instantiation for each variable; in the mixed case there are both types of variables.

Apart from the standard (singular) E-unification problems, one can also combine multiple E-unification problems and try to find one substitution that solves them all. Such a combined E-unification problem is called a *simultaneous E-unification problem*

**Definition 4.1.6 (Simultaneous E-unification problem)**
*Let $R$ be a finite set of E-unification problems. Does a substitution $\theta$ exist, such that $\theta$ is a solution for each E-unification problem $r$ in $R$?*

We need to solve simultaneous E-unification problems to close all leafs of a tableau simultaneously.

### 4.1.1 Decidability and complexity

Unfortunately, [Bec97] states that universal E-unification is undecidable. This means that in general it is not possible to determine whether a solution exists for some universal E-unification problem. As a result of this, mixed E-unification (which contains universal equalities) is also undecidable. Rigid E-unification, however, is decidable. So, given any arbitrary rigid E-unification problem, it can be determined in a finite amount of time whether a solution to the problem exists. Unfortunately, simultaneous rigid E-unification, which is needed for equational tableaux, is undecidable.

An algorithm to find a solution to a rigid E-unification problem is given in [DV96]. This algorithm is *not* complete. Thus, it is not guaranteed that the algorithm finds a solution for any arbitrary solvable rigid E-unification problem. However, as is proved in [DV96], the algorithm is complete when used in combination with the tableau method for first-order logic with equality. This is obviously very nice for our causes, since our aim is to prove formulas in first-order logic with equality.

Solving a rigid E-unification problem is NP-complete, as is shown in [GSNP88]. So, naturally, the algorithm described in [DV96] is also NP-complete. If the context of the E-unification problem (i.e. the set $E$ of equalities of the problem) consists of merely ground equalities, Shostak's algorithm can be used to solve the problem in polynomial time. This algorithm is described in [Sho84].

### 4.1.2 Rigid E-unification and the tableau method

This section describes how E-unification problems can be used in combination with the tableau method in such a way that first-order logic formulas with equality can be proved using the tableau method. The general idea is to try to identify E-unification problems in tableau leafs, and close such leafs by solving the E-unification problems.

As described in chapter 3, some leaf of a tableau can be closed if a contradiction is found in its label. For example, let $\gamma$ be some leaf of the tableau. Suppose (the label of) $\gamma$ contains some inequality $s \neq t$. Now the equalities that are in $\gamma$ are used to try to show that $s = t$ can be concluded from this context, resulting in a contradiction. This is equal to solving a E-unification problem. The type of E-unification problem that is to be solved depends on

the type of equalities that are present in the problem.

The set of all E-unification problems in some leaf of a tableau can be defined as follows (following the definition in [Bec97]).

**Definition 4.1.7 (Extracting E-unification problems)**
*Let $E(\gamma)$ denote the set of all equalities in the label of tableau leaf $\gamma$. The set $EP(\gamma)$ of E-unification problems in $\gamma$ consists of:*

$E(\gamma) \vdash_\forall s = t$, *for all inequalities* $s \neq t$ *on* $\gamma$

If one of the E-unification problems in $EP(\gamma)$ has a solution, leaf $\gamma$ can be closed using that solution.

As mentioned before, the type of E-unification problems that need to be solved depends on the type of equalities in the problem. Since universal and mixed E-unification problems are undecidable, we do not want to take formulas containing bound formulas into account, since it is then not decidable whether a leaf can be closed. Therefore, only rigid E-unification problems should be considered, since they are decidable.

As stated before, simultaneous rigid E-unification is needed to close all leafs simultaneously. We also stated that simultaneous rigid E-unification is undecidable. Therefore, to keep the tableau method semi-decidable using rigid E-unification, we need to close leafs one at a time. Doing this implies that when a unifier is found that closes a certain leaf, it must be checked whether this unifier is compatible with all other unifiers that close other leafs.

To ensure this, all equalities that occur in a tableau are considered to be rigid. [DV96] introduces the calculus $\mathcal{BSE}$, which can be used to solve rigid E-unification problems. Furthermore, a proof is provided that the tableau method in combination with rigid E-unification is sound and complete. This theorem will be formally stated after introducing the calculus $\mathcal{BSE}$ in the next chapter.

# Chapter 5

# Solving a Rigid E-Unification problem

This chapter describes a method to solve rigid E-unification problems, along with some interesting properties.

## 5.1 Ground problems

First we look at the case in which the rigid E-unification problem that is to be solved consists of ground equations. In this case, we can use an efficient algorithm defined by Shostak in [Sho84]. A corrected version of this algorithm is given in [RS01].

## 5.2 Introducing constraints

Generally, a rigid E-unification will contain equations with free variables. In this case, the algorithm of Shostak can not be used. This section describes the method to solve an arbitrary rigid E-unification problem, which can be summarized as follows:

- We rewrite the problem using the *calculus* $\mathcal{BSE}$. This results in a new (hopefully more simple) rigid E-unification problem. Each rewrite step gives rise to a set of *ordered constraints*. The rewrite step is only valid if this constraint set is satisfiable. Several steps are taken to check satisfiability of a constraint set:

1. Using a rewrite system $\mathcal{R}$ that preserves satisfiability, the constraint set is transformed into a disjunction of *solved forms*. This transformation uses properties of LPO to simplify the terms in the constraint set.
2. For each solved form we construct a set of all possible *simple systems* that can be derived from it. Such a simple system defines an ordering on all subterms in the solved form.
3. We check whether at least one of the simple systems generated by a solved form is satisfiable. A simple system is satisfiable, if it is not *trivially bottom*. If there exists such a simple system, the original constraint set is satisfiable and the rewrite rule of the calculus $\mathcal{BSE}$ may be applied.

- Repeat the above steps, until the equality resolution rule of the calculus $\mathcal{BSE}$ is applicable. After applying equality resolution, the resulting problem is trivial. If no rule of the calculus $\mathcal{BSE}$ is applicable, we conclude that no solution exists for the problem.

Step 2 of the process of determining satisfiability of a constraint set is implemented using graph structures that represent solved forms. We derive all simple systems by traversing such a graph. During this traversal we eliminate trivially bottom simple systems on the fly, thus combining step 2 with step 3. This process is described in more detail in chapter 6.

We will now explain the method in more detail. Constraints are defined in this section, following the definition given in [DV96][1].

**Definition 5.2.1 (Constraint)**
*Let $s$ and $t$ be terms. A* constraint *is an expression of the form $s = t$, or of the form $s \succ t$. The first type is called an* equality constraint *and the latter an* inequality constraint

A set of constraints is called a *constraint set*. An important question that needs to be answered very often in the method for solving rigid E-unification is whether some constraint set is *satisfiable*

[1] In [DV96] constraints are defined to be a set of expressions of the form $s = t$, or $s \succ t$. These expressions are called equality constraints, respectively inequality constraints. Effectively, this definition defines constraints to be a set of (equality/inequality) constraints. Because this convention seems to be rather confusing, we use the terminology *constraint set* in this thesis to indicate a set of (equality/inequality) constraints

**Definition 5.2.2 (Constraint solution)**
*A substitution $\theta$ is a solution to some constraint $s = t$ (respectively, $s \succ t$), if $\theta(s)$ and $\theta(t)$ are ground and $\theta(s) = \theta(t)$ (respectively, $\theta(s) \succ \theta(t)$.*

**Definition 5.2.3 (Constraint set satisfiability)**
*A substitution $\theta$ is a solution to a constraint set, if it is a solution to each constraint in the constraint set. A constraint set is* satisfiable *if it has a solution.*

## 5.3 Calculus $\mathcal{BSE}$

Now that constraints have been introduced, this section combines a constraint set with a rigid E-unification problem and introduces the calculus $\mathcal{BSE}$.

**Definition 5.3.1 (Constraint rigid E-unification problem)**
*Let $\mathcal{R}$ be a rigid E-unification problem $E \vdash_\forall s = t$, and let $\mathcal{C}$ be a constraint set. Then a pair consisting of $\mathcal{R}$ and $\mathcal{C}$ is called a* constraint rigid E-unification problem *(denoted as $\mathcal{R} \cdot \mathcal{C}$).*

The calculus $\mathcal{BSE}$ is defined in terms of such constraint rigid E-unification problems. It takes a constraint rigid E-unification problem and rewrites it into a new constraint rigid E-unification problem. The following rewrite rules define the calculus $\mathcal{BSE}$.

