Implementing rigid e-unification

Implementing Rigid E-unification

Michael Franssen (m.franssen@tue.nl)
Eindhoven University of Technology,
Dept. of Mathematics and Computer Science,
Den Dolech 2, 5612 AZ Eindhoven, The Netherlands

Abstract

Rigid E-unification problems arise naturally in automated theorem provers that deal with equality. While there is a lot of theory about rigid E-unification, only few implementations exist. Since the problem is NP-complete, direct implementations of the theory are slow. In this paper we discuss how to implement a rigid E-unifier, focussing on efficiency. First, we introduce an efficient representation of unifying substitutions to implement a regular Robinson unification algorithm. Next, we discuss the algorithm to compute rigid E-unifiers as proposed by Degtyarev et al. [4] and we discuss how to solve the symbolic ordering constraint as proposed by Comon [1] and Nieuwenhuis [10]. Finally, we discuss how rigid E-unification can be implemented efficiently. However, the worst case is still exponential.

1 Introduction

Rigid E-unification problems arise naturally in automated theorem provers for first-order logic. The theorem prover transforms a first-order formula into a less complicated form, removing existential quantifiers by skolemization. Universal quantifiers are usually instantiated (repeatedly). Since it is hard to come up with good instances, one often uses a fresh variable to instantiate the universally quantified formula. A useful value for the variable is then found by unification later on. This variable is called a rigid variable, since it can be instantiated only once during the construction of the proof.

If the logic of the theorem prover allows for equations, a typical problem that has to be solved is $s_1 = t_1, \ldots, s_n = t_n \vdash s = t$: does equality $s = t$ hold, given the equalities $s_1 = t_1, \ldots, s_n = t_n$. Shostak [12] proposed a uniform treatment for this problem, but this only works for ground formulas without free variables. However, as stated above, the formulas we consider may contain rigid variables.

Therefore, the theorem prover should be able to compute a rigid E-unifier to solve these kinds of problems. A rigid E-unifier is a substitution θ from rigid variables to ground formulas, such that $\theta(s_1 = t_1), \ldots \theta(s_n = t_n) \vdash \theta(s = t)$. The rigid E-unification problem is to compute such a θ for $s_1 = t_1, \ldots, s_n = t_n \vdash s = t$.

In [4] a complete tableau calculus is proposed using rigid E-unification. Degtyarev et al. present not only the tableau calculus, but also a $\mathcal{BSE}$ calculus to compute rigid E-unifiers. Their calculus uses solutions to rigid E-unification problems to close individual tableau leaves and combines those to compute a closure for the entire tableau. A more direct approach would be to try to compute the solution to a simultaneous rigid E-unification problem as proposed by [7], but this proved to be undecidable [3].

The $\mathcal{BSE}$ calculus used by Degtyarev et al. uses a symbolic ordering constraint that has to be satisfiable for the solution to be correct. Although they 'assume that there is an effective procedure for checking constraint satisfiability', this turns out to be a complicated problem in itself and is NP-complete.

Comon [1] proposed to use the lexicographical path ordering to check such constraints. For this, the original constraint is rewritten by $\rightarrow_R$ into a set of solved forms. Each solved form then gives rise to an (exponential) number of so called simple systems. The original constraint is satisfiable if at least one of the simple systems is. Deciding satisfiability of a simple systems is, as the name suggests, simple (it can be done in linear time).

Nieuwenhuis [10] simplified the decidability of simple systems, by loosening their definition. Even though his method still uses an exponential number of simple systems, one has to make less case distinctions to decide on satisfiability.

Even though all this theory is available only few implementations exists of first-order theorem provers that allow equalities. Two examples are Spass [13] and Prins [8]. Spass is based on the connection or resolution method and uses other ways than rigid E-unification to deal with equalities, which are beyond the scope of this paper. Prins is based on semantic tableaux, but uses completion based methods to deal with equalities. The advantage of using rigid E-unification to handle equalities is that it is easier to integrate in existing theorem provers, since it does not change the rules of the prover itself. An efficient implementation is to our knowledge not described in literature so far.

In this paper, we follow the lines of Degtyarev, Comon and Nieuwenhuis to actually implement a rigid E-unifier and provide guidelines to do this as efficiently as possible in the context of an NP-complete problem.

Section 2 of this paper describes the concepts and definitions we need. In Section 3 we discuss an efficient implementation of Robinson unification, to introduce a simple, yet compact, representation of unifying substitutions. Section 4 describes the theory of solving rigid E-unification problems: Subsection 4.1 describes the $\mathcal{BSE}$ calculus found in [4], Subsection 4.2 describes the rewrite system $\rightarrow_R$ to compute solved forms as defined in [1] and Subsection 4.3 describes how to compute simple systems from these solved forms and how to decide their satisfiability following [10]. In Section 5 we implement a rigid E-unifier following the same three steps, providing details to gain efficiency for each step. Finally, the results are discussed in Section 6.

2 Preliminaries

Definition 1 (formulas) *The grammar for formulas F is given by*

$$\begin{aligned} F &::= \top \mid \bot \mid F \wedge F \mid F \vee F \mid T = T \\ T &::= \mathcal{F}(T^*) | \mathcal{V} \end{aligned}$$

Where $\mathcal{V}$ is a set of variables and $\mathcal{F}$ is a set of function symbols. Function symbols of arity 0 are called constants. The semantics of $=$ is equality in the model. That is, if in a model $\models s = t$ holds, s and t are mapped to the same value. From this it also follows that $=$ represents Leibniz equality, meaning that if $s = t$ holds, all occurrences of s in any term T may be replaced by t without changing the semantics of T.

$FV(P)$ denotes the free variables of a (set of) formula(s) P as usual. A substitution $\theta : \mathcal{V} \to T$ is a mapping from variables to terms.

Definition 2 (Lexicographical Path Ordering LPO) *Let $s \simeq f(s_1, \ldots, s_n)$ and $t \simeq g(t_1, \ldots, t_m)$ be two terms. Let $>_{\mathcal{F}}$ be an arbitrary ordering of the function symbols in $\mathcal{F}$ (called the precedence ordering). Then $s \succ t$ iff one of the following holds:*

1. $(\exists i : 1 \leq i \leq m : s_i \succ t)$
2. $f >_{\mathcal{F}} g \wedge (\forall j : 1 \leq j \leq m : s \succ t_j)$
3. $f \simeq g \wedge (\exists j : 1 \leq j \leq n : (\forall i : 1 \leq i < j : s_i = t_i) \\ \wedge\ s_j \succ t_j \\ \wedge\ (\forall k : j < k \leq n : s \succ t_k))$

The LPO is total on ground terms. We use $\simeq$ to denote syntactic equality of terms.

