SAT Encoding of LPO Termination

BACHELOR THESIS

by

Jannes Timm

Vrije Universiteit Amsterdam:

Department of Computer Science
Section Theoretical Computer Science

Supervisors:

Femke van Raamsdonk (f.van.raamsdonk@vu.nl)
Jörg Endrullis (j.endrullis@vu.nl)

September 18, 2018

Abstract

In this thesis we describe how to algorithmically solve the decision problem of LPO (Lexicographic Path Order) termination by transforming the problem to SAT (Satisfiability of a Formula in Propositional Logic). In order to prove that a TRS (First-order Term Rewriting System) with a finite set of function symbols Σ and a finite set of rules R is LPO-terminating one needs to find a suitable precedence (order over Σ) that can be used to construct a LPO. The process of finding this precedence can be implemented by transforming the problem to SAT and using an efficient SAT-solver instead of manually trying all possible precedences.

Contents

Chapter 1

Introduction

Proving termination of TRS is generally undecidable, but there exist many methods for proving termination for some particular classes of TRS. Among those one can differentiate between semantical methods and syntactical methods, where path orderings are an important class of syntactical methods. Through finding a particular path ordering one can prove termination, whereas not finding the particular path ordering does not prove non-termination. The recursive path ordering (RPO) as introduced by Dershowitz is a classical example [1]. In this paper we limit the exploration of transformation to satisfiability in propositional logic (SAT) to the lexicographic path order (LPO), introduced by Kamin and Lévy [2]. First, we provide some of the necessary background to finite first-order term rewriting systems and proofs of termination by reduction orders. We introduce the defining conditions for LPO in their recursive form and show how to apply them to an example TRS $\mathcal{D}$, proving it to be LPO-terminating.

We show how to encode the decision problem of LPO termination to SAT, such that we can find a precedence $>_p$, that can be used to construct a LPO, if the given TRS is LPO-terminating. A short analysis of the space complexity of the resulting propositional logic formula is given, where two possible approaches of constructing the formula are contrasted. We illustrate the procedure by applying it to the same example TRS $\mathcal{D}$. After clarifying the encoding algorithm conceptually, we discuss how an actual Java program implements the encoding procedure and obtains a precedence if the input TRS is LPO-terminating. The implementation of the main algorithms are shown and a short analysis of time complexity of the encoding procedure is given.

Chapter 2

Background

As we restrict ourselves to first-order TRS in our LPO termination to SAT transformation, we give a short overview of first-order terms and the corresponding first-order TRS. Afterwards the decision problem of termination and corresponding proofs of termination are discussed. We assume the reader is familiar with propositional logic, specifically the property of satisfiability of a propositional logic formula. A SAT-solver is a program which accepts a propositional logic formula F as input and determines if the given formula is satisfiable. If it is satisfiable it returns an assignment for the variables which makes F *True*. An assignment just assigns each variable contained in the formula a value of *True* or *False*. For a more thorough background on SAT and SAT-solvers see Biere et al [3].

2.1 First-order Terms

First-order terms are the objects of first-order term rewriting systems. They are defined over a signature Σ.

Definition 1. A signature Σ is a finite set of function symbols F, G, .. with fixed arities. The arity of a function symbol is a natural number, specifying the number of arguments the function takes. Functions with arity 0 are called constants.

Terms are built from an alphabet consisting of disjoint sets of Σ and a countably infinite set of variables Var. The following definition is copied verbatim from Terese [4]:

Definition 2. The terms over Σ, denoted $Ter(\Sigma)$, are defined inductively as follows:

(i) $x \in Ter(\Sigma)$ for every $x \in Var$.

(ii) If F is a n-ary function symbol ($n \geq 0$) and $t_1, ..., t_n \in Ter(\Sigma)$, then $F(t_1, ..., t_n) \in Ter(\Sigma)$. By this notation it is understood that the constant symbols of Σ (case $n = 0$) are in $Ter(\Sigma)$ - we write C instead of $C()$.

For example given the signature $\Sigma = \{0, S, A, M\}$ with corresponding arities $0, 1, 2, 2$ and variables x and y we can define terms like 0, $S(x)$, $A(x, y)$ and $M(y, x)$. The following section shows how this definition of terms can be combined with rewrite rules to construct term rewriting systems.

2.2 Term Rewriting Systems

Definition 3. A term rewriting system (TRS) over a signature Σ is defined by a finite set of rewrite rules R. A rewrite rule r always has the form $t \to s$, with $t, s \in Ter(\Sigma)$, where t is not allowed to be a variable and all variables in s must also occur in t. Formally a TRS $T = (\Sigma, R)$.