**Definition 5.3.2 (Calculus $\mathcal{BSE}$)**
*Left rigid basic superposition:*

$$\frac{E \cup \{l = r, s[p] = t\} \vdash_\forall e \cdot \mathcal{C}}{E \cup \{l = r, s[r] = t\} \vdash_\forall e \cdot \mathcal{C} \cup \{l \succ r, s[p] \succ t, l = p\}} \text{ (LRBS)}$$

*Right rigid basic superposition:*

$$\frac{E \cup \{l = r\} \vdash_\forall s[p] = t \cdot \mathcal{C}}{E \cup \{l = r\} \vdash_\forall s[r] = t \cdot \mathcal{C} \cup \{l \succ r, s[p] \succ t, l = p\}} \text{ (RRBS)}$$

*Equality resolution:*

$$\frac{E \vdash_\forall s = t \cdot \mathcal{C}}{\vdash_\forall s = s \cdot \mathcal{C} \cup \{s = t\}} \text{ (ER)}$$

Several restrictions must be met before these rules may be applied:

1. The constraint set at the conclusion of the rule is satisfiable. This is necessary to assure correctness of the method, as well as termination (see [DV96] for a proof).
2. The equation on the right-hand side of the rigid equation at the premise of the rule is not of form $q = q$, for some term $q$. This ensures that the algorithm terminates after equality resolution is applied.
3. Term $p$ is not a variable in both basic superposition rules.
4. $s[r] \neq t$ in the left basic superposition rule. Applying the left basic superposition rule with $s[r] = t$ would add no new information to the context.
5. In both basic superposition rules $p \neq r$. Otherwise, if $p = r$, the conclusion of a basic superposition rule would be the same as the premise, since $p$ is replaced by $r$ in either the left-hand or right-hand side of the rigid equation.

## 5.4 Constraint Satisfiability

The first condition on the basic superposition rules of the calculus $\mathcal{BSE}$ determines that these rules may only be applied if the constraint at the conclusion of the rule is satisfiable with respect to the reduction ordering. Therefore, a method is needed to determine satisfiability of such constraints.

The reduction ordering used is the lexicographic path ordering. Thus, a procedure is needed to check satisfiability of lexicographic path ordering constraints. This chapter describes the procedure used, which is based on [Nie93] and [Com90].

### 5.4.1 Overview of the procedure

A graphical representation of all steps needed to solve a rigid E-unification problem is given in figure 5.1. Determining satisfiability of a constraint (indicated by the dotted arrows in the figure) using this procedure involves three steps. First the constraint is transformed into a disjunction of *solved forms* by rewriting the constraint. The terms in each solved form are ordered in all possible ways to obtain a set of *simple systems*. Finally, using this set of simple systems, satisfiability of the original constraint can be determined.

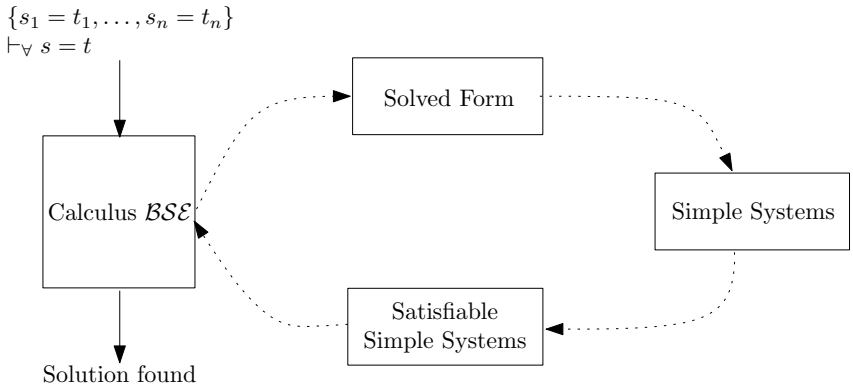



Figure 5.1: Overview of solving a rigid E-unification problem.

This section describes these three steps in detail (along with definitions of the newly introduced concepts).

#### Solved form transformation

[Com90] describes a set of rewrite rules $\mathcal{R}$ that transforms an ordering constraint into a corresponding disjunction of solved forms. It is important to note that these rules preserve satisfiability: an ordering constraint is satisfiable if the disjunction of solved forms (obtained by using $\mathcal{R}$) is satisfiable.

Each of the rules assumes constraints to be in disjunctive normal form, so after each rule application a normalization is performed. Solved forms are defined in [Com90] as follows[2]:

**Definition 5.4.1 (Solved Forms)**
*A solved form is either* $\top$, $\bot$, *or a formula*
$x_1 = t_1 \wedge \ldots \wedge x_n = t_n \wedge u_1 \succ v_1 \wedge \ldots \wedge u_m \succ v_m$,
*where*

- $x_1, \ldots, x_n$ *are variables occurring only once in the formula,*
- *for each* $i \in \{1, \ldots, m\}$, $u_i$ *or* $v_i$ *is a variable,*

[2]The definition actually does not contain the fact that a solved form might consist of multiple disjuncts. [Com90] claims in his paper that all normal forms obtained by applying the rewrite rules $\mathcal{R}$ are simple systems. However, normal forms can be found that do not adhere to the definition of a solved form. After an email correspondence with Comon, it was concluded that solved forms are actually disjunctions of what is called a solved form in [Com90].

- *for each index* $i \in \{1, \ldots, m\}$, $u_i$ *is not a subterm of* $v_i$ *nor* $v_i$ *of* $u_i$.

The rules described in [Com90] are listed in table 5.1. However, [Com90] is a bit sloppy in the proof of completeness of the reduction rules. Comon states that $\mathcal{R}$ is complete if any normal form for $\rightarrow^*_{\mathcal{R}}$ is a solved form. But, using only the rules from $\mathcal{R}$, this is not always true. The derivation stated in example 1 of [Com90] can not be made using the rules in $\mathcal{R}$, without adding a *subsumption* rule to remove disjuncts.

**Definition 5.4.2 (Subsumption)**
*A clause* $C_1$ *subsumes clause* $C_2$ *if every literal in* $C_1$ *also occurs in* $C_2$.

[JMS04] makes (and proves) the following observation: if clause $C_1$ subsumes clause $C_2$, then $C_1$ implies $C_2$. Hence, the following rule is introduced based on that observation.

**Definition 5.4.3 (Subsumption Rule)**
$C_1 \vee C_2 \vee \ldots \vee C_n \rightarrow_R C_2 \vee \ldots \vee C_n$, *if* $C_1$ *subsumes* $C_2$.

The rule simply removes disjuncts that are redundant for determining satisfiability of the term. Let term $T \equiv C_1 \vee C_2$ and assume that clause $C_1$ subsumes $C_2$, then – following from the above observation – $C_1 \rightarrow C_2$. Assume $T$ is satisfiable and $C_2$ does not hold, then $C_1$ must hold, but this would imply that $C_2$ holds. So, $C_2$ must hold for $T$ to be satisfiable, regardless of the value of $C_1$. Therefore, clause $C_1$ can be discarded.

#### From solved forms to simple systems

The next step in determining satisfiability of the constraint is to derive all *simple systems* from the solved forms of the constraint. [Com90] and [Nie93] use (slightly) different definitions of simple systems. However, this slight difference results in an entire extra step that is needed to determine satisfiability when using the definition of [Com90]. Therefore, the definition of [Nie93] is used and presented here:

**Definition 5.4.4 (Simple System)**
*A* system *is a conjunction of equations and inequations. A* simple system *is a system* $\mathcal{I}$ *satisfying the following properties:*

- *There exists a finite set of terms* $\{t_1, \ldots, t_n\}$ *such that* $\mathcal{I} \equiv \bigwedge_{1<i\leq n} t_{i-1} \succ t_i$ *or* $t_{i-1} = t_i$.
- *Every subterm of a term* $t_i$ *is some term* $t_j$, *with* $i < j$.