Definition 3 (Most general unifier) *Let s and t be terms. A unifier θ is a substitution such that $\theta(s) \simeq \theta(t)$, where $\simeq$ denotes syntactical equality. θ is called a most general unifier if for any unifier θ' there exists a substitution θ'' such that $\theta' = \theta'' \circ \theta$. The most general unifier is unique.*

Definition 4 (Rigid E-unification problem) *Let $s_1 = t_1, \ldots, s_n = t_n$ be a list of equations and let $s = t$ be a single goal equation. The rigid E-unification* *problem denoted by $s_1 = t_1, \ldots, s_n = t_n \vdash s = t$ is stated as follows: Is there a substitution θ, such that $\theta(s) = \theta(t)$ holds in all models in which all of $\theta(s_1) = \theta(t_1), \ldots, \theta(s_n) = \theta(t_n)$ hold? (i.e. is there a substitution θ such that $\theta(s_1 = t_1 \wedge \ldots \wedge s_n = t_n \Rightarrow s = t)$ is a tautology?). This problem is shown to be NP-complete [6].*

3 Efficient Robinson unification

In this section we discuss an efficient algorithm to compute most general unifiers based on the algorithm by Robinson [11]. Although this algorithm is not directly used for rigid E-unification, its discussion is useful for several purposes: (1) it will be used to demonstrate the techniques we will use later on. (2) regular unification is embedded in the computation of solved forms needed for rigid E-unification. (3) Regular unification is useful in its own right.

A unification algorithm computes the most general unifier for two expressions e_1 and e_2. That is, it computes a substitution θ, such that $\theta(e_1) \simeq \theta(e_2)$ if it exists. The most general unifier θ of e_1 and e_2 also has the property that any other unifier θ' can be written as $\theta'' \circ \theta$ for certain θ''. An algorithm to compute most general unifiers for first-order formulas was first described by Robinson in [11].

In [9], Martelli and Montanari describe an efficient unification algorithm. Their algorithm, however, requires somewhat complicated data-structures and requires the tree-representation of formulas to be inspected in a specific order. In this section, we will construct an efficient unification algorithm that uses no complicated data-structures and that does not impose any specific order in which the tree-representation of formulas has to be inspected. This is important, since in an automated theorem prover most unification attempts will fail and most datastructures that are constructed will be discarded.

We start with a straightforward implementation of a basic Robinson unification algorithm:

function unify(S :set of $T = T$) : $\mathcal{V} \to T$ |[
  if $S = \emptyset$ $\to$ **return** id
  else
    $s = t :\in S$;
    $S := S \setminus \{s = t\}$
    if $s \in \mathcal{V}$ $\to$
      if $s = t$ $\to$ **return** unify(S)
      elseif $s \in FV(t)$ $\to$ **abort** // occurs check
      else return $(s \mapsto t) \circ$unify($(s \mapsto t)(S)$)
      fi
    elseif $t \in \mathcal{V}$ $\to$ **return** unify($S \cup \{t = s\}$)
    else
      let $f(s_1, \ldots, s_n) :: s, g(t_1, \ldots, t_m) :: t$;
      if $f \equiv g$ $\to$ **return** unify($S \cup \{s_i = t_i \mid 1 \leq i \leq n\}$)
      else abort // function mismatch
      fi
    fi
  fi
]|

By **let** $f(s_1, \ldots, s_n) :: s$ we mean that s must have the form $f(s_1, \ldots, s_n)$ and that we will use f and s_1 till s_n to denote its components.

Correctness and termination of this algorithm follow from the observation that the set of most general unifiers of S does never change under application of the algorithm and that the overall complexity of the formulas in S decreases[1].

Note that every substitution in S is followed by the incorporation of the corresponding mapping in θ. Hence, instead of S, we can maintain a set S' of equations and maintain invariant that $\theta(S') = S$. We then get the following version of the unification algorithm:

[1]the formula $s \mapsto t$ is considered less complex than $t \mapsto s$ in the case where $s \in \mathcal{V}$ and $t \notin \mathcal{V}$.

```
function unify(θ : V → T; S' :set of T = T) : V → T |[
  if S' = ∅ → return θ
  else
    s = t :∈ S';
    S' := S' \ {s = t}
    if θ(s) ∈ V →
      if θ(s) = θ(t) → return unify(θ, S')
      elseif θ(s) ∈ FV(θ(t)) → abort // occurs check
      else return unify([θ(s) ↦ θ(t)] ∘ θ, S')
      fi
    elseif θ(t) ∈ V → return unify(θ, S' ∪ {t = s})
    else
      let f(θ(s_1), ..., θ(s_n)) :: θ(s), g(θ(t_1), ..., θ(t_m)) = θ(t);
      if f ≡ g → return unify(θ, S' ∪ {s_i = t_i | 1 ≤ i ≤ n})
      else abort // function mismatch
      fi
    fi
  fi
]|
```

We will show how efficiency can be increased by choosing appropriate representations for S and θ.

The substitution θ can be represented as a list

$$\theta' =< [x_1 \mapsto t_1], \dots, [x_n \mapsto t_n] >$$

of one-point mappings. θ is then computed θ_0, where

$$\begin{aligned} &\theta_i = [x_{i+1} \mapsto \theta_{i+1}(t_{i+1})] \circ \dots \circ [x_n \mapsto \theta_n(t_n)] \\ &\theta_n = id \end{aligned}$$

The substitution $[x \mapsto \theta(t)] \circ \theta$ is then represented by $< [x \mapsto t], \theta' >$, where θ' is the list representation of θ. That is, instead of changing a substitution θ, we simply prepend $[x \mapsto t]$ to a list representation θ', where x and t are (sub)terms that are already available.

Note that the x_i in θ' are unique. We define $\theta'(x_i) = t_i$ for $1 \leq i \leq n$.

During the computation of the unifier, it is necessary to compare the first symbols of terms, i.e. the variable if the terms are variables and the function symbol if the terms are function applications. Whatever the representation of our unifier θ is, we must be able to compute the first symbol of $\theta(s)$ for a term s. Therefore, we use the following lemma.

Lemma 5 *Let $\theta' =< [x_1 \mapsto t_1], \dots, [x_n \mapsto t_n] >$ be a list of one-point mappings and let θ be the corresponding substitution as defined above. Also assume that x_i does not occur in $\theta_i(t_i)$ for any i, and that all x_i are different. Let v be a variable, such that $v = x_i$ for certain i. Then $\theta(v) = \theta(t_i)$.*

Proof:

$$
\begin{array}{cl}
 & \theta(v) \\
= & \{\text{definition of } \theta\} \\
 & ([x_1 \mapsto \theta_1(t_1)] \circ \ldots \circ [x_n \mapsto \theta_n(t_n)])(v) \\
= & \{v \notin \{x_{i+1}, \ldots, x_n\}\} \\
 & ([x_1 \mapsto \theta_1(t_1)] \circ \ldots \circ [x_i \mapsto \theta_i(t_i)])(v) \\
= & \{v = x_i; \text{ definition of } \circ\} \\
 & ([x_1 \mapsto \theta_1(t_1)] \circ \ldots \circ [x_{i-1} \mapsto \theta_{i-1}(t_{i-1})])(\theta_i(t_i)) \\
= & \{x_i \text{ does not occur in } \theta_i(t_i)\} \\
 & ([x_1 \mapsto \theta_1(t_1)] \circ \ldots \circ [x_i \mapsto \theta_i(t_i)])(\theta_i(t_i)) \\
= & \{\text{definition of } \circ \text{ and } \theta_i\} \\
 & \theta(t_i)
\end{array}
$$