We continue using the signature $\Sigma = \{0, S, A, M\}$ and variables $x, y \in Var$ from the previous section. The table 2.1 shows a an example of a TRS $\mathcal{D}$ (modified Dedekind). This TRS will serve as a running example in the following sections.

1. $A(0, x) \to x$
2. $A(S(x), y) \to A(x, S(y))$
3. $M(x, 0) \to 0$
4. $M(x, S(y)) \to A(x, M(x, y))$

Table 2.1: modified Dedekind ($\mathcal{D}$)

The left-hand side of a rewrite rule is often called a redex and the right-hand side is often called a contractum. Here we do not formally define substitution and contexts, but just give an intuitive explanation. A substitution is a function that maps a set of variables to to a set of terms. The definition can be extended to be valid for the context of terms, then it maps a set of terms to another set of terms. Specifically, it substitutes variables inside a term with other terms (or leaves them as is). For a substitution σ that replaces all occurrences of variable x with variable y we write $\sigma : [x \backslash y]$. A context is an incomplete term C, a term with one or more "holes". These holes can be replaced by terms, we write $C[t_1, ..., t_n]$.

Definition 4. The application of a rewrite rule $r = t \to s$ to a term m is called a rewrite step (or reduction step). The rewrite rule r can be applied to term m if there exists a substitution σ, such that m contains $\sigma(t)$, formally $m = C[\sigma(t)]$. The rewrite step contracts $\sigma(t)$ to $\sigma(s)$, we write $C[\sigma(t)] \to_r C[\sigma(s)]$. We can concatenate many rewrite steps (using different rewrite rules) to obtain a rewrite sequence, reducing a term m to a term n: $m \to_r ... \to_r n$, we write $m \twoheadrightarrow n$. A rewrite sequence can also be infinite if the application of rewrite rules always yields another term n containing a redex, such that we can always rewrite $n \to_r o$, with some rule $r \in R$..

For example given the term $A(0, M(z, 0))$ we can apply rule 1 of $\mathcal{D}$ to perform a first reduction step $A(0, M(z, 0)) \to M(z, 0)$ and then rule 3 to perform a second reduction step $M(z, 0) \to 0$. Here we applied the substitutions $\sigma_1 : [x \backslash M(z, 0)]$ and $\sigma_2 : [x \backslash z]$ in order to be able to apply the reduction rules.

2.3 Termination

A TRS is terminating if and only if it does not admit an infinite rewrite sequence given any term. In other words, for every term $m \in Ter(\Sigma)$ it must hold that only a finite amount of rewrite steps can be taken, such that m is reduced to a term n ($m \twoheadrightarrow n$) that can not be reduced any further (then term n is called a normal form

1. $A(0, x) \to x$

Table 2.2: Terminating TRS

1. $A(x, y) \to A(y, x)$

Table 2.3: Non-Terminating TRS

of t). To illustrate this here are two single-rule TRS, one which is terminating, one which is not terminating.
For the terminating TRS in Table 2.2 any valid term we construct will only allow a finite rewrite sequence as the redex contracts to one of its arguments and therefore the term can only become smaller. At some point it must reach a normal form, that can not be reduced any further, and that is where the rewrite sequence stops. For the non-terminating TRS in Table 2.3 we can construct a term $A(m, n)$ with any terms m and n, which admits an infinite rewrite sequence $A(m, n) \to_r A(n, m) \to_r A(m, n) \to_r \ldots$. This shows that commutivity laws can not be represented in a terminating TRS.

If we merge the two previous examples and construct a TRS with both their rules we obtain another non-terminating TRS (Table 2.4). Given the term $A(0, x)$ we can apply rule 1 resulting in a single rewrite step and a finite rewrite sequence. But since we could also apply rule 2 infinitely many times it is also possible to obtain an infinite rewrite sequence, which makes the TRS non-terminating.

1. $A(0, x) \to x$
2. $A(x, y) \to A(y, x)$

Table 2.4: Non-Terminating TRS

Termination of term rewriting systems is undecidable. However, many proof methods for classes of TRS have been developed. In order to discuss the use of orders for termination some definitions and theorems are needed first.

Theorem 1. *A TRS (Σ, R) is terminating if and only if there exists a well-founded order $\succ$ on $Ter(\Sigma)$ such that $t \succ s$ for every $t, s \in Ter(\Sigma)$ for which $t \to_r s$, with $r \in R$. A well-founded order is a strict order $\succ$ which does not admit an infinite descending sequence $t_0 \succ t_1 \succ \ldots$.*

One can hardly use this fact alone in order to prove termination, since any non-trivial signature Σ (one containing non-constant function symbols) will allow for infinitely many pairs of terms $t, s \in Ter(\Sigma)$. Luckily this theorem can be extended by using the notion of a reduction order on $Ter(\Sigma)$.