**Equality Rules**

$(D_1)$ $f(v_1, \ldots, v_n) = f(u_1, \ldots, u_n) \rightarrow_R v_1 = u_1 \wedge \ldots \wedge v_n = u_n$

$(C_1)$ $f(v_1, \ldots, v_n) = g(u_1, \ldots, u_m) \rightarrow_R \bot$, if $f \neq g$

$(R)$ $x = t \wedge P \rightarrow_R x = t \wedge P\{x \mapsto t\}$,
if $x$ is a variable, $x \notin Var(t)$, $P$ is a conjunction of equations and inequations, $x \in Var(P)$ and, if $t$ is a variable, then $t \in Var(P)$.

$(O_1)$ $s = t[s]_p \rightarrow_R \bot$, if $p \neq \Lambda$.

**Inequality Rules**

$(D_2)$ $f(v_1, \ldots, v_n) \succ g(u_1, \ldots, u_m) \rightarrow_R$
$f(v_1, \ldots, v_n) \succ u_1 \wedge \ldots \wedge f(v_1, \ldots, v_n) \succ u_m$,
if $f \succ_F g$

$(D_3)$ $f(v_1, \ldots, v_n) \succ g(u_1, \ldots, u_m) \rightarrow_R$
$v_1 \geq g(u_1, \ldots, u_m) \vee \ldots \vee v_n \geq g(u_1, \ldots, u_m)$,
if $g \succ_F f$

$(D_4)$ $f(v_1, \ldots, v_n) \succ f(u_1, \ldots, u_n) \rightarrow_R$

$$\begin{array}{lll} & (v_1 \succ u_1 \wedge f(v_1, \ldots, v_n) \succ u_2 \wedge \ldots \wedge f(v_1, \ldots, v_n) \succ u_n) \\ \vee & (v_1 = u_1 \wedge v_2 \succ u_2 \wedge \ldots \wedge f(v_1, \ldots, v_n) \succ u_n) \\ \vee & \ldots \\ \vee & (v_1 = u_1 \wedge v_2 = u_2 \wedge \ldots \wedge v_n \succ u_n) \\ \vee & v_1 \geq f(u_1, \ldots, u_n) \vee \ldots \vee v_n \geq f(u_1, \ldots, u_n) \end{array}$$

$(O_2)$ $t[s]_p \succ s \rightarrow_R \top$, if $p \neq \Lambda$.

$(O_3)$ $s \succ t[s] \rightarrow_R \bot$

$(T_1)$ $s \succ t \wedge t \succ s \rightarrow_R \bot$

$(T_2)$ $s = t \wedge s \succ t \rightarrow_R \bot$

Table 5.1: Transformation rules to solved form

The difference between the definitions in both papers is that [Nie93] allows equations as well as inequations in simple systems, while [Com90] only allows inequations. Simple systems are written as follows: $t_1 \# t_2 \# t_3 \# t_4 \# \ldots \# t_n$, where $\#$ is $\succ$ or $=$.

In order to determine satisfiability of a simple system, as described in the next section, a special element has to be added to each simple system: the *first limit ordinal* $\omega$.
The first limit ordinal is defined as follows in [Nie93]:

**Definition 5.4.5 (First limit ordinal (term))**
*Let $f$ and 0 be the smallest non-constant and constant function symbols.*

- *$f$ is unary: $\omega = g(0, \ldots, 0)$, where $g$ is the smallest function greater than $f$,*
- *$f$ is not unary: $\omega = f(0, \ldots, 0, t, 0)$, where $t$ is the second smallest ground term.*

Note that a first limit ordinal does not always exist. If the smallest non-constant function $f$ is unary and there is no other function greater than $f$, the first limit ordinal is undefined. If $\omega$ exists, we can split a simple system in its *natural* and *non-natural* part. The natural part contains all terms $t$ for which $\omega \succ t$, while the non-natural part contains terms $t$ such that $t \succ \omega$. The natural part is denoted as $\mathcal{N}$.

Now we need to calculate the set of simple systems from a solved form. Let $s$ be the solved form for which we want to calculate the corresponding set of simple systems. We take the following steps to build the set of simple systems $SimSys(s)$ generated by $s$:

1. Add $0 < \omega$ to $s$, if $\omega$ exists (otherwise add $0 = 0$ to ensure that 0 is a term in $s$) and determine the list of all subterms of $s$, say $Sub(s)$.

2. Let $k$ be the number of elements in $Sub(s)$. Each simple system generated by $s$ will have length $k$. We calculate all permutations $Perm(s)$ of the items in $Sub(s)$.

3. Let $p \in Perm(s)$. Now, we can generate a set of candidate simple systems $css(p)$ from $p$ by inserting relations between its items $t_1, \ldots, t_k$ in all possible ways (e.g. $css(p)$ will contain simple systems $t_1 \succ t_2 \succ$

$\ldots$ , $t_1 = t_2 \succ \ldots$ , $t_1 \succ t_2 = \ldots$ , etc.). Hence, we interpret $p$ as $t_1 \# t_2 \# \ldots \# t_k$, where $\#$ is $\succ$ or $=$.

4. For each $c \in css(p)$, check whether $c$ adheres to the definition of simple systems (e.g. a term $t$ is greater than all of its proper subterms) and check whether it adheres to the relations specified in solved form $s$ (i.e. if $t_i \succ t_j$, or $t_i = t_j$ in $s$, then $t_i$ must also be larger than, respectively equal to, $t_j$ in $c$). If $c$ conforms to these conditions, then $c \in SimSys(s)$.

5. Repeat steps 3 and 4 for all $p \in Perm(s)$.

#### Determining satisfiability of simple systems

In [Com90] it is proved that a solved form is satisfiable if and only if at least one of the simple systems generated from the solved form is satisfiable. If we combine this result with the fact that the solved form was obtained from the original constraint using satisfiability preserving rewrite rules, we can conclude that the original constraint is satisfiable if at least one simple system is satisfiable.

Since we are working with an ordering that is total on ground terms, each term has a successor. We define the successor function *succ* on terms as follows:

**Definition 5.4.6 (Successor)**
*Let 0 and $f$ be the smallest constant and non-constant function symbols. Furthermore, let $\mathcal{C} = \{c_1, \ldots, c_n\}$ be the set of constant function symbols in $\mathcal{F}$, where $c_1 \succ_\mathcal{F} \ldots \succ_\mathcal{F} c_n = 0$.*
*Successor $succ(t)$ of term $t$ is defined as follows:*

- $succ(c_{i+1}) = c_i$,
- $succ(c_1) = f(0, \ldots, 0)$,
- *if* $t \equiv f(0, \ldots, 0, t') \in \mathcal{N}$, *then* $succ(t) = f(0, \ldots, 0, succ(t'))$,
- *in all other cases,* $succ(t) = f(0, \ldots, 0, t)$.

Using the successor function, terms can be assigned an ordinal number. This is nicely explained in [Nie93] using the example given below.

Let $\mathcal{F} = \{f, a, 0\}$, where $f$ is binary and $a$ and 0 constants. Furthermore, let $f \succ_{\mathcal{F}} a \succ_{\mathcal{F}} 0$.
Then the terms are, in increasing order with relation to $\succ$:

| Ordinal | 0 | 1 | 2 | 3 | 4 | 5 | | $\omega$ | |
|---|---|---|---|---|---|---|---|---|---|
| Term | 0 | $a$ | $f(0,0)$ | $f(0,a)$ | f(0,f(0,0)) | $f(0,f(0,a))$ | ... | $f(a,0)$ | ... |

As can be seen in this example, our definition of $\omega$ given earlier actually defines the term numbered with the ordinal number $\omega$. $\omega$ as an ordinal number has the following definition:

**Definition 5.4.7 (First limit ordinal (ordinal number))**
*$\omega$ is the first ordinal number that is neither 0, nor the successor of another ordinal number. I.e. $\omega$ is the least upper bound of the natural numbers.*

In the rest of this thesis the context will make clear whether we are talking about $\omega$ as a term, or as a number.

Terms $t$ in the natural part of a simple system are smaller than $\omega$, and thus they have a corresponding natural value. The natural value of $t \in \mathcal{N}$ is denoted by $|t|$.

Nieuwenhuis introduces the notion of *trivially bottom* to determine satisfiability of a simple system in [Nie93].