□

This lemma allows us to (repeatedly) apply a one-point mapping to the terms s and t in order to compute a partial unfolding of $\theta(s)$ and $\theta(t)$. Application will be repeated until a non-variable is encountered or until the full image is computed. This will eliminate most terms of the form $\theta(x)$ in the algorithm, which now becomes:

function unify(θ' :list of $\mathcal{V} \mapsto T; S'$:set of $T = T$) :list of $\mathcal{V} \mapsto T$ |[
  if $S' = \emptyset \ \rightarrow$ **return** θ'
  else
    $s = t :\in S'$;
    $S' := S' \setminus \{s = t\}$
    while $(\exists e.[s \mapsto e] \in \theta')$ **do** $s := \theta'(s)$ **od**
    while $(\exists e.[t \mapsto e] \in \theta')$ **do** $t := \theta'(t)$ **od**
    if $s \in \mathcal{V} \ \rightarrow$
      if $s = t \ \rightarrow$ **return** unify(θ', S')
      elseif $s \in FV(\theta(t)) \ \rightarrow$ **abort** // occurs check
      else return unify($< [s \mapsto t], \theta' >, S'$)
      fi
    elseif $t \in \mathcal{V} \ \rightarrow$ **return** unify($\theta', S' \cup \{t = s\}$)
    else
      let $f(\theta'(s_1), \ldots, \theta'(s_n)) :: s, g(\theta'(t_1), \ldots, \theta'(t_m)) :: t$;
      if $f \equiv g \ \rightarrow$ **return** unify($\theta', S' \cup \{s_i = t_i \mid 1 \leq i \leq n\}$)
      else abort // function mismatch
      fi
    fi
  fi
]|

The only occurrence of θ in this version is in the guard $s \in FV(\theta(t))$. Since we do not want a direct representation of θ, but represent θ by θ', we will check for occurrences of s in $\theta(t)$ with the following algorithm:

function occursCheck(θ' :list of $\mathcal{V} \mapsto T$; $v : \mathcal{V}$; $s : T$) : **bool** |[
  while $(\exists e.[s \mapsto e] \in \theta') \rightarrow s := \theta'(s)$;
  if $s \in \mathcal{V} \rightarrow$ **return** $s = v$
  else
    let $f(\theta'(s_1), \ldots, \theta'(s_n)) :: s$;
    return $(\bigvee i : 1 \leq i \leq n$:occursCheck$(\theta', v, s_i))$
  fi
]|

Finally, we can note that the only reason the set S' is maintained is to pick one element from it and alter θ' accordingly. If we pick this element at the time of the recursive call, we can eliminate the need for S' altogether, yielding our final algorithm for regular unification:

function unify(θ' :list of $\mathcal{V} \mapsto T; s, t : T$) :list of $\mathcal{V} \mapsto T$ |[
  while $(\exists e.[s \mapsto e] \in \theta')$ **do** $s := \theta'(s)$ **od**
  while $(\exists e.[t \mapsto e] \in \theta')$ **do** $t := \theta'(t)$ **od**
  if $s \in \mathcal{V} \rightarrow$
    if $s = t \rightarrow$ **return** θ'
    elseif occursCheck$(\theta', s, t) \rightarrow$ **abort** // occurs check
    else return $< [s \mapsto t], \theta' >$
    fi
  elseif $t \in \mathcal{V} \rightarrow$ **return** unify(θ', t, s)
  else
    let $f(\theta'(s_1), \ldots, \theta'(s_n)) :: s, g(\theta'(t_1), \ldots, \theta'(t_m)) :: t$;
    if $f \equiv g \rightarrow$
      foreach $i : 1 \leq i \leq n$ **do** θ' :=unify(θ', s_i, t_i) **od**
      return θ'
    else abort // function mismatch
    fi
  fi
]|

Our final algorithm does not impose any order in which subterms of s and t are unified. Also, the only required data structure is a list of pairs, in which the first element of each pair refers to a variable and the second element refers to the sub-term that should be substituted for this variable. These sub-terms already exist within the original terms s and t to be unified, so the data structure is a list of pairs of pointers. Also, the length of this list cannot exceed the number of free variables occurring in s and t. Such a data structure requires very little memory and can be manipulated efficiently. Especially in a context where most of the unification attempts will fail (e.g. in an automated theorem prover) this has the advantage that no data has to be copied and no time is wasted in substitutions to represent partial unifiers that will fail during a later stage of construction.

Operationally the final algorithm can be understood as a synchronous tree traversal of the terms s and t to be unified. Whenever a variable x is encountered in, for instance, s, one checks if a substitute for x exists in θ'. If so, then the synchronous tree traversal continues with the substitute for x. If not, an occurs check is performed and θ' is extended to include the mapping $x \mapsto e$, where e is the corresponding sub term in t.

4 Solving rigid equation problems

In [4] a calculus $\mathcal{BSE}$ is used to rewrite the goal equation into a trivial form. For each rewrite step, conjuncts are added to a constraint, which has to be satisfiable. About computing the satisfiability of this constraint they state: *"We assume that there is an effective procedure for checking constraint satisfiability"* and they provide a reference to (among others) [10]. However, when implementing Degtyarev's calculus, most of the work involves checking constraint satisfiability. Also, interactions and optimizations that involve both the $\mathcal{BSE}$ calculus and the constraint checking are not addressed.

The approach to solve rigid equation problems used in this paper can be summarized as follows:

- Use the calculus $\mathcal{BSE}$ to rewrite the goal into a trivial form (i.e. $s = s$), provided that the constraint generated by the rules is satisfiable. The $\mathcal{BSE}$ calculus requires the Leibniz property of the semantics of $=$.
- In order to check satisfiability of the constraint, first rewrite it into a number of solved forms.
- To check satisfiability of a solved form, generate the corresponding simple systems and check if a satisfiable simple system exists.

In the following subsections, we will briefly describe each of these steps.