Definition 5. A reduction order on $Ter(\Sigma)$ is a well-founded order on $Ter(\Sigma)$ that is

(i) closed under substitutions; if $t \succ s$ and σ is an arbitrary substitution, then $\sigma(t) \succ \sigma(s)$, and
(ii) closed under contexts; if $t \succ s$ and C is an arbitrary context, then $C[t] \succ C[u]$.

A reduction order $\succ$ on $Ter(\Sigma)$ is called compatible with a TRS (Σ, R) if $t \succ s$ for every rewrite rule $t \to s \in R$.

Using the notion of a compatible reduction order on $Ter(\Sigma)$ it now becomes easier to proof termination of a TRS.

Theorem 2. *A TRS (Σ, R) is terminating if and only if it admits a compatible reduction order $\succ$ on $Ter(\Sigma)$.*

2.3.1 Termination proofs by reduction order

Since the TRS we deal with have finitely many rules, a compatible reduction order, if it exists, is effectively computable. Syntactical methods based on theorem 2 that lift a precedence $>_p$ on a signature Σ to a reduction order $\succ$ have been devised. A precedence on Σ is just a well-founded order on Σ. Two important classes of these methods are the recursive path order and Knuth-Bendix. Here we will only discuss a special version of the recursive path order; the lexicographic (recursive) path order.

Chapter 3

Lexicographic Path Order

3.1 Definition

In the following we will use a slighly modified version of the Definition 5.4.12 from the textbook by Baader and Nipkow to define the lexicographic path order [5].

Definition 6. Let Σ be a finite signature and $>_p$ be a well-founded order on Σ. The lexicographic path order $>_{lpo}$ on $T(\Sigma, V)$ induced by $>_p$ is defined as follows: t $>_{lpo}$ s iff:

(LPO1) *$s \in Var(t)$ and $t \neq s$, or*
(LPO2) *$t = f(t_1, ..., t_m)$ and $s = g(s_1, ..., s_n)$, and*
(LPO2a) *there exists i, $1 \leq i \leq m$ with $t_i \geq_{lpo} s$, or*
(LPO2b) *$f > g$ and $t >_{lpo} s_j$ for all j, $1 \leq j \leq n$, or*
(LPO2c) *$f = g$, $t >_{lpo} s_j$ for all j, $1 \leq j \leq n$, and there exists i, $1 \leq i \leq m$, such that $t_1 = s_1, ..., t_{i-1} = s_{i-1}$ and $t_i >_{lpo} s_i$.*

The lexicographic path order is a reduction order on $Ter(\Sigma)$, see Buchholz [6]. We call a TRS LPO-terminating if it admits a LPO for some precedence $>_p$.

3.2 Examples

In this section we apply the LPO definition to our running example TRS $\mathcal{D}$ and we show that there exists a well-founded order $>_p$ over the signature that can be lifted to the lexicographic path order. After introducing the SAT-encoding of LPO termination we will use $\mathcal{D}$ again to show how a SAT-solver would find this well-founded order $>_p$ over the signature Σ. We now show that the precedence $M >_p A >_p S >_p 0$ can be lifted to $>_{lpo}$, with all $t \to s \in R$ satisfying $t >_{lpo} s$, which shows that $\mathcal{D}$ is terminating.

For the first and third rewrite rules it is trivial to show $t >_{lpo} s$ as the condition in LPO1 applies. For the other two rules cases of LPO2 apply.

Rewrite rule 2 ($A(S(x), y) \to A(x, S(y))$) is a case of LPO2c:

$A = A$
- ▶ $A(S(x), y) >_{lpo} x$ (LPO1)
- ▶ $A(S(x), y) >_{lpo} S(y)$ (LPO2b):
  - $A >_p S$
  - ▶ $A(S(x), y) >_{lpo} y$ (LPO1)
- ▶ for $i = 1$, $t_i >_{lpo} s_i$: $S(x) >_{lpo} x$

Rewrite rule 4 ($M(x, S(y)) \to A(x, M(x, y))$) is a case of LPO2b:

$M >_p A$
- ▶ $M(x, S(y)) >_{lpo} x$ (LPO1)
- ▶ $M(x, S(y)) >_{lpo} M(x, y)$ (LPO2c):
  - $M = M$
  - ▶ $M(x, S(y)) >_{lpo} x$ (LPO1)
  - ▶ $M(x, S(y)) >_{lpo} y$ (LPO1)
  - ▶ for $i = 2$, $t_1 = s1$, $t_i >_{lpo} s_i$: $x = x$, $S(y) >_{lpo} y$ (LPO1)

As we have shown that all left-hand sides of the rewrite rules are heavier terms than the right-hand sides in the LPO and LPO is a reduction order, according to Theorem 2 the TRS $\mathcal{D}$ is terminating. We will confirm this finding without a given precedence $>_p$, by encoding the problem into SAT.