**Definition 5.4.8 (Trivially Bottom)**
*Let $k$ be $max(|K|, 1)$, where $|K|$ denotes the number of constants smaller than the smallest non-constant function.*
*A simple system $S$ is trivially bottom if and only if*

1. *$s = t$ with $top(s) \neq top(t)$, or*
2. *$f(s_1, \ldots, s_p) = f(s'_1, \ldots, s'_p)$ and $\exists i : 1 \leq i \leq p : s_i \neq s'_i$, or*
3. *$s = t$ and $t$ is a proper subterm of $s$ or vice versa, or*
4. *$f(s_1, \ldots, s_p) \succ t$ with $top(t) \succ_F f$ and $\nexists i : 1 \leq i \leq p : s_i \succeq t$, or*
5. *$f(s_1, \ldots, s_p) \succ f(s'_1, \ldots, s'_p)$ and $\langle s_1, \ldots, s_p \rangle \not\succ_s^{lex} \langle s'_1, \ldots, s'_p \rangle$, or*
6. *$\omega \succ f(0, \ldots, 0, t)$ and there are strictly more than $k$ operators $\succ$ between $f(0, \ldots, 0, t)$ and $t$, or*

7. $\omega \succ t$ *with* $t$ *in* $\mathcal{N}$*, and strictly more than* $|t|$ *operators* $\succ$ *between* $t$ *and* 0*, or*

8. $f(0,\ldots,0,t) \succ t \succ \omega$ *and there is strictly more than one operator* $\succ$ *between* $f(0,\ldots,0,t)$ *and* $t$*.*

If a simple system adheres to – at least – one of the first five items listed in the definition of trivially bottom, it is unsatisfiable because it directly contradicts the definition of LPO or syntactic equality. The last three items are a little less obvious. We will shortly explain why these lead to unsatisfiable simple systems.

First we look at property six. Assume simple system $s$ contains $\omega \succ f(0,\ldots,0,t)$. Following the definition of a simple system, subterm $t$ must be smaller than $f(0,\ldots,0,t)$. Both $f(0,\ldots,0,t)$ and $t$ are in $\mathcal{N}$. According to the definition of the successor function, the successor of $t$ depends on the structure of $t$. If $t$ is a constant, its successor is the next constant, or the smallest non-constant function if $t$ is the largest constant. In the worst case, $t$ equals 0. $succ(0)$ is the first constant greater than 0, if it exists. Similarly, $succ(succ(0))$ is the constant thereafter, if it exists. We know that there are $k-1$ constants that are greater than 0, so after applying the successor function $k$ times, we will arrive at the smallest non-constant function instantiated with zeroes. So, at most $k \succ$ operators are allowed between $t$ and $f(0,\ldots,0,t)$.

Next, we discuss item seven. Term $t$ is in the natural part of the simple system. Its natural value is $|t|$. If there are more than $|t|$ operators $\succ$ between $t$ and 0, say $|t|+1$, the natural value of $t$ is $|t|+1$ according to the ordering in the simple system. This can never be correct and thus such a simple system is unsatisfiable.

Finally, we take a look at property eight. If $f(0,\ldots,0,t) \notin \mathcal{N}$, it follows directly from the definition of the successor function that $succ(t) = f(0,\ldots,0,t)$. So, since there is no term between term $t$ and its successor, at most one operator $\succ$ is allowed between $t$ and $f(0,\ldots,0,t)$ in the simple system.

Now we know when a simple system is trivially bottom. [Nie93] contains a proof that a simple system $S$ is satisfiable if and only if it is not trivially bottom. So, determining satisfiability of the original constraint is now

equivalent to checking whether the set of simple systems contains at least one simple system that is not trivially bottom.

# Chapter 6

# Implementation

The previous chapter introduced the method used to solve rigid E-unification problems. This chapter discusses the implementation of this method.

In order to transform arbitrary formulas and terms to skolem negation normal form, the rewrite rules suggested in chapter 2 have been implemented. The transformation of a constraint set to its solved form is a straightforward implementation of the rewrite system $\mathcal{R}$. How our implementation derives satisfiable simple systems from a solved form is discussed in more detail.

Finally, a short overview of the tool written for this project is given. It allows a user to define a rigid E-unification problem. Then the tool can determine whether or not the problem is solvable.

## 6.1 Simple Systems

In order to calculate all possible simple systems from some solved form and determine satisfiability of each simple system in an efficient way, a graph structure is introduced. This graph structure will be based on the solved form. From this graph structure all simple systems can be determined in such a way that trivially bottom checks can be performed on the fly. This way we avoid constructing many simple systems that turn out the be trivially bottom anyway. If a simple system is obtained by the graph traversal algorithm, the original constraint set was satisfiable, since we have a satisfiable simple system.

### 6.1.1 Using directed acyclic graphs

The graph used is called a *simple systems graph* and is a directed acyclic graph (DAG) in which terms are stored in nodes. If there is an edge from node $n$ to node $m$, this means that the term stored in $n$ is strictly smaller than the term stored in $m$ (i.e. $n \prec m$). As its name suggests, a DAG contains no cycles.

#### Constructing the simple systems graph

The first step in constructing a simple systems graph for a solved form $s$ is to determine all subterms that occur within $s$, because each simple system must contain each of these subterms according to the definition of a simple system.

Furthermore, in order to determine satisfiability of the simple system it is necessary to add the first limit ordinal $\omega$ (and its subterms!) as well as the smallest constant 0 to the simple system.

As stated above, the graph stores $\succ$ relations between terms that are forced by the definition of $LPO$ or by the solved form.

**Definition 6.1.1 (Minimal term)**
*A minimal term $t$ is a term for which the following holds:* $(\forall s : s \in \mathcal{T} \wedge s \neq t : s \nsim t \vee t \preceq s)$

So, a minimal term is either smaller than or incomparable to all other terms. Several minimal terms can exist, all of which have to be incomparable to each other.

Ensuring that terms are added in ascending order can be done by sorting the terms and adding the terms to the graph afterwards in the obtained order, or by simply determining a minimal term each time a term needs to be inserted.

The insertion of terms in ascending order assures that all terms currently stored in the simple systems graph when adding some term $t$ are smaller than or incomparable to $t$.

### Creation algorithm

Using the observations made, the following algorithm can be used to create a simple systems graph. It uses a depth-first traversal trying to insert a node. If for some node $n$, term $t$ is already appended at a path starting from $n$, then $t$ will not be added as a direct son of $n$. If it was not appended at a path starting from $n$, term $t$ was apparently incomparable to all terms located on paths starting from $n$. In this case, it is checked whether $n \prec t$ holds. If this is true, $t$ is added as a direct son of $n$ in a new branch.

Pseudo-code for the algorithm described is given below. Input of the algorithm is a solved form *solvedform*, output is a simple systems graph $ssg = (V, E)$. The procedure starts by adding term 0 to the graph. This term will be the root node, so it must be *strictly* smaller than all other terms.

BUILDSSG(*solvedform*)

```
1  V, E ← {0}, ∅
2  st ← "all subterms occuring in solvedform and ω" \ 0
3  do st ≠ ∅ →
4      ▷ insert s in ssg
5      pick s ∈ st, s.t. (∀t : t ∈ st : s ⪯ t ∨ s ⋈ t)
6      st ← st \ {s}
7      V ← V ∪ {s}
8      DFSInsert(s, 0, (V, E))
9  od
```

DFSINSERT($s, n, (V, E)$)

```
 1  inserted ← false
 2  if n ≺ s →
 3      foreach w ∈"sons of n" →
 4          inserted ← inserted ∨ DFSInsert(s, w, (V, E))
 5      hcaerof
 6      if ¬inserted →
 7          E ← E ∪ {n → s}
 8          inserted ← true
 9      fi
10  fi
11  return inserted
```

#### Using the simple systems graph to obtain all simple systems

Obtaining all simple systems without equalities from a simple systems graph corresponds to the problem of finding all topological sorts of the graph. A topological sort of some DAG is defined in [Ste01] as a linear ordering of all nodes in the DAG, such that if there is an edge from node $n$ to node $m$, $m$ appears after $n$ in the ordering.

Knuth has shown in [Knu68] that finding a topological sort for some DAG can be done in linear time. However, finding *all* topological sorts is in NP.