4.1 The $\mathcal{BSE}$ calculus

The rigid equation is rewritten according to the calculus $\mathcal{BSE}$, given in Figure 1. These rules are used to rewrite the goal equation into a trivial form (i.e. $e{\simeq}e$). However, in order to guarantee progress during rewriting, we only want to replace bigger terms by smaller ones, according to some ordering (e.g. the lexicographical path ordering (lpo)). Such an ordering is total on ground terms, but since the terms also contain rigid variables, it is not always clear which side of the equation is the bigger term. Therefore, a constraint $\mathcal{C}$ is maintained, to which conjuncts are added that claim that rewriting takes places in the right direction. A constraint $\mathcal{C}$ is called satisfiable if there exists a substitution for the rigid variables such that $\mathcal{C}$ holds. Hence, if $\mathcal{C}$ is satisfiable all rewritings that have been applied result in smaller terms. Therefore, after applying each $\mathcal{BSE}$ rule, one has to check satisfiability of the constraint.

$$lrbs \quad \frac{E \cup \{l = r, s[p] = t\} \vdash_\forall e \cdot \mathcal{C}}{E \cup \{l = r, s[r] = t\} \vdash_\forall e \cdot \mathcal{C} \wedge l \succ r \wedge s[p] \succ t \wedge l \simeq p} \quad p \notin \mathcal{V}, s[r] \not\equiv t$$

$$rrbs \quad \frac{E \cup \{l = r\} \vdash_\forall s[p] = t \cdot \mathcal{C}}{E \cup \{l = r\} \vdash_\forall s[r] = t \cdot \mathcal{C} \wedge l \succ r \wedge s[p] \succ t \wedge l \simeq p} \quad p \notin \mathcal{V}$$

$$er \quad \frac{E \vdash_\forall s = t \cdot \mathcal{C}}{\vdash_\forall s = s \cdot \mathcal{C} \wedge s \simeq t} \quad s \not\equiv t$$

Figure 1: The calculus $\mathcal{BSE}$: left rigid basic superposition (lrbs), right rigid basic superposition (rrbs) and equality resolution (er). A rule may only be applied if the resulting constraint is satisfiable.

4.2 Solved forms

A constraint imposes an ordering of terms, which are not necessarily ground. Hence, indirectly, this ordering puts some restrictions on the values that may be assigned to the rigid variables contained in these terms.

The rewrite system $\rightarrow_R$ defined in Figures 2 and 3 (taken from [1]) is used to make the restrictions on the rigid variables explicit. That is, based on the definition of LPO, the restrictions are rewritten such that they explicitly limit the possible values for rigid variables. The normal form of this system is either $\top$ (if the constraint trivially holds), $\perp$ (if the constraint is trivially inconsistent), or it is a disjunction of constraints of the form:

$$x_1 \simeq t_1 \wedge \ldots \wedge x_n \simeq t_n \wedge u_1 \succ v_1 \wedge \ldots \wedge u_m \succ v_m,$$

Equality rules:

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

(C_1) $f(v_1, \ldots, v_n) \simeq g(u_1, \ldots, u_m) \rightarrow_R \perp$
if $f \not\equiv g$

(R) $x \simeq t \wedge P \rightarrow_R x \simeq t \wedge P[x := t]$
if $x \in FV(P) \setminus FV(t)$, P is a conjunction of (in)equations and $t \in \mathcal{V} \Rightarrow t \in FV(P)$.

(O_1) $s \simeq t[s] \rightarrow_R \perp$
if $s \not\equiv t[s]$

Figure 2: The rules of $\rightarrow_R$ that deal with equality. Note that these are equal to the rules of regular Robinson unification.

Inequality rules:

(D_2) $f(v_1, \ldots, v_n) \succ g(u_1, \ldots, u_m) \to_R$
$f(v_1, \ldots, v_n) \succ u_1 \wedge \ldots \wedge f(v_1, \ldots, v_n) \succ u_m$
if $f \succ_{\mathcal{F}} g$.

(D_3) $f(v_1, \ldots, v_n) \succ g(u_1, \ldots, u_m) \to_R$
$v_1 \succeq g(u_1, \ldots, u_m) \vee \ldots \vee v_n \succeq g(u_1, \ldots, u_m)$
if $g \succ_{\mathcal{F}} f$.

(D_4) $f(v_1, \ldots, v_n) \succ f(u_1, \ldots, u_n) \to_R$
$v_1 \succeq f(u_1, \ldots, u_n) \vee \ldots \vee v_n \succeq f(u_1, \ldots, u_n)$
$\vee\ (v_1 \succ u_1 \wedge f(v_1, \ldots, v_n) \succ u_2 \wedge \ldots \wedge f(v_1, \ldots, v_n) \succ u_n)$
$\vee\ (v_1 \simeq u_1 \wedge v_2 \succ u_2 \wedge \ldots \wedge f(v_1, \ldots, v_n) \succ u_n)$
$\vee\ \ldots$
$\vee\ (v_1 \simeq u_1 \wedge v_2 \simeq u_2 \wedge \ldots \wedge v_n \succ u_n)$
$\vee\ v_1 \succeq f(u_1, \ldots, u_n) \vee \ldots \vee v_n \succeq f(u_1, \ldots, u_n)$

(O_2) $t[s] \succ s \to_R \top$
if $t[s] \not\equiv s$.

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

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

(T_2) $s \simeq t \wedge s \succ t \to_R \bot$

Figure 3: The rules of $\to_R$ that deal with inequality.

where $x_1, \ldots x_n$ are variables not occurring in $t_1, \ldots, t_n, u_1, \ldots u_m, v_1, \ldots v_m$ and where for every $i, 1 \leq i \leq m$ either u_i or v_i is a variable and v_i is not a subterm of u_i or vica versa. Every disjunct in this formula is called a *solved form*. Hence, in a solved form every conjunct either states the value of a rigid variable, or it constitutes an upper or lower bound for the value of a rigid variable. The part $x_1 \simeq t_1 \wedge \ldots \wedge x_n \simeq t_n$ is called the solved part and the part $u_1 \succ v_1 \wedge \ldots \wedge u_m \succ v_m$ is called the constrained part of the solved form. If $m = 0$ we have a solution for the constraint and do not have to take the next step.

4.3 Simple systems

In order to decide whether or not a solved form $x_1 \simeq t_1 \wedge \ldots \wedge x_n \simeq t_n \wedge u_1 \succ v_1 \wedge \ldots \wedge u_m \succ v_m$, is satisfiable, one has to find out if a ground substitution for the rigid variables exists such that the constraint holds according to LPO. The solved form provides the substitutes for $x_1, \ldots, x_n$. For the constrained part, a set of simple systems has to be computed. For each simple system it can be checked whether or not it is inconsistent. The solved form (and hence also the original constraint) is satisfiable iff there is a simple system for the solved form that is not inconsistent.

The set of simple systems for the constrained part c of a solved form is computed as follows:

- Compute the set $sub(c)$ of all sub terms occurring in c.
- Consider all possible orderings of $sub(c)$, where every sub term s_2 of $s_1 \in sub(c)$ occurs after s_1.
- For each possible ordering, put either $\simeq$ or $\succ$ between the terms in all possible ways that are consistent with the original constrained part.

Example 6 *Consider a constrained part $c = f(g(x)) \succ y$. The set $sub(c)$ is then $\{f(g(x)), g(x), x, y\}$. The possible orderings of these sub terms are*

$$\begin{array}{lllllllll}
(1) & f(g(x)) & , & g(x) & , & x & , & y \\
(2) & f(g(x)) & , & g(x) & , & y & , & x \\
(3) & f(g(x)) & , & y & , & g(x) & , & x \\
(4) & y & , & f(g(x)) & , & g(x) & , & x
\end{array}$$

When inserting either $\succ$ or $\simeq$ between these terms consistently with the constrained part, we get a list of 27 simple systems:

$$\begin{array}{ccc}
(1) & (2) & (3) \\
f(g(x)) \simeq g(x) \simeq x \succ y & f(g(x)) \simeq g(x) \succ y \simeq x & f(g(x)) \succ y \simeq g(x) \simeq x \\
f(g(x)) \simeq g(x) \succ x \simeq y & f(g(x)) \simeq g(x) \succ y \succ x & f(g(x)) \succ y \simeq g(x) \succ x \\
f(g(x)) \simeq g(x) \succ x \succ y & f(g(x)) \succ g(x) \simeq y \simeq x & f(g(x)) \succ y \succ g(x) \simeq x \\
f(g(x)) \succ g(x) \simeq x \simeq y & f(g(x)) \succ g(x) \simeq y \succ x & f(g(x)) \succ y \succ g(x) \succ x \\
f(g(x)) \succ g(x) \simeq x \succ y & f(g(x)) \succ g(x) \succ y \simeq x & \\
f(g(x)) \succ g(x) \succ x \simeq y & f(g(x)) \succ g(x) \succ y \succ x & \\
f(g(x)) \succ g(x) \succ x \succ y & &
\end{array}$$

Ordering (4) does not produce any simply system, since $y \simeq f(g(x))$ and $y \succ f(g(x))$ are both inconsistent with the constrained part c.

For simple systems it is possible to check whether or not they are satisfiable, by checking if they are trivially bottom. To check if a simple system is trivially bottom, we check if it contains any of the following (subscript $_s$ means that the (in)equality holds according to the simple system s):

1. $f(s_1, \ldots s_n) \simeq_s g(t_1, \ldots, s_m)$ with f different from g.
2. $f(s_1, \ldots, s_p) \simeq_s f(s'_1, \ldots, s'_p)$ and $(\exists i \in 1 \ldots p \; . \; \neg(s_i \simeq_s s'_i))$.
3. $s \simeq_s t$ and t is a proper subterm of s or vica versa.
4. $f(s_1, \ldots, s_p) \succ_s t$ with $top(t) \succ_{\mathcal{F}} f$ and $\neg(\exists i \in 1 \ldots p \; . \; s_i \succeq_s t)$.
5. $f(s_1, \ldots, s_p) \succ_s f(s'_1, \ldots, s'_p)$ and $\neg(\langle s_1, \ldots, s_p \rangle \succ_s^{lex} \langle s'_1, \ldots, s'_p \rangle)$.

5 Implementing Rigid-E unification

To implement rigid-E unification, one has to follow the steps described above. In the following sections we will discuss how each step can be implemented efficiently. The $\mathcal{BSE}$ calculus can be implemented fairly directly, since its complexity is limited. Still we will provide some efficiency considerations. Computing solved forms for the constraint of the $\mathcal{BSE}$ calculus will be done using the techniques from Section 3 for Robinson unification. When checking satisfiability of the constrained part of a solved form, we are mainly concerned about avoiding to compute the entire set of simple systems. Instead, we will construct an algorithm to search for one satisfiable simple system that meets the constraint.

5.1 Implementing the $\mathcal{BSE}$ calculus

The $\mathcal{BSE}$ calculus can be implemented almost directly. However, one should still try to avoid computing the same result more than once. For this, we will consider the application of a $\mathcal{BSE}$ rule to the rigid E-unification problem $E \vdash s = t \cdot \mathcal{C}$, where $E = \{s_1 = t_1, \ldots, s_n = t_n\}$ is represented by a list of equations, $s = t$ is called the goal of the unification problem and $\mathcal{C}$ is the satisfiable constraint so far. Also, we will take into account the previous $\mathcal{BSE}$ rule applied to the unification problem (if any). The general strategy for applying a rule is then as follows: first check the side-conditions, then extend the constraint $\mathcal{C}$ piece by piece and check satisfiability after every extension, and finally compute E' and $s' = t'$ to generate the rigid E-unification problem that makes up the conclusion of the $\mathcal{BSE}$ rule. After the rule was applied, we first try to complete this derivation by recursion (depth first). If application of the rule does not lead to a solution, we try the next possible application by backtracking. We terminate the proof search if all derivations have been tried or as soon as a solution has been found.

ER: The rule *er* is only tried initially and then only if the previous rule applied was *rrbs*. The reason for this is, that if the rule could not successfully be applied before and the goal did not change, it will not be applicable now, since the constraint will only become more restrictive. If *er* is applied successfully, the rigid E-unification is solved. Hence, if *er* should be tried, it should be tried first.

RRBS: The *rrbs* rule is applied if the *er* could not be applied successfully, since *rrbs* will change the goal and hopefully, *er* can be applied afterwards. According to the *rrbs* rule, we have to take the following steps:

1. Extend $\mathcal{C}$ with $s \succ t$. If this fails, *rrbs* is not applicable[2].
2. For each equation $l = r$ in E:
3. Try to extend $\mathcal{C} \wedge s \succ t$ with $l \succ r$. If this fails, pick another equation.

[2]Of course one also has to try $t \succ s$, but we will not include this symmetry in our discussion

4. Compute all sub-terms of s that are not variables. Call this set S.
5. For each sub-term p in S:
6. Try to extend $\mathcal{C} \wedge s \succ t \wedge l \succ r$ with $l \simeq p$. If this fails, pick another sub-term.
7. Since $\mathcal{C} \wedge s \succ t \wedge l \succ r \wedge l \simeq p$ is satisfiable, we can successfully apply *rrbs*. Therefore, we compute $s[r]$ and recursively try to solve the rigid E-unification problem $E \vdash s[r] = t \cdot (\mathcal{C} \wedge s \succ t \wedge l \succ r \wedge l \simeq p)$. If this fails, we pick another sub-term and continue. If it does not fail, we also have a solution to the original unification problem.
8. If all sub-terms have been tried for all equations and no solution has been found, *rrbs* is not the next rule to be applied.

If the previous rule applied was *lrbs*, we need not to consider all equations in E. *lrbs* is tried after *rrbs* has been tried and also *lrbs* does not change the goal. Hence, all equations in E that have not been altered by *lrbs* have already been tried on the goal with *rrbs* during the previous step. Therefore, we will restrict the choice of $l = r$ to the equation that has been altered by *lrbs* in the previous step and we only have to try all equations in E initially and if the previous rule was *rrbs*.

LRBS: The rule *lrbs* is applied only if *er* and *rrbs* failed. The reason for this is that it only changes equations in E and hence does not directly alter or solve the goal. Also, since two equations are selected from E, there it can be applied in very many ways. We use two indices i and j with $i \neq j$ to indicate the chosen equations. $s_i = t_i$ corresponds to $l = r$ in the *lrbs* rule and $s_j = t_j$ corresponds to $s[p] = t$. The construction of the constraint is similar to *rrbs* and will not be discussed here again. Assume that j ranges from 1 till n in an outer loop and i ranges from 1 till n in an inner loop. If a combination of i and j leads to a successful application of *lrbs*, we do not need to try the full ranges again in the derivation that follows: apparently, all combinations i', j' with $1 \leq i' \leq n$ and $1 \leq j' \leq j$ failed. The combinations that need to be tried within the remainder of the derivation are i', j' with $i' = j$ and $1 \leq j' < j$ (since the equation at position j has changed) and those with $1 \leq i' \leq n$ and $j \leq j' \leq n$ (again $j = j'$ is included since the equation at position j has changed).

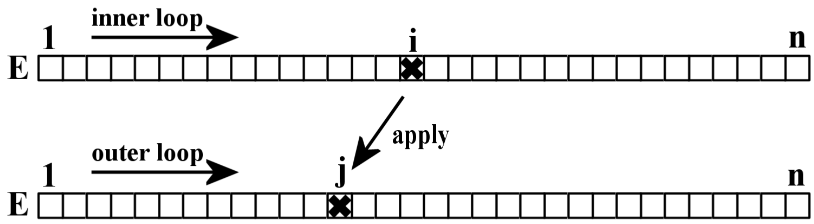

Figure 4: Selecting equations to apply LRBS

Unfortunately the $\mathcal{BSE}$ strategy above does not avoid all double computations, but it already restricts them enough to get an efficient and fast implementation.

5.2 Computing solved forms

In Section 3 we computed a most general unifier without performing any substitutions on the expressions being unified and without using complicated data structures. Also, we did not need to represent a set of equations to be unified. In this section, we will use the same representation for substitutions and also omit a set of equations and inequations to compute a set of solved forms for a constraint. However, this time we will not start with a straightforward implementation and then transform the algorithm into something more efficient, but rather present the final algorithm and explain its workings.

Notation and representation

The $\mathcal{BSE}$ calculus maintains a constraint as a conjunction of (in)equalities. The rewrite system $\rightarrow_R$ is then used to compute a disjunction of solved forms. However, before implementing it, we will alter $\rightarrow_R$ slightly. First of all, O1 will only be used for the case where s is a variable. In all other cases, either D1, C1 or R applies, or the equation is part of the solved form. Also, we will omit T2 altogether, since it is superfluous. Like O1, we only need to consider the case where s is a variable. Then T2's left hand side $s \simeq t \wedge s \succ t$ can be rewritten by R to $s \simeq t \wedge t \succ t$ and then by O3 to $\perp$.

In the implementation, a single solved form will be represented by a type called "Solved". Solved consists of two parts: (1) θ', which is a list of one point mappings representing the solved part in the same way that the unifier θ was represented in Section 3. (2) cp', which is a set of inequalities where at least one of the left hand side or the right hand side is a variable not occurring in the domain of θ'. The set cp' represent the constrained part of the solved form. Like the set S in our regular unification algorithm, the list cp' represents the inequations $\theta(cp')$, where θ is the substitution represented by θ' of the same solved form. Therefore cp' may be represented by a simple list of pairs where one element refers to a variable and the other element refers to a sub term that already existed in the formulas making up the original constraint. Since no substitutions will be actually performed, no sub terms have to be copied.

If sol represents a solved form (i.e. $sol : Solved$), its components are denoted as $sol.\theta'$ and $sol.cp'$ and sol may be written as $\langle \theta', cp' \rangle$.

Note that the solved form $\top$ is represented by an empty list of one-point mappings along with an empty set of inequalities. The solved form $\perp$ cannot be represented in this way. However, since we compute a set of solved forms, rather than a single solved form, we represent $\perp$ by the empty set of solved forms. This is conform to the interpretation of a solved form as a conjunction of partial constraints ($\top$ is the unity of $\wedge$) and the interpretation of a set of solved forms as a disjunction of solved forms ($\perp$ is the unity of $\vee$).

Since the $\rightarrow_R$ normal form of a constraint can consist of several solved forms, we compute a set of solved forms rather than a single solved form. The type representing a set of solved forms is denoted by "{Solved}".

Abstract code

The abstract algorithm to compute solved forms for a given (in)equality is given in below. We provide line numbers to simplify the discussion that follows:

```
0  function makeEqual(sol :Solved; s, t : T) : {Solved} |[
1    while (∃e.s ↦ e ∈ sol.θ′) →  s := sol.θ′(s);
2    while (∃e.t ↦ e ∈ sol.θ′) →  t := sol.θ′(t);
3    if s ∈ 𝒱  →
4      if s = t  →  return {sol} //special case of rule D1
5      elseif (occursCheck(sol.θ′, s, t)  →  return ∅ //rule O1
6      else return addSubst(sol, s, t) //rule R
7      fi
8    elseif t ∈ 𝒱  → return makeEqual(sol, t, s)
9    else
10     let f(sol.θ′(s_1), ..., sol.θ′(s_n)) :: s, g(sol.θ′(t_1), ..., sol.θ′(t_m)) :: t;
11     if f ≡ g  →  //rule D1
12       SOL:= {sol};
13       foreach i : 1 ≤ i ≤ n do H :=makeEqualSet(SOL, s_i, t_i) od;
14       return SOL
15     else return ∅ //rule C1
16     fi
17   fi
18 ]|

19 function makeEqualSet(SOL: {Solved}; s, t : T) : {Solved} |[
20   return (⋃ sol : sol ∈SOL:makeEqual(sol, s, t))
21 ]|

22 function addSubs(sol :Solved; v : 𝒱; t : T) : {Solved} |[
23   H := {⟨< (v ↦ t), sol.θ′ >, ∅⟩};
24   for(s≻t ∈ sol.cp′) do H :=makeGreaterSet(H, s, t) od;
25   return H
26 ]|
```