To illustrate the use of LPO further here is another very simple example which is not terminating and therefore can not be LPO-terminating either.

1. $A \to F(A)$

Table 3.1: non-terminating TRS

Here we only have one rule containing the two function symbols A and F, where A is a constant and F is unary. Without the application of any definition it is already easy to see that this rule can not be terminating, as it allows infinite rewriting of the produced A in the right hand side. When we try to check the LPO conditions for this system we see that only LPO2b and the precedence $A >_p F$ could potentially yield a LPO, but the next condition that $A >_{lpo} A$ trivially fails., therefore we can not find a LPO. As mentioned earlier, this result does not prove non-termination. We will revisit this TRS in the next chapter and show how the encoding procedure will yield a contradiction as the resulting formula.

Chapter 4

SAT Encoding

4.1 Description

To encode the procedure of finding a precedence $>_p$ over Σ which induces $>_{lpo}$ over the terms of rewrite rules R we will use Definition 6. Given any TRS T with rewrite rules $R(r_1, ..., r_n)$ and signature Σ the question if there exists a precedence on the function symbols in Σ that induces a LPO $>_{lpo}$ with $t >_{lpo} s$ for all $t \to s \in R$ can be translated to propositional logic by using the following encoding procedure. By checking satisfiability of the encoding we can not only prove termination (if satisfiable), but also find a valid precedence through one satisfying assignment of the variables.

Define propositional formula

$$F = F_1 \wedge F_2 \tag{4.1}$$

where F is true iff there exists an ordering $>_{lpo}$ with $t >_{lpo} s$ for all $t \to s \in R$ for the TRS T. F_1 encodes the necessary properties of the precedence $>_p$ on Σ and F_2 encodes the conditions for the lexicographic path order. There are two kinds of propositional variables that we use.

1. $P_{a,b}$, which is true iff $a >_p b$, where $a, b \in \Sigma$ and $>_p$ is the precedence on Σ.
2. $R_{t,s}$, which is true iff $t >_{lpo} s$, where $t, s \in Ter(\Sigma)$ and $>_{lpo}$ is the lexicographic path order induced by the precedence $>_p$.

Define propositional formula

$$F_1 = \underbrace{\left(\bigwedge_{a,b\in\Sigma} (P_{a,b} \leftrightarrow \neg P_{b,a})\right)}_{\text{A}} \wedge \underbrace{\left(\bigwedge_{a\in\Sigma} \neg P_{a,a}\right)}_{\text{B}} \wedge \underbrace{\left(\bigwedge_{a,b,c\in\Sigma} ((P_{a,b} \wedge P_{b,c}) \to P_{a,c})\right)}_{\text{C}} \tag{4.2}$$

The formula F_1 ensures that the order $>_p$ encoded by the P variables conforms to all the necessary conditions to be a precedence over Σ. Part A encodes anti-symmetry, Part B encodes irreflexivity and Part C encodes transitivity, therefore $>_p$ is a strict order. The only condition left is that of well-foundedness.

Lemma 3. *An order is well-founded if it is an order over a finite set, it is transitive and irreflexive.*

Proof. Assume we have a finite set A and an order $>$ defined on all possible pairs of elements $e_i \in A$, which is transitive and irreflexive. Assume $>$ is not well-founded. It follows there must be some infinite decending sequence $e_x > ... > e_y > ... > e_y$ ..., which necessarily contains some repeating element e_y. Then due to transitivity it must hold that $e_y > e_y$, which contradicts our assumption of irreflexivity of $>$. Therefore $>$ must be a well-founded order. $\square$

Since Σ is a finite set and $>_p$ is strict, it follows that $>_p$ is also well-founded and thus qualifies as a precedence over Σ.

Define propositional formula

$$F_2 = \bigwedge_{t \to s \in R} R_{t,s} \tag{4.3}$$

For each rewrite rule $t \to s \in R$ it must hold that $t >_{lpo} s$, which is captured by the $R_{t,s}$ variables. A $R_{t,s}$ variable is *True* iff the conditions for $t >_{lpo} s$ hold, so like the definition of the LPO conditions $R_{t,s}$ is defined recursively. Therefore each $R_{t,s}$ variable must be unrolled to a formula containing more R and P variables. Here we give the recursive definition of $R_{t,s}$, which has to be applied to every rule to generate the full F_2 formula.