The algorithm presented here depends on searching for the set of smallest terms in the simple systems graph, and using each of these smallest terms as start symbol of a set of simple systems after removing it from the simple systems graph. Then the rest of the terms are added recursively. Pseudo-code for the algorithm described is listed below. Input of the graph is a simple systems graph $G = (V, E)$. Furthermore, $l \rightarrow^* r$ indicates that there is a path from node $l$ to node $r$.

ConstructOrderings$(V, E)$

1 **return** $ConstructVisit(0, V \setminus \{0\}, E)$

ConstructVisit$(t_i \# \ \ldots \ \# \ t_n, V, E)$

```
 1  if V ≠ ∅ →
 2      result ← ∅
 3      min ← {v ∈ V | (∀w : w ∈ V : v ⪯ w ∨ v ≁ w)}
 4      foreach m ∈ min →
 5          result ← result ∪ ConstructVisit(m ≻ t_i # ... # t_n,
                                              V \ {m}, E)
 6          if t_i → m ∉ E →
 7              result ← result ∪ ConstructVisit(m = t_i # ... # t_n,
                                                  V \ {m}, E)
 8          fi
 9      hcaerof
10  fi
11  else result ← {t_i # ... # t_n}
12  return result
```

This algorithm will find the set of all simple systems. Afterwards we need to check whether there is at least one satisfiable simple system in this set.

To do so, we need to check whether there is a simple system that is not trivially bottom.

We make the following observation, in order to eliminate simple systems that are trivially bottom in a more efficient way. Using our algorithm (i.e. always inserting the smallest term not yet added), it is impossible to add a term to a partially constructed simple system that is trivially bottom in such a way that after insertion the simple system is no longer trivially bottom. Thus, if a partially constructed simple system is trivially bottom it will never be, nor become, satisfiable.

With this observation, we can make our algorithm much more efficient. Before adding a term (either as an equality or an inequality) to some partially constructed simple system, we can check whether the obtained simple system is not trivially bottom. If the obtained simple system $s$ is found to be trivially bottom, we know that any simple system $t\# \ldots \#s$ is also trivially bottom. Therefore, we do not need to consider any extensions of such a simple system, since it is already trivially bottom. The ConstructVisit algorithm is extended as follows:

ConstructVisit$(t_i \# \ \ldots \ \# \, t_n, V, E)$

1 **if** $V \neq \emptyset \rightarrow$
2 $\quad$ *result* $\leftarrow \emptyset$
3 $\quad$ *min* $\leftarrow \{v \in V \mid (\forall w : w \in V : v \preceq w \vee v \nsim w)\}$
4 $\quad$ **foreach** $m \in$ *min* $\rightarrow$
5 $\quad\quad$ **if** "$m \succ t_i \# \ \ldots \ \# \, t_n$ not trivially bottom" $\rightarrow$
6 $\quad\quad\quad$ *result* $\leftarrow$ *result* $\cup$ *ConstructVisit*$(m \succ t_i \# \ \ldots \ \# \, t_n,$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad V \setminus \{m\}, E)$
7 $\quad\quad$ **fi**
8 $\quad\quad$ **if** $t_i \rightarrow m \notin E \wedge$"$m = t_i \# \ \ldots \ \# \, t_n$ not trivially bottom" $\rightarrow$
9 $\quad\quad\quad$ *result* $\leftarrow$ *result* $\cup$
$\qquad\qquad\qquad\qquad\qquad$ *ConstructVisit*$(m = t_i \# \ \ldots \ \# \, t_n,$
$\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad V \setminus \{m\}, E)$
10 $\quad\quad$ **fi**
11 $\quad$ **hcaerof**
12 **fi**
13 **else** *result* $\leftarrow \{t_i \# \ \ldots \ \# \, t_n\}$
14 **return** *result*

Note that this algorithm can be improved further by using the fact that as soon as one non-trivially bottom simple system is found, the original constraint is satisfiable. Thus, if we find one simple system, the algorithm can stop.

The approach described above depends on checking whether adding some term $t$ to a simple system leads to a simple system that is trivially bottom. We recall the definition of trivially bottom and indicate how each point can be checked. Let $a$ be the term that is added to some simple system $sys \equiv s_i \# \ldots \# s_n$.

1. $s = t$ with $top(s) \neq top(t)$: if $a \succ s_i \# \ldots \# s_n$, this can not occur. Otherwise, if $a = s_i \# \ldots \# s_n$, we need to determine terms $t'$ in $sys$ defined to be equal to $a$. $a = s_i \# \ldots \# s_n$ is trivially bottom if $top(a) \neq top(t')$ for one such $t'$.

2. $f(s_1, \ldots, s_p) = f(s'_1, \ldots, s'_p)$ and $(\exists i : 1 \leq i \leq p : s_i \neq s'_i)$: if $a \succ s_i \# \ldots \# s_n$ or $a$ is constant, this can not occur. Otherwise, $a \equiv f(s_1, \ldots, s_p)$. We need to check whether there are terms $t'$ in $sys$, s.t. $a = t'$. If such terms $t'$ exists, $a = s_i \# \ldots \# s_n$ is trivially bottom if for one of these terms $t' \equiv f(s'_1, \ldots, s'_p)$ there is an index $i$, such that $s'_i \neq s_i$ in the simple system. This can be checked on the fly, because we know for certain that all arguments of $a$ and $t'$ are already in the simple system, since they are subterms of $a$ and $t'$.

3. $s = t$ and $t$ is a proper subterm of $s$ or vice versa: if $a$ is a proper subterm of any term $t' \in sys$, a path $a \rightarrow^* t'$ would have been in the simple systems graph. Thus, by construction this case can not occur.

4. $f(s_1, \ldots, s_p) \succ t$ with $top(t) \succ_F f$ and $(\nexists i : 1 \leq i \leq p : s_i \geq t)$: if $a = s_i \# \ldots \# s_n$, or $a$ is not of form $f(s_1, \ldots, s_p)$, this case is not applicable. Otherwise, to check whether $a \succ s_i \# \ldots \# s_n$ is trivially bottom we need to look at all terms $t' \in sys$, such that $a \succ t'$. Then $a \succ s_i \# \ldots \# s_n$ is trivially bottom, if there exists one such term $t'$ that is smaller than all parameters of $a$. Again, since all subterms of $a$ are already in $ss$, this can be checked on the fly.

5. $f(s_1, \ldots, s_p) \succ f(s'_1, \ldots, s'_p)$ and $\langle s_1, \ldots, s_p \rangle \not\succ_s^{lex} \langle s'_1, \ldots, s'_p \rangle$: if $a = s_i \# \ldots \# s_n$, or $a$ is not of the form $f(s_1, \ldots, s_p)$, this case is not applicable. Otherwise, to check whether $a \succ s_i \# \ldots \# s_n$ is trivially bottom we need to look at all terms $t' \in sys$, such that $a \succ t'$ and

$t' = f(s'_1, \ldots, s'_p)$. Then $a \succ ss$ is trivially bottom, if there exists one such term $t'$, such that $(\nexists j : 1 \leq j \leq n : (\forall i : 1 \leq i < j : s_i = s'_i))$. Since all subterms of $a$ are already in $ss$, this can be checked on the fly.

6. $\omega \succ f(0, \ldots, 0, t)$ and there are strictly more than $k$ operators $\succ$ between $f(0, \ldots, 0, t)$ and $t$: If $a$ is not of the form $f(0, \ldots, 0, t)$, $a = s_i \# \ldots \# s_n$ or $a \notin \mathcal{N}$, this case is not applicable. Otherwise, we can simply count the number of $\succ$ operators that occur between $a$ and $t$. By construction we know for sure that $t$ must be in the graph, since it is a proper subterm of $t$.

7. $\omega \succ t$ with $t$ in $\mathcal{N}$, and strictly more than $|t|$ operators $\succ$ between $t$ and 0: If $a = s_i \# \ldots \# s_n$ or $a \notin \mathcal{N}$, this case is not applicable. Otherwise, count the number of $\succ$ operators that occur between $a$ and 0. If this number exceeds $|t|$, $a \succ s_i \# \ldots \# s_n$ is trivially bottom.