27 **function** makeGreater(sol :Solved; $s, t : T$) : {Solved} |[
28 **while** $(\exists e.s \mapsto e \in sol.\theta) \rightarrow\ s := sol.\theta'(s)$;
29 **while** $(\exists e.t \mapsto e \in sol.\theta) \rightarrow\ t := sol.\theta'(t)$;
30 **if** $(s \in \mathcal{V}) \vee (t \in \mathcal{V})\ \rightarrow$
31 **if** $s \in \mathcal{V} \wedge$ occursCheck$(sol.\theta', s, t) \rightarrow$ **return** $\emptyset$ //rule O3
32 **elseif** $t \in \mathcal{V} \wedge$occursCheck$(sol.\theta', t, s) \rightarrow$ **return** $\{sol\}$ //rule O2
33 **elseif** $(t \succ s) \in sol.cp' \rightarrow$ **return** $\emptyset$ //rule T1
34 **else return** $\{\langle sol.\theta', sol.cp' \cup \{s \succ t\}\rangle\}$ //irreducable $s \succ t$
35 **if**
36 **else**
37 **let** $f(\theta'(s_1), \ldots, \theta'(s_n)) :: s, g(\theta'(t_1), \ldots, \theta'(t_m)) :: t$;
38 **if** $f >_{\mathcal{F}} g\ \rightarrow\ //D2$
39 $H := \{sol\}$;
40 **foreach** $i : 1 \leq i \leq m$ **do** $H :=$makeEqualSet(H, s, t_i) **od**;
41 **return** H
42 **elseif** $g >_{\mathcal{F}} f\ \rightarrow\ //D3$
43 **return** ($\bigcup i : 0 \leq i \leq n$: makeEqual$(sol, s_i, t) \cup$
makeGreater(sol, s_i, t))
44 **else** $//f \equiv g\ \rightarrow D4$
45 $S =$ ($\bigcup i : 1 \leq i \leq n$: makeEqual$(sol, s_i, t) \cup$
makeGreater(sol, s_i, t));
46 $eq := \{sol\}$;
47 **for** $i := 1$ **to** n **do**
48 $H :=$makeGreater(eq, s_i, t_i);
49 **foreach** $j : i + 1 \leq j \leq n$ **do** $H :=$makeGreaterSet(H, s, t_j) **od**;
50 $S := S \cup H$;
51 $eq :=$makeEqualSet(eq, s_i, t_i)
52 **od**;
53 **return** S
54 **fi**
55 **fi**
56]|

57 **function** makeGreaterSet(SOL: {*Solved*}; $s, t : T$) : {*Solved*} |[
58 **return** ($\bigcup sol : sol \in SOL$:*makeGreater*$(sol, s, t)$)
59]|

• **makeEqual** (lines 0 till 18) implements the rules D1, C1, R and O1. makeEqual is almost the same as the function unify in Section 3 to compute a regular Robinson unification. However, it takes a solved form as input and not just a list of one point mappings. The idea is that the solved form is extended such that it also unifies $\theta(s)$ and $\theta(t)$ if possible. It returns a set of solved forms that satisfy this requirement. Note that this function will never abort, but instead return an empty set of solutions.

• **makeEqualSet** (lines 19 till 21) is a convenience function to apply the previous function to a set of solved forms instead of a single solved form and return the union of the results.

• **addSubs** (lines 22 till 26) implements the rule R and is called only by makeEqual. Instead of only adjusting θ', which is as trivial as it was for unify, addSubs also has to re-evaluate the constrained part of the solved form *sol*, since these inequatinos also might contain references to the variable v. The re-evaluation takes place by using makeGreater to add all the inequations of $sol.cp'$ to a solved form initially containing only $sol.\theta'$. This results in a complicated mutually recursive pattern between makeEqual an makeGreater. Termination of this recursion is discussed later.

• **makeGreater** (lines 27 till 56) implements all inequality rules of $\rightarrow_R$, except T2. The idea of this function is that the solved form *sol* is extended to a solved form that also satisfies $\theta(s) \succ \theta(t)$. Instead of adding the constraint to a set and rewriting it to a solved form, the rewriting takes place directly and only if $\theta(s) \succ \theta(t)$ is part of the solved form, $s \succ t$ is added to $sol.cp'$. Lines 28 and 29 unfold the substitution far enough to decide on the first function symbol of $\theta(s)$ and $\theta(t)$, just like is done in unify. The rules T1, O2 and O3 are applied in the cases where $\theta(s)$ or $\theta(t)$ are a variable (lines 30 till 35). Rule D2 (lines 38 till 41) is computed by recursively adding all the constraints of D2's right hand side to the solved form. Since every addition may return a set of solved forms, *sol* is put in a set and makeGreaterSet is used instead of makeGreater. Rule D3 (lines 42 and 43) simply unites the solved forms obtained from extending *sol* with all disjuncts of the right hand side of $D3$. Finally, D4 (lines 44 till 53) is computed in two steps. In line 45 the disjuncts $v_1 \succeq f(u_1, \ldots, u_n)$ till $v_n \succeq f(u_1, \ldots, u_n)$ are combined with *sol* to create the initial set S of solved forms that make up the result. Then, in line 46 till 52 the remaining disjuncts of D4 are computed one by one in variable H and joined with S. *eq* invariantly contains all solved forms obtained by extending *sol* with the equalities $s_1 \simeq t_1, \ldots s_{i-1} \simeq t_{i-1}$. Using this invariant H can be initialized efficiently.

• **makeGreaterSet** (lines 57 till 59) is, like makeEqualSet, a convenience function to apply the function makeGreater to a set of solved forms instead of a single solved form and return the union of the results.

Termination of the algorithm can be seen as follows: every recursive call either deals with subterms of the original arguments (calls to makeEqual, makeEqualSet, makeGreater or makeGreaterSet) or it eliminates one of the free variables (calls to addSubs).

5.3 Computing a satisfiable simple system

When computing simple systems and checking their satisfiability, the vast amount of possible simple systems causes problems. Therefore, we will not attempt to implement an algorithm that produces all simple systems and then check the satisfiability of each system. Instead, we will build a graph representing all simple systems that are consistent with the lpo and the constraints given by the solved form. We can then use a backtracking algorithm on this graph to construct the corresponding simple systems. During backtracking, we can compute if the partial simple system obtained so far satisfies all constraints and cut off the backtracking early if it does not. Also, we abort the algorithm altogether as soon as one satisfiable simple system is found, since this is sufficient to conclude that the original constraint is satisfiable.

In the simple systems graph the vertices represent sub terms and the edges represent the lpo-relationships between them. That is, the graph is constructed as follows:

- Construct a list of all sub terms of all terms in the constrained part.
- Sort this list according to the partial ordening defined by the lpo (The ordening is partial, since the sub terms contain variables).
- Represent the graph G by keeping track of the following: (1) the set of all vertices, say $G.V$ (2) the set of minimal vertices, say $G.M$ (3) for each vertex the set of direct successors, say $G.s$.
- Construct the simple systems graph, adding al sub terms from small to large. This is simpler than adding them in random order, since we never have to consider terms in the graph that are bigger than the term being added.
- Add the edges representing the relations imposed by the constrained part (By definition the left and right hand side of each inequation in the constrained part are incomparable, hence those edges cannot yet exist). When adding these additional edges, we need to check if cycles are introduced. If so, there will be no satisfiable simple systems, since the requirements contradict each other.

To check a reduced set of possible simple systems for satisfiability, we use the algorithm described below. This algorithm will search through all simple systems s that are consistent with the simple systems graph (i.e. if according to the graph $t_1 \succ t_2$, then $t_1 \succ_s t_2$). The arguments of the function have the following meaning:

- G is the representation of the simple systems graph.
- S is a partial simple system constructed so far. Initially, S is empty.
- in is an array stating for every vertex v the number of incoming edges when all vertices already in S are removed from the graph. If $in[v] > 0$, v may not be used to extend S, since obviously other vertices have to be added first. If v is already in S then $in[v] = -1$.
- $level$ is the number of $\succ$ symbols already in S. Initially, this is 0.