If $s \in Var(t)$ and $t \neq s$ **(LPO1)**:

$$R_{t,s} = \text{True}$$

If t and s can be decomposed as $t = f(t_1, ..., t_m)$ and $s = g(s_1, ..., s_n)$ **(LPO2)**:

$$R_{t,s} = A \vee B \vee C \text{ , with}$$

$$A = \begin{cases} \bigvee_{i=1}^{m} R_{t_i,s} & \text{if } \neg\exists i.t_i = s \\ \text{True} & \text{if } \exists i.t_i = s \end{cases}$$

$$B = \begin{cases} P_{f,g} \wedge \bigwedge_{j=1}^{n} R_{t,s_j} & \text{if } f \neq g \\ \text{False} & \text{otherwise} \end{cases}$$

$$C = \begin{cases} \bigwedge_{j=1}^{n} R_{t,s_j} \wedge R_{t_i,s_i} & \text{if } f = g \wedge \exists i.1 \leq i \leq m, \left(\bigwedge_{h=1}^{i-1} t_h = s_h\right) \wedge (t_i \neq s_i) \\ \text{False} & \text{otherwise} \end{cases}$$

Else,

$$R_{t,s} = \text{False}$$

When generating the encoding of sub-formula F_2 there are two different approaches regarding the variables of type $R_{t,s}$. They can either be replaced recursively by their definitions which ultimately only contain variables of type $P_{a,b}$ (Version 1) or they can be left as is inside the formula (Version 2). In the latter case F_2 must be extended by adding clauses of form $R_{t,s} \leftrightarrow A \vee B \vee C$ for each variable of type $R_{t,s}$[1], not only the $R_{t,s}$ variables where t and s and left-hand and right-hand side in the rules, but also all the $R_{t,s}$ variables generated in the recursive calls.

4.2 Space Complexity

The size of the formula F is dependent on the size of the set Σ in general, the amount of rewrite rules R, and the sizes of the terms t and s in the rules $r = t \to s$. Let n be the size of Σ. We can see that the subformula F_1 will generate $3 * n^3 + 2 * n^2 + n$ P-variables. Part A generates 2 P-variables for each pair (a, b) of symbols, part B just generates 1 P-variable for each symbol, part C generates 3 P-variables for each triple (a, b, c). So the size of F_1 has an asymptotic upper bound of $O(n^3)$.

The size of F_2 varies for the two proposed versions. The size of Version 1 depends on the number of rules and the maximum size of the rules, where size is defined by the product of the amount of function symbols in the left and right term. Let o be the number of rules and $f_t * f_s$ the size of the biggest rule, where f_t is the number of function symbols in the left term and f_s is the number of function symbols in the right term. Then $o * f_t * f_s$ is an upper bound for the size of F_2. The subformula $R_{t,s}$ for any rule $r \in R$ can't exceed the size $f_t * f_s$ because for every function symbol $a \in \Sigma$ inside the left term only the variables $P_{a,b}$, with $b \in \Sigma$ and b inside the right term can potentially be generated.

The size of Version 2 depends on the number of rules and the maximum size of the rules, where here size is defined by the product of the number of function symbols *and* variables in the left and right term. This size of a rule directly corresponds to the maximum amount of recursive calls that are made when constructing the subformula $R_{t,s}$ corresponding to the rule (further explanation follows in the implementation section). Furthermore the size of Version 2 also depends on the amount of arguments in the left and the right term. Let o be the number of rules and $g_t * g_s$ the size of the biggest rule, where g_t is the number of function symbols and variables in the left term and g_s is the number of function symbols and variables in the right term. Let a_t be the maximum number of arguments in any left term and a_s the maximum number of arguments in any right term. Then $o * g_t * g_s * (a_t + a_s + 1)$ is an upper bound for the size of F_2. The subformula $R_{t,s}$ for any rule $r \in R$ can't exceed the size $g_t * g_s * (a_t + a_s + 1)$ because a maximum of $g_t * g_s$ R-variables can be generated, which are added as conjuncts in the form of bi-implications. The maximum size of such a bi-implication is $(a_t + a_s + 1)$, part A contributing a_t R-variables and part B or C contributing a_s R-variables and 1 P- or R-variable. We can see an example of this in the following Example section.

[1]here we mean adding each clause by means of conjunctions

4.3 Examples

To illustrate the procedure of encoding to SAT we revisit the rule 4 from our running example, the modified Dedekind TRS. We only look at F_2 and selectively unroll the $R_{t,s}$ variables and substitute back. To keep the subscripts readable the left term is referred to as $left$ and the right term as $right$.

4. $M(x, S(y)) \to A(x, M(x, y))$

Table 4.1: modified Dedekind rule 4

1. $\underline{R_{left,right} \leftrightarrow (R_{x,right} \vee R_{S(y),right}) \vee (P_{M,A} \wedge R_{left,x} \wedge R_{left,M(x,y)})}$
2. $R_{x,right} \leftrightarrow False$
3. $R_{S(y),right} \leftrightarrow (R_{y,right}) \vee (P_{S,M} \wedge R_{S(y),x} \wedge R_{S(y),M(x,y)})$
4. $R_{y,right} \leftrightarrow False$
5. $R_{S(y),x} \leftrightarrow False$
6. substitute 4,5 in 3: $R_{S(y),right} \leftrightarrow False$
7. substitute 2,6 in 1: $\underline{R_{left,right} \leftrightarrow (P_{M,A} \wedge R_{left,x} \wedge R_{left,M(x,y)})}$
8. $R_{left,x} \leftrightarrow True$
9. $R_{left,M(x,y)} \leftrightarrow (R_{x,M(x,y)} \vee R_{S(y),M(x,y)}) \vee (R_{left,x} \wedge R_{left,y} \wedge R_{S(y),y})$
10. $R_{x,M(x,y)} \leftrightarrow False$
11. $R_{S(y),M(x,y)} \leftrightarrow ((R_{y,M(x,y)}) \vee (P_{S,M} \wedge R_{S(y),x} \wedge R_{S(y),y})$
12. $R_{y,M(x,y)} \leftrightarrow False$
13. substitute 5,12 in 11: $R_{S(y),M(x,y)} \leftrightarrow False$
14. substitute 10,13 in 9: $R_{left,M(x,y)} \leftrightarrow (R_{left,x} \wedge R_{left,y} \wedge R_{S(y),y})$
15. $(R_{left,x} \wedge R_{left,y} \wedge R_{S(y),y}) \leftrightarrow True$
16. substitute 15 in 14: $R_{left,M(x,y)} \leftrightarrow True$
17. substitute 8,16 in 7: $\underline{R_{left,right} \leftrightarrow P_{M,A}}$ **(Result)**

This result is the same as the one in the previous section, in order for $left >_{lpo} right$ to hold it must be that $M >_p A$. For smaller examples like the modified Dedekind system one can obtain the precedence without using a SAT-solver to solve the obtained propositional formula. But for systems with longer and more rules it will not be trivial to see which precedence (if any) would satisfy the formula.

Here we can also see that Version 2 of F_2 would be much bigger in size than Version 1, all of the bi-implications (some of which are not even included in their

1. $A \to F(A)$

Table 4.2: non-terminating TRS

full form, e.g line 15) would be conjuncts in one big conjunction, whereas fully unrolling the R-variables leaves us with just one single P-variable.

We return to the previous example (Table 4.2) which is not LPO-terminating and show the application of our algorithm to it, which yields an unsatisfiable $F_2 = R_{A,F(A)}$.

1. $R_{A,F(A)} \leftrightarrow P_{A,F} \wedge R_{A,A}$
2. $R_{A,A} \leftrightarrow False$
3. substitute 2 in 1: $R_{A,F(A)} \leftrightarrow False$

The recursive call to $R_{A,A}$ yields a *False*, since neither the LPO1 nor the LPO2 conditions are met.

Chapter 5

Implementation

The SAT encoding algorithm was fully implemented in a Java program, which takes a TRS as input and produces a possible precedence $>_p$ in case the TRS is LPO-terminating. The Java implementation makes use of three external libraries, "commons-cli" to parse command line options, "tweety" for propositional logic and "sat4j" for SAT-solving. In total it consists of around 1300 lines of Java code, among which 500 lines are tests.

The program reads the input and constructs a list of rewrite rules from it. A set of all function symbols is extracted from the rules and formula F_1 is constructed according to the definition given earlier. For each rule a subformula $R_{t,s}$ is constructed by means of a recursive procedure. The subformulas are connected as conjuctions to obtain formula F_2 and the conjunction $F_1 \wedge F_2$ is transformed to conjunctive normal form (CNF)[1]. For a description of an algorithm that transforms any given propositional logic formula to CNF see Huth and Ryan [8]. The resulting formula F in CNF is given to the SAT-solver, which returns an assignment of the P-variables if F is satisfiable. The assignment of P-variables determines a precedence $>_p$, which is easily constructed from the variables and returned as output.

5.1 Core Algorithms

Here we present the implementation of the encoding, and the reconstruction of the precedence from a satisfying variable assignment for F.

5.1.1 Encoding Procedure

In the following pseudocode implementations of the recursive function $R_{t,s}$ and a helper function $\bigwedge R_{t,s_j}$ are presented. The actual implementation in Java uses methods of the propositional formula objects to construct conjunctions and disjunctions,

[1]Here we use a standard procedure provided by the *tweety* library of converting to an equivalent formula. This can result in an exponentially bigger formula, which is unwanted. Therefore it would be better to transform to a equisatisfiable formula, which is only bigger by a linear factor [7]. However, performance is not the goal of this thesis, instead we focus on exposition and analysis of the encoding.

but here we replaced some method calls with logical combinators to make the reading easier. The $R_{t,s}$ function is used for every rewrite rule of the input TRS and the resulting formulas act as conjuncts in the construction of F_2. We limit the exposition here to Version 1 of F_2 and skip the construction of F_1, as it is trivial.

Algorithm: Pseudocode of recursive function $R_{t,s}$

```
Input: Term t, Term s, HashMap propositions
Output: PropositionalFormula $R_{t,s}$
if $t$ is variable then
    return False
if $s$ is variable then
    if $s \in t$ then
        return True
    else
        return False

PropositionalFormula result = new Disjunction
result = result $\vee$ lpo2a(...)
if s.symbol != t.symbol then
    result = result $\vee$ lpo2b(...)
else
    result = result $\vee$ lpo2c(...)
return result
```

The algorithm's subcalls to lpo2a, lpo2b and lpo2c are actually inline code in the original implementation, but here we present them as extra procedure calls to improve readability.

Algorithm: lpo2a

```
for Term $t_i$: t.args do
    if $t_i$ equals $s$ then
        return True
Disjunction lpo2a = new Disjunction
for Term $t_i$: t.args do
    lpo2a = lpo2a $\vee$ $R_{t,s}(t_i, s, propositions)$
return lpo2a
```

Algorithm: lpo2b

```
Proposition $P_{f,g}$ = propositions.get("$P_{f,g}$")
PropositionalForumula lpo2b = $P_{f,g} \wedge \bigwedge R_{t,s_j}(t, s, propositions)$
return lpo2b
```

Algorithm: lpo2c

```
int iMax = Math.max(t.numArgs, s.numArs)
for int i = 1; i <= iMax; i++ do
    if $t_i$ != $s_i$ then
        PropositionalFormula $R_{t_i,s_i} = R_{t,s}(t.args.get(i),\ s.args.get(i),\ propositions)$
        PropositionalFormula lpo2c = $R_{t_i,s_i} \wedge \bigwedge R_{t,s_j}(t,\ s,\ propositions)$
        return lpo2c
return False
```

Both the lpo2b and lpo2c part of the algorithm make use of a helper function which constructs the formula $\bigwedge R_{t,s_j}$. This is an actual extra procedure to avoid code duplication in the main procedure.

Algorithm: Pseudocode of helper function $\bigwedge R_{t,s_j}$

Input: Term t, Term s, HashMap propositions

Output: PropositionalFormula $\bigwedge_{j=1}^{n} R_{t,s_j}$, where n = s.numArgs

```
PropositionalFormula result = new Conjunction
for Term $s_j$: s.args do
    result = result $\wedge$ $R_{t,s}(t,\ s_j,\ propositions)$
return result
```

5.1.2 Construction of Precedence

Algorithm: Java code of constructPrecedence(...)

Input: Set<Proposition> model, Set<String> functionSymbols

Output: NavigableSet<String>

```
TreeSet<String> prec = new TreeSet<>( new Comparator<String>() {
  @Override
  public int compare(String a, String b) {
    if a.equals(b) then
      return 0
    Proposition $P_{a,b}$ = generatePVariable(a,b)
    if model.contains($P_{a,b}$) then
      return 1
    else
      return -1
  }
})
prec.addAll(functionSymbols)
return prec
```

The procedure above shows how the precedence in form of an ordered set of the function symbols is constructed. It takes a set of P-variables that constitute the satisfying assignment for F (all P-variables that have to be $True$) and the set Σ as arguments. The presented code is very Java specific, so we elaborate to generalize. The *NavigableSet* interface and its implementation *TreeSet* represent an ordered set. The *TreeSet* takes a *Comparator* as its input, which specifies at what position elements are inserted. The *Comparator*'s method *compare(String a, String b)* returns 0 in case both symbols are equal, 1 in case $a > b$, -1 in case $b > a$. Thus, after inserting all function symbols the ordered set *prec* represents a precedence $>_p$.

5.2 Time Complexity

In order to determine an upper bound for the runtime of the given algorithm for generating $R_{t,s}$ we need to determine the maximum amount of recursive calls that can be made. The amount of recursive calls is directly dependent on the number of repetitions of the statements inside the for-loops. This holds true even though the first loop in the lpo2a-part of the code does not contain any calls because the second loop has the same amount of repetitions, so that both loops together effectively execute 2 times as many statements as recursive calls are made. The important detail is that this factor is constantly 2. Therefore we can regard all the other work that is done as constant and only look at the maximum amount of calls. Working with memoization prevents repeated calls with the same (sub)terms as input, every call that has been made must cache its result. This is not visible in the given pseudocode to keep the code clear, but only few alterations would have to be made to implement this memoization (e.g an extra *HashMap* argument which keeps the cached results).

Here we define the size of a term as the number of variables and function symbols within it. Then the maximum amount of recursive calls (including the original first call to the procedure) for a rule $t \to s \in R$ (t and s being the input to the procedure) is $size(t) * size(s)$.

5.3 Examples

The full implementation has been tested with various different systems. For example, given the Ackermann TRS (Table 5.1) as input, the program returns the precedence $A >_p S >_p Z$. Another example is the TRS 5a (Table 5.2) from Dershowitz [9]. The program finds the possible precedence $N >_p A >_p O$. As a last example we take the TRS 5c (Table 5.3) from Dershowitz, which yields an unsatisfiable formula F and therefore is not LPO-terminating. But it can be shown to be terminating using another proof method [9].

1. $A(0, y) \to S(y)$
2. $A(S(x), 0) \to A(x, S(0))$
3. $A(S(x), S(y)) \to A(x, A(S(x), y))$

Table 5.1: Ackermann

1.	$N(N(x)) \rightarrow x$
2.	$N(O(x,y)) \rightarrow A(N(x),N(y))$
3.	$N(A(x,y)) \rightarrow O(N(x),N(y))$
4.	$A(x,O(y,z)) \rightarrow O(A(x,y),A(x,z))$
5.	$A(O(y,z),x) \rightarrow O(A(y,x),A(z,x))$

Table 5.2: Example 5a from Dershowitz

1.	$N(N(x)) \rightarrow x$
2.	$N(O(x,y)) \rightarrow A(N(N(N(x))),N(N(N(y))))$
3.	$N(A(x,y)) \rightarrow O(N(N(N(x))),N(N(N(y))))$

Table 5.3: Example 5c from Dershowitz

Here we note again that the program only finds one possible precedence, a total order on the set Σ. In many cases a partial order would be sufficient as a precedence $>_p$ that can be lifted to $>_{lpo}$, some function symbols may be positioned anywhere in a total order. Consequently in a case like this, there exist multiple possible total orders that are valid precedences.

Chapter 6

Conclusion

Transforming the decision problem of LPO termination to SAT is an easy way of algorithmically solving the question whether any TRS is LPO-terminating, and thus terminating. The LPO definition perfectly translates to a recursive procedure which can construct a propositional logic formula F encapsulating the requirements for a LPO and a necessary precedence $>_p$. Checking if these requirements can be fulfilled then becomes the question of satisfiability of the constructed formula F. Using a SAT-solver to determine one or all satisfying assignment(s) we can construct one or all possible precedences $>_p$, which can be lifted to LPO. In case F is unsatisfiable we can conclude the given TRS is not LPO-terminating, so no conclusion regarding termination can be made.

Bibliography

[1] N. Dershowitz, “Termination of rewriting”, *Journal of Symbolic Computation*, vol. 3, no. 1-2, pp. 69–115, 1987.

[2] S. Kamin, “Two generalizations of the recursive path ordering”, *Unpublished manuscript*, 1980.

[3] A. Biere, M. Heule, and H. van Maaren, *Handbook of Satisfiability.* IOS press, 2009, vol. 185.

[4] M. Bezem, J. W. Klop, and R. de Vrijer, *Term rewriting systems.* Cambridge University Press, 2003, vol. 55.

[5] F. Baader and T. Nipkow, *Term rewriting and all that.* Cambridge University Press, 1999.

[6] W. Buchholz, “Proof-theoretic analysis of termination proofs”, *APAL*, vol. 75, pp. 57–65, 1994.

[7] D. Sheridan, “The optimality of a fast cnf conversion and its use with sat”, *SAT*, vol. 2, 2004.

[8] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems.* Cambridge university press, 2004, ch. 1.5.2, pp. 58–65.

[9] N. Dershowitz, “Orderings for term-rewriting systems”, *Theoretical Computer Science*, vol. 17, no. 3, pp. 279–301, 1982.

Appendix

Implementation

Code: https://github.com/jannes-t/lpo-checker

Used libraries:

1. tweety: http://tweetyproject.org/lib/index.html
2. sat4j: http://www.sat4j.org/
3. commons-cli: https://commons.apache.org/proper/commons-cli/