8. $f(0, \ldots, 0, t) \succ t \succ \omega$ and there is strictly more than one operator $\succ$ between $f(0, \ldots, 0, t)$ and $t$: If $a$ is not of the form $f(0, \ldots, 0, t)$, $a = s_i \# \ldots \# s_n$ or $a \in \mathcal{N}$, this case is not applicable. Otherwise, we can count the number of $\succ$ operators that occur between $a$ and $t$. We know that $t$ must be in the graph, since it is a proper subterm of $t$. If the number exceeds one, $a \succ s_i \# \ldots \# s_n$ is trivially bottom.

## 6.2 Solving Tool

A main result of the project is a tool with which users can define and solve rigid E-unification problems. It is part of the `auto` package of Cocktail and written in Java. Therefore, it should be quite straightforward to integrate the solving method with the tableau method that is currently used in Cocktail.

### 6.2.1 Overview

A screenshot of the tool is depicted in figure 6.1. We will briefly discuss the fields numbered by 1 through 4 in the screenshot.

1. Here users can define Cocktail terms into the context of Cocktail. Cocktail needs to know that functions and variables exist, before they can be used.

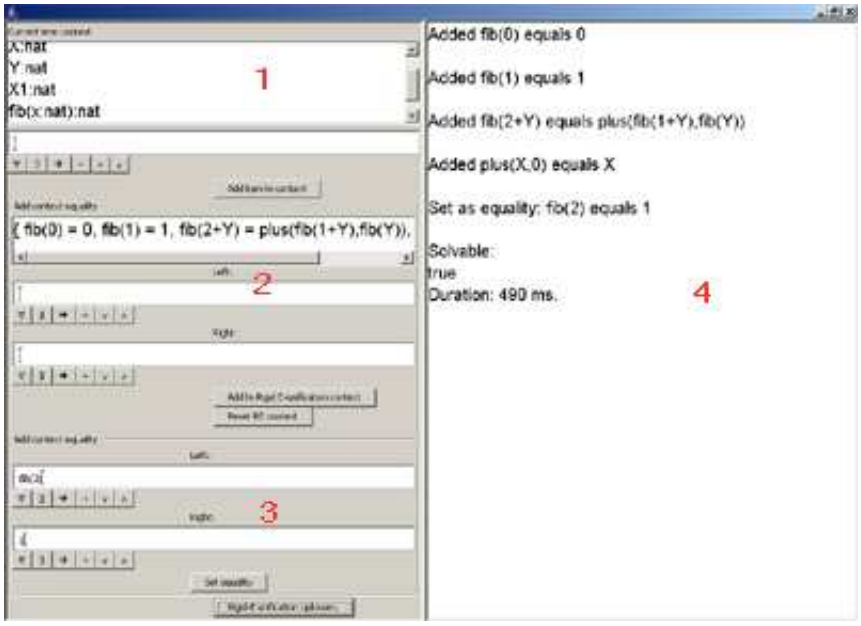



Figure 6.1: Screenshot of the tool to solve rigid E-unification problems.

2. This field shows the context of the rigid E-unification problem. New equalities can be added by using the input fields below it. The input fields were already present in Cocktail and provide parsing of entered formulas. If they are not correct, or terms in the formula are not defined in the context of Cocktail, a parser error will be returned.

3. These input fields can be used to define the goal of the rigid E-unification problem.

4. Output of the tool, showing what the user has done. Also, after the user presses the 'Rigid E-Unification Oplossen' button (translation: Solve the rigid E-unification problem), the result will be shown in this field (i.e. true or false). Furthermore, in the standard output stream the derivation is shown for a solvable problem (i.e. which calculus rules were applied).

# Chapter 7

# Results

This chapter gives an overview of the achieved results of the masters project. Furthermore, some suggestions are provided with regard to future work.

## 7.1 Project overview

The project consisted of a lot of research. We first had to read and try to understand papers concerning rigid E-unification. Soon a very promising paper [DV96] was found. However, it soon became clear that the real problem behind solving rigid E-unification problems is to check satisfiability of a constraint set. Therefore, research was performed on checking satisfiability of ordered constraints.

Next to this research, we needed to actually implement an algorithm to solve a rigid E-unification problem. This was completely done in the context of Cocktail, to make sure that it should be not too difficult to incorporate the algorithm in Cocktail later on. The implementation consists of about 4500 lines of code, including comments. A very simple test application has been written, in which a rigid E-unification problem can be defined, as well as solved.

Cocktail will become a more useful tool if it is able to automatically construct a proof using equations. Most interesting theorems that need to be proved when deriving a program from its specification depend on such reasoning. The work done in this project can serve as the basis for this extension of Cocktail.

## 7.2 Future work

There still is a lot of work that should be done. We give some suggestions for such work.

- First of all, actually incorporate the rigid E-unification algorithm in Cocktail's tableau prover. It should be fairly easy to do so, since the implementation made for this project is completely based on the Cocktail code.

- In chapter 3 another method is suggested to reason with equations in a tableau prover: add additional tableau-expansion rules. It might be very interesting to also implement this suggestion. Then this approach can be compared with rigid E-unification. Lots of automated theorem provers exist and usually each one uses one particular approach, for example the tableau method using syntactical unification. Comparing such different approaches is always a bit biased, since one tool might be implemented more efficient than some other tool. Cocktail contains different automated theorem proving approaches. Because each approach uses the same framework (i.e. Cocktail) the approaches can be compared more realistically.

- Cocktails ATP, once extended with rigid E-unification, can be compared to other theorem provers that allow reasoning about equations. Examples of such theorem provers are PrInS (see [Gie02])[1] and Spass (see [Inf07]). As noted above, such comparisons will always be biased, because these theorem provers use a framework different than that of Cocktail.

- The implementation of the rigid E-unification solver is very slow. It would be well worth the effort to find out whether the algorithm can be implemented more efficiently. However, it must be kept in mind that solving rigid E-unification problems is in NP.

- Cocktail allows users to keep a context containing definitions, theorems, lemmas, etc. This context is needed to automatically find proofs for theorems. However, the running time of the rigid E-unification solving algorithm grows exponentially with the size of the problem context. Therefore, it would be very nice if only those items from

[1] If someone succeeds in actually running PrInS, I would be more than interested in this astonishing accomplishment.

the context of Cocktail are placed in the context of rigid E-unification problems, that are needed to solve the problem. A possible approach to this would be to only include formulas from the context of Cocktail that contain at least one subterm of the righthand-side of the rigid E-unification problem that is to be solved. These formulas will then form the context of the rigid E-unification problem. This will probably lead to a small context for the rigid E-unification problem. If no solution is found for the problem, the context can be extended. If no solution is found after $N$ extensions of the context, for some predefined $N$, we conclude that the problem has no solution. This conclusion might be false, effectively leaving open a leaf that could have been closed if we would have taken more of the context into account. However, as described in chapter 3, Cocktail currently also uses bounds on the size of the tableau and the running time of the algorithm. This $N$ would thus be nothing more than just another bound.

- A very nice property of Cocktail is that its proofs can be translated to $\lambda$-terms in $\lambda P-$. See [Fra00] for more information about $\lambda$-terms and the translation. Using such $\lambda$-terms, correctness of the proof can be checked. Unfortunately, when adding rigid E-unification to the tableau prover, this translation will no longer be possible. This is due to the fact that terms are transformed to skolem negation normal form. After replacing an existential quantifier with a skolem function, information is lost about the original formula. In general, it seems to be impossible to generate the original formula from its skolemized form. However, a study could be performed that confirms this theory, or contradicts it leading to a revised translation scheme.

# Appendix A

# Proving fib(2) = 1

This appendix is added to illustrate the tableaux method extended with rigid E-unification.
Everything is denoted in the terminology of Cocktail. For instance, the natural numbers are defined by 0 and a successor function $s$. For example, in Cocktail 1 is defined as $s(0)$.

First, we define a plus operator $p$ over the natural numbers.

**Definition A.0.1 (Plus)**
$(\forall x : x \in \mathbb{N} : p(x, 0) = x)$
$(\forall x : x \in \mathbb{N} : (\forall y : y \in \mathbb{N} : p(x, s(y)) = s(p(x, y))))$

Let $fib$ denote the Fibonacci function. $fib$ can be defined as follows:

**Definition A.0.2 (Fibonacci)**

$$\begin{array}{ll} & fib(0) = 0 \\ & fib(s(0)) = s(0) \\ (\forall x : x \in \mathbb{N} : & fib(s(s(x))) = p(fib(s(x)), fib(x)) \end{array}$$

Now, given these definitions, we are going to prove $fib(s(s(0)) = s(0)$.