- *levels* is an array stating for every vertex v with $in[v] = 0$ the minimum value that *level* must have before it may be used to extend S. *levels* is updated during recursion whenever a vertex v is selected to extend S. The idea is that when v is (and the corresponding edges are) removed from G, the vertices $G.s(v)$ may not be prepended to extend S, unless at least one $\succ$ has been inserted after v.

```
function findSimpleSystem(G :Graph; S :SimpleSystem;
                          in : G.V → int; level : int;
                          levels : G.V → int
                         ) {SimpleSystem}|[
  r := ∅;
  if | S |=| G.V | →  r := {S}
  else
    foreach v ∈ G.V do
      if in[v] = 0 →
        in[v] := −1;
        foreach n ∈ G.s[v] do
          in[n] := in[n] − 1;
          levels[n] := level + 1
        od;
        if level > levels[v]∧valid(v≃S) →
          r := r∪findSimpleSystem(G, v≃S, in, level, levels)
        fi;
        if valid(v≻S)∧ | S |≠ 0 →
          r := r∪findSimpleSystem(G, v≻S, in, level + 1, levels)
        fi
        foreach n ∈ G.s[v] do in[n] := in[n] + 1 od;
        in[v] := 0
      fi
    od
  fi;
  return r
]|
```

The function *valid* checks the satisfiability constraints of the simple system that is its argument. It assumes that S is already satisfiable and only checks whether or not the extension (either $v\simeq$ or $v\succ$) introduces inconsistencies. This is done by simply using the satisfiability constraints for simple systems stated in Section 4.3. Note that constraint 3 does not need to be checked, since by construction we get that if s is a proper sub term of t or vice versa, then $s\simeq t$ will never be added to S.

For instance, when applied to Example 6, only 7 simple systems are inspected by the algorithm:

$$\begin{array}{lll} f(g(x)){\succ}g(x){\succ}x{\simeq}y & f(g(x)){\succ}g(x){\succ}y{\simeq}x & f(g(x)){\succ}y{\simeq}g(x){\succ}x \\ f(g(x)){\succ}g(x){\succ}x{\succ}y & f(g(x)){\succ}g(x){\simeq}y{\succ}x & f(g(x)){\succ}y{\succ}g(x){\succ}x \\ & f(g(x)){\succ}g(x){\succ}y{\succ}x & \end{array}$$

Note that any simple system which contains $t_1{\simeq}t_2$ has the same solutions as the same simple system in which t_1 and t_2 are swapped. Therefore, extending a simple system with $t{\simeq}$ will only be considered if t is greater than that of the topmost term of the simple system according to some total ordering on terms (e.g. the alphabetic ordering on textual representation of terms). This simple addition cuts down the inspected simple systems to 5:

$$\begin{array}{lll} f(g(x)){\succ}g(x){\succ}x{\succ}y & f(g(x)){\succ}g(x){\succ}y{\simeq}x & f(g(x)){\succ}y{\simeq}g(x){\succ}x \\ & f(g(x)){\succ}g(x){\succ}y{\succ}x & f(g(x)){\succ}y{\succ}g(x){\succ}x \end{array}$$

Also, in practice the computation is aborted as soon as a satisfiable simple system is found, since this already implies that the original constraint is satisfiable as well.

6 Results

In this paper we have shown how to fully implement a rigid E-unification algorithm. Although the problem itself has been shown to be NP-complete [6], we paid much attention to efficiency.

The exponential character of the problem is only due to the checking of constraint satisfiability of constrained parts of solved forms. However, since in our implementation al solved forms are computed first, we only need to perform these checks if there are no solved forms without a constrained part.

In our practical tests so far it turns out that this is hardly ever necessary and hence, most rigid E-unifications problems are solved without the exhaustive search for a satisfiable simple system. This yields a very efficient implementation for practical purposes. For instance, the following rigid E-unification problem (to compute Fibonacci number 2, with p is plus, s as successor and f as Fibonacci) was solved in 1.235ms by our Java implementation:

$$\begin{array}{l} p(X,0) = X, p(P,Q) = p(Q,P), \\ f(0) = 0, f(s(0)) = s(0), f(s(s(Y))) = p(f(Y), f(s(Y))) \\ \vdash f(2) = 1 \end{array}$$

Our next step will be the embedding of this rigid E-unifier in the tableaux based theorem prover of Cocktail [5]. Here we will also consider optimizations of the $\mathcal{BSE}$ calculus when many similar rigid E-unification problems have to be solved. Also, we are interested in the possibility of translating the trees generated by the $\mathcal{BSE}$ calculus into λ-terms, since this would allow the full embedding of the entire automated theorem prover in interactive systems like Coq [2].

References

[1] H. Comon, *Solving symbolic ordering constraints*, International Journal of Foundations of Computer Science **1** (1990), no. 4, 387–412.

[2] Coq, *The Coq proof assistant*, URL: `http: //coq. inria.fr/`, 2008.

[3] A. Degtyarev and A. Voronkov, *The undecidability of simultaneous rigid E-unification*, Theoretical Computer Science **166** (1996), no. 1–2, 291–300.

[4] A. Degtyarev and A. Voronkov, *What you always wanted to know about rigid E-unification*, Journal of Automated Reasoning **20** (1998), 47–80.

[5] M. Franssen, *Cocktail: A tool for deriving correct programs*, Ph.D. thesis, Eindhoven University of Technology, 2000.

[6] J.H. Gallier, P. Narendran, D. Plaisted, and W. Snyder, *Rigid E-unification is NP-complete*, In Proc. IEEE Conference on Logic in Computer Science (LICS), IEEE, 1988, pp. 338–346.

[7] J.H. Gallier, S. Raatz, and W. Snyder, *Theorem proving using rigid E-unification: Equational matings*, IEEE Conference on Logic in Computer Science (LICS), IEEE, 1987, pp. 338–346.

[8] M. Giese, *Proof search without backtracking for free variable tableaux*, Ph.D. thesis, Fakultät für Informatik, Universität Karlsruhe, July 2002.

[9] Alberto Martelli and Ugo Montanari, *An efficient unification algorithm*, ACM Transactions on Programming Languages and Systems **4** (1982), no. 2.

[10] R. Nieuwenhuis, *Simple LPO constraint solving methods*, Information Processing Letters **47** (1993), no. 2, 65–69.

[11] J.A. Robinson, *A machine oriented logic based on the resolution principle*, Journal of the Association for Computing Machinery **12** (1965), no. 1, 23–41.

[12] R.E. Shostak, *Deciding combinations of theories*, **31** (1984), no. 1, 1–12.

[13] C. Weidenbach, B. Afshordel, E. Keen, C. Theobalt, and D. Topić, *Spass theorem prover*, URL: `http: //spass.mpi-sb.mpg.de/`, Max-Planck-Institut für Informatik, 2007.