



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



*Official Master's Degree in Software
Engineering, Formal Methods and
Information Systems*

Universitat Politècnica de València

Departamento de Sistemas Informáticos
y Computación

Term Rewriting Systems .Net
Framework

Author:
Arnal Julián, Salvador

Supervisor:
Lucas Alba, Salvador

Academic Year 2012/2013

To Loanny, the reason
I smile each day since we met each other.

ABSTRACT

This thesis presents the implementation of a framework to work with term rewriting systems. Term rewriting systems provide a complete computational model which is very close to functional programming. Its formulation is equational logic and it can also be used to reason about program properties, not only of functional programs but also of programs written in other programming languages.

Our framework has been implemented in a tool called TRS.Tool (built on .Net and available at <http://TRS.JarCode.Net>) that covers three main objectives:

1. Determine the basic properties of a given term rewriting system (signature, set of rules conservativeness, linearity, etc. . .).
2. Calculate the critical pairs of the term rewriting system and determine its orthogonality.
3. Apply the rewriting rules over a given term.

The tool is available as a web based application created using this framework.

RESUMEN

Esta tesis presenta la implementación de un marco para trabajar con sistemas de reescritura de términos. Los sistemas de reescritura de términos proporcionan un modelo de cómputo completo que es muy cercano a la programación funcional. Su formulación es la de la lógica ecuacional y puede ser utilizado también para razonar sobre las propiedades de los programas, no solo de los programas funcionales, sino también de programas escritos en otros lenguajes de programación.

Nuestro marco que ha sido implementado en una herramienta llamada TRS.Tool (creada en .Net y disponible en <http://TRS.JarCode.Net>) que cubre tres objetivos principales:

1. Determinar las propiedades básicas de un sistema de reescritura de términos dado (signatura, conjunto de reglas, conservativo, lineal, etc...).
2. Calcular los pares críticos del sistema de reescritura de términos y determinar su ortogonalidad.
3. Aplicar las reglas de reescritura sobre un término dado.

La herramienta está disponible como una aplicación web creada utilizando este marco.

ACKNOWLEDGMENTS

Writing this master thesis would have been impossible without the help and support of the kind people around me. Here I can give particular mention to some of them only. It is a difficult task to acknowledge all people who influenced my life and contributed, in different ways, to this thesis. I am very grateful to all my friends and colleagues. However, I would like to add some personal words.

First and foremost I offer my sincerest gratitude to my supervisor Prof. Salvador Lucas, I appreciate his vast knowledge and skill in many areas. Without his assistance and guidance this thesis wouldn't be possible.

A very special thanks goes out to my family, my brother and my parents, for their support during my entire life and, in particular, I must acknowledge Loanny, my friend, my love, my partner, my life: you came into my life recently but without your love, kindness and encouragement I would not have finished this master.

I would like to acknowledge my friends for their moral support and motivation, which drives me to give my best. Alejandro, Ana, Carolina, Elena, Jorge, Luis, Mario, Max, Pedro, Pilar, Rodrigo, Silvia, . . . the list is endless.

I know most of you will never understand a word of this thesis, but you all made it possible.

Thanks to one and all.

Boro Arnal

CONTENTS

Abstract	II
Resumen	III
Acknowledgments	IV
Contents	V
List of Figures	VIII
List of Tables	XII
List of Examples	XV
1 Introduction	1
1.1 Objectives	5
1.2 Plan of the Thesis	5
2 Preliminaries	7

2.1	Basic concepts	7
2.1.1	Signature, variables and terms	7
2.1.2	Substitutions, matching and unification	11
2.1.3	Rewrite Rules and Term Rewriting Systems	12
2.2	Confluence	15
2.2.1	Local confluence and Confluence	15
2.2.2	Critical pairs	16
2.2.3	Orthogonality	18
2.2.4	Decidability	20
3	Framework Model	22
3.1	Term Rewriting System Module	24
3.1.1	Symbol Class	26
3.1.2	Term Class	28
3.1.3	Rule Class	29
3.1.4	TermRewritingSystem Class	31
3.2	Critical Pair Module	35
3.2.1	Substitution Class	35
3.2.2	Position Class	36
3.2.3	CriticalPair Class	37
3.3	Execution Module	38
3.3.1	Execution Tree Class	38
3.3.2	ExecutionNode Class	40
4	Implementation	47
4.1	Unification	47
4.2	Matching	51

4.3	Confluence decision	54
5	TRS.Tool	60
5.1	TPDB Format	61
5.2	Use Example	63
5.2.1	Input Data	64
5.2.2	Terms	66
5.2.3	Matching and unification	68
5.2.4	Rules	69
5.2.5	Critical Pairs	70
5.2.6	Term Execution	75
6	Testing the framework	84
7	Conclusions	89
	Bibliography	91

LIST OF FIGURES

1	Term tree structure	11
2	Local confluence and Confluence	16
3	Critical Pair graph	18
4	Critical Pair convergence graph	20
5	Term Rewriting System Module Diagram	23
6	Critical Pair Module Diagram	23
7	Execution Module Diagram	24
8	Term Rewriting System Module Class Diagram	25
9	Symbol Class	26
10	Symbol inheritance structure	27
11	Term Class	28
12	Rule Class	30
13	TermRewritingSystem Class	31
14	Term Rewriting System in memory representation	34
15	Critical Pair Module Class Diagram	35

16	Substitution Class	36
17	Position Class	36
18	CriticalPair Class	37
19	Execution Module Diagram	38
20	ExecutionTree Class	39
21	ExecutionNode Class	41
22	ExecutionTree Init	42
23	ExecutionTree Step 1	42
24	ExecutionTree Step 2	43
25	ExecutionTree Step 3	43
26	ExecutionTree Step 4	43
27	ExecutionTree Step 5	44
28	ExecutionTree Step 6	44
29	ExecutionTree Step 7	45
30	ExecutionTree Step 8	45
31	GetTrace data structure returned	46
32	Unification example 1, step 1	49
33	Unification example 1, step 2	49
34	Unification example 1, step 3	49
35	Unification example 2, step 1	50
36	Unification example 2, step 2	50
37	Unification example 2, step 3	50
38	Matching example 1, step 1	52
39	Matching example 1, step 2	52
40	Matching example 1, step 3	52

41	Matching example 2, step 1	53
42	Matching example 2, step 2	53
43	Matching example 2, step 3	54
44	Confluence algorithm flow chart	55
45	Non confluence example	57
46	Non termination example	58
47	Non confluence example	59
48	Tree without deforestation	59
49	Web tool interface	60
50	Tools on Rewriting	61
51	TRS input	63
52	Result tabs	63
53	Input data tab	64
54	Terms tab	67
55	Matching and unification	68
56	Matching table detail	69
57	Unification table detail	69
58	Rules tab	69
59	Critical pair tab	71
60	First critical pair calculation	72
61	Second critical pair calculation	72
62	First critical pair convergence	73
63	Second critical pair convergence	74
64	Term execution tab	75
65	Term execution root node	76

66	Term execution root node expansion	76
67	Term execution node tool tip	76
68	Term execution with deforested node	77
69	Term execution canonical node	77
70	Full term execution tree	77
71	Term execution tree	77
72	Normalizing form rewrite sequence	78
73	Example 1	79
74	Example 1 “optimized” version	80
75	Example 2	82
76	Example 2 “optimized” version	83
77	Test detailed report	86
78	5.trs test loaded	86
79	15.trs results	87
80	Test total values	87
81	Test dispersion graph	88

LIST OF TABLES

1	TRS.ToString()	65
2	Variables and signature	66
3	Variables and signature	66
4	Term t analysis	67
5	Term T_{3L} analysis	68
6	Rule R_1 analysis	70
7	Rule R_2 analysis	70
8	Critical pair calculation	71
9	Critical pair convergence	73
10	Critical pairs	74
11	TRS properties	75
12	Example 1 properties	79
13	Example 1 “optimized” version critical pairs	80
14	Example 1 “optimized” version properties	81
15	Example 2 properties	82

16	Example 2 “optimized” version critical pairs	83
17	Example 2 “optimized” version properties	83
18	Confluence property test	84
19	Locally Confluence property test	85
20	Terminating property test	85

LIST OF EXAMPLES

1	Confluence.	3
2	Termination.	4
3	Signature, variables and terms.	8
4	Matching and unification.	12
5	Rewrite Rules.	13
6	Critical Pairs.	16
7	Critical Pairs convergence.	19
8	Term Rewriting System Representation.	32
9	Execution Module.	41
10	Unification algorithm example traces.	48
11	Matching algorithm example traces.	51
12	Non confluence.	56
13	Non termination, first case.	57
14	Non termination, second case.	58
15	Confluence Analysis.	78

16 Termination Analysis.	81
----------------------------------	----

CHAPTER 1

INTRODUCTION

In Software Engineering and Programming, formal methods are understood as a collection of mathematical techniques for developing software and hardware systems. In software engineering they can be used to exploit the power of mathematical notations and proofs. Formal methods cover hundreds of methodologies, languages and notations that can be used to specify and verify properties.

Some formal methods rely on the use of *Term Rewriting Systems* (*TRSs*). There are various areas where the term rewriting systems play an important role in the study of computational procedures. Rewrite rules can be used to specify properties and requirements of programs. And using the theory of rewriting systems, we can check them.

We all have used rewriting systems, without knowing it, since school. When we simplify an arithmetic equation, we can do it as follows:

$$(3 + 4) \cdot (4 + 2) \rightarrow 7 \cdot (4 + 2) \rightarrow 7 \cdot 6 \rightarrow 42 \quad (1)$$

We can see this sequence as performing elementary simplification steps within some context. The elementary steps are the basic operations of addition and multiplication that we can easily represent by means of simple *rules* like $(0 + 1 \rightarrow 1, 0 + 2 \rightarrow 2, 0 + 3 \rightarrow 3, \dots, 0 \cdot 1 \rightarrow 0, 0 \cdot 2 \rightarrow 0, \dots)$. The contexts are arithmetic expressions with a ‘hole’ where the elementary step

is done. On the first step of the sequence (1) the context is $\square \cdot (4 + 2)$, where \square denotes the hole, and $3 + 4 \rightarrow 7$ is the elementary step. This means that \square is $3 + 4$ before and 7 after.

We are applying a sequence of elementary steps, till we obtain a term (i.e., 42) that can't be reduced anymore: this is the normal form.

Some important properties of this system are:

- **Non deterministic:** there are steps where we can choose different rules to be applied, for instance:

$$(3+4) \cdot (4+2) \rightarrow 7 \cdot (4+2) \rightarrow 7 \cdot 4 + 7 \cdot 2 \rightarrow 7 \cdot 4 + 14 \rightarrow 28 + 14 \rightarrow 42 \quad (2)$$

It's another valid rewrite sequence. In general, the rewriting process is *non-deterministic*.

- **Uniqueness of normal forms:** all possible different rewrite sequences will always produce the *same* normal form.
- **Terminating:** Every rewrite sequence can be extended to reach a normal form. This property is not easy to check. Note that we have excluded the usual commutative property of addition and product to make it terminating. Otherwise we could have a sequences like $0 + 1 \rightarrow 1 + 0 \rightarrow 0 + 1 \rightarrow \dots$
- **Confluent:** Divergent sequences starting from a given expression admit a join into a common expression.

Each of those properties can be used to investigate how to choose finite computation sequences (normalization), the absence of infinite computations (termination), the determinism of computations (confluence), the possibility of obtain completely defined information at the end of the computations (completely definedness), etc...

Two of these properties are specially important for what they represent: confluence and termination. Termination ensures that there is no infinite computation and some outcome will be obtained in finite time from an initial expression. Confluence guarantees that such an outcome will be unique if computations are *exhaustive* i.e., they stop yielding a normal form. Overall, we can think of such rewriting computations (to normal form) as being deterministic.

Decidability of these properties is a main problem, so most studies in the field restrict the attention to specific classes of term rewriting systems (left-linear, ground, growing, . . .) to develop decision methods.

Term rewriting systems can be used to model algorithms and then study their properties or check their correctness. In the following we show two simple examples that illustrates the importance of the properties of confluence and termination, and why having tools to automatically check them is useful.

Example 1: Confluence.

Consider the following term rewriting system¹ \mathcal{R} :

$$\text{add}(0, x) \rightarrow x \tag{3}$$

$$\text{add}(s(x), y) \rightarrow s(\text{add}(x, y)) \tag{4}$$

$$\text{prod}(0, x) \rightarrow 0 \tag{5}$$

$$\text{prod}(s(x), y) \rightarrow \text{add}(y, \text{prod}(x, y)) \tag{6}$$

$$\text{fact}(0) \rightarrow s(0) \tag{7}$$

$$\text{fact}(s(x)) \rightarrow \text{prod}(s(x), \text{fact}(x)) \tag{8}$$

It implements the computation of the factorial of a number. The natural numbers are represented in Peano notation, i.e. with a constant symbol 0 and a function symbol s (called *successor*), we write 0 for 0, $s(0)$ for 1, $s(s(0))$ for 2, . . ., and so on. In general $n \in \mathbb{N}$ is represented as the n -th application of s to 0 (written $s^n(0)$).

This rewrite system is terminating and confluent, so we will always get a result (a normal form) from the evaluation of an expression like $\text{fact}(s^n(0))$ and this result will always be the same (namely, $s^{n!}(0)$, i.e., the Peano representation of $n!$), disregarding of the particular choice of reduction sequence.

We can try to improve the algorithm by adding a new rule to try to speed up the execution by reducing the number of steps that are needed to obtain the solution²:

$$\text{prod}(s(0), y) \rightarrow y \tag{9}$$

This new rule represents the property $1 \cdot y = y$. If we study the properties

¹You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-1>

²You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-2>

of the new system, now it is not confluent. This means that we may get different answers depending on the reduction sequence.

Example 2: Termination.

Consider now the following term rewriting system³ \mathcal{R} :

$$\text{add}(0, x) \rightarrow x \tag{10}$$

$$\text{add}(s(x), y) \rightarrow s(\text{add}(x, y)) \tag{11}$$

$$\text{prod}(0, x) \rightarrow 0 \tag{12}$$

$$\text{prod}(s(x), y) \rightarrow \text{add}(y, \text{prod}(x, y)) \tag{13}$$

that implements the addition and multiplication operations. We can prove it terminating and confluent. We can try to improve it adding by a new rule⁴:

$$\text{add}(x, x) \rightarrow \text{prod}(s(s(0)), x) \tag{14}$$

Which represents the fact $x + x = 2 \cdot x$. What we are trying to do with this new rule is to reduce the number of rewriting steps by transforming the addition of a number to itself, into the multiplication of that number by 2.

Unfortunately, again we have obtained a worse system, because now it is not terminating nor confluent.

As a conclusion of these examples, we can say that when designing or optimizing rewrite systems, we greatly benefit from having appropriate analysis tools available.

Our TRS.Tool, whose design and use is described in this document, is able to prove that the TRS \mathcal{R} in Example 1 is confluent before adding the “optimizing rule” (9). Furthermore, TRS.Tool can also prove that the extended TRS \mathcal{R}' which is obtained after adding the rule (9) to \mathcal{R} fails to be confluent.

With regard to the TRS \mathcal{R} in Example 2, termination tools like MUTERM⁵ can prove termination of \mathcal{R} before adding the rule (14). And tools like AProVE⁶ are able to *disprove* termination of the extended TRS. Furthermore, TRS.Tool can be used to show not only that \mathcal{R} is confluent, but also that \mathcal{R}' is *neither* confluent *nor* terminating.

³You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-3>

⁴You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-4>

⁵<http://zenon.dsic.upv.es/muterm/>

⁶<http://aprove.informatik.rwth-aachen.de/>

1.1 Objectives

The main goal of this thesis is the implementation of a framework that enables the user (programmer, software developer, or interested searcher) to work with term rewriting systems.

This framework provides for three main functionalities:

1. Determine the basic structure of a given term rewriting system: signature, arity functions, set of rules and syntactic properties like conservativeness, linearity, etc
2. Compute the critical pairs of the term rewriting system and determine its orthogonality.
3. Apply the rewriting rules over a given term.

Because of my personal experience with programming languages I decided to implement it using C#.Net as it provides some benefits over other languages and technologies as: total object orientation, a complete class library, fully integrated IDE, . . . and the library created can be used directly in F#, a functional language integrated in the .Net framework.

We have also developed a web tool using this framework to show how does it work. The tool is suitable to be used in teaching the main concepts and notions of term rewriting as done in several courses of the Master of Software Engineering, Formal Methods and Information Systems like:

- 32574 / FSA / Fundamentos de Ingeniería del Software Automática.
- 32582 / DLP / Diseño de Lenguajes de Programación.
- 32576 / TEP / Terminación de Programas.
- 30184 / TSD / Tecnología Software Declarativa.
- 32575 / ATM / Métodos Ágiles y Tecnología Multiparadigma.

1.2 Plan of the Thesis

This thesis is organized in the following chapters:

-
- **Chapter 2:** we present a brief introduction to term rewriting theory. We present the basic concepts and the notation used.
 - **Chapter 3:** we explain the implementation model of our framework to work with term rewriting systems.
 - **Chapter 4:** we explain in detail the main algorithms implemented in the framework.
 - **Chapter 5:** we present the web tool created with the use of the framework.
 - **Chapter 6:** we conclude with the analysis of the framework created and the future work that can be done to made it more powerful.

CHAPTER 2

PRELIMINARIES

This is a brief introduction to the theory of term rewriting. We present the basic concepts and the notation used. More detailed information can be found in [1, 2, 6]

2.1 Basic concepts

2.1.1 Signature, variables and terms

A *signature* \mathcal{F} is a finite set of function symbols $\{f, g, h \dots\}$, each having a fixed *arity*, which establishes the number of its ‘arguments’, given by a mapping $ar : \mathcal{F} \rightarrow \mathbb{N}$. A function symbol a with arity of 0 ($ar(a) = 0$) is called a *constant*.

Let \mathcal{X} be a countable infinite set of *variables* $\{x, y, z, \dots\}$ where $\mathcal{F} \cap \mathcal{X} = \emptyset$. The set of *terms* over \mathcal{F} and \mathcal{X} , denoted by $\mathcal{T}(\mathcal{F}, \mathcal{X})$, is the least set satisfying:

1. All variables are terms: if $x \in \mathcal{X}$, then $x \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,
2. All constants are terms: if $\{a \in \mathcal{F} \wedge ar(a) = 0\}$, then $a \in \mathcal{T}(\mathcal{F}, \mathcal{X})$,

3. A function call is a term if all the parameters are terms: if $\{f \in \mathcal{F} \wedge ar(f) = k > 0 \wedge t_i \in \mathcal{T}(\mathcal{F}, \mathcal{X}) \forall i \in [1, k]\}$, then $f(t_1, \dots, t_k) \in \mathcal{T}(\mathcal{F}, \mathcal{X})$.

For any term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, the set of variables occurring in t is denoted as $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t) = \emptyset$, then we say that t is *ground*. The set of ground terms is denoted by $\mathcal{T}(\mathcal{F})$ (rather than $\mathcal{T}(\mathcal{F}, \emptyset)$).

A term is said to be *linear* if it has no multiple occurrences of a single variable.

The symbol labeling the *root* of t is denoted as $Root(t)$:

1. The root of a variable x is x : if $x \in \mathcal{X}$, then $Root(x) = x$,
2. The root of a constant a is a : if $\{a \in \mathcal{F} \wedge ar(a) = 0\}$, then $Root(a) = a$,
3. The root of a term $f(t_1, \dots, t_k)$ is f : $Root(f(t_1, \dots, t_k)) = f$.

Positions are represented by a (possibly empty) sequence of natural numbers. The empty sequence Λ denotes the root position of a term. The length of a position p is $|p|$.

$$p \in \mathcal{P}os = \{\Lambda\} \cup \{i.q \mid i \in \mathbb{N}_{>0} \wedge q \in \mathcal{P}os\} \quad (15)$$

$\mathcal{P}os(t)$ denotes the set of positions in t , and refer to sub-trees in the syntactic (tree) structure of the terms. We write $t|_p$ to denote the sub-term of t at position p . The depth of sub-term $s = t|_p$ is the length $|p|$ of p .

$$\mathcal{P}os(t) = \begin{cases} \{\Lambda\} & \text{if } t \in \mathcal{X} \\ \{\Lambda\} \cup \bigcup_{1 \leq i \leq k} i.\mathcal{P}os(t_i) & \text{if } t = f(t_1, \dots, t_k) \end{cases} \quad (16a)$$

$$\mathcal{P}os(t) = \begin{cases} \{\Lambda\} & \text{if } t \in \mathcal{X} \\ \{\Lambda\} \cup \bigcup_{1 \leq i \leq k} i.\mathcal{P}os(t_i) & \text{if } t = f(t_1, \dots, t_k) \end{cases} \quad (16b)$$

Positions of non-variable symbols in t are denoted as $\mathcal{P}os_{\mathcal{F}}(t)$, and $\mathcal{P}os_{\mathcal{X}}(t)$ are the positions of variables.

$$\mathcal{P}os_{\mathcal{F}}(t) = \{p \in \mathcal{P}os(t) \mid Root(t|_p) \in \mathcal{F}\} \quad (17)$$

$$\mathcal{P}os_{\mathcal{X}}(t) = \{p \in \mathcal{P}os(t) \mid Root(t|_p) \in \mathcal{X}\} \quad (18)$$

Example 3: Signature, variables and terms.

Let $\mathcal{F} = \{f, g, h, a, b\}$ be a set of symbols such that $ar(f) = 2, ar(g) = 1, ar(h) = 3, ar(a) = 0, ar(b) = 0$.

Let $\mathcal{X} = \{x, y, z\}$ be a set a variables.

The following terms do not belong to $\mathcal{T}(\mathcal{F}, \mathcal{X})$:

$$f(a) \tag{19}$$

$$g \tag{20}$$

$$x(b) \tag{21}$$

$$h(f(a, b), a) \tag{22}$$

$$j(a, b) \tag{23}$$

Functions f, g and h are used with a wrong arity on (19), (20) and (22) respectively. On (21) the symbol x is used as a function when it is a variable and on (23), the symbol j does not belongs to \mathcal{F} .

The following terms belong to $\mathcal{T}(\mathcal{F}, \mathcal{X})$:

$$t_1 = f(x, b) \tag{24}$$

$$t_2 = g(b) \tag{25}$$

$$t_3 = f(a, f(g(x), f(h(f(x, b), a, g(y)), g(b)))) \tag{26}$$

$$t_4 = h(f(a, b), g(x), a) \tag{27}$$

$$t_5 = h(f(a, z), g(x), a) \tag{28}$$

$$t_6 = h(f(x, b), a, g(y)) \tag{29}$$

$$t_7 = x \tag{30}$$

$$t_8 = b \tag{31}$$

$$t_9 = f(g(a), a) \tag{32}$$

Some properties of terms t_1, \dots, t_8 are:

$$\mathcal{V}ar(t_1) = \mathcal{V}ar(t_4) = \mathcal{V}ar(t_7) = \{x\} \quad (33)$$

$$\mathcal{V}ar(t_2) = \mathcal{V}ar(t_8)\mathcal{V}ar(t_9) = \emptyset \quad (34)$$

$$\mathcal{V}ar(t_3) = \mathcal{V}ar(t_6) = \{x, y\} \quad (35)$$

$$\mathcal{V}ar(t_5) = \{x, z\} \quad (36)$$

$$\{t_2, t_8, t_9\} \in \mathcal{T}(\mathcal{F}) \quad (37)$$

$$Root(t_1) = Root(t_3) = Root(t_9) = f \quad (38)$$

$$Root(t_2) = g \quad (39)$$

$$Root(t_4) = Root(t_5) = Root(t_6) = h \quad (40)$$

$$Root(t_7) = x \quad (41)$$

$$Root(t_8) = b \quad (42)$$

$$t_6|_1 = t_1 \quad (43)$$

$$t_3|_{2.2.2} = t_2 \quad (44)$$

$$t_3|_{2.2.1} = t_6 \quad (45)$$

$$t_1|_1 = t_7 \quad (46)$$

$$t_1|_2 = t_8 \quad (47)$$

$$\mathcal{P}os_{\mathcal{F}}(t_5) = \{\Lambda, 1, 2, 3, 1.1\} \quad (48)$$

$$\mathcal{P}os_{\mathcal{X}}(t_5) = \{1.2, 2.1\} \quad (49)$$

$$\mathcal{P}os_{\mathcal{F}}(t_2) = \{\Lambda, 1, 1.1\} \quad (50)$$

$$\mathcal{P}os_{\mathcal{X}}(t_2) = \emptyset \quad (51)$$

Terms $\{t_2, t_8\}$ are ground, terms $\{t_1, t_2, t_4, t_5, t_6, t_7, t_8\}$ are lineal.

Figure 1 shows the tree representation of term t_3 .

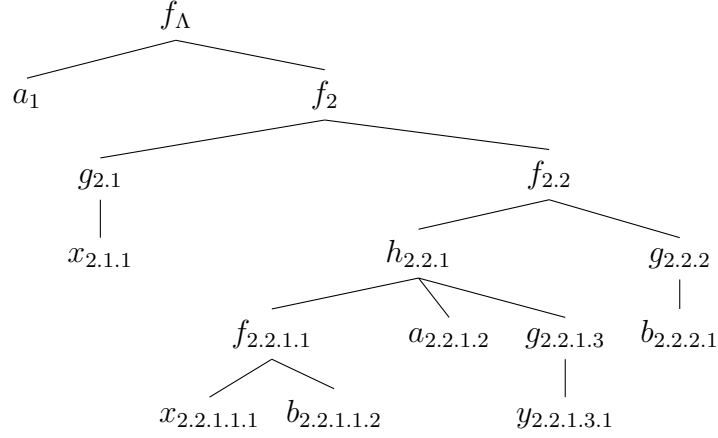


Figure 1: Term tree structure

2.1.2 Substitutions, matching and unification

We denote with $t[s]_p$ the term obtained from t by replacing the subterm at position p ($t|_p$) with the term s .

A *substitution* σ is a mapping from \mathcal{X} into $\mathcal{T}(\mathcal{F}, \mathcal{X})$ such that only a finite number of non-trivial bindings ($\sigma(x) \neq x$) are allowed.

$$\sigma : \mathcal{X} \rightarrow \mathcal{T}(\mathcal{F}, \mathcal{X}) \quad (52)$$

Therefore, we write substitutions as a set of the form $\sigma = \{x_1 \mapsto t_1, \dots, x_n \mapsto t_n\}$ denoting that for each $i \in \{1, n\}$, the variable x_i is mapped to term t_i . The domain of a substitution σ is denoted by $\mathcal{D}om(\sigma)$, and is the finite set:

$$\mathcal{D}om(\sigma) = \{x \in \mathcal{X} \mid \sigma(x) \neq x\} \quad (53)$$

The identity, or empty substitution, is denoted by ϵ (Note that $\mathcal{D}om(\epsilon) = \emptyset$):

$$\epsilon(x) = x \quad \forall x \in \mathcal{X} \quad (54)$$

Let $l, s, t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$:

- A substitution σ such that $\sigma(l) = t$ is called a *matcher* of l against t (we also say that l matches t with substitution σ , or that t is an instance of l).

- A substitution σ such that $\sigma(l) = \sigma(t)$ is called a unifier of s and t (we also say that s and t unify with substitution σ).
- A substitution σ such that $\sigma^{-1} \circ \sigma = \sigma \circ \sigma^{-1} = \epsilon \wedge \text{Dom}(\sigma) = \mathcal{X}$ is called a *renaming* substitution.
- If two terms s and t unify, then there is a unique (up to renaming) *most general unifier* (mgu) θ such that for every other unifier σ , there is a substitution τ such that $\tau \circ \theta = \sigma$

Example 4: Matching and unification.

Consider the terms t_1, \dots, t_8 in Example 3.

The substitution $\sigma = \{z \rightarrow b\}$ unifies t_4 and t_5 , because $\sigma(t_4) = \sigma(t_5)$. $\text{Dom}(\sigma) = \{z\}$. This substitution is also a matcher of t_5 against t_4 because $\sigma(t_5) = t_4$.

The substitution σ is the mgu of t_4 and t_5 . We can use $\sigma' = \{z \rightarrow b, x \rightarrow b\}$ to unify t_4 and t_5 , but it is not the mgu (the substitution τ such that $\tau \circ \sigma = \sigma'$ is $\tau = \{x \rightarrow b\}$). In this case $\text{Dom}(\sigma') = \{z, x\}$. The new substitution σ' is not a matcher of t_5 against t_4 because $\sigma'(t_5) \neq t_4$.

2.1.3 Rewrite Rules and Term Rewriting Systems

A *rewrite rule* is an ordered pair (l, r) , written $l \rightarrow r$, where l and r are terms over a signature \mathcal{F} and a set of variables \mathcal{X} such that l is not a variable, and all variables occurring in r already occur in l . Formally:

- $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{X})$
- $l \notin \mathcal{X}$
- $\text{Var}(r) \subseteq \text{Var}(l)$

We say that the *left-hand side* (*lhs*) of the rule is l , and the *right-hand side* (*rhs*) is r .

A rewrite rule $l \rightarrow r$, is:

- Left-linear if l is a linear term.

- Right-linear if r is a linear term.
- Linear if both l and r are linear terms.
- Collapsing if $r \in \mathcal{X}$. That is, if the right side is just a variable.
- Duplicating if there is a variable x with more occurrences in r than in l .
- Conservative if $\mathcal{V}ar(l) = \mathcal{V}ar(r)$. (All variables in the left-hand side, also occur in the right-hand side)
- Destructive if $\mathcal{V}ar(l) \supset \mathcal{V}ar(r)$. (Some variables in the left-hand side, are missing in the right-hand side)
- Right-ground if r is a ground term.
- Ground if both l and r are ground terms.

Given a signature \mathcal{F} , a *Term Rewriting System* (TRS) is a pair $\mathcal{R} = (\mathcal{F}, R)$ such that R is a set of rewrite rules over the signature \mathcal{F} .

An instance $\sigma(l)$ of the left-hand side l of a rule $l \rightarrow r$ is called a *redex* (reducible expression) of the rule.

A term t rewrites to a term u at position p with the rule $l \rightarrow r$ and the substitution σ , written $t \xrightarrow[p]{l \rightarrow r} u$, (or simply $t \rightarrow u$), if $s|_p = \sigma(l)$ and $s = t[\sigma(r)]_p$. Such a term t is called reducible. Irreducible terms are said to be in normal form.

A term rewriting system $\mathcal{R} = (\mathcal{F}, R)$ is called left-linear (right-linear, linear, conservative) if each rule $l \rightarrow r \in \mathcal{R}$ is left-linear (right-linear, linear, conservative). On the other hand, \mathcal{R} is called collapsing (duplicating) if at least one of the rules $l \rightarrow r \in \mathcal{R}$ is collapsing (duplicating).

A term rewriting system is called *looping* if it admits a rewrite sequence $t \rightarrow^* C[\sigma(t)]$ for some term t , some context $C[\]$, and some substitution σ

Example 5: Rewrite Rules.

The following rules are non-valid rewrite rules:

$$x \rightarrow f(a, b) \tag{55}$$

$$g(x) \rightarrow f(x, y) \tag{56}$$

In (55) the *lhs* is a variable, and in (56) the *rhs* contains variables than doesn't occur in the *lhs*.

Let's consider the Term Rewriting System formed by the following valid rewrite rules¹:

$$f(a, x) \rightarrow h(x, x, a) \quad (57)$$

$$g(x) \rightarrow f(a, x) \quad (58)$$

$$f(y, y) \rightarrow y \quad (59)$$

$$h(x, y, a) \rightarrow g(x) \quad (60)$$

$$f(g(x), a) \rightarrow g(g(x)) \quad (61)$$

$$b \rightarrow a \quad (62)$$

$$f(a, x) \rightarrow a \quad (63)$$

We can say:

- Rewriting rules (57), (58), (60), (61), (62) and (63) are left-linear.
- Rewriting rules (58), (59), (61), (62) and (63) are linear.
- Rewriting rules (58), (59), (60), (61), (62) and (63) are right-linear.
- Rewriting rule (59) is collapsing.
- Rewriting rule (57) is duplicating.
- Rewriting rules (57), (58), (59), (61) and (62) are conservative.
- Rewriting rules (60) and (63) are destructive.
- Rewriting rules (62) and (63) are right-ground.
- Rewriting rule (62) is left-ground.
- Rewriting rule (62) is ground.

If we take $t_9 = f(g(a), a)$ from Example 3, the next rules can be applied:

- Rule (58) at position 1 with $\sigma = \{x \rightarrow a\}$.
- Rule (61) at position Λ with $\sigma = \{x \rightarrow a\}$.

¹You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-5>

After applying them, we get:

- $f(f(a, a), a)$ and
- $g(g(a))$,

respectively.

2.2 Confluence

Confluence is the property of term rewriting systems ensuring that terms in a system can be rewritten in more than one way, to yield the same final result.

Termination is another fundamental property of rewrite systems which guarantee the existence of at least one normal form for any term. For terminating and confluent systems such normal form is unique.

Formally a term rewriting system is *terminating* if there is no infinite sequence $t = t_1 \rightarrow t_2 \rightarrow \dots$ for any term t .

In the following, we concentrate the attention in the confluence property. We formally define it and also introduce some simple methods to prove confluence of term rewriting systems.

2.2.1 Local confluence and Confluence

Let $\mathcal{R} = (\mathcal{F}, R)$ be a term rewriting system. We say that \mathcal{R} is:

- Locally confluent if, for all $s, t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, whenever $s \rightarrow t$ and $s \rightarrow t'$, there is $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $t \rightarrow^* u$ and $t' \rightarrow^* u$.
- Confluent if, for all $s, t, t' \in \mathcal{T}(\mathcal{F}, \mathcal{X})$, whenever $s \rightarrow^* t$ and $s \rightarrow^* t'$, there is $u \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that $t \rightarrow^* u$ and $t' \rightarrow^* u$.

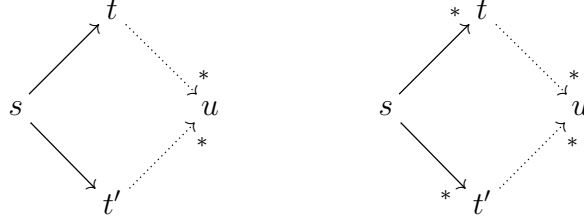


Figure 2: Local confluence and Confluence

2.2.2 Critical pairs

Let $l \rightarrow r$ and $l' \rightarrow r'$ be two rules of a term rewriting system and suppose these rules share no variables ($\mathcal{V}ar(l) \cap \mathcal{V}ar(l') = \emptyset$). If l_p is a sub-term of l at position p ($l_p = l|_p$) such that l_p is not a variable ($l_p \notin \mathcal{X}$), and l_p and l' unify with mgu σ ($\sigma(l_p) = \sigma(l')$), then the pair $\langle \sigma(l)[\sigma(r')]_p, \sigma(r) \rangle$ is called the *critical pair* and p is called critical or overlapping position of the pair.

A critical pair $\langle s, t \rangle$ is *convergent* if there is a term u such that $s \rightarrow^* u$ and $t \rightarrow^* u$.

Example 6: Critical Pairs.

The term rewriting system in the Example² 5, has four critical pairs:

1. Rule (61) with rule (58) at position 1.
2. Rule (57) with rule (59) at position Λ .
3. Rule (57) with rule (63) at position Λ .
4. Rule (63) with rule (59) at position Λ .

Let's study the first critical pair:

$$l \rightarrow r = (61) = f(g(x), a) \rightarrow g(g(x)) \quad (64)$$

$$l' \rightarrow r' = (58) = g(x') \rightarrow f(a, x') \quad (65)$$

²You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-5>

Notice that l and l' should not share any variable, because that, we must rename the variables in l' and r'

$$p = 1 \tag{66}$$

$$l_p = l|_1 = g(x) \tag{67}$$

$$\sigma = \{x' \rightarrow x\} \tag{68}$$

σ is the m.g.u. of l_p and l' ($\sigma(l_p) = \sigma(l') = g(x)$).

$$\sigma(l) = f(g(x), a) \tag{69}$$

$$\sigma(r') = f(a, x) \tag{70}$$

$$\sigma(r) = g(g(x)) \tag{71}$$

$$s = \sigma(l)[\sigma(r')]_p = f(f(a, x), a)[f(a, x)]_1 = f(f(a, x), a) \tag{72}$$

Now we have the two terms that form the critical pair: $\langle (72), (71) \rangle$, i.e.:

$$\langle f(f(a, x), a), g(g(x)) \rangle \tag{73}$$

The study of critical pairs is essential for the analysis of confluence of term rewriting systems. The existence of a critical pair implies a point of divergence in the rewriting computation, i.e. the same redex can be rewritten in different ways.

As we said before, a critical pair represents a divergence. For instance, the critical pair (73) is the result of the divergence that exist when we try to reduce the term $\sigma l = f(g(x), a)$: we have two choices: either using the rule (61) at position Λ or else use the rule (58) at position 1. If we try both rules, we will get $f(f(a, x), a)$ and $g(g(x))$ respectively, this is the critical pair.

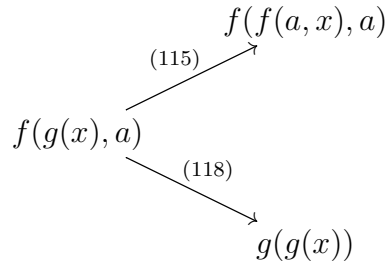


Figure 3: Critical Pair graph

2.2.3 Orthogonality

There are two kinds of critical pairs that are relevant for the analysis of local confluence:

- When the critical position is the root position ($p = \Lambda$) the critical pair is an *overlay*.
- When the two terms of the critical pair are equal, the critical pair is called *trivial*. Note that trivial critical pairs are *trivially* convergent

A critical pair with both properties is called a *trivial overlay*.

A suitable criterion to determine the confluence of a term rewriting system relies on the notions of orthogonality. A left-linear term rewriting system is:

- *Orthogonal* if it has no critical pair.
- *Almost orthogonal* if its critical pairs are trivial overlays.
- *Weakly orthogonal* if its critical pairs are trivial.

An Orthogonal term rewriting system is almost orthogonal, and almost orthogonal term rewriting systems are weakly orthogonal.

The interest of orthogonality for proving confluence is due to Huet and Lévy's theorem [4]:

$$\text{Weakly orthogonal term rewriting systems are confluent.} \quad (74)$$

If a term rewriting system has nontrivial critical pairs, we can study their convergence. According to Huet's theorem [3]:

A Term rewriting system is locally confluent if only if
all its critical pairs are convergent. (75)

If all critical pairs of a TRS \mathcal{R} are convergent, then \mathcal{R} is locally confluent. Obviously, confluence implies local confluence, but the opposite claim is not true.

Example 7: Critical Pairs convergence.

Consider again the critical pair (73) in Example 6.

This critical pair is not trivial. We must try to test it for convergence. If the term rewriting system is terminating we only need to apply rules until we get to the normal forms. Otherwise we must apply the rules carefully to avoid entering into a loop.

For the first term $f(f(a, x), a)$, we have this rewriting sequence:

$$f(f(a, x), a) \tag{76}$$

$$\xrightarrow[1]{f(a, x) \rightarrow a \quad \sigma = \epsilon} \tag{77}$$

$$f(a, a) \tag{78}$$

$$\xrightarrow[\Lambda]{f(y, y) \rightarrow y \quad \sigma = \{y \rightarrow a\}} \tag{79}$$

$$a \tag{80}$$

And for the second one, $g(g(x))$, the rewriting sequence is:

$$g(g(x)) \tag{81}$$

$$\xrightarrow[\Lambda]{g(x) \rightarrow f(a, x) \quad \sigma = \{x \rightarrow g(x)\}} \tag{82}$$

$$f(a, g(x)) \tag{83}$$

$$\xrightarrow[\Lambda]{f(a, x) \rightarrow a \quad \sigma = \{x \rightarrow g(x)\}} \tag{84}$$

$$a \tag{85}$$

So, the critical pair is convergent

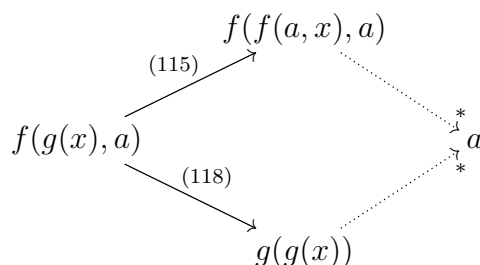


Figure 4: Critical Pair convergence graph

The rest of critical pairs are:

$$\langle a, h(a, a, a) \rangle \quad (86)$$

$$\langle a, h(x, x, a) \rangle \quad (87)$$

$$\langle a, a \rangle \quad (88)$$

Critical pair (88) is trivial, hence convergent. The other two critical pairs ((86) and (87)) are not trivial, but they are convergent.

Huet's Theorem ensures that the system is locally confluent because all critical pairs are convergent.

2.2.4 Decidability

The problem of checking confluence of a term rewriting system is undecidable, but it is decidable for some special cases.

The *Newman's lemma*, states that:

$$\begin{aligned} &\text{A terminating term rewriting system is confluent if} \\ &\text{it is locally confluent.} \end{aligned} \quad (89)$$

As a consequence of Huet's Theorem and Newman's Lemma, we have the following decidability result for terminating term rewriting systems [5]:

$$\begin{aligned} &\text{Confluence of terminating term rewriting systems with} \\ &\text{a finite number of rules is decidable.} \end{aligned} \quad (90)$$

Given a terminating and finite term rewriting system, as the number of rules is finite, the number of critical pairs will be finite too.

If not all the critical pairs are trivial, as the term rewriting system is terminating, we can reduce the terms to its canonical forms, in finite time, and determine whether the critical pairs are convergent or not by just checking the obtained normal forms for equality.

CHAPTER 3

FRAMEWORK MODEL

The Framework is divided into three main submodules:

- **Term Rewriting System Module:** This module is the responsible to store the data structure representing a term rewriting system.
- **Critical Pair Module:** Contains the necessary classes to calculate and manage the critical pairs of a term rewriting system.
- **Execution Module:** It is used to trace the execution of a term rewriting system over a term.

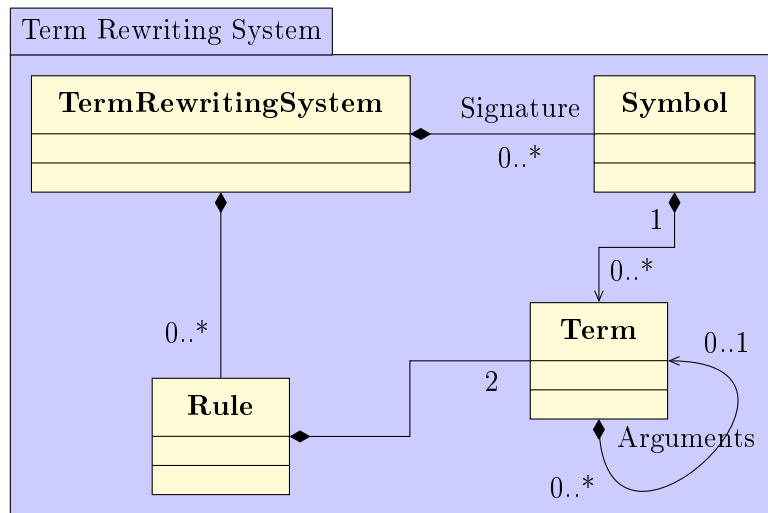


Figure 5: Term Rewriting System Module Diagram

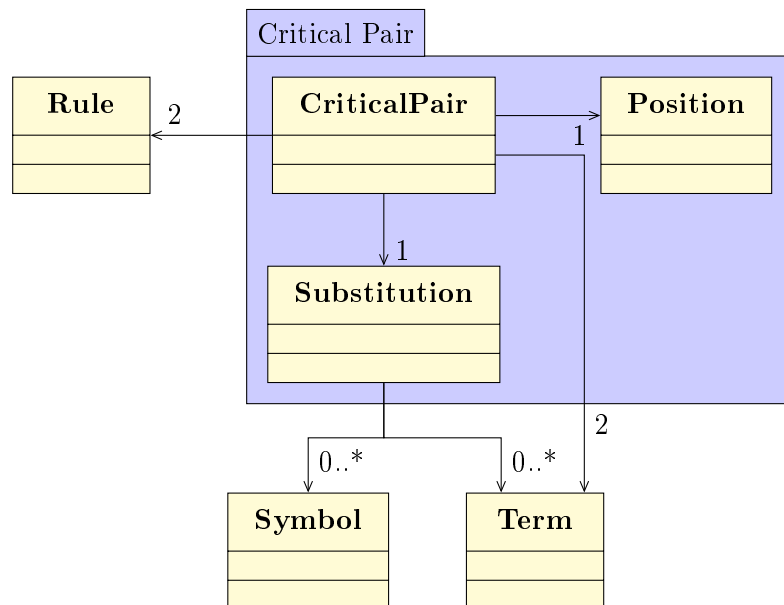


Figure 6: Critical Pair Module Diagram

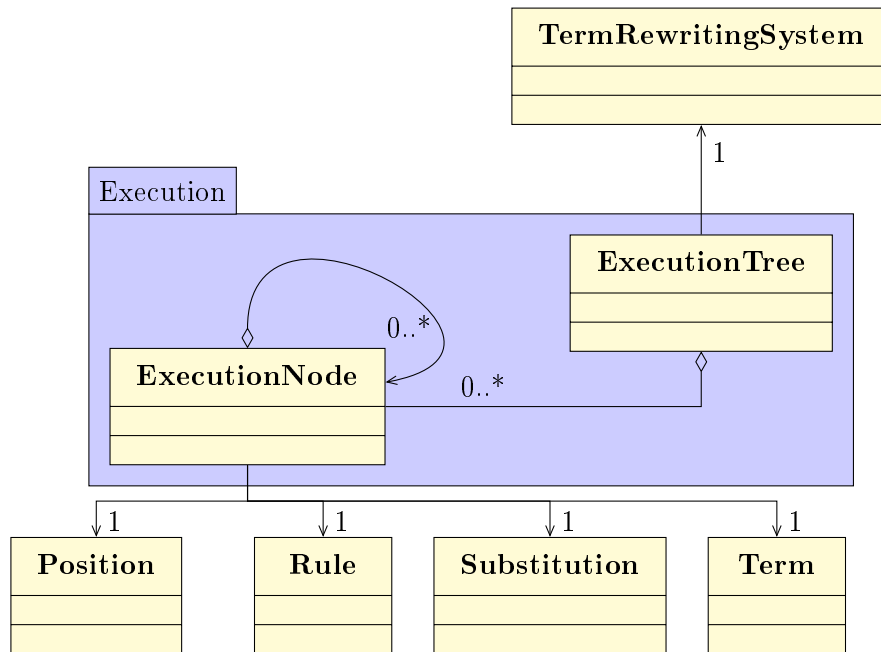


Figure 7: Execution Module Diagram

Additionally there is an auxiliary class, `Parser`, which is responsible to parse the `.trs` files and build a `TermRewritingSystem` object.

3.1 Term Rewriting System Module

The Term Rewriting System Module is the core of the framework. It represents the data structure that stores a term rewriting system and allows access to its properties.

The data structure which has been chosen to represent a `Term` is based on the syntactic tree. Each `Term` object represents a node in that tree, and has a reference to `Symbol` on it. This representation is show more clear in figure 14. Each symbol on the term rewriting system is represented by one (and only one) `Symbol` object, and it stores a list of all the nodes on the trees where it is used.

The reason for proceeding in this way was that this makes the code for the implementation of the matching, unification and substitution algorithms clearer. The implementations has been done in the recursive mode, and the cost for all of them is linear (see Chapter 4).

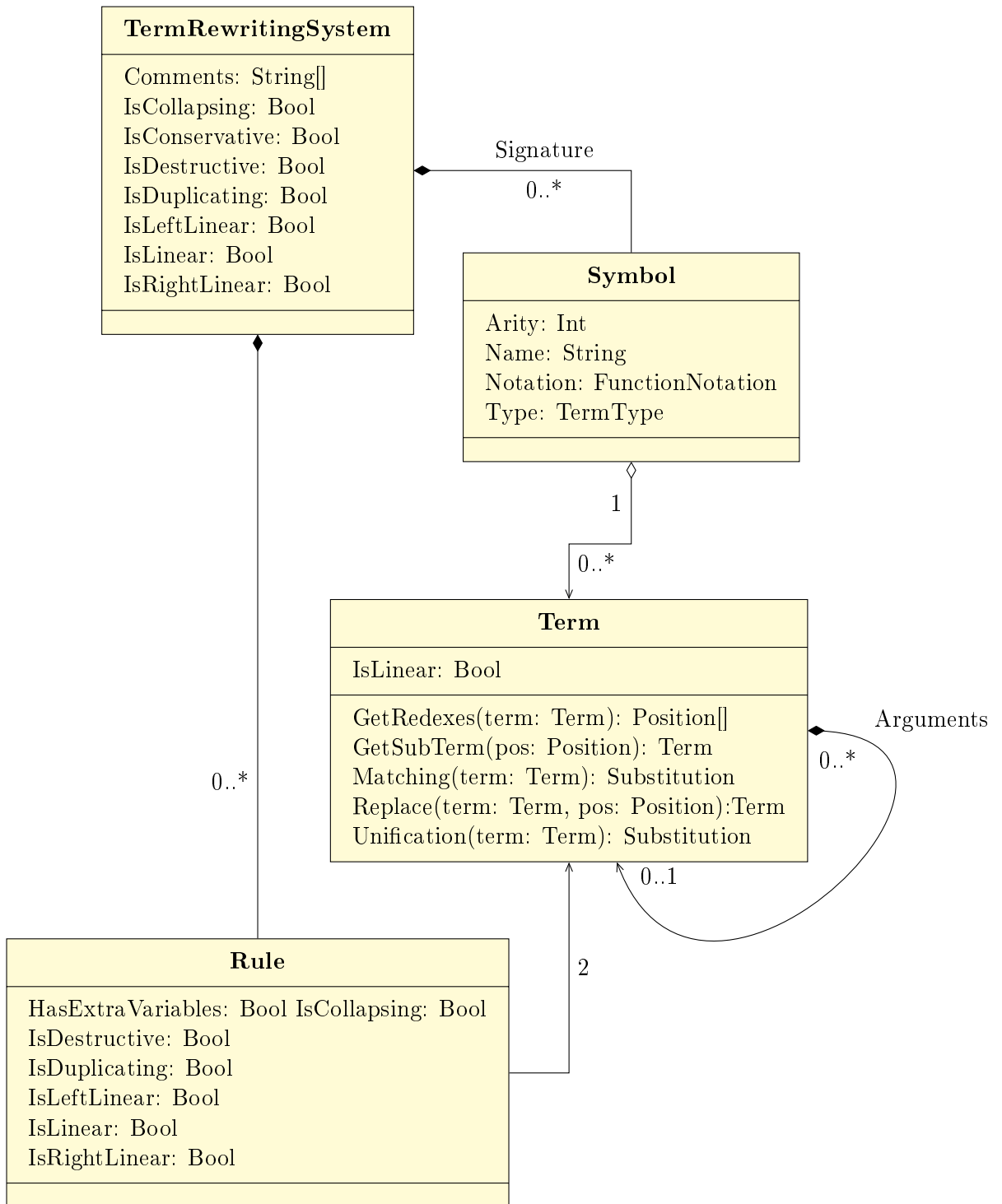


Figure 8: Term Rewriting System Module Class Diagram

3.1.1 Symbol Class

The `Symbol` Class is used to store the characteristics of each symbol in the term rewriting system signature. A `Symbol` could be a “*variable*”, a “*function*” or a “*constant*”. A `Symbol` object represents only one kind of them, and all the uses of it on the terms are stored in a array of instantiations (`.Instances[0]`, `.Instances[1]`, `.Instances[2]`, ...)

Symbol
Arity: Int Instances: List<Term> Name: String Notation: FunctionNotation Type: TermType
Init(type: TermType, name: String, arity: Int, notation: FunctionNotation) Instantiate(Parent: Term): Term

Figure 9: Symbol Class

The data types of `Type` and `Notation` (`TermType` and `FunctionNotation`) are enumerations, whose possible values are `[Function, Variable, Constant]` and `[Null, Prefix, Infix, Postfix]` respectively.

The `Notation` attribute represents the way a function is printed out, and on non function symbols must be `Null`.

The `Arity` attribute is used on function symbols only. For all other symbols must be 0. As its theoretical counterpart, it represents number of ‘arguments’, this is, the number of child subterms it must have.

The `Init` function is used by the constructor method and checks the class constraints, i.e.:

- If the symbol is a variable or a constant, then the `Arity` must be 0 and the `Notation` `Null`.
- If the symbol is a function the `Arity` must be greater than 0 and `Notation` distinct than `Null`.
- If the symbol is a function with `Infix Notation` the `Arity` must be 2.

- If the symbol is a function with `Postfix Notation` the `Arity` must be 1.

The `Instantiate` methods create an new `Term` object and adds it on the instances list.

In the design phase, a first version for this class was representing more accurately the concepts ‘Function’, ‘Variable’ and ‘Constant’ from the theory, because they were separated classes for each one, with the appropriate inheritance structure.

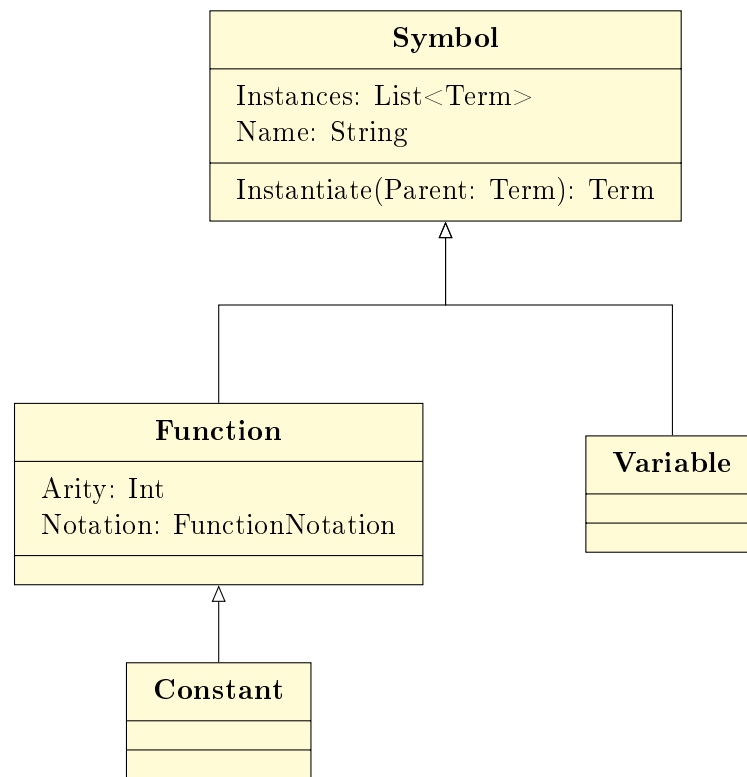


Figure 10: Symbol inheritance structure

With this implementation the `Type` property and the `Init` method is not longer necessary, because each class has the restrictions about its own attributes.

The reason for not choosing this structure is making the code simpler. Since the term object must work with the super class `Symbol`, it will be necessary to check the real class of the object each time and do the appropriate casting to access its properties and methods. So we think will be more easy

to have only one class with a property that identifies its type and control by code the restrictions in the rest of properties.

3.1.2 Term Class

Each `Term` object represents an instantiation of a symbol, so it has a reference to its `Symbol` object. The `Term` object acts as a node into the tree representation of the term, and each one is the root symbol of the subterm it represents. If the `Term` is not a root of the main term, then it has a reference to its `Parent`, and if it is a function, then it also has references to the `Arguments` sub-terms.

The attribute `IsLinear` represents the “*Linear*” property of the term, this is if it has no multiple occurrences of a single variable.

To speed up the execution time of some algorithms, each `Term` object keeps a dictionary to track the number of instantiations each `Symbol` has into the term represented by the object. This corresponds to the `Symbols` attribute. Each time the term changes (by adding, removing or replacing subterms at any depth in the tree structure) this dictionary updates its values.

Term
Arguments: List<Term> IsLinear: Bool Parent: Term Symbol: Symbol Symbols: SortedList<Symbol, Int>
Clone(term: Parent): Term GetSubTerm(pos: Position): Term Matching(term: Term): Substitution Replace(pos: Position, term: Term): Term Split(pos: Position): String[] ToString(): String Unification(term: Term): Substitution

Figure 11: Term Class

The public methods of the class are:

- **Clone:** It clones the current Term and returns a duplicate.
- **GetSubTerm:** Returns the subterm at an specified position.
- **Matching/Unification:** Calculates the substitution which is necessary to match/unify the given term. If no matching/unification is possible, it returns `null`
- **Replace:** Replaces the subterm at the specified position with the one which is passed as an argument.
- **Split:** It takes a position as an argument and splits the textual representation of the term into three parts: the part before the position, the subterm at the position and the part after the position. This method is useful to print out the Term when you need to mark a subterm inside (e.g. by underlining it).
- **ToString:** Converts the Term into its text representation.

The storage in memory of a term corresponds to the syntactic tree structure of the term. Each node in the tree is a Term Object. The leaves on the tree are Term Objects whose Symbol value is of arity 0. The number of branches each node has is determined by the arity of the symbol.

3.1.3 Rule Class

Each rule in a term rewriting system is represented by a Rule object. It has references to two Term objects: the left-hand side term and the right-hand side.

The object also has properties representing the main Rule attributes (Collapsing, Destructive, Duplicating, ...), based on the attributes of left and right terms.

Rule
HasExtraVariables: Bool IsCollapsing: Bool IsConservative: Bool IsDestructive: Bool IsDuplicating: Bool IsLeftLinear: Bool IsLinear: Bool IsRightLinear: Bool Left: Term Right: Term
Clone(term: Parent): Term Apply(term: Term, pos: Position, sub: Substitution): Term GetCriticalPairs(r: Rule): List<CriticalPair> GetRedexes(t: Term): List<Pair<Position, Substitution>> ToString(): String

Figure 12: Rule Class

The `Rule` class has three important methods:

- **Apply**: It takes a term, a position, and a substitution as parameters. It returns the result of applying the current rule to the specified term at the position with the given substitution.
- **GetCriticalPairs**: It takes another (or the same) rule as parameter, and returns the list of critical pairs for both rules.
- **GetRedexes**: Given a `Term` returns the list of redexes of the rule for that term. A redex is a pair formed by a `Position` of the `Term` and a `Substitution`.

It also has some methods that are implemented in all classes of the framework: `Clone` and `ToString`. These methods works in a similar way in all classes, as explained for the `Term` class.

3.1.4 TermRewritingSystem Class

The `TermRewritingSystem` Class is the main class of the framework. It allows the creation of a term rewriting system and provides methods to check its properties.

TermRewritingSystem
Comments: List<String> Constants: SortedList<String, Symbol> Functions: SortedList<String, Symbol> IsCollapsing: Bool IsConservative: Bool IsDestructive: Bool IsDuplicating: Bool IsLeftLinear: Bool IsLinear: Bool IsRightLinear: Bool Rules: List<Rule> Terms: List<Term> Variables: SortedList<String, Symbol>
Clone(): TermRewritingSystem GetCriticalPairs(): List<CriticalPair> Instantiate(Parent: Term): Term NewRule(left: Term, right: Term): Rule NewSymbol(type:TermType,name:String,notation: FunctionNotation,arity:int):Symbol Parse(String term): Term ToString(): String

Figure 13: TermRewritingSystem Class

Each `TermRewritingSystem` object contains four collections:

- **List of Rules:** An array of the rules that form the term rewriting system. Each one is accessible by its position in the array (`.Rules[0]`, `.Rules[1]`, `.Rules[2]`, ...).
- **List of Variables:** A dictionary storing the `Symbol` objects representing each variable used in the term rewriting system. The variables are accessible by the string representing its name (`.Variables["x"]`, `.Variables["y"]`, `.Variables["z"]`, ...)
- **List of Functions:** A dictionary storing the `Symbol` objects representing each function used in the term rewriting system. The functions are accessible by the string representing its name (`.Functions["f"]`, `.Functions["g"]`, `.Functions["h"]`, ...)

- **List of Constants:** A dictionary storing the Symbol objects representing each constant used in the term rewriting system. The constants are accessible by the string representing its name (`.Constants["a"]`, `.Constants["b"]`, `.Constants["c"]`, ...)

The signature of the term rewriting system is represented by the union of the collections `Constants` and `Functions`.

The object also has methods to inform about the main term rewriting system (`Collapsing`, `Conservative`, `Destructive`, ...).

The methods `Instantiate`, `NewRule` and `NewSymbol` are used to create new objects of type `Term`, `Rule` or `Symbol` in the current `TermRewritingSystem` object.

Example 8: Term Rewriting System Representation.

This example illustrates the source code which is needed to build a term rewriting system and the representation in memory of the created structure.

Consider the following term rewriting system with only one rule:

$$f(a, x) \rightarrow f(g(a), b) \tag{91}$$

```
// Create the TRS
TermRewritingSystem trs = new TermRewritingSystem();

// Create the Symbols
Symbol s_a = new Symbol(TermType.Constant, "a");
Symbol s_b = new Symbol(TermType.Constant, "b");
Symbol s_f = new Symbol(TermType.Function, "f", FunctionNotation.Prefix, 2);
Symbol s_g = new Symbol(TermType.Function, "g", FunctionNotation.Prefix, 1);
Symbol s_x = new Symbol(TermType.Variable, "x");

// Add Symbols to the TRS
trs.Constants.Add(s_a);
trs.Constants.Add(s_b);
trs.Functions.Add(s_f);
trs.Functions.Add(s_g);
trs.Variables.Add(s_x);

// Create the left term
Term t_f1 = new Term(s_f);
s_f.Instances.Add(t_f1); // f(?, ?)

Term t_a1 = new Term(s_a);
s_a.Instances.Add(t_a1);
t_f1.Arguments.Add(t_a1);
t_a1.Parent = t_f1; // f(a, ?)

Term t_x1 = new Term(s_x);
```



```

s_x.Instances.Add(t_x1);
t_f1.Arguments.Add(t_x1);
t_x1.Parent = t_f1;           // f(a, x)

// Create the right term
Term t_f2 = new Term(s_f);
s_f.Instances.Add(t_f2);     // f(?, ?)

Term t_g1 = new Term(s_g);
s_g.Instances.Add(t_g1);
t_f2.Arguments.Add(t_g1);
t_g1.Parent = t_f2;         // f(g(?), ?)

Term t_a2 = new Term(s_a);
s_a.Instances.Add(t_a2);
t_g1.Arguments.Add(t_a2);
t_a2.Parent = t_g1;         // f(g(a), ?)

Term t_b1 = new Term(s_b);
s_b.Instances.Add(t_b1);
t_f2.Arguments.Add(t_b1);
t_b1.Parent = t_f2;         // f(g(a), b)

// Create the rule
Rule r1 = new Rule();        // ? -> ?
r1.Left = t_f1;              // f(a, x) -> ?
r1.Right = t_f2;             // f(a, x) -> f(g(a), b)

// Add the rule to the TRS
trs.Rules.Add(r1);

```

This code includes all the steps which are necessary to create the term rewriting system, but in the framework there are methods that automatize most of the work. The following code is a more compact but equivalent version of the previous one.

```

// Create the TRS
TermRewritingSystem trs = new TermRewritingSystem();

// Create the symbols
trs.NewSymbol(TermType.Constant, "a")
trs.NewSymbol(TermType.Constant, "b")
trs.NewSymbol(TermType.Function, "f", 2)
trs.NewSymbol(TermType.Function, "g")
trs.NewSymbol(TermType.Variable, "x")

// Create the left Term
Term t_f1 = trs.Instantiate("f"); // f(?, ?)
trs.Instantiate("a", t_f1);       // f(a, ?)
trs.Instantiate("x", t_f1);       // f(a, x)

// Create the right Term
Term t_f2 = trs.Instantiate("f"); // f( ? , ?)
Term t_g1 = trs.Instantiate("g", t_f2); // f(g(?), ?)
trs.Instantiate("a", t_g1);       // f(g(a), ?)

```

```

trs.Instantiate("b", t_f2);           // f(g(a), b)

// Create the Rule
trs.NewRule(t_f1, t_f2);           // f(a, x) -> f(g(a), b)

```

The long version of the code has been included to clarify how the objects and links between them are created, but the short version must be used because it builds the correct structure automatically. In fact, the `Symbol`, `Term` and `Rule` constructors are private to the package, and can be called only through the correspondent `TermRewritingSystem` object method (`NewSymbol`, `Instantiate` and `NewRule` respectively).

There is an even more compact version for creating terms, using the `Parse` method. The way this method can be used to build the terms is:

```

// Create the left Term
Term t_f1 = trs.Parse("f(a,x)");    // f(a, x)

// Create the right Term
Term t_f2 = trs.Parse("f(g(a),b)"); // f(g(a), b)

```

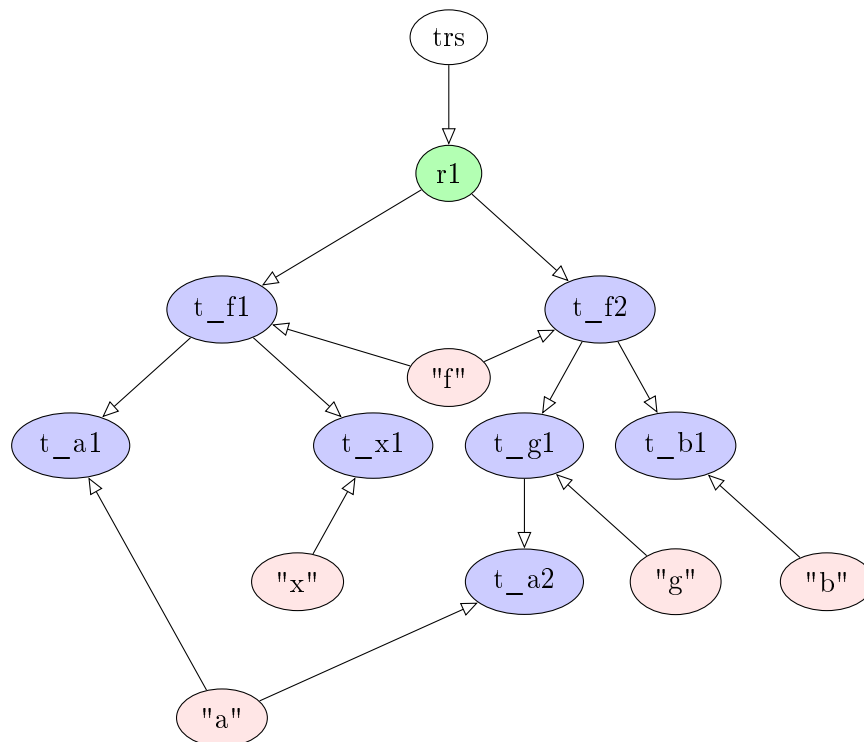


Figure 14: Term Rewriting System in memory representation

Figure 14 represents the objects structure created with the previous code. Blue balloons are Terms and the red ones are the Symbols. The white balloon is the TermRewritingSystem object and the green one is the Rule object. In order to make the graph more clear, the link between the Symbols and the TermRewritingSystem object is not represented.

3.2 Critical Pair Module

This module stores the information concerning with a Critical Pair.

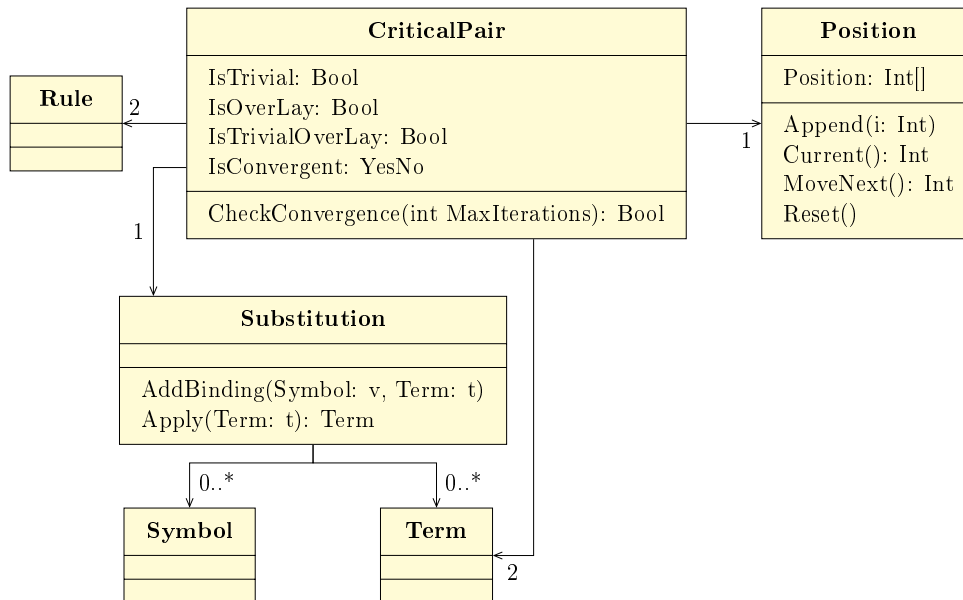


Figure 15: Critical Pair Module Class Diagram

3.2.1 Substitution Class

The Substitution class is a list of *bindings*, i.e., pairs Variable-Term. Each pair is internally named *binding*, the method `AddBinding` is used to add new bindings to the substitution, as it takes a `Symbol` object and a `Term` object as arguments. The method checks its type (must be `Variable`); If it fails, an exception is raised.

Substitution
Bindings: SortedList<Symbol, Term>
Clone(): Substitution Apply(Term: t): Term AddBinding(Symbol: v, Term: t) ToString(): String

Figure 16: Substitution Class

The method `Apply` takes a `Term` as argument and returns a new `Term` as the result of applying the substitution to the term. The original `Term` is not modified.

3.2.2 Position Class

`Position` is an auxiliary class that indicates a position of a symbol into a term. Internally it is an array of integers, and implements the interfaces `Enumerable<int>` and `IEnumerator<int>`, so it makes easier for the rest of algorithms to explore a term.

A `Position` object has no a reference to any object of the tree structure of a `Term`; it only stores a position which is not associated to any `Term`, e.g. "1.2.2.1.3".

Position
Pos: List<Int> Current: Int
Clone(): Position Current(): Int MoveNext(): Bool Reset() Append(i: Int) ToString(): String

Figure 17: Position Class

3.2.3 CriticalPair Class

This class is the main one in the module, as it is used to store the elements that define a critical pair, i.e.:

- an ordered pair of Rules,
- a Position, on the left-hand side of the first Rule,
- a Substitution, that unifies the subterm of the left-hand side of the first Rule at the specified Position with the left side of the second Rule. Here, we assume that both rules share no variables.
- an ordered pair of Terms,

CriticalPair
FirstRule: Rule FirstTerm: Term IsTrivial: Bool IsOverLay: Bool IsTrivialOverLay: Bool IsConvergent: YesNo Pos: Position SecondRule: Rule SecondTerm: Term Sigma: Substitution
CheckConvergence(int MaxIterations): Bool Clone(): CriticalPair ToString(): String

Figure 18: CriticalPair Class

The class has a main method: `CheckConvergence`. It tries to check whether the two Terms that form the critical pair converge to the same term. As we don't know if the term rewriting system is terminating, the method tries 'all' the possible of rewritings of each term. To avoid an infinite loop, `MaxIterations` bounds the number of iterations before stopping the search for convergence. If the method does not find an answer and reaches the maximum iterations, the method will return `false`; otherwise it will return `true`.

The data type of `IsConvergent` is an enumeration whose possible values are `[Yes, No, DontKnow]`. It will take the value `DontKnow` when the method `CheckConvergence` is unable to determine whether the critical pair converges or not.

3.3 Execution Module

The execution module is used to trace the rewriting steps over a term by a term rewriting system. An execution covers all possible rewritings over a term. The tree structure is used to represent all possible substitutions we can make. Each node represents one step of the rewriting process, and each different rewrite possibility generates a new branch on the tree.

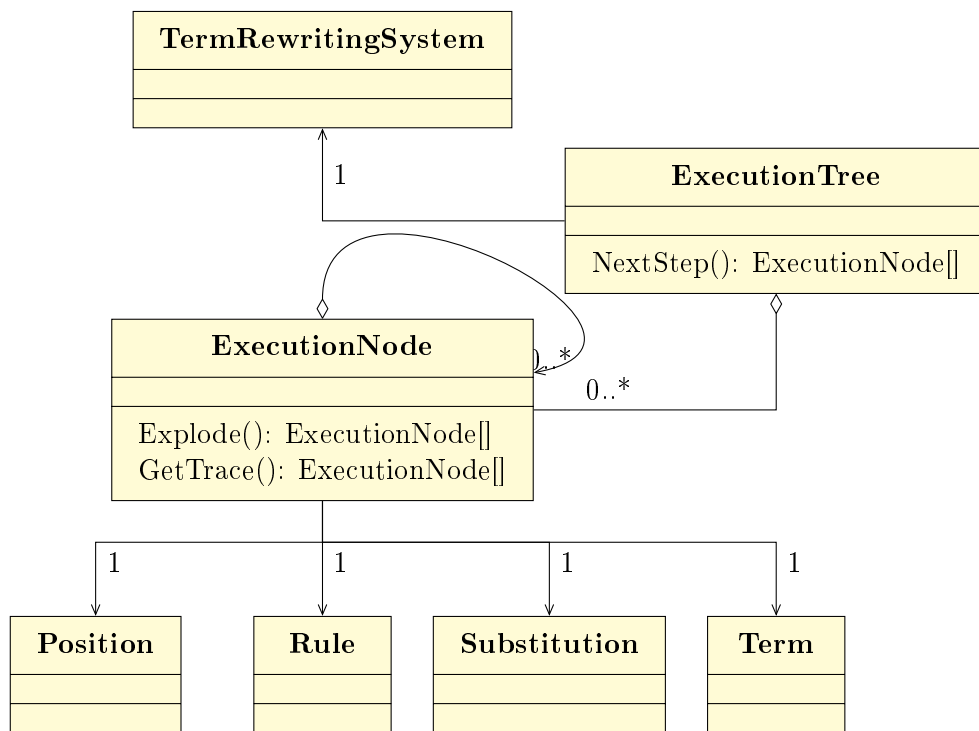


Figure 19: Execution Module Diagram

3.3.1 Execution Tree Class

`ExecutionTree` is the entry point for building a trace for the rewrites over a term. It contains references to two objects:

- **Root:** The `ExecutionNode` root of the tree. It stores the starting `Term` which is used for the substitution.
- **TRS:** The term rewriting system which is used in the rewrite process.

`ExecutionTree` has three collections of `ExecutionNode`:

- **Nodes:** All the execution nodes generated during the execution process.
- **CanonicalNodes:** The nodes with a `Term` in normal form, that is, no rule from the `TermRewritingSystem`, can be applied.
- **NewNodes:** Nodes that were created in a previous iteration of the execution process and are waiting to be processed.

ExecutionTree
CanonicalNodes: List<ExecutionNode> Nodes: List<ExecutionNode> NewNodes: List<ExecutionNode> Root: ExecutionNode TRS: TermRewritingSystem
NextStep(): List<ExecutionNode> Find(t: Term): ExecutionNode

Figure 20: `ExecutionTree` Class

The way this class works is:

1. When an object is created, the root node is stored on `CanonicalNodes` or `NewNodes` depending if the root `Term` is in normal form or not.
2. Now we begin an iterative process: each step is launched by calling the `NextStep` method.
3. When the `NextStep` method is called, it takes an `ExecutionNode` out from the `NewNodes` list and calls the `Explode` method on it.
4. Every `ExecutionNode` which is returned by the `Explode` method is stored in the `Nodes` list.

5. Any `ExecutionNode` which is returned by the `Explode` method with a `Term` in normal form is stored in the `CanonicalNodes` list.
6. The rest of `ExecutionNode` which are not added on a previous iteration to the `Nodes` list are stored in the `NewNodes` list. The `ExecutionNode` not added to `NewNodes` list are flagged as `Deforested`
7. The end of the iterative process is reached when the `NextStep` method returns a `Null` value. This means that there is no node on the `NewNodes` list.

If the term rewriting system is not terminating, we can fall in an eternal loop, and never get out from the iterative process. If the terms generated in the rewriting process are finite, even though the term rewriting system is not terminating, the iterative process will finish.

3.3.2 ExecutionNode Class

A `ExecutionNode` object stores a single step in the rewrite process. The properties of the objects are:

- `Term`: Is the current term.
- `Deforest`: Is a boolean that indicates the current `Term` has been generated before on another `ExecutionNode` from the same `ExecutionTree`. If it is `true`, it is not necessary to continue the rewrite process from this point because we will generate the same branches.
- `Childs` and `Parent`: Links to parent and child `ExecutionNode` in the tree structure. The `Child` list will be empty until the `Explode` method is called.
- `Pos`, `Rule`, `Sigma`: Those are the `Rule`, `Position` and `Substitution` that used over the `Term` from the parent `ExecutionNode` generates the current `Term`.

ExecutionNode
Deforested: Bool Childs: List<ExecutionNode> IsCanonical: YesNo Parent: ExecutionNode Pos: Position Rule: Rule Sigma: Substitution Term: Term
Explode(): List<ExecutionNode> GetTrace(): List<ExecutionNode>

Figure 21: ExecutionNode Class

The method `Explode` calculates all child `ExecutionNodes` resulting from the application of all possible rules to the current `Term`. Method `GetTrace` returns an ordered list of all the `ExecutionNodes` from the root to the current node.

Example 9: Execution Module.

This example traces how the execution module works. Given the term rewriting system from the Example¹ 5:

$$f(a, x) \rightarrow h(x, x, a) \quad (92)$$

$$g(x) \rightarrow f(a, x) \quad (93)$$

$$f(y, y) \rightarrow y \quad (94)$$

$$h(x, y, a) \rightarrow g(x) \quad (95)$$

$$f(g(x), a) \rightarrow g(g(x)) \quad (96)$$

$$b \rightarrow a \quad (97)$$

$$f(a, x) \rightarrow a \quad (98)$$

Let's trace how the `ExecutionTree` is built for `Term`:

$$f(g(x), a) \quad (99)$$

We assume that the `trs` object is created according to the term rewriting system definition

¹You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-5>

```
Term t = trs.Parse("f(g(x),a)");

\\ Initialize the ExecutionTree Object
ExecutionTree eTree = new ExecutionTree(t);
```

After the initialization, the structure of the tree will be:

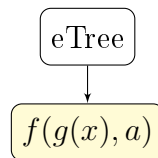


Figure 22: ExecutionTree Init

Yellow nodes represent the ones stored in the `NewNodes` list. Now we can start to iterate over the `ExecutionTree`:

```
List<ExecutionNode> newNodes = null;
do{
    newNodes = eTree.NextStep();
}while(newNodes != null);
```

This code will iterate until the `NewNodes` list is empty. We should add a counter to the loop to avoid infinite loops, and stop if a predefined number of iterations is reached. For the current example, though, the counter is not necessary. Now we can see, through the next figures, how the `ExecutionTree` grows.

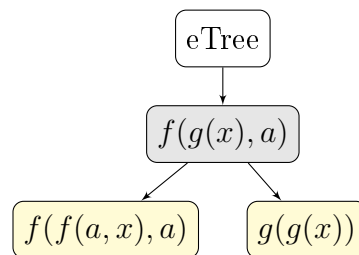


Figure 23: ExecutionTree Step 1

The node that is currently being Explode is represented in gray. Branches generated by this node represent different options on applying the

rules from the term rewriting system. Those new nodes are added to the NewNodes list, waiting to be Explode and thus generating new branches.

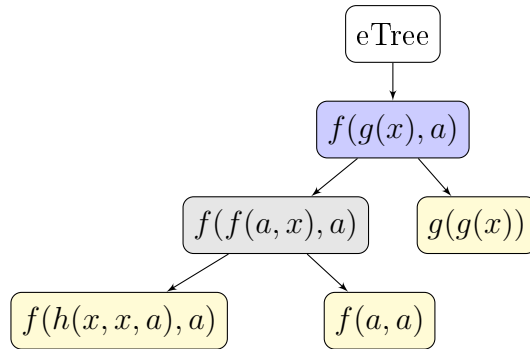


Figure 24: ExecutionTree Step 2

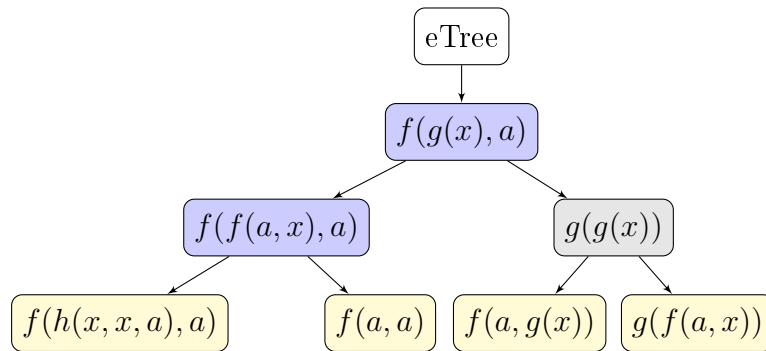


Figure 25: ExecutionTree Step 3

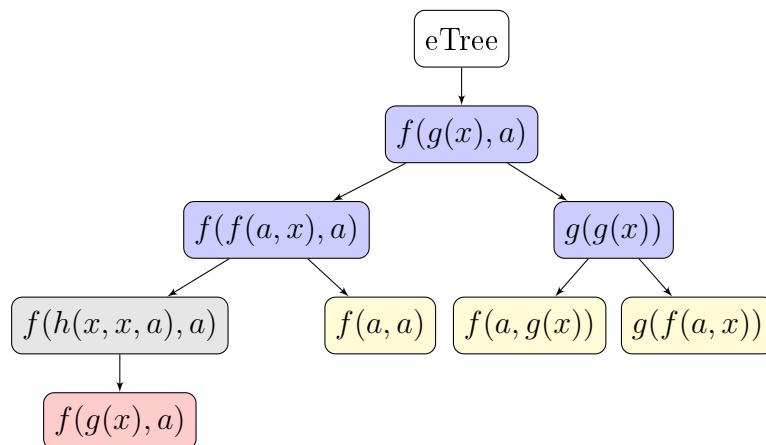


Figure 26: ExecutionTree Step 4

When the tree detects a node that previously appeared during the execution, this new node is deforested, and it is not added to the `NewNodes` list. This kind of nodes are represented in red.

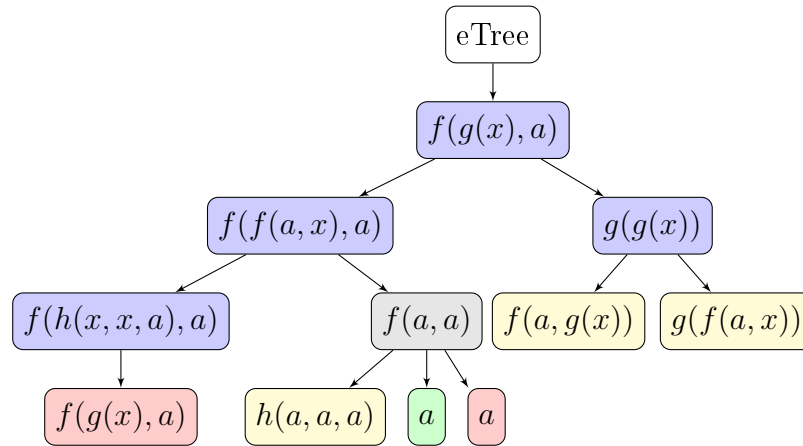


Figure 27: ExecutionTree Step 5

When a new node contains a `Term` in normal form (represented in green), it enters in the `CanonicalNodes` list, not in `NewNodes`. Only the first occurrence of a canonical term is stored in the `CanonicalNodes` list, the rest are considered deforested nodes.

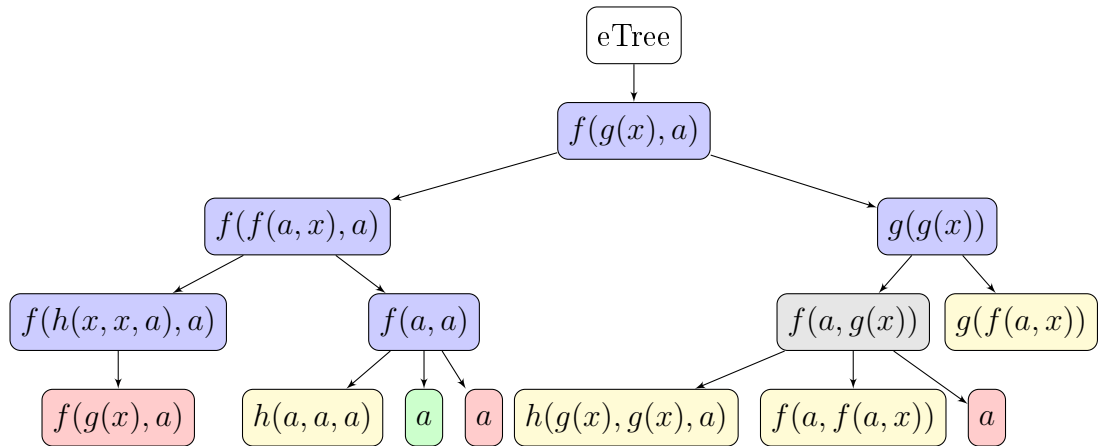


Figure 28: ExecutionTree Step 6

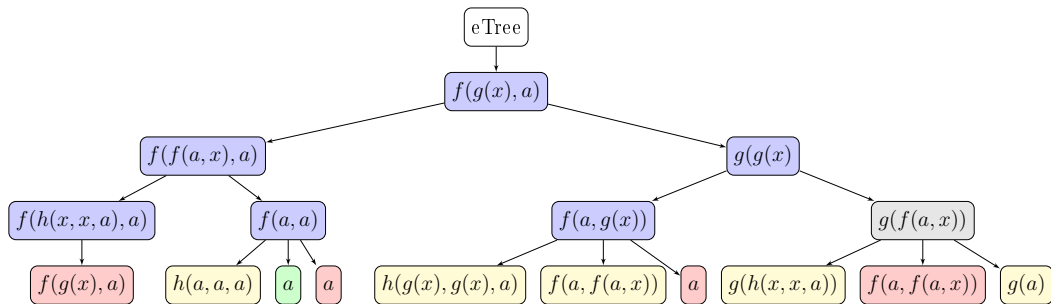


Figure 29: ExecutionTree Step 7

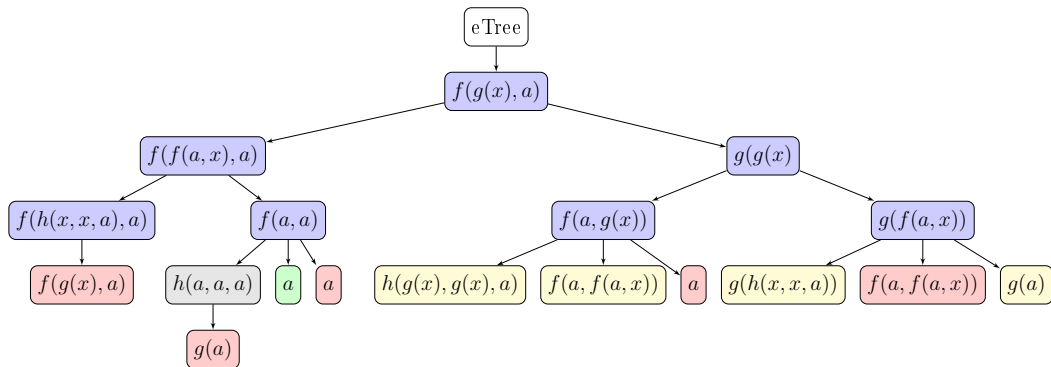


Figure 30: ExecutionTree Step 8

We can continue the loop until no more new nodes remain to be processed. Finally we will get all branches deforested. Thus we only have to find a normal form, the one obtained in Figure 27 (a).

If we call the method `GetTrace` from the `ExecutionNode` in the `CanonicalNodes` list, we will get this information:

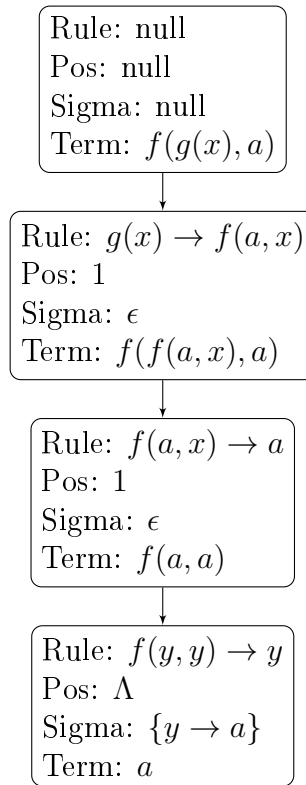


Figure 31: GetTrace data structure returned

CHAPTER 4

IMPLEMENTATION

Some of the most important algorithms of the framework are the unification and matching algorithm. For this reason, we are going to explain in detail how these algorithms have been implemented.

4.1 Unification

This method is based on Paterson-Wegman linear unification algorithm [7]. Since terms are represented in memory as trees, the algorithm explores both terms to be unified in parallel. This exploration is done *depth-first* (starting from the root and exploring as deep as possible along each branch before backtracking).

The algorithm starts with an empty substitution $\sigma = \epsilon$. For each step of the exploration, there are three possible cases:

1. The current symbol on both terms are equal. Then, continue with the unification.
2. The current symbol on both terms are different and none of them is a variable. Then, the unification fails.

3. The current symbol on both terms are different and at least one of them is a variable. Then, add a new binding to the substitution with the variable and the sub-term. Now:
 - (a) If the variable is also used on σ and the new binding is different from the one on σ , then the unification fails.
 - (b) If the variable is used on the right side of the new binding, then the unification fails.
 - (c) If the variable is not used on σ , we apply the new binding to the right side of all bindings on σ . Then we continue with the unification.

Note that when both symbols are variables, we can create the rule in one direction or another.

The original Paterson-Wegman algorithm has some errors that were discovered and fixed by D.D. Champeaux [7]. We have taken them into account in the algorithm that we implement here.

1. Each node has a list with the number of occurrences of each variable on it. The unification is possible only if the variable of the substitution is not used on it.
2. Each time a new binding is added to the substitution, it is applied to all existing bindings in the substitution.

With those changes the algorithm is still linear (with respect to the sum of nodes and edges in the tree graph of the terms).

Example 10: Unification algorithm example traces.

Let $f(a, x)$ and $f(a, g(x))$ be two terms that we want to unify.

The algorithm starts with $\sigma = \epsilon$ and pointers to the respective roots of the term (marked in red on the figures).

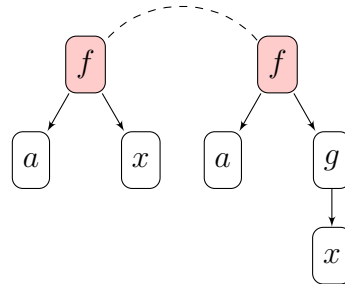


Figure 32: Unification example 1, step 1

As both nodes have the same symbol, we continue with the exploration of the tree:

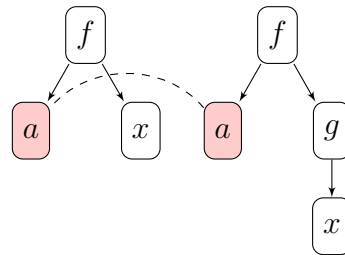


Figure 33: Unification example 1, step 2

Again, the nodes have the same symbol, and because we are in a leaf of the tree, we must do backtracking to explore the other branch.

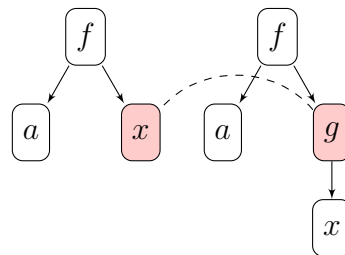


Figure 34: Unification example 1, step 3

In this case, the nodes are different, one has the subterm x and the second $g(x)$. Since one of the terms is a variable, we must create a new binding which is added to the substitution: $x \mapsto g(x)$. However, since the variable x is used in $g(x)$, the unification fails. And this means that both terms does not unify.

Let's see another example:

Let $f(x, y)$ and $f(g(y), a)$ be two terms that we want to unify.

The algorithm starts with $\sigma = \epsilon$ and pointers to the roots of each term.

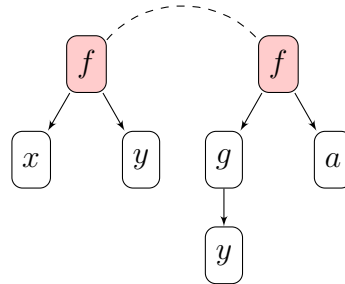


Figure 35: Unification example 2, step 1

We continue with the exploration.

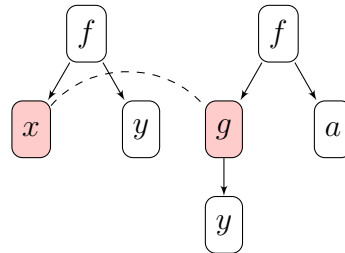


Figure 36: Unification example 2, step 2

As in the previous example, the nodes are different, one has the subterm x and the second $g(y)$, since one of the terms is a variable, we must create a new binding: $x \mapsto g(y)$. Now variable x is not used in $g(y)$ and the unification can continue. We add the new binding to the substitution: $\sigma = \epsilon \cup \{x \mapsto g(y)\}$

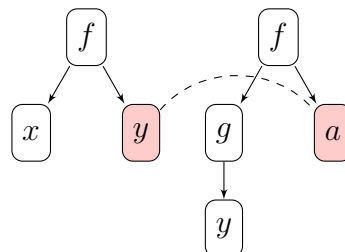


Figure 37: Unification example 2, step 3

Again we have different nodes: one with sub-term y and the other one with sub-term a , since one of the terms is a variable, we must create a new binding: $y \mapsto a$. Before adding the new binding to the substitution, we apply the new binding to the existing one in the substitution. The substitution becomes: $\sigma = \{x \mapsto g(a), y \mapsto a\}$

We have finished the tree exploration: both terms unify with $\sigma = \{x \mapsto g(a), y \mapsto a\}$

4.2 Matching

The matching algorithm can be considered as a particular case of the unification algorithm because the way they work is the same, with the difference that the unification is done in both directions and matching is only one direction. As in the unification, the algorithm explores both terms in parallel using a depth-first exploration.

The algorithm starts with an empty substitution $\sigma = \epsilon$. On each exploration step there are three possible cases:

- The current symbol on both terms are equal: continue with the matching.
- The current symbol on both terms are different: the first term is not a variable: the matching fails.
- The current symbol on both terms are different and the first term is a variable: we are going to add a new binding to the substitution with the variable part from the first term and the term part from the second one. Now:
 - If the variable is also used in σ and the new binding is different from the one in σ , the matching fails.
 - If the variable is not used in σ , we continue with the matching.

Note that in this case the substitution is always made in the same direction, so the variable must be allocated on the first term.

Example 11: Matching algorithm example traces.

We are going to illustrate the matching algorithm with the same terms from Example 10

Let $f(a, x)$ and $f(a, g(x))$ be two terms that we want to match.

The algorithm starts with $\sigma = \epsilon$ and pointers to the roots of each term.

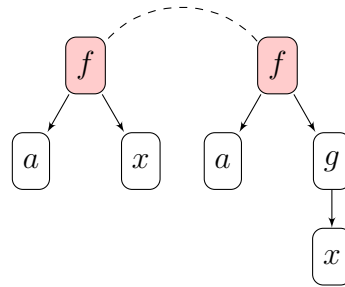


Figure 38: Matching example 1, step 1

As both nodes have the same symbol, we continue with the tree exploration:

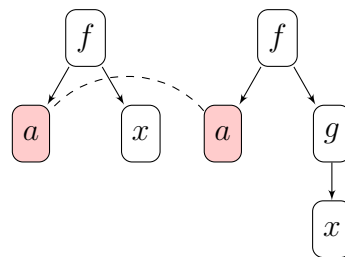


Figure 39: Matching example 1, step 2

Again, the nodes have the same symbol. Since we are in a leaf of the tree, we must do backtracking to explore the other branch.

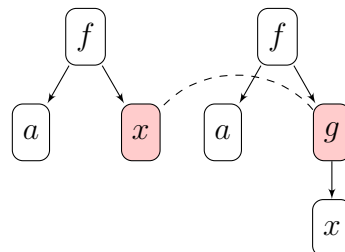


Figure 40: Matching example 1, step 3

In this case, the nodes are different, one has the subterm x and the second $g(x)$. Since one of the terms is a variable, we must create a new binding: $x \mapsto g(x)$. In the matching algorithm we don't have the restriction that the variable of the binding can't be used in the term part so we add the new binding to the substitution.

Since we end the exploration, the term $f(a, x)$ matches with the term $f(a, g(x))$ with the substitution $\sigma = \{x \mapsto g(x)\}$.

Let's see the second example:

Let $f(x, y)$ and $f(g(y), a)$ be two terms that we want to match.

The algorithm starts with $\sigma = \epsilon$ and pointers to the roots of each term.

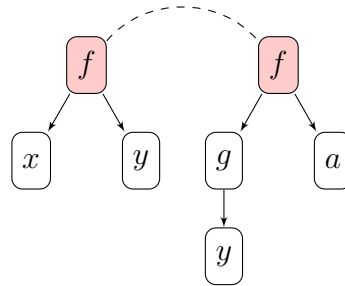


Figure 41: Matching example 2, step 1

We continue with the exploration.

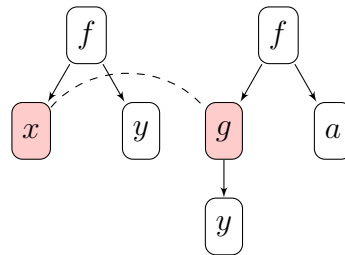


Figure 42: Matching example 2, step 2

The nodes are different, one has the subterm x and the other $g(y)$. Since the first is a variable, we must create a new binding: $x \mapsto g(y)$. So we add it to the substitution: $\sigma = \{x \mapsto g(y)\}$

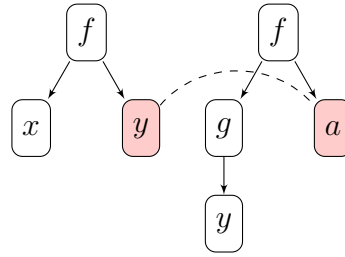


Figure 43: Matching example 2, step 3

Again we have different nodes, one with y and the other with a . We create the binding $y \mapsto a$, and add it to the substitution: $\sigma = \{x \mapsto g(y), y \mapsto a\}$

We have finished the tree exploration: the term $f(x, y)$ matches with the term $f(g(y), a)$ with the substitution $\sigma = \{x \mapsto g(y), y \mapsto a\}$.

4.3 Confluence decision

Figure 44 shows the flow chart for studying confluence of a term rewriting system on the basis of the analysis of its critical pairs.

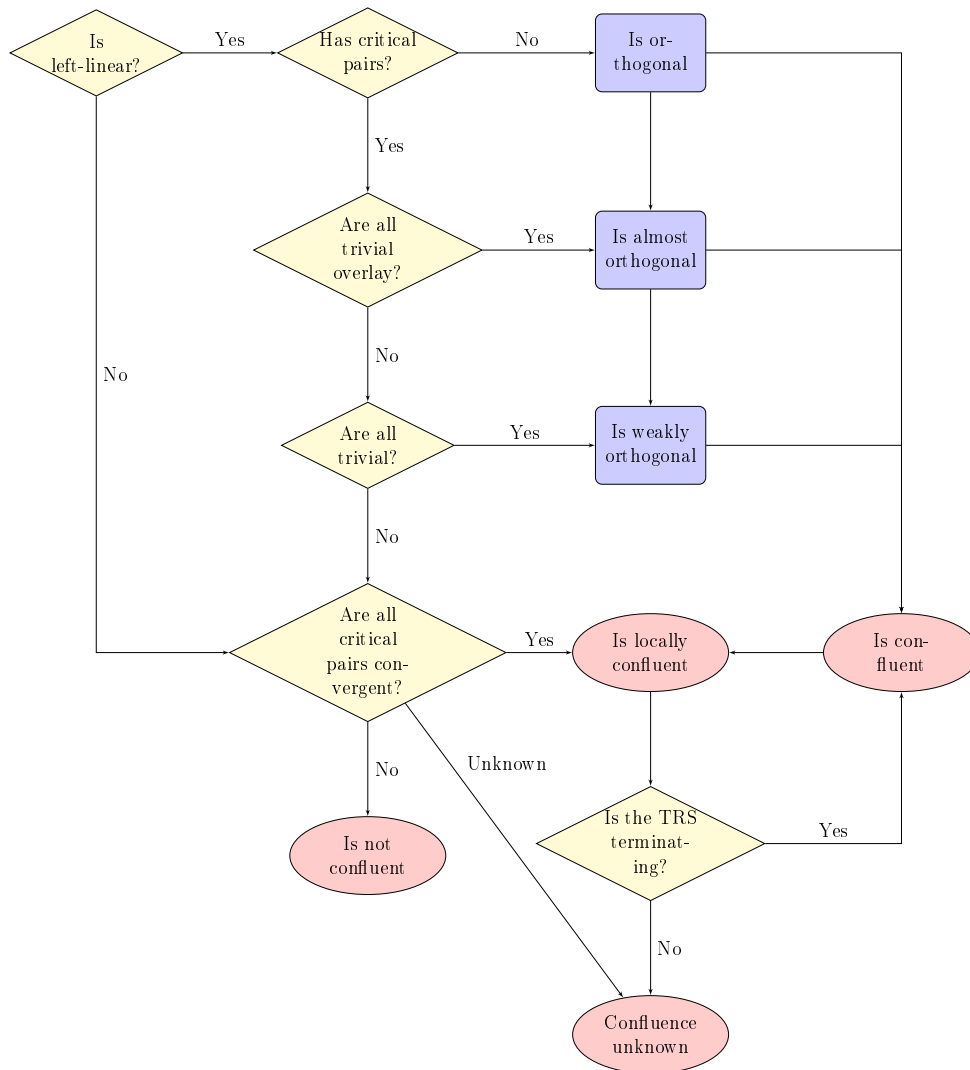


Figure 44: Confluence algorithm flow chart

If the term rewriting system is not orthogonal (or almost/weakly orthogonal), because it is not left-linear or some of its critical pairs is not trivial, we must check the convergence of the critical pairs. This point is critical, because, right now, the framework does not provide any method to determine if the term rewriting system is terminating or not.

The method which is used to check for convergence of critical pairs is quite simple: the execution module is used to try all possible rewrites over each term in the critical pair, and then checking for an eventual equality of the results waiting at some point. The problem is that, if the term rewriting

system is not terminating, then the module could be running forever without reaching this point. Since the exploration of the reduction space of a term can be limited by using an additional counter which is a parameter of the process, if the process reaches this limit without any convergence, the critical pair is flagged as “convergence unknown”.

There are three particular cases where we can obtain useful information when such “confluence unknown” state is obtained:

- If exploring any of the terms in the critical pair we get two (or more) different normal forms, then we can conclude that the term rewriting system is *not confluent*, regardless of the result for the convergence of the critical pair and the local confluence property.
- If, while looking for convergence of a critical pair, any of the terms has exhausted the exploration of its reduction space before reaching the specified limit of rewrite steps without obtaining a normal form, then the term rewriting system is nonterminating. This is because the execution tree has all its leaves deforested, and there are loops to other terms in the execution tree, so there are infinite loops.
- If, while looking for a convergence point for a critical pair, a loop has been detected, i.e., any term contains a subterm that previously occurs in the rewrite sequence, we can conclude that the term rewriting system is nonterminating (see the definition of looping TRS in page 13).

In the following, we illustrate these remarks with some corresponding examples:

Example 12: Non confluence.

Consider the following rewriting system¹:

$$b \rightarrow a \tag{100}$$

$$b \rightarrow c \tag{101}$$

$$c \rightarrow b \tag{102}$$

$$c \rightarrow d \tag{103}$$

One of its critical pairs is:

$$\langle c, a \rangle \tag{104}$$

¹You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-8>

The critical pair is convergent because:

$$c \rightarrow b \rightarrow a \quad (105)$$

But this is the execution tree created while looking for this convergence:

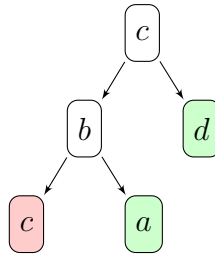


Figure 45: Non confluence example

The tree has two different normal forms (the green ones), so the term rewriting system is not confluent.

Example 13: Non termination, first case.

Consider the following rewriting system²:

$$+ (0, x) \rightarrow x \quad (106)$$

$$+ (-x, x) \rightarrow 0 \quad (107)$$

$$+ (+x, y), z \rightarrow +(x, +(y, z)) \quad (108)$$

$$+ (x, y) \rightarrow +(y, x) \quad (109)$$

$$- (0) \rightarrow 0 \quad (110)$$

$$- (-x) \rightarrow x \quad (111)$$

$$- (+x, y) \rightarrow +(-x, -(y)) \quad (112)$$

One of its critical pairs is:

$$\langle -(+y, x), +(-x, -(y)) \rangle \quad (113)$$

If we explore all possible rewrites over the first term $-(+y, x)$ we will get this tree:

²You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-6>

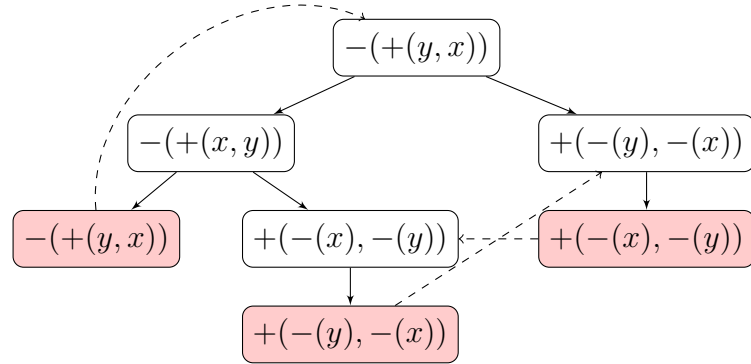


Figure 46: Non termination example

The deforested nodes (the ones that have been created before on the tree) are marked with red background. The tree can't be further explored, but no normal form has been found, because every rewrite sequence will make an eternal loop with terms of the tree, so we can conclude that this term rewriting system is nonterminating.

Example 14: Non termination, second case.

Consider the rewriting system from Example 12.

In Figure 45 we can see that the second occurrence of the node c is deforested, and the first occurrence of it is in the path to the root node, so here is an infinite loop in a rewriting sequence: $c \rightarrow b \rightarrow c \rightarrow b \rightarrow \dots$. We can conclude that term rewriting system is nonterminating.

Note that the rewrite process can be represented as a tree due to the deforestation. Without deforestation, the representation is a directed graph and the term rewriting system is nonterminating if there is a cycle in the graph.

The cycles the framework can detect are of the kind of Example 12, where there is a rewrite rule that gives back a term to a previous term in the same branch of the tree (see Figure 47). There are other cycles that can't be detected because, since the tree is deforested, the cycle involves different branches in the tree, e.g. in Figure 46, the loop $+(-x, -y) \rightarrow +(-y, -x) \rightarrow +(-x, -y)$ is not detected as a loop by the framework.

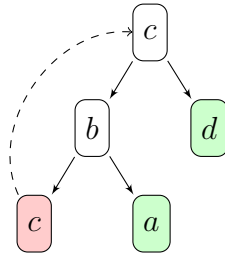


Figure 47: Non confluence example

If we suppress deforestation, the framework will be eventually able to detect all loops, as shown in Figure 48, because the deforestation cuts the rewrite sequence before it reaches the same term twice in the same branch.

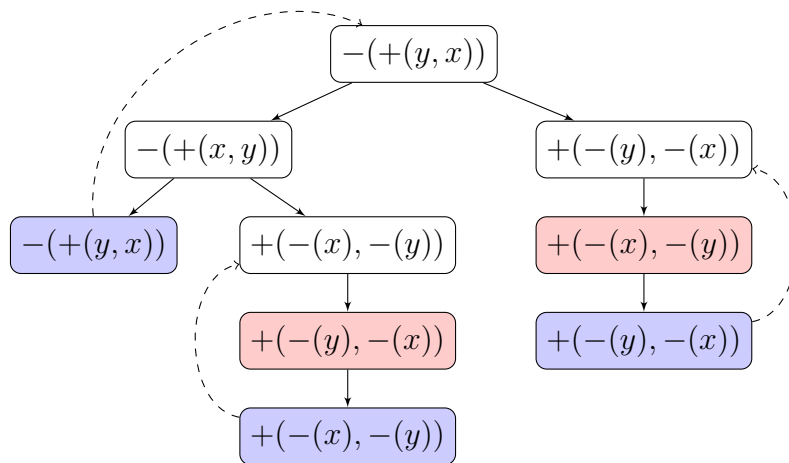


Figure 48: Tree without deforestation

The first case when the framework concludes the nontermination of the term rewriting system is useful when the tree has all its leaves deforested without detecting any loop involving a single branch in the tree reduction.

CHAPTER 5

TRS.TOOL

To illustrate the use of the framework in a real application. A web tool has been developed to test all functionalities. It can be used in:

`http://TRS.JarCode.Net`

The interface is very simple, there is a textarea where we can type the target term rewriting system. We can also upload a file from the local computer with the term rewriting system definition. You can specify the limit of rewrites while searching for convergence of critical pairs. This limit is for each critical pair.



Figure 49: Web tool interface

TRS.Tool has been included in the list of rewriting tools maintained by Nao Hirokawa at the Japan Advanced Institute of Science Technologies (JAIST)¹.

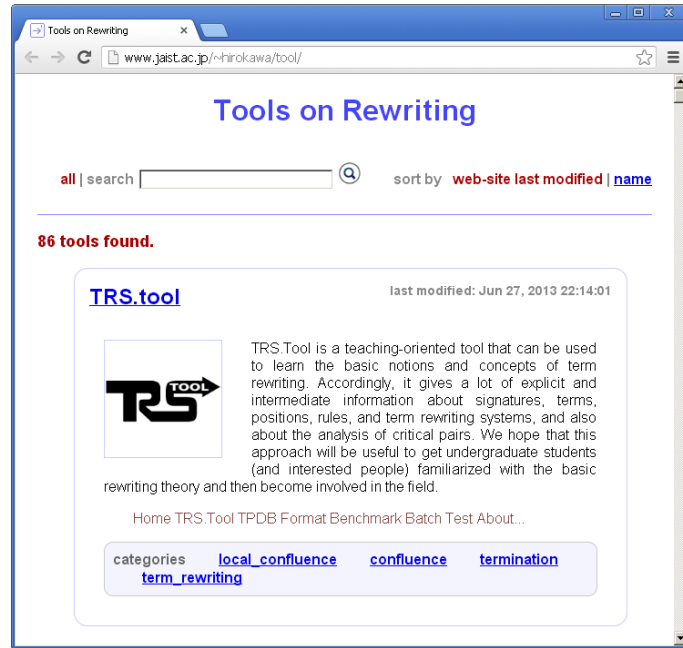


Figure 50: Tools on Rewriting

5.1 TPDB Format

Our choice of input format for Term Rewriting Systems is a very popular one: the TPDB format, which is used since 2003 in the Termination Problem Data Base (TPDB). For more information about the original format you can visit the website of the Termination Problem Data Base². Here is a brief description:

```

spec          ::= (decl) spec | ε
decl          ::= VAR idlist | FUNCTIONS funclist |
                CONSTANTS idlist | RULES listofrules |
                TERMS listofterms | COMMENTS listofcoments
idlist        ::= id idlist | ε
funclist      ::= funcdef funclist | ε
  
```

¹<http://www.jaist.ac.jp/~hirokawa/tool/>

²<https://www.lri.fr/~marche/tpdb/format.html>

```

funcdef      ::= _id_ | _id | id(int)
listofrules  ::= rule listofrules |  $\varepsilon$ 
rule         ::= term -> term
term         ::= id | id() | id(termlist) | (term)id(term) |
                (term)id
termlist     ::= term, termlist | term
listofterms  ::= id = term listofterms |  $\varepsilon$ 
listofcoments ::= string listofcoments |  $\varepsilon$ 

```

Notes about the grammar:

- *id* are non-empty sequences of characters except space, '(', ')', '"', '=' and ', '; and excluding the special sequence '->' and keywords VAR, FUNCTIONS, CONSTANTS, RULES, TERMS and COMMENTS.
- *string* are sequences of any characters between double quotes.
- *int* are non-empty sequences of digits.
- A symbol occurring in a RULES section which has not been used before is assumed to denote a function symbol, and must be used afterwards always with the same arity.

Theses are the changes introduced in the original format:

- Removed THEORY and STRATEGY sections. There is no need to remove them from the input data, because any unknown section is ignored.
- Added FUNCTIONS and CONSTANTS sections. They are optional, and the parser can deduce them from the RULES sections, but has been considered important to include them because can reduce the number of mistakes produced typing the rules.
- Added TERMS section. It is used to pass to the tool the terms that have to be use in the execution module.
- Added infix and postfix notation for functions, e.g. $(op1) + (op2)$ and $(op) ++$ respectively.

An example of use of this format is shown in the next section.

5.2 Use Example

To test the functionality of the web tool we are going to show a simple execution of the tool. The term rewriting system for the test is³:

$$f(a, x) \rightarrow h(x, x, a) \quad (114)$$

$$g(x) \rightarrow f(a, x) \quad (115)$$

$$f(y, x) \rightarrow y \quad (116)$$

$$h(x, y, a) \rightarrow g(x) \quad (117)$$

$$f(g(x), a) \rightarrow g(g(x)) \quad (118)$$

Also, for testing the execution module, we are going to use the term:

$$t = f(g(x), a) \quad (119)$$

First step is write the term rewriting system data into the TPDB Format, and input it on the textarea.

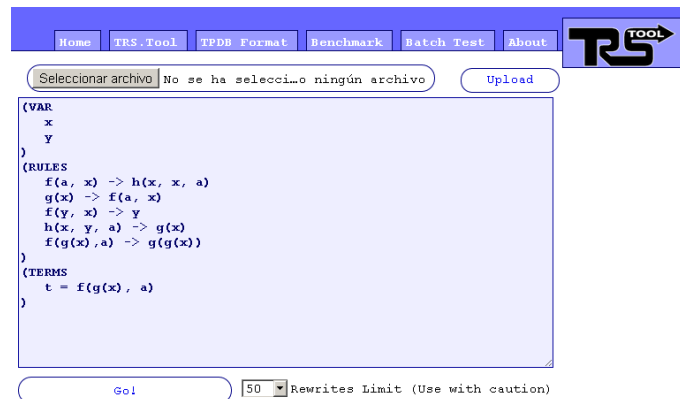


Figure 51: TRS input

Once the term rewriting system has been introduced you can push the button «Go!» and the web tool does all the operations and presents the results. Those have been classified in 6 groups, each one in a different tab: Input Data, Terms, Matching and unification, Rules, Critical pairs and Term Execution.

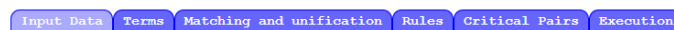


Figure 52: Result tabs

³You can test it with TRS.Tool: <http://TRS.JarCode.Net/?t=-7>

5.2.1 Input Data

Shows the same information introduced by the user. The main purpose is to check that all data have been correctly parsed.

Input Data	Terms	Matching and unification	Rules	Critical Pair	Execution
<pre> TRS.ToString() (VAR x y) (FUNCTIONS f(2) g(1) h(3)) (CONSTANTS a) (RULES f(a,x) -> h(x,x,a) g(x) -> f(a,x) f(y,x) -> y h(x,y,a) -> g(x) f(g(x),a) -> g(g(x))) (TERMS t = f(g(x),a) T0L = f(a,x) T0R = h(x,x,a) T1L = g(x) T1R = f(a,x) T2L = f(y,x) T2R = y T3L = h(x,y,a) T3R = g(x) T4L = f(g(x),a) T4R = g(g(x))) (COMMENTS) </pre>	<pre> Variables and Signature X = [x, y] F = {f, g, h, a} ar(f) = 2, ar(g) = 1, ar(h) = 3, ar(a) = 0 </pre>				
	<pre> Terms t = f(g(x),a) T0L = f(a,x) T0R = h(x,x,a) T1L = g(x) T1R = f(a,x) T2L = f(y,x) T2R = y T3L = h(x,y,a) T3R = g(x) T4L = f(g(x),a) T4R = g(g(x)) </pre>				
			<pre> Rules R0 = f(a,x) -> h(x,x,a) R1 = g(x) -> f(a,x) R2 = f(y,x) -> y R3 = h(x,y,a) -> g(x) R4 = f(g(x),a) -> g(g(x)) </pre>		

Figure 53: Input data tab

The information that is shown here is organized into four tables:

- *TRS.ToString()*: uses the ToString method of the TermRewritingSystem Object created to print out the term rewriting system in TPDB format, see Table 1.


```

TRS.ToString()

(VAR
    x
    y
)
(FUNCTIONS
    f(2)
    g(1)
    h(3)
)
(CONSTANTS
    a
)
(RULES
    f(a,x) -> h(x,x,a)
    g(x) -> f(a,x)
    f(y,x) -> y
    h(x,y,a) -> g(x)
    f(g(x),a) -> g(g(x))
)
(TERMS
    t = f(g(x),a)
    T0L = f(a,x)
    T0R = h(x,x,a)
    T1L = g(x)
    T1R = f(a,x)
    T2L = f(y,x)
    T2R = y
    T3L = h(x,y,a)
    T3R = g(x)
    T4L = f(g(x),a)
    T4R = g(g(x))
)
(COMMENTS
)

```

Table 1: TRS.ToString()

- *Variables and Signature*: Shows the set of variables and the signature (function symbols and arity functions), see Table 2

Variables and Signature
$X = \{x, y, z\}$
$F = \{f, g, h, a\}$ $\text{ar}(f) = 2, \text{ar}(g) = 1, \text{ar}(h) = 3, \text{ar}(a) = 0$

Table 2: Variables and signature

- **Terms:** Lists all terms with labels to be used in the rest of the tabs. The terms from the rules are numbered from 0 to n , and marked with "L" or "R", so the term T_{0L} is the left-hand side of the first rule, the term T_{3R} is the right-hand side of the fourth rule, and so on..., see Table 3

Terms
$t = f(g(x), a)$
$T_{0L} = f(a, x)$
$T_{0R} = h(x, x, a)$
$T_{1L} = g(x)$
$T_{1R} = f(a, x)$
$T_{2L} = f(y, x)$
$T_{2R} = y$
$T_{3L} = h(x, y, a)$
$T_{3R} = g(x)$
$T_{4L} = f(g(x), a)$
$T_{4R} = g(g(x))$

Table 3: Variables and signature

5.2.2 Terms

It draws the tree structure for each term and its basic properties.

Input Data	Terms	Matching and unification	Rules	Critical Pair	Execution		
$t = f(g(x), a)$ $f \mid \Lambda$ $g \mid 1$ $a \mid 2$ $x \mid 1.1$ Var(t) = { x } t is not Ground t is Linear	$T_{0L} = f(a, x)$ $f \mid \Lambda$ $a \mid 2$ $x \mid 2$ Var(T_{0L}) = { x } T_{0L} is not Ground T_{0L} is Linear	$T_{0R} = h(x, x, a)$ $h \mid \Lambda$ $x \mid 1$ $x \mid 2$ $a \mid 3$ Var(T_{0R}) = { x } T_{0R} is not Ground T_{0R} is not Linear	$T_{1L} = g(x)$ $g \mid \Lambda$ $x \mid 1$ Var(T_{1L}) = { x } T_{1L} is not Ground T_{1L} is Linear	$T_{1R} = f(a, x)$ $f \mid \Lambda$ $a \mid 1$ $x \mid 2$ Var(T_{1R}) = { x } T_{1R} is not Ground T_{1R} is Linear	$T_{2L} = f(y, x)$ $f \mid \Lambda$ $y \mid 1$ $x \mid 2$ Var(T_{2L}) = { x, y } T_{2L} is not Ground T_{2L} is Linear	$T_{2R} = y$ $y \mid \Lambda$ $y \mid 1$ Var(T_{2R}) = { y } T_{2R} is not Ground T_{2R} is Linear	$T_{3L} = h(x, y, a)$ $h \mid \Lambda$ $x \mid 1$ $y \mid 2$ $a \mid 3$ Var(T_{3L}) = { x, y } T_{3L} is not Ground T_{3L} is Linear
$T_{3R} = g(x)$ $g \mid \Lambda$ $x \mid 1$ Var(T_{3R}) = { x } T_{3R} is not Ground T_{3R} is Linear	$T_{4L} = f(g(x), a)$ $f \mid \Lambda$ $g \mid 1$ $a \mid 2$ $x \mid 1.1$ Var(T_{4L}) = { x } T_{4L} is not Ground T_{4L} is Linear	$T_{4R} = g(g(x))$ $g \mid \Lambda$ $g \mid 1$ $x \mid 1.1$ Var(T_{4R}) = { x } T_{4R} is not Ground T_{4R} is Linear					

Figure 54: Terms tab

As an example, this is the information for terms t and T_{3L}

$t = f(g(x), a)$
$f \mid \Lambda$
$g \mid 1$ $a \mid 2$ $x \mid 1.1$
Var(t) = { x }
t is not Ground
t is Linear

Table 4: Term t analysis

Matching			
$T_{1L} = g(x)$	matches	$T_{4R} = g(g(x))$	with $\sigma = \{x \rightarrow g(x)\}$
$T_{2L} = f(y, x)$	matches	$t = f(g(x), a)$	with $\sigma = \{x \rightarrow a, y \rightarrow g(x)\}$
$T_{2L} = f(y, x)$	matches	$T_{0L} = f(a, x)$	with $\sigma = \{y \rightarrow a\}$
$T_{2L} = f(y, x)$	matches	$T_{1R} = f(a, x)$	with $\sigma = \{y \rightarrow a\}$
$T_{2L} = f(y, x)$	matches	$T_{4L} = f(g(x), a)$	with $\sigma = \{x \rightarrow a, y \rightarrow g(x)\}$
$T_{2R} = y$	matches	$t = f(g(x), a)$	with $\sigma = \{y \rightarrow f(g(x), a)\}$
$T_{2R} = y$	matches	$T_{0r} = f(a, x)$	with $\sigma = \{y \rightarrow f(a, x)\}$

Figure 56: Matching table detail

Unification			
$t = f(g(x), a)$	and	$T_{2L} = f(y, x)$	unifies with $\sigma = \{x \rightarrow a, y \rightarrow g(a)\}$
$t = f(g(x), a)$	and	$T_{2R} = y$	unifies with $\sigma = \{y \rightarrow f(g(x), a)\}$
$T_{0L} = f(a, x)$	and	$T_{2L} = f(y, x)$	unifies with $\sigma = \{y \rightarrow a\}$
$T_{0L} = f(a, x)$	and	$T_{2R} = y$	unifies with $\sigma = \{y \rightarrow f(a, x)\}$
$T_{0R} = h(x, x, a)$	and	$T_{2R} = y$	unifies with $\sigma = \{y \rightarrow h(x, x, a)\}$
$T_{0R} = h(x, x, a)$	and	$T_{3L} = h(x, y, a)$	unifies with $\sigma = \{x \rightarrow y\}$
$T_{1r} = a(x)$	and	$T_{2D} = v$	unifies with $\sigma = \{v \rightarrow a(x)\}$

Figure 57: Unification table detail

For each pair of terms such that the matching or unification is possible, the table contains the substitution σ that makes it possible. If a pair of terms do not unify or match, they are not included in the corresponding table.

5.2.4 Rules

Analyses of the properties of the rules.

Input Data	Terms	Matching and unification	Rules	Critical Pair	Execution
$R_0 = f(a, x) - h(x, x, a)$	$R_1 = g(x) - f(a, x)$	$R_2 = f(y, x) - y$			
R_0 is Left-Linear	R_1 is Left-Linear	R_2 is Left-Linear			
R_0 is not Right-Linear	R_1 is Right-Linear	R_2 is Right-Linear			
R_0 is not Linear	R_1 is Linear	R_2 is Linear			
R_0 is not Collapsing	R_1 is not Collapsing	R_2 is Collapsing			
R_0 is Duplicating	R_1 is not Duplicating	R_2 is not Duplicating			
R_0 is Conservative	R_1 is Conservative	R_2 is not Conservative			
R_0 is not Destructive	R_1 is not Destructive	R_2 is Destructive			
$R_3 = h(x, y, a) - g(x)$	$R_4 = f(g(x), a) - g(g(x))$				
R_3 is Left-Linear	R_4 is Left-Linear				
R_3 is Right-Linear	R_4 is Right-Linear				
R_3 is Linear	R_4 is Linear				
R_3 is not Collapsing	R_4 is not Collapsing				
R_3 is not Duplicating	R_4 is not Duplicating				
R_3 is not Conservative	R_4 is Conservative				
R_3 is Destructive	R_4 is not Destructive				

Figure 58: Rules tab

There is a table for each rule that indicates its properties. Here are the tables for R_1 and R_2 :

$R_1 = g(x) \rightarrow f(a, x)$
R_1 is Left-Linear
R_1 is Right-Linear
R_1 is Linear
R_1 is not Collapsing
R_1 is not Duplicating
R_1 is Conservative
R_1 is not Destructive

Table 6: Rule R_1 analysis

$R_2 = f(y, x) \rightarrow y$
R_2 is Left-Linear
R_2 is Right-Linear
R_2 is Linear
R_2 is Collapsing
R_2 is not Duplicating
R_2 is not Conservative
R_2 is Destructive

Table 7: Rule R_2 analysis

5.2.5 Critical Pairs

This option obtains and analyses the critical pairs.

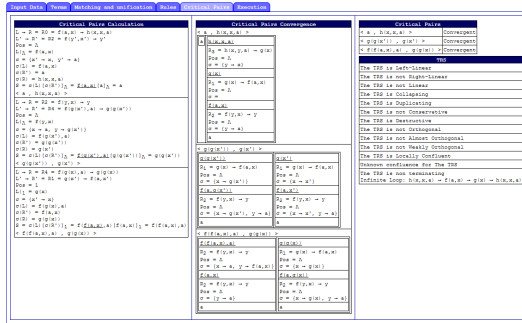


Figure 59: Critical pair tab

There are four tables on this tab:

- *Critical pair calculation*: There is a cell for each critical pair.

Critical Pairs Calculation	
$L \rightarrow R = R0 = f(a, x) \rightarrow h(x, x, a)$ $L' \rightarrow R' = R2 = f(y', x') \rightarrow y'$ $Pos = \Delta$ $L _{\Delta} = f(a, x)$ $\sigma = \{x' \rightarrow x, y' \rightarrow a\}$ $\sigma(L) = f(a, x)$ $\sigma(R') = a$ $\sigma(R) = h(x, x, a)$ $S = \sigma(L)[\sigma(R')]_{\Delta} = f(a, x)[a]_{\Delta} = a$ $\langle a, h(x, x, a) \rangle$	
$L \rightarrow R = R2 = f(y, x) \rightarrow y$ $L' \rightarrow R' = R4 = f(g(x'), a) \rightarrow g(g(x'))$ $Pos = \Delta$ $L _{\Delta} = f(y, x)$ $\sigma = \{x \rightarrow a, y \rightarrow g(x')\}$ $\sigma(L) = f(g(x'), a)$ $\sigma(R') = g(g(x'))$ $\sigma(R) = g(x')$ $S = \sigma(L)[\sigma(R')]_{\Delta} = f(g(x'), a)[g(g(x'))]_{\Delta} = g(g(x'))$ $\langle g(g(x')), g(x') \rangle$	
$L \rightarrow R = R4 = f(g(x), a) \rightarrow g(g(x))$ $L' \rightarrow R' = R1 = g(x') \rightarrow f(a, x')$ $Pos = 1$ $L _1 = g(x)$ $\sigma = \{x' \rightarrow x\}$ $\sigma(L) = f(g(x), a)$ $\sigma(R') = f(a, x)$ $\sigma(R) = g(g(x))$ $S = \sigma(L)[\sigma(R')]_1 = f(f(a, x), a)[f(a, x)]_1 = f(f(a, x), a)$ $\langle f(f(a, x), a), g(g(x)) \rangle$	

Table 8: Critical pair calculation

Here is the outcome for the first and second critical pairs:

$$\begin{array}{l}
L \rightarrow R = R0 = f(a, x) \rightarrow h(x, x, a) \\
L' \rightarrow R' = R2 = f(y', x') \rightarrow y' \\
Pos = \Lambda \\
L|_{\Lambda} = f(a, x) \\
\sigma = \{x' \rightarrow x, y' \rightarrow a\} \\
\sigma(L) = f(a, x) \\
\sigma(R') = a \\
\sigma(R) = h(x, x, a) \\
S = \sigma(L) [\sigma(R')]_{\Lambda} = \underline{f(a, x)} [a]_{\Lambda} = a \\
\langle a, h(x, x, a) \rangle
\end{array}$$

Figure 60: First critical pair calculation

$$\begin{array}{l}
L \rightarrow R = R2 = f(y, x) \rightarrow y \\
L' \rightarrow R' = R4 = f(g(x'), a) \rightarrow g(g(x')) \\
Pos = \Lambda \\
L|_{\Lambda} = f(y, x) \\
\sigma = \{x \rightarrow a, y \rightarrow g(x')\} \\
\sigma(L) = f(g(x'), a) \\
\sigma(R') = g(g(x')) \\
\sigma(R) = g(x') \\
S = \sigma(L) [\sigma(R')]_{\Lambda} = \underline{f(g(x'), a)} [g(g(x'))]_{\Lambda} = g(g(x')) \\
\langle g(g(x')), g(x') \rangle
\end{array}$$

Figure 61: Second critical pair calculation

- *Critical pair convergence*: For nontrivial critical pairs the tool tries to test their convergence:

Critical Pairs Convergence	
< a , h(x,y,a) >	
a	$h(x,y,a)$ $R_3 = h(x,y,a) \rightarrow g(x)$ Pos = Δ $\sigma = \{y \rightarrow x\}$
	$g(x)$ $R_1 = g(x) \rightarrow f(a,x)$ Pos = Δ $\sigma =$
	$f(a,x)$ $R_2 = f(y,x) \rightarrow y$ Pos = Δ $\sigma = \{y \rightarrow a\}$
	a
< g(g(x')), g(x') >	
g(g(x'))	$R_1 = g(x) \rightarrow f(a,x)$ Pos = Δ $\sigma = \{x \rightarrow g(x')\}$
g(x')	$R_1 = g(x) \rightarrow f(a,x)$ Pos = Δ $\sigma = \{x \rightarrow x'\}$
f(a,g(x'))	$R_2 = f(y,x) \rightarrow y$ Pos = Δ $\sigma = \{x \rightarrow g(x'), y \rightarrow a\}$
f(a,x')	$R_2 = f(y,x) \rightarrow y$ Pos = Δ $\sigma = \{x \rightarrow x', y \rightarrow a\}$
	a
< f(f(a,x),a) , g(g(x)) >	
f(f(a,x),a)	$R_2 = f(y,x) \rightarrow y$ Pos = Δ $\sigma = \{x \rightarrow a, y \rightarrow f(a,x)\}$
g(g(x))	$R_1 = g(x) \rightarrow f(a,x)$ Pos = Δ $\sigma = \{x \rightarrow g(x)\}$
f(a,x)	$R_2 = f(y,x) \rightarrow y$ Pos = Δ $\sigma = \{y \rightarrow a\}$
f(a,g(x))	$R_2 = f(y,x) \rightarrow y$ Pos = Δ $\sigma = \{x \rightarrow g(x), y \rightarrow a\}$
	a

Table 9: Critical pair convergence

If the critical pair converges, the rewrite sequence is detailed for each part of the critical pair. Here is the detail for checking convergence of the second and fourth critical pairs:

< a , h(x,x,a) >	
a	$h(x,x,a)$ $R_3 = h(x,y,a) \rightarrow g(x)$ Pos = Δ $\sigma = \{y \rightarrow x\}$
	$g(x)$ $R_1 = g(x) \rightarrow f(a,x)$ Pos = Δ $\sigma =$
	$f(a,x)$ $R_2 = f(y,x) \rightarrow y$ Pos = Δ $\sigma = \{y \rightarrow a\}$
	a

Figure 62: First critical pair convergence

$\langle g(g(x')), g(x') \rangle$	
$g(g(x'))$	$g(x')$
$R_1 = g(x) \rightarrow f(a, x)$ Pos = Λ $\sigma = \{x \rightarrow g(x')\}$	$R_1 = g(x) \rightarrow f(a, x)$ Pos = Λ $\sigma = \{x \rightarrow x'\}$
$f(a, g(x'))$	$f(a, x')$
$R_2 = f(y, x) \rightarrow y$ Pos = Λ $\sigma = \{x \rightarrow g(x'), y \rightarrow a\}$	$R_2 = f(y, x) \rightarrow y$ Pos = Λ $\sigma = \{x \rightarrow x', y \rightarrow a\}$
a	a

Figure 63: Second critical pair convergence

- *Critical pairs*: It lists all critical pairs and their main properties (trivial, convergent, etc...):

Critical Pairs	
$\langle a, h(x, x, a) \rangle$	Convergent
$\langle g(g(x')), g(x') \rangle$	Convergent
$\langle f(f(a, x), a), g(g(x)) \rangle$	Convergent

Table 10: Critical pairs

- *TRS Properties*: Finally we get the properties of the term rewriting system. If the nontermination has been concluded, the infinite loop is printed out:

TRS
The TRS is Left-Linear
The TRS is not Right-Linear
The TRS is not Linear
The TRS is Collapsing
The TRS is Duplicating
The TRS is not Conservative
The TRS is Destructive
The TRS is not Orthogonal
The TRS is not Almost Orthogonal
The TRS is not Weakly Orthogonal
The TRS is Locally Confluent
Unknown confluence for The TRS
The TRS is non terminating Infinite Loop: $h(x,x,a) \rightarrow g(x) \rightarrow f(a,x) \rightarrow \underline{h(x,x,a)}$

Table 11: TRS properties

5.2.6 Term Execution

If a term has been specified, the rewrite rules are applied over it. To avoid an infinite loop and breaking down the server with non terminating term rewriting systems and infinite computations, a limit of rewrites is specified by the user when the process is launched. If the process finds a normal form, the detailed trace is shown.

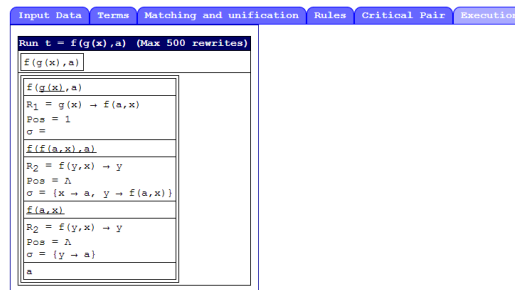


Figure 64: Term execution tab

For each term specified in the TERMS section of the TPDB file, a table is created. The first row of the table shows the root node of the execution tree, which is the term itself:

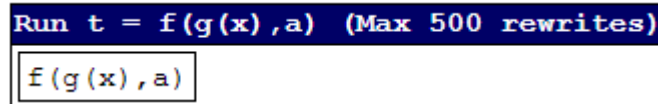


Figure 65: Term execution root node

If you click on any node of the tree execution, it is expanded and shows the terms that can be reached in one step by applying the rules of the term rewriting system:

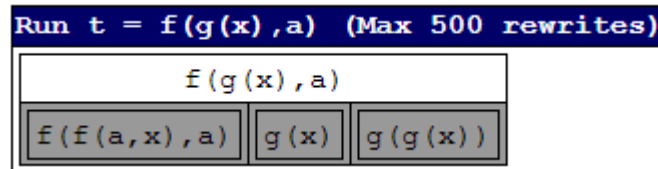


Figure 66: Term execution root node expansion

The nodes that result from this expansion are shown in gray color background.

If we put the cursor over any node of the execution tree, a tool tip is shown with the rule, the position and the matching substitution that are used in the reduction step.

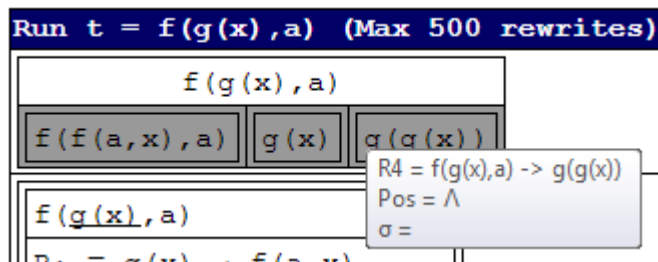


Figure 67: Term execution node tool tip

When we reach a node that previously occurred in the tree, it is flagged as deforested, so you can't continue with this branch. Deforested nodes are represented in red color background.

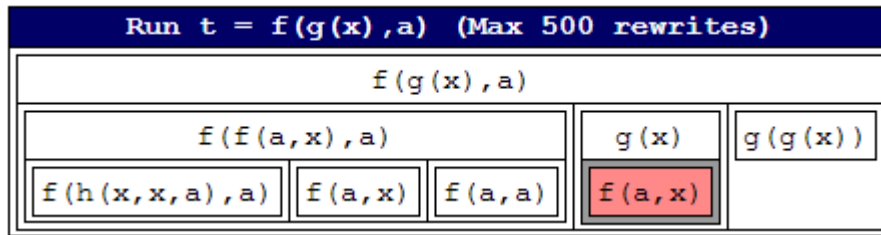


Figure 68: Term execution with deforested node

When we reach a node such that no rules apply (the term is in normal form), it is represented in green color background.

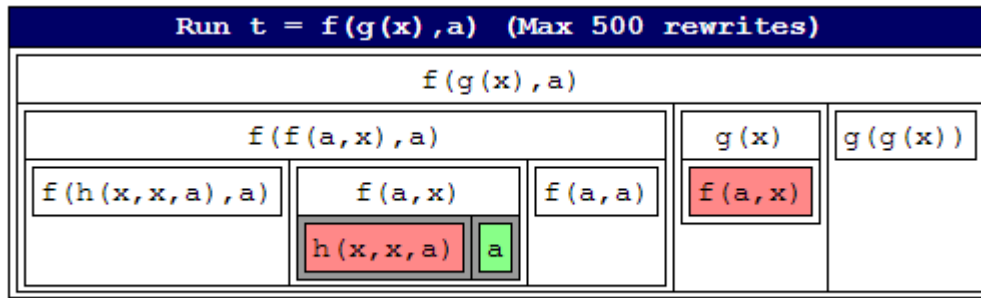


Figure 69: Term execution canonical node

Since the tree explores all possible rewrites, the full tree can be huge:

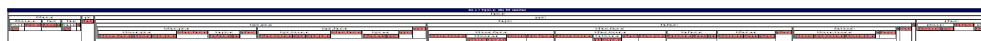


Figure 70: Full term execution tree

If we have expanded too much the tree, we can click again over any node to collapse the branches we want to hide

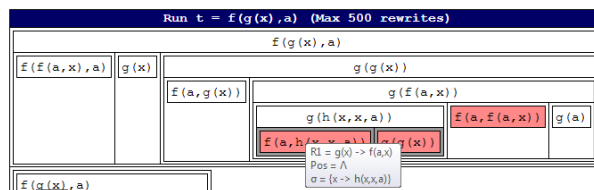


Figure 71: Term execution tree

Finally, for each normal form the detailed rewrite sequence is shown:

$f(g(x), a)$
$R_1 = g(x) \rightarrow f(a, x)$
$Pos = 1$
$\sigma =$
$f(f(a, x), a)$
$R_2 = f(y, x) \rightarrow y$
$Pos = \Lambda$
$\sigma = \{x \rightarrow a, y \rightarrow f(a, x)\}$
$f(a, x)$
$R_2 = f(y, x) \rightarrow y$
$Pos = \Lambda$
$\sigma = \{y \rightarrow a\}$
a

Figure 72: Normalizing form rewrite sequence

Only the first occurrence of each different normal form is shown. In the example, the normal form a can be reached in different ways, but only one of them is shown. If the term rewriting system is not confluent, then there is the possibility to get different normal forms, and a rewrite sequence for each of them will be shown.

Example 15: Confluence Analysis.

Consider again the term rewriting system in Example 1:

$$add(0, x) \rightarrow x \quad (120)$$

$$add(s(x), y) \rightarrow s(add(x, y)) \quad (121)$$

$$prod(0, x) \rightarrow 0 \quad (122)$$

$$prod(s(x), y) \rightarrow add(y, prod(x, y)) \quad (123)$$

$$fact(0) \rightarrow s(0) \quad (124)$$

$$fact(s(x)) \rightarrow prod(s(x), fact(x)) \quad (125)$$

You can test it with TRS.Tool at:

<http://TRS.JarCode.Net/?t=-1>

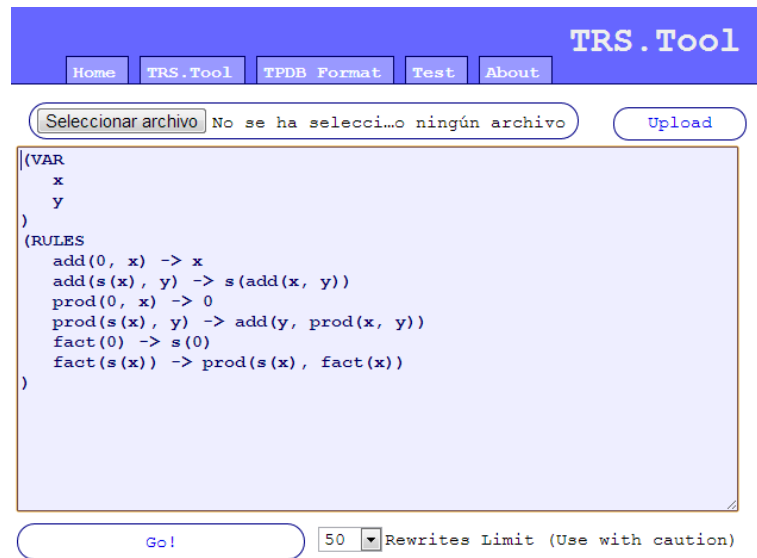


Figure 73: Example 1

If we test the term rewriting system with TRS.Tool, we can check that it has no critical pairs and it is actually confluent, due to its orthogonality.

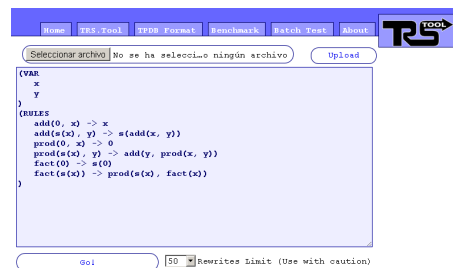


Table 12: Example 1 properties

Now, if we add the new rule (9):

<http://TRS.JarCode.Net/?t=-2>

No se ha seleccionado ningún archivo

```

(VAR
  x
  y
)
(RULES
  add(0, x) -> x
  add(s(x), y) -> s(add(x, y))
  prod(0, x) -> 0
  prod(s(x), y) -> add(y, prod(x, y))
  fact(0) -> s(0)
  fact(s(x)) -> prod(s(x), fact(x))
  prod(s(0), y) -> y
)

```

Rewrites Limit (Use with caution)

Figure 74: Example 1 “optimized” version

The TRS.Tool finds a non convergent critical pair, so the term rewriting system is not confluent.

Critical Pairs	
< y , add(y,prod(0,y)) >	Non Convergent

Table 13: Example 1 “optimized” version critical pairs

TRS
The TRS is Left-Linear
The TRS is not Right-Linear
The TRS is not Linear
The TRS is Collapsing
The TRS is Duplicating
The TRS is not Conservative
The TRS is Destructive
The TRS is not Orthogonal
The TRS is not Almost Orthogonal
The TRS is not Weakly Orthogonal
The TRS is not Confluent
The TRS is not Locally Confluent
Unknown terminator for The TRS

Table 14: Example 1 “optimized” version properties

Example 16: Termination Analysis.

Consider again the term rewriting system in Example 2:

$$\text{add}(0, x) \rightarrow x \quad (126)$$

$$\text{add}(s(x), y) \rightarrow s(\text{add}(x, y)) \quad (127)$$

$$\text{prod}(0, x) \rightarrow 0 \quad (128)$$

$$\text{prod}(s(x), y) \rightarrow \text{add}(y, \text{prod}(x, y)) \quad (129)$$

You can test it with TRS.Tool at:

<http://TRS.JarCode.Net/?t=-3>

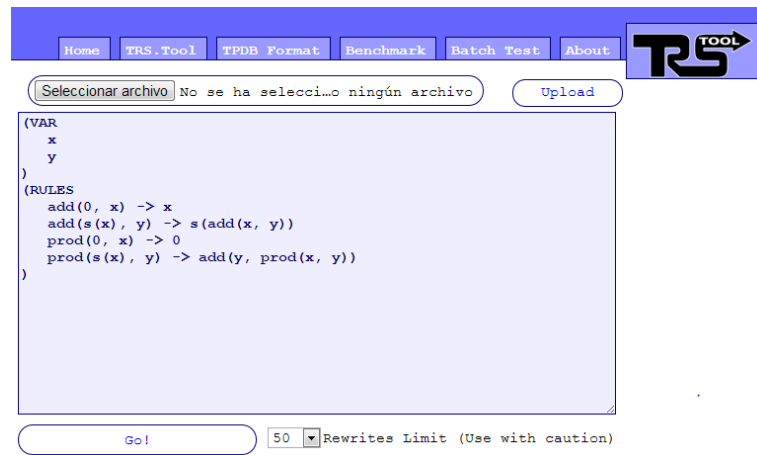


Figure 75: Example 2

If we test the term rewriting system with TRS.Tool, we can check that it has no critical pairs and it is actually confluent, due to its orthogonality.

TRS
The TRS is Left-Linear
The TRS is not Right-Linear
The TRS is not Linear
The TRS is Collapsing
The TRS is Duplicating
The TRS is not Conservative
The TRS is Destructive
The TRS is Orthogonal
The TRS is Almost Orthogonal
The TRS is Weakly Orthogonal
The TRS is Confluent
The TRS is Locally Confluent
Unknown terminator for The TRS

Table 15: Example 2 properties

Now, if we add the new rule (14):

<http://TRS.JarCode.Net/?t=-4>

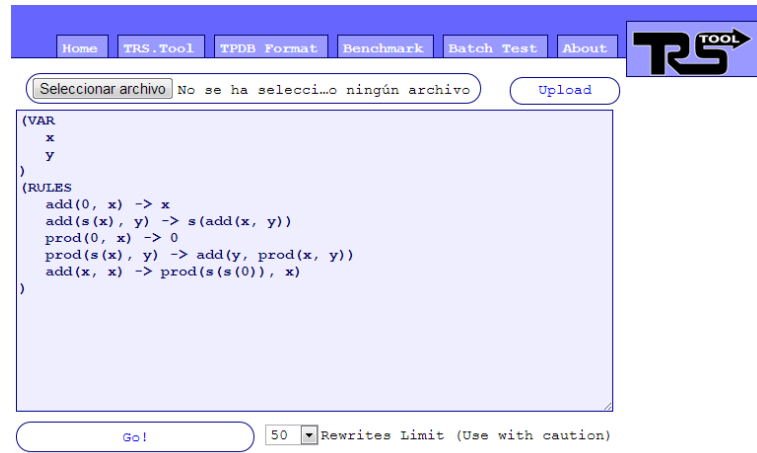


Figure 76: Example 2 “optimized” version

The TRS.Tool finds a non convergent critical pair, so the term rewriting system is not confluent.

Critical Pairs	
$\langle \text{prod}(s(s(0)), 0), 0 \rangle$	Convergent
$\langle \text{prod}(s(s(0)), s(x)), s(\text{add}(x, s(x))) \rangle$	Non Convergent

Table 16: Example 2 “optimized” version critical pairs

Furthermore, TRS.Tool detects the new system is nonterminating.

TRS
The TRS is not Left-Linear
The TRS is not Right-Linear
The TRS is not Linear
The TRS is Collapsing
The TRS is Duplicating
The TRS is not Conservative
The TRS is Destructive
The TRS is not Orthogonal
The TRS is not Almost Orthogonal
The TRS is not Weakly Orthogonal
The TRS is not Confluent
The TRS is not Locally Confluent
The TRS is non terminating
Infinite Loop: $\text{prod}(s(s(0)), 0) \rightarrow \text{add}(0, \text{prod}(s(0), 0)) \rightarrow \text{add}(0, \text{add}(0, \text{prod}(0, 0))) \rightarrow \text{add}(0, \text{add}(0, 0)) \rightarrow \text{add}(0, \text{prod}(s(s(0)), 0))$

Table 17: Example 2 “optimized” version properties

CHAPTER 6

TESTING THE FRAMEWORK

In order to test the correct functionality of the framework, we have decided to use the set of term rewriting systems from the Confluence Competition ¹. For this test we have set 500 as the limit of rewrites.

The test set contains 216 term rewriting system tagged with their properties. After trying them with TRS.Tool, we obtain the following results for confluence and local confluence:

	CoCo Tag	TRS.Tool	<i>Success ratio</i>
Confluent	171	5	<i>2.9 %</i>
Non Confluent	27	14	<i>51.9 %</i>
Success	198	19	<i>9.6 %</i>
Unknown	18	199	
Total	216	216	

Table 18: Confluence property test

These results are not so impressive because, currently, TRS.Tool has only one way to check confluence of a system: the check for orthogonality properties (and in the set of problems there are only 5 orthogonal systems).

¹<http://termcomp-devel.uibk.ac.at/hzank1/cops/>

	CoCo Tag	TRS.Tool	<i>Success ratio</i>
Locally Confluent	163	201	<i>123.3 %</i>
Non Locally Confluent	12	12	<i>100.0 %</i>
Success	175	213	<i>121.7 %</i>
Unknown	9	3	
Total	216	216	

Table 19: Locally Confluence property test

The target of the Confluence Competition is proving confluence rather than local confluence. In contrast, TRS.Tool determines the local confluence of some systems that have this information missed on the set.

TRS.Tool determines the local confluence of 38 systems that are not tagged as locally confluent:

- 31 (1, 19, 20, 22...): Tagged as confluent, but not as locally confluent. Probably, since confluence implies local confluence, the tagging for local confluence was just missed as directly entailed from positive confluence tag.
- 2 (126, 216): Have no tags about confluence or local confluence, but in the comments it is said that they are confluent.
- 3 (21, 212, 214): Tagged as nonconfluent and not as locally confluent.
- 2 (62, 175): No tag about confluence nor local confluence.

	CoCo Tag	TRS.Tool	<i>Success ratio</i>
Terminating	35	0	<i>0.0 %</i>
Non Terminating	181	129	<i>71.3 %</i>
Success	216	129	<i>59.7 %</i>
Unknown	0	87	
Total	216	216	

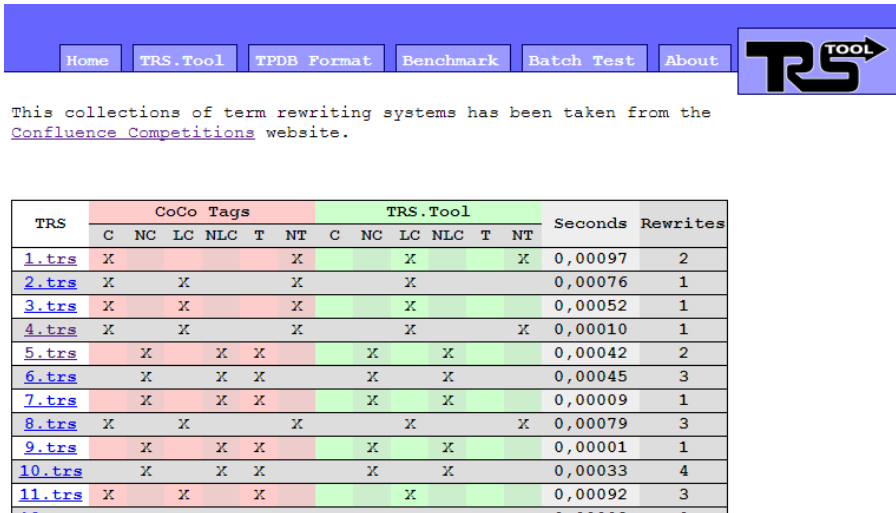
Table 20: Terminating property test

TRS.Tool implements a method to detect nonterminating systems (see section 4.3 and Examples 13 and 14), and it was able to discover 71.3% of them.

A detailed report of the test can be viewed on the website:

<http://TRS.JarCode.Net/Test.html>

Here you can see the individual result for each test, and the result for each of those properties: confluence, nonconfluence, local confluence, non-local confluence, termination and nontermination.

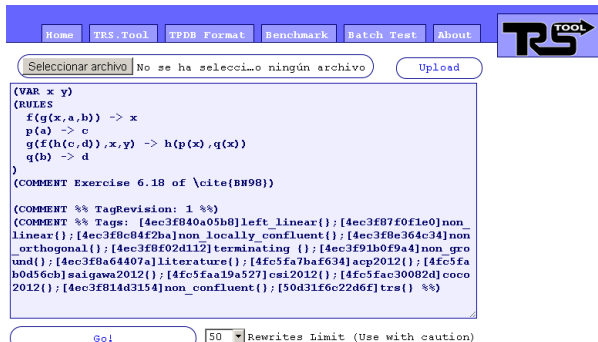


The screenshot shows the TRS Tool website interface. At the top, there is a navigation menu with buttons for Home, TRS.Tool, TPDB Format, Benchmark, Batch Test, and About. To the right is the TRS TOOL logo. Below the menu, a text block states: "This collections of term rewriting systems has been taken from the [Confluence Competitions](#) website." Below this is a table with the following structure:

TRS	CoCo Tags						TRS.Tool						Seconds	Rewrites
	C	NC	LC	NLC	T	NT	C	NC	LC	NLC	T	NT		
1.trrs	X					X		X			X		0,00097	2
2.trrs	X					X		X			X		0,00076	1
3.trrs	X		X			X		X			X		0,00052	1
4.trrs	X		X			X		X			X		0,00010	1
5.trrs		X		X	X			X		X			0,00042	2
6.trrs		X		X	X			X		X			0,00045	3
7.trrs		X		X	X			X		X			0,00009	1
8.trrs	X		X			X		X			X		0,00079	3
9.trrs		X		X	X			X		X			0,00001	1
10.trrs		X		X	X			X		X			0,00033	4
11.trrs	X		X		X			X			X		0,00092	3
12.trrs	X		X		X			X			X		0,00000	0

Figure 77: Test detailed report

If you click on the name of one file tested, you can test it again with TRS.Tool.



The screenshot shows the TRS Tool interface with the file 5.trrs loaded. The interface includes a navigation menu, a file selection area with a "Seleccionar archivo" button and an "Upload" button, and a text area containing the test file's content:

```
(VAR x y)
(RULES
  f(g(x,a,b)) -> x
  p(x) -> c
  g(f(h(c,d),x,y) -> h(p(x),q(x))
  q(b) -> d
)
(COMMENT Exercise 6.18 of \cite{BN98})

(COMMENT %% TagRevision: 1 %%)
(COMMENT %% Tags: [4ec3f84a05b8]left_linear();[4ec3f87f0f1e0]non_linear();[4ec3f8c84f2ba]non_locally_confluent();[4ec3f8e364c34]non_orthogonal();[4ec3f8f02d112]terminating ();[4ec3f91b0f9a4]non_gro_mad();[4ec3f9a64407a]l1_termination();[4fc5fa7baf634]acp2012();[4fc5fab0d5fcb]saigawa2012();[4fc5fae19a527]csi2012();[4ec5fac300694]coco2012();[4ec3f814d3154]non_confluent();[50d31f6c22a6f]trs() %%)
```

At the bottom, there is a "Go!" button and a "Rewrites Limit (Use with caution)" dropdown menu set to "50".

Figure 78: 5.trrs test loaded

The column “Seconds” indicates the amount of time the analysis of the term rewriting system takes. As the precision of the Stopwatch object

from .Net which is used to measure the time is 1 millisecond, and some tests take less than that, we have done each test 10 times and calculate the average time. The last column “Rewrites” indicates how many rewrites were necessary to determine the (non)convergence of all critical pairs.

We have removed system number 15 from the test, because it is a complex term rewriting system with 15 critical pairs, and some of them reach the limit of 500 rewrites without any convergence. The time for this one took almost 2 hours and the TRS.Tool couldn’t determine any property. It will be necessary to create a timer to cut the execution of the process and give a timeout exception.

12.trs	△	△	△	△	△	△	△	△	△	0,00008	0	
13.trs	X	X	X	X	X	X	X	X	X	0,00040	4	
14.trs	X	X	X	X	X	X	X	X	X	0,00026	3	
15.trs	-	-	-	-	-	X	-	-	-	-	7193,3580	3036
16.trs	X	X	X	X	X	X	X	X	X	0,00015	0	
17.trs	X	X	X	X	X	X	X	X	X	0,00062	0	
18.trs	X	X	X	X	X	X	X	X	X	0,00011	0	

Figure 79: 15.trs results

At the end of the table there is a row with the total values. The average time for the 215 systems was 239 milliseconds. This value could be lower, because of the educational objective of the tool, it analyses all the critical pairs. But this is not necessary: if we find a notconvergent critical pair, then it is not necessary to check the rest and we can get an answer early.

211.trs	△	△	△	△	△	△	△	△	△	0,00247	0			
212.trs	X	X	X	X	X	X	X	X	X	0,00028	3			
213.trs	X	X	X	X	X	X	X	X	X	0,00060	4			
214.trs	X	X	X	X	X	X	X	X	X	0,02617	4			
215.trs	X	X	X	X	X	X	X	X	X	0,03196	4			
216.trs	X	X	X	X	X	X	X	X	X	0,04908	49			
Total	171	27	163	12	35	180	5	14	201	12	0	129	51,35253	6374

Figure 80: Test total values

Here is a dispersion graph of the time which is necessary to get the result in milliseconds. Note that only 5 examples required more than 1 second.

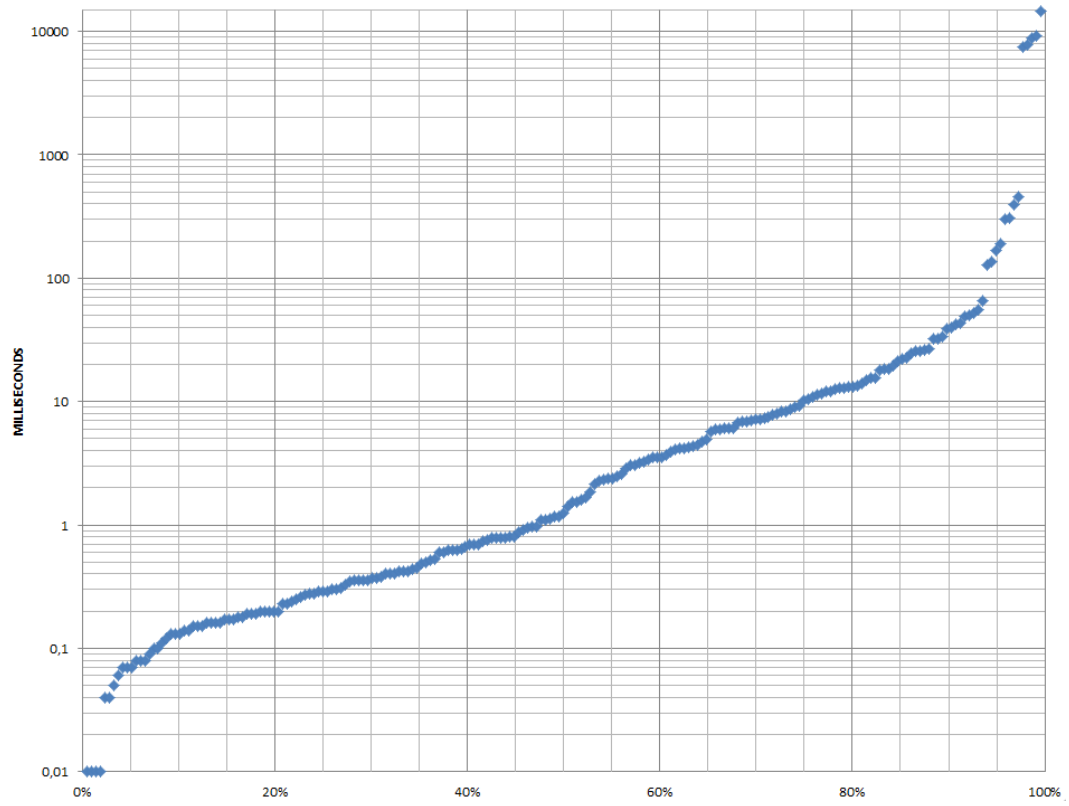


Figure 81: Test dispersion graph

CHAPTER 7

CONCLUSIONS

The main goal of this thesis is the implementation of a framework that enables the user to work with term rewriting systems, providing these functionalities:

1. Determine the basic structure and syntactic properties of a given term rewriting system.
2. Compute the critical pairs of the term rewriting system and determine its orthogonality, local confluence and confluence.
3. Compute resolution sequences for term rewriting.

Another important target of the thesis is the development of a web tool which is suitable to be used in teaching the main rewriting concepts.

Our implementation fits the goals, but there are a lot of additional developments that can be included and on some points can be optimized.

One of the first targets that should be addressed in the future is the optimization of the execution module to make it more smart. Currently, almost all branches of the execution tree are explored and the deforestation method is quite poor. It should be improved to detect loops of non-terminating term rewriting systems. As a second step in the optimization of the execution

module we should define heuristics to determine the best rewriting rule to be applied in order to get a given term or a normal form.

Another line of future work is the verification of more properties for the term rewriting systems: reducibility, termination, correctness, etc... Termination is especially attractive due to its relevance and use in confluence tests.

Finally I think the methods implemented to decide confluence can be applied to a very restricted subclass of term rewriting system (weakly orthogonal and terminating). Extended methods that apply to more general TRSs could be implemented in the future

The current version of the tool can be used as a starting point for a more powerful tool if the development continues. Also, TRS.Tool gives much more intermediate information about the process of the analysis of the critical pairs than other existing tools. This is very useful for the students to assimilate the basic concepts of this topics and getting involved in the field. Actually, this was the main motivation to start this project and implement TRS.Tool.

BIBLIOGRAPHY

- [1] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.
- [2] Marc Bezem, Jan Willem Klop, and Roel de Vrijer. *Term Rewriting Systems by “Terese”*. Cambridge Tracts in Theoretical Computer Science, Vol 55, Cambridge University Press, 2003.
- [3] Gérard Huet. Confluent reductions: Abstract properties and applications to term rewriting systems: Abstract properties and applications to term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.
- [4] Gérard Huet and Jean-Jacques Lévy. Computations in orthogonal rewriting systems, i and ii. *Computational Logic, Essays in Honor of Alan Robinson*, pages 396–443, 1991.
- [5] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.
- [6] Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer Publishing Company, Incorporated, 1st edition, 2002.
- [7] M. S. Paterson and M. N. Wegman. Linear unification. In *Proceedings of the eighth annual ACM symposium on Theory of computing*, STOC ’76, pages 181–186, New York, NY, USA, 1976. ACM.