We start off by creating a tableau root with the following label:

$$\begin{array}{ll} \Gamma = & \{\ (\forall x : x \in \mathbb{N} : p(x, 0) = x), \\ & (\forall x : x \in \mathbb{N} : (\forall y : y \in \mathbb{N} : p(x, s(y)) = s(p(x, y)))), \\ & fib(0) = 0, \\ & fib(s(0)) = s(0), \\ & (\forall x : x \in \mathbb{N} : fib(s(s(x))) = p(fib(s(x)), fib(x)), \\ & fib(s(s(0)) \neq s(0)\} \end{array}$$

The label $\Gamma$ contains the formulas in the definitions of $p$ and $fib$, as well as the negation of the formula that we want to prove.

The only tableau expansion rule that is applicable on the root node is the $\delta$ rule. There are several universally quantified formulas. We apply the $\delta$ rule several times to obtain the equalities needed to prove the theorem. After doing so, the tableau depicted in figure A.1 is obtained.

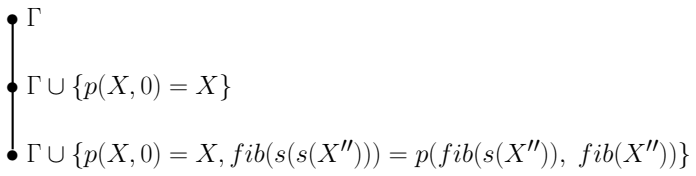



Figure A.1: Tableau obtained after applying the $\delta$ rule two times

The tableau only has one leaf. If we are able to close this leaf, the proof is complete. We will try to close the leaf by using rigid E-unification. To do so, we check whether the label of the leaf contains an inequality. Such an inequality does indeed exist: $fib(s(s(0))) \neq s(0)$. The negation of this formula will serve as the right-hand side of a rigid E-unification problem. The context of the problem consists of all equalities in the label of the leaf. Thus, we identify the following rigid E-unification problem:

$$\{\ p(X,0) = X,\ fib(0) = 0,\ fib(s(0)) = s(0),$$
$$fib(s(s(X''))) = p(fib(s(X'')),\ fib(X''))\ \}\ \vdash fib(s(s(0))) = s(0)$$

If this problem has a solution, the leaf can be closed, since a contradiction in the label is then found. We try to solve this rigid E-unification problem using the $\mathcal{BSE}$-calculus.

We start with an empty constraint set. We try to apply the lrbs-rule, with $l : fib(0)$, $r : 0$, $s : p(fib(s(X'')), fib(X''))$, $t : fib(s(s(X''))$ and $p : fib(X'')$. This step is shown below. Term $l$ is indicated by a single line, while term $p$ is indicated with a double line.

$$\frac{\begin{array}{c}\{\ p(X,0) = X,\ \underline{fib(0)} = 0,\\ fib(s(0)) = s(0),\ fib(s(s(X''))) = \overline{p(fib(s(X'')), \underline{\underline{fib(X'')}}})\}\\ \vdash fib(s(s(0))) = s(0) \cdot \emptyset\end{array}}{\begin{array}{c}\{\ p(X,0) = X,\ fib(0) = 0,\\ fib(s(0)) = s(0),\ fib(s(s(X''))) = p(fib(s(X'')),\ 0)\}\\ \vdash fib(s(s(0))) = s(0)\cdot\\ \{fib(0) \succ 0,\ p(fib(s(X'')), fib(X'')) \succ fib(s(s(X''))),\ fib(0) = fib(X'')\}\end{array}}\ (\textsc{lrbs})$$

We have to check whether the constraint set obtained by applying this rewrite step is satisfiable, since that is a condition that has to be met before the rule can be applied. We will only show this satisfiability check for this rewrite step, since showing it for all steps would require far too much space.

The first step in checking satisfiability of a constraint set is to rewrite it to its solved form using the transformation rules shown in table 5.1. The constraint set contains three elements for which we show the transformation result[1]:

1. $fib(0) \succ 0 \xrightarrow{O_2}_{\mathcal{R}} \top$
2. $p(fib(s(X'')), fib(X'')) \succ fib(s(s(X'')) \xrightarrow{D_2}_{\mathcal{R}}$
   $p(fib(s(X'')), fib(X'')) \succ s(s(X''))$
3. $fib(0) = fib(X'') \xrightarrow{D_1}_{\mathcal{R}} 0 = X''$

After performing these transformations and combining the results we obtain formula
$p(fib(s(X'')), fib(X'')) \succ s(s(X'')) \wedge 0 = X''$.
On this formula transformation rule $R$ can be applied, after which we obtain the following solved form: $p(fib(s(0)), fib(0)) \succ s(s(0)) \wedge 0 = X''$.

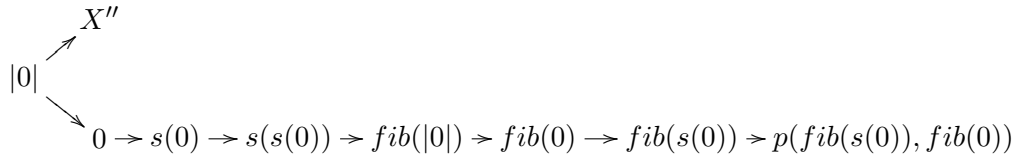
Now that we have calculated the solved form of the constraint set, we need to calculate all simple systems that are not trivially bottom. Each simple system must contain first limit ordinal $\omega$, as well as the smallest term $|0|$. Term $|0|$ is newly introduced and defined to be strictly smaller than all other terms, since our problem already contains regular term 0. In this case, the smallest function according to the precedence ordering is $s$. This is a unary function, so according to the definition of $\omega$ we get $\omega = fib(|0|)$.

[1] $\mathcal{F} : p \succ_{\mathcal{F}} fib \succ_{\mathcal{F}} s \succ_{\mathcal{F}} 0$

To obtain all simple systems that are not trivially bottom we construct the simple systems graph. First of all, we calculate the set of all subterms contained in the solved form, $\omega$ and $|0|$:

$$\{\ |0|,\ 0,\ X'',\ s(0),\ s(s(0)),\ fib(0),\ fib(s(0)),\ fib(|0|),\ p(fib(s(0)),\ fib(0))\ \}$$

Using the algorithm described in chapter 6, the graph depicted below is obtained.




The next step is to derive all simple systems from this graph that are not trivially bottom. As soon as one such simple system exists, the original constraint set is satisfiable. In this case almost the entire ordering is fixed by the simple systems graph. Only variable $X''$ has to be added in an appropriate position. We will suffice by showing that at least one satisfiable simple system exists:

$$p(fib(s(0)), fib(0)) \succ fib(s(0) \succ fib(0) \succ fib(|0|) \succ s(s(0)) \succ s(0) \succ X'' = 0 \succ |0|$$

Now we can conclude that the constraint set is satisfiable. So, the lrbs rule can be applied. Each rule of the $\mathcal{BSE}$ calculus expands the constraint set. For this example, we will transform the constraint set to its solved form before applying the next rule. This simplifies the constraint sets of which satisfiability needs to checked. We will now show the complete derivation. The interested reader may verify satisfiability of each rewrite step.

First we apply the lrbs-rule to rewrite $fib(X'')$ to 0. It is clear that in order to do so, $X''$ has to be equal to 0. The lrbs-rule is applied with the following parameters:
$l : fib(0)$, $r : 0$, $s : p(fib(s(X'')), fib(X''))$, $t : fib(s(s(X''))$ and $p : fib(X'')$

$$\frac{\begin{array}{c}\{\ p(X,0) = X,\ \underline{fib(0)} = 0,\\ fib(s(0)) = s(0),\ fib(s(s(X''))) = p(fib(s(X'')),\ \underline{\underline{fib(X'')}})\}\\ \vdash fib(s(s(0))) = s(0) \cdot \emptyset\end{array}}{\begin{array}{c}\{\ p(X,0) = X,\ fib(0) = 0,\\ fib(s(0)) = s(0),\ fib(s(s(X''))) = p(fib(s(X'')), 0)\}\\ \vdash fib(s(s(0))) = s(0)\cdot\\ \{p(fib(s(0)), fib(0)) \succ s(s(0)),\ X'' = 0\}\end{array}}\ (\textsc{lrbs})$$

Now we can apply the lrbs-rule again, to transform $fib(s(X''))$ to $s(0)$. The following parameters are used:
$l : fib(s(0)), r : s(0), s : p(fib(s(X'')), 0), t : fib(s(s(X'')))$ and $p : fib(s(X''))$

$$\frac{\begin{array}{c}\{\ p(X,0) = X,\ fib(0) = 0,\\ \underline{fib(s(0))} = s(0),\ fib(s(s(X''))) = p(\underline{\underline{fib(s(X''))}}, 0)\}\\ \vdash fib(s(s(0))) = s(0)\cdot\\ \{p(fib(s(0)), fib(0)) \succ s(s(0)),\ 0 = X''\}\end{array}}{\begin{array}{c}\{\ p(X,0) = X,\ fib(0) = 0,\\ fib(s(0)) = s(0),\ fib(s(s(X''))) = p(s(0), 0)\} \quad \vdash fib(s(s(0))) = s(0)\cdot\\ \{p(fib(s(0)), fib(0)) \succ s(s(0)),\ p(fib(s(0)), 0) \succ s(s(0)),\ 0 = X''\}\end{array}}\ (\textsc{lrbs})$$

Using the lrbs-rule, we now transform $p(s(0), 0)$ to $X$. The following parameters are used:
$l : p(X, 0), r : X, s : p(s(0), 0), t : fib(s(s(X''))$ and $p : p(s(0), 0)$

$$\frac{\begin{array}{c}\{\ \underline{p(X,0)} = X,\ fib(0) = 0,\\ fib(s(0)) = s(0),\ fib(s(s(X''))) = \underline{\underline{p(s(0), 0)}}\} \quad \vdash fib(s(s(0))) = s(0)\cdot\\ \{p(fib(s(0)), fib(0)) \succ s(s(0)),\ p(fib(s(0)), 0) \succ s(s(0)),\ 0 = X''\}\end{array}}{\begin{array}{c}\{\ p(X,0) = X,\ fib(0) = 0, \quad fib(s(0)) = s(0),\ fib(s(s(X''))) = X\}\\ \vdash fib(s(s(0))) = s(0)\cdot\\ \{p(fib(0), fib(s(0))) \succ s(s(0)),\ p(0, fib(s(0))) \succ s(s(0)), p(s(0), 0) \succ s(s(0)),\\ X'' = 0, X = s(0)\}\end{array}}\ (\textsc{lrbs})$$

Now the context contains $fib(s(s(X'')) = X$, while the right-hand side of

the rigid E-unification problem is $fib(s(s(0))) = s(0)$. So, we use the rrbs-rule to unify $fib(s(s(X'')))$ with $fib(s(s(0)))$, transforming $fib(s(s(0)))$ to $X$. The rrbs-rule is applied with the following parameters:
$l : fib(s(s(X'')))$, $r : X$, $s : fib(s(s(0)))$, $t : s(0)$ and $p : fib(s(s(0)))$

$$\frac{\begin{array}{c}\{\ p(X,0) = X,\ p(X',Y') = P(Y',X'),\ fib(0) = 0,\\ fib(s(0)) = s(0),\ \underline{fib(s(s(X')))} = X\} \qquad \vdash \underline{\underline{fib(s(s(0)))}} = s(0)\cdot\\ \{p(fib(0), fib(s(0))) \succ s(s(0)),\ p(0, fib(s(0))) \succ s(s(0)), p(s(0),0) \succ s(s(0)),\\ X'' = 0, X = s(0)\}\end{array}}{\begin{array}{c}\{\ p(X,0) = X,\ p(X',Y') = P(Y',X'),\ fib(0) = 0,\\ fib(s(0)) = s(0),\ fib(s(s(X''))) = X\} \qquad \vdash X = s(0)\cdot\\ \{p(fib(0), fib(s(0))) \succ s(s(0)),\ p(0, fib(s(0))) \succ s(s(0)), p(s(0),0) \succ s(s(0)),\\ fib(s(s(0))) \succ s(0), X'' = 0, X = s(0)\}\end{array}}\ (\textsc{rrbs})$$

Finally, we can apply equality resolution. The constraint set does not change by applying this rule, since it already contains constraint $X = s(0)$. The following parameters are used:
$s : X$ and $t : s(0)$.

$$\frac{\begin{array}{c}\{\ p(X,0) = X,\ p(X',Y') = P(Y',X'),\ fib(0) = 0,\\ fib(s(0)) = s(0),\ fib(s(s(X''))) = X\} \qquad \vdash X = s(0)\cdot\\ \{p(fib(0), fib(s(0))) \succ s(s(0)),\ p(0, fib(s(0))) \succ s(s(0)), p(s(0),0) \succ s(s(0)),\\ fib(s(s(0))) \succ s(0), X'' = 0, X = s(0)\}\end{array}}{\begin{array}{c}\vdash X = X\cdot\\ \{p(fib(0), fib(s(0))) \succ s(s(0)),\ p(0, fib(s(0))) \succ s(s(0)), p(s(0),0) \succ s(s(0)),\\ fib(s(s(0))) \succ s(0), X'' = 0, X = s(0)\}\end{array}}\ (\textsc{er})$$

Since $X = X$ is a tautology, we are done. We have successfully used the $\mathcal{BSE}$-calculus to show that given context $\Gamma$ it can be derived that $fib(s(s(0))) = s(0)$. Since the label of the (only) leaf of our tableau contains $fib(s(s(0))) \neq s(0)$, we have found a contradiction. Thus, the leaf can be closed, concluding our proof.

# Bibliography


[Bec97] Bernhard Beckert. Semantic tableaux with equality. *Journal of Logic and Computation*, 7(1):39–58, 1997.

[Com90] Hubert Comon. Solving symbolic ordering constraints. *International Journal of Foundations of Computer Science*, 1(4):387–412, 1990.

[DV96] Anatoli Degtyarev and Andrei Voronkov. What you always wanted to know about rigid e-unification. In *Logics in Artificial Intelligence*, pages 50–69, 1996.

[Fra00] M.G.J. Franssen. *Cocktail: A Tool for Deriving Correct Programs*. PhD thesis, Eindhoven University of Technology, 2000.

[Gie02] Martin Giese. *Proof search without backtracking for free variable tableaux*. PhD thesis, Universität Karlsruhe, 2002.

[GSNP88] Jean H. Gallier, Wayne Snyder, Paliath Narendran, and David A. Plaisted. Rigid e-unification is np-complete. In *LICS*, pages 218–227. IEEE Computer Society, 1988.

[HR00] Michael R. A. Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, Cambridge, England, 2000.

[Inf07] Max-Planck-Institut Informatik. Spass theorem prover, 2007.

[JMS04] Jan Otop Jerzy Marcinkowski and Grzegorz Stelmaszek. On a semantic subsumption test, 2004.

[Kal90] Anne Kaldewaij. *Programming: the derivation of algorithms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[Knu68] Donald E. Knuth. *Fundamental Algorithms*. Addison-Wesley, 1968.

[Nie93] Robert Nieuwenhuis. Simple LPO constraint solving methods. *Information Processing Letters*, 47(2):65–69, 1993.

[NW01] Andreas Nonnengart and Christoph Weidenbach. Computing small clause normal forms. In *Handbook of Automated Reasoning*, pages 335–367. 2001.

[RS01] Harald Rue$\beta$ and Natarajan Shankar. Deconstructing shostak. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*, page 19, Washington, DC, USA, 2001. IEEE Computer Society.

[Sho84] Robert E. Shostak. Deciding combinations of theories. *J. ACM*, 31(1):1–12, 1984.

[Smu68] Raymond M. Smullyan. *First-Order Logic*. Ergebnisse Der Mathematik und Irher Grenzgebiete, Band 43. Springer-Verlag, 1968.

[Ste01] Thomas H. Cormen; Charles E. Leiserson; Ronald L. Rivest; Clifford Stein. *Intro-duction to Algorithms, second edition*. The MIT Press and McGraw-Hill, 2001.