Universidad Politécnica de Valencia
Departamento de Sistemas Informáticos y Computación
Doctorado en Informática

Ph.D. Thesis

Rule-based Software Verification and Correction

Candidate:
Demis Ballis

Supervisors:
Prof. María Alpuente
Prof. Moreno Falaschi

June 2006

Author's e-mail: dballis@dsic.upv.es

Author's address:

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera, s/n
46022 Valencia
España

To my grandfather Pino

Abstract

The increasing complexity of software systems has led to the development of sophisticated formal methodologies for verifying and correcting data and programs. In general, establishing whether a program behaves correctly w.r.t. the original programmer's intention or checking the consistency and the correctness of a large set of data are not trivial tasks as witnessed by many case studies which occur in the literature.

In this dissertation, we face two challenging problems of verification and correction. Specifically, the verification and correction of declarative programs, and the verification and correction of Web sites (i.e. large collections of semistructured data).

Firstly, we propose a general correction scheme for automatically correcting declarative, rule-based programs which exploits a combination of bottom-up as well as top-down *inductive learning* techniques. Our hybrid methodology is able to infer program corrections that are hard, or even impossible, to obtain with a simpler, automatic top-down or bottom-up learner. Moreover, the scheme will be also particularized to some well-known declarative programming paradigm: that is, the functional logic and the functional programming paradigm.

Secondly, we formalize a framework for the automated verification of Web sites which can be used to specify integrity conditions for a given Web site, and then automatically check whether these conditions are fulfilled. We provide a rule-based, formal specification language which allows us to define syntactic as well as semantic properties of the Web site. Then, we formalize a verification technique which detects both incorrect/forbidden patterns as well as lack of information, that is, incomplete/missing Web pages. Useful information is gathered during the verification process which can be used to repair the Web site. So, after a verification phase, one can semi-automatically infer some possible corrections in order to fix the Web site. The methodology is based on a novel rewriting-like technique, called *partial rewriting*, in which the traditional pattern matching mechanism is replaced by a more suitable technique (*tree simulation*) for recognizing patterns inside semistructured documents.

Sommario

La crescente complessità dei sistemi software ha reso necessario lo sviluppo di sofisticate metodologie formali per la verifica e la correzione di dati e programmi. In generale, stabilire se un programma si comporta correttamente rispetto alle intenzioni originali del programmatore o controllare la consistenza e la correttezza di grandi insiemi di dati non sono compiti triviali, come è testimoniato dai numerosi casi di studio presenti in letteratura.

In questa tesi affrontiamo due interessanti problemi di verifica e correzione: la verifica e correzione di programmi dichiarativi e la verifica e correzione di siti Web (i.e. collezioni di dati semistrutturati).

In primo luogo, proponiamo uno schema generale per la correzione automatica di programmi dichiarativi *rule-based*, il quale sfrutta una combinazione di tecniche di *inductive learning* sia *top-down* che *bottom-up*. La nostra metodologia ibrida è in grado di inferire correzioni particolarmente ardue, o addirittura impossibili, da ottenere con un più semplice sistema di *learning* puramente *top-down* o *bottom-up*. Inoltre, istanzieremo lo schema a due ben noti paradigmi di programmazione dichiarativa: il paradigma logico funzionale e il paradigma funzionale.

In secondo luogo, formalizzeremo un *framework* per la verifica automatica di siti Web che può essere utilizzato per specificare alcune condizioni di integrità di un sito, le quali successivamente possono essere controllate automaticamente. Forniremo un linguaggio di specifica *rule-based* che ci permette di definire proprietà sia sintattiche che semantiche di un sito Web. Quindi, formalizzeremo una tecnica di verifica in grado di riconoscere modelli scorretti/vietati come pure pagine Web incomplete/mancanti. Il processo di verifica permette di raccogliere informazioni che possono essere utili per la riparazione del sito Web. Pertanto, al termine di tale fase, è possibile inferire alcune correzioni in maniera semi-automatica. La nostra metodologia si basa su una nuova tecnica *rewriting-like*, chiamata *partial rewriting*, la quale sostituisce il tradizionale meccanismo di *pattern matching* con una tecnica di *matching* più conveniente (*tree simulation*) che facilita il riconoscimento di modelli in un documento semistrutturato.

Resumen

La creciente complejidad de los sistemas software ha conducido al desarrollo de metodologías formales para la verificación y la corrección de datos y programas. Generalmente, establecer si un programa se comporta según las intenciones originales del programador o controlar la consistencia y la corrección de grandes conjuntos de datos no son tareas triviales, como atestiguan los numerosos casos de estudio que encontramos en la bibliografía.

En esta tesis, abordamos dos problemas abiertos de verificación y corrección. En concreto, la verificación y corrección de programas declarativos y la verificación y corrección de sitios Web (es decir, conjuntos de datos semiestructurados).

En primer lugar, se ha definido un esquema general para la corrección automática de programas declarativos basados en reglas, que explota una combinación de técnicas de aprendizaje inductivo *top-down* y *bottom-up*. Nuestra metodología híbrida es capaz de inferir correcciones que son arduas, o incluso imposibles, de conseguir con un sistema más simple de aprendizaje automático puramente *top-down* o *bottom-up*. Además se ha particularizado el esquema general a dos paradigmas de programación declarativa bien conocidos: el paradigma lógico funcional y el paradigma funcional.

En segundo lugar, se ha formalizado un marco para la verificación automática de sitios Web, que se puede usar para especificar condiciones de integridad sobre ellos, y luego comprobar automáticamente si estas condiciones se satisfacen. Por un lado, hemos definido un lenguaje de especificación basado en reglas, que permite definir propiedades tanto sintácticas como semánticas de un sitio Web. Por otro lado, se ha formalizado una técnica de verificación que detecta patrones incorrectos/prohibidos y carencia de información, es decir pàginas Web incompletas o ausentes. Durante el proceso de verificación, se recoge información útil, que puede ser usada para la reparación del portal. Por lo tanto, después de la fase de verificación, también es posible inferir algunas posibles correcciones para arreglar de manera semi-automática el sitio Web erróneo. Nuestra metodología se fundamenta en una nueva técnica basada en reescritura (*partial rewriting*), en la cual se remplaza el tradicional mecanismo de *pattern matching* con una técnica de ajuste más conveniente (*tree simulation*) que facilita el reconocimiento de patrones en un documento semiestructurado.

Resum

La creixent complexitat dels sistemes software ha conduït al desenvolupament de metodologies formals per a la verificació i la correcció de dates y programes. Generalment, establir si un programa es comporta seguint les intencions originals del programador o controlar la consistència i la correcció de grans conjunts de dates, no son tarees trivials, com manifesten els numerosos casos d'estudio que trobem en la bibliografia.

En aquesta tesi, abordem dos problemes oberts de verificació i correcció. En concret, la verificació i correcció de programes declaratius i la verificació i correcció de *Web site* (es decir, conjunts de dates semiestructurats).

En primer lloc, s'ha definit un esquema general per la correcció automàtica de programes declaratius basats en regles, que exploten una combinació de técniques de aprenentatge inductiu *top-down* i *bottom-up*. La nostra metodologia híbrida pot inferir correccions que son àrdues, o inclòs imposibles, de aconseguir con un sistema més simple d'aprenentatge automàtic purament *top-down* o *bottom-up*. A més més s'ha particularitzat l'esquema general a dos paradigmes de programació declarativa ben coneguts: el paradigma lògic funcional i el paradigma funcional.

En segon lloc, s'ha formalitzat un marc per la verificació automàtica de *Web sites*, que es pot utilitzar per la especificació de condicions de integritat d'ells, i després comprovar automàticament si estes condicions es satisfeixen. D'una banda, hem definit un llenguatge d'especificació basat en regles, que permeteixen la definició de propietats sintáctiques i semàntiques de un *Web site*. De l'altra, s'ha formalitzat una tècnica de verificació que detecta patrons incorrectes/prohibits i carència d'informació, per eixemple pàgines Web incompletes o mancants. Durant el procés de verificació, es recol·lecteix l'informació útil, que pot ser utiltzata per la reparació del *Web site*. Per tant, després de la fase de verificació, també se poden inferir algunes posibles correccions per a arranjar semi-automàticament el *Web site* erroni. La nostra metodologia descansa en una nova tècnica basata en *rewriting*, en la qual es sustitueix el tradicional mecanisme de *pattern matching* con una tècnica de ajustatge més convenient (*tree simulation*) que simplifica el reconeixement de patrons en un document semiestructurat.

Acknowledgments

First of all, I would like to express my gratitude to my advisors Moreno Falaschi and María Alpuente for their dedication and patience. They showed me the beauty and the complexity of the *declarative world*, giving me the opportunity to work in this marvelous research area. I owe most of what I have learned to them and I will never be able to thank them enough.

I also wish to acknowledge my Italian and Spanish colleagues and friends who shared with me rooms, sorrow and happiness throughout my PhD course. Sometimes, I wonder how they were able to stand me. So, thanks to Nicola, Marco, Davide, Alicia, Josep, Vicent, Claudio, Santi, and to everybody I missed who deserves a mention.

My deep gratitude also goes to Sofía. She taught me not to take myself too seriously, since *Life is a carnival*. I think I learned the lesson and I am sure I will never forget it.

I am infinitely grateful to Gute. Her smiling face provided me the energy for the final rush. I feel so lucky to have met such a positive, and optimistic person, who is always ready to solve my problems by cooking tasteful *cepelinai*. Thanks, Gute!

Finally, I must thank my family for the untiring support and the unconditional trust. They have always encouraged me to follow my dreams. Thanks mom, dad, and Gigi. You do not know how much you helped me.

May, 2005.

Demis Ballis

Contents

Introduction v

I Automated Program Correction 1

1 A Multiparadigm Correction Scheme 3
1.1 Exploiting debugger outcomes . . . 4
1.2 Inductive learning . . . 5
1.3 Correction scheme . . . 6

2 Preliminaries 9
2.1 Foundations . . . 9
2.1.1 Terms and equations . . . 9
2.1.2 Substitutions and syntactic unification . . . 10
2.1.3 $\mathcal{V}$-Herbrand base and program semantics . . . 11
2.2 Programs as term rewriting systems . . . 11
2.3 The narrowing relation . . . 12
2.4 Conditional programs and narrowing . . . 14

3 Correction of Functional Logic Programs 17
3.1 Denotation of functional logic programs . . . 19
3.2 Diagnosis of declarative programs . . . 20
3.3 Correction method . . . 21
3.3.1 Automatic generation of positive and negative example sets . . . 21
3.3.2 Specialization operators . . . 22
3.3.3 Top-down correction algorithm . . . 23
3.3.4 Correctness of the algorithm . . . 24
3.4 Improving the algorithm . . . 25
3.5 Automated correction system . . . 29
3.5.1 Experimental evaluation . . . 30

4 Correction of First-Order Functional Programs 31
4.1 Denotation of functional programs . . . 33
4.1.1 Concrete semantics . . . 33
4.1.2 Abstract semantics . . . 35
4.2 The correction problem . . . 36
4.3 How to generate example sets automatically . . . 36
4.4 Example-guided unfolding . . . 37

4.4.1 The unfolding operator 38
4.4.2 The top-down correction algorithm 39
4.4.3 Correctness of algorithm TDCorrectorF 42

II Web Site Verification 45

5 A Language for Verifying Web sites 47
5.1 Basic notions 48
5.2 Denotation of Web sites 51
5.3 Web specification language 53
5.4 Partial rewriting 56
5.4.1 Page simulations 56
5.4.2 Rewriting Web page templates 59
5.5 The verification framework 61
5.5.1 Detecting correctness errors 61
5.5.2 Detecting completeness errors 64
5.6 Implementation 71

Conclusions 73

A Context Sensitive Rewriting and Over-Generality 77
A.1 Deciding over-generality by context sensitive rewriting 77
A.1.1 Context Sensitive Rewriting 77
A.1.2 Testing $\mathcal{R} \vdash E^+$ 78
A.1.3 Extending the decision algorithm to CTRSs 81

B Some technicalities 83
B.1 Proofs of the technical results of Chapter 3 83
B.2 Proofs of the technical results of Chapter 4 87
B.3 Proofs of the technical results of Chapter 5 91

Bibliography 101

List of Figures

1.1 Automated generation of incorrectness and incompleteness symptoms. 4
1.2 Hybrid correction scheme. 7

4.1 Coin-flip specification. 34
4.2 Wrong coinflip OBJ program. 41
4.3 Unfolded coinflip program. 42

5.1 Term representation of $f(h(a), X)$. 49
5.2 Examples of valid and non valid markings for the term $f(h(a), X)$. . . 50
5.3 An XML document and its corresponding encoding as a ground term $\mathtt{p}$. 51
5.4 An example of Web site 52
5.5 Page simulation between $\mathtt{p}_1$ and $\mathtt{p}_2$. 57
5.6 Non-minimal and minimal simulations. 59
5.7 Minimal non-injective simulation 59
5.8 A screenshot of the system. 71

Introduction

Data and programs are the basic ingredients of software systems. In order to solve a problem, we need programs which behave in a *correct* way w.r.t. the programmer's intention and which operate on *correct* sets of data. This consideration has lead several researchers to tackle the problem of verifying and correcting data and programs. In this dissertation, we will try to give a little contribution to this area, developing methodologies for automatically verifying and correcting programs, specifically functional and functional logic programs. Moreover, we will also focus on the verification of properties, such as integrity constraints, on large collections of data. We will formalize a high level language to detect and repair possible data inconsistencies within Web sites (i.e sets of semistructured data, e.g., XML/HTML documents).

Verifying and correcting programs

Program debugging has always played an important role in software development and much effort has been spent in formalising diagnosis and bug-locating techniques. As a matter of fact, every programming language is nowadays equipped with debugging facilities which can aid users to develop error-free programs at different levels of automation. Moreover, these tools permit to avoid laborious, time-expensive proof-reading sessions.

Especially in the context of declarative programming (functional, logic and functional logic programming) a lot of diagnosis methodologies have been successfully developed in the last thirty years, giving rise to a research area called *declarative diagnosis*, whose main aim consists in trying to automate the debugging process of declarative programs as much as possible. The diagnosis problem has been addressed following a plethora of distinct approaches. However, a rough taxonomy of the debugging methodologies might be given as follows.

- Tracing methodologies;
- Oracle-guided methodologies;
- Bottom-up, immediate consequence $T_{\mathcal{R}}$-based methodologies.

The taxonomy mentioned above provides an increasing order of automation in the debugging process.

Roughly speaking, tracers allow us to visualize step-by-step the program execution, which might be useful in order to locate bugs, even if —due to the complexity of the operational semantics of declarative programs— the information obtained by tracing is often difficult to understand. Some declarative languages (e.g. ALF [61], Babel [84], Curry [64], and Haskell [106]) are equipped with tracing tools which are based

on suitable extended box models which help to display the run-time execution [63, 21]. Unfortunately, these diagnosis systems do not enforce program correctness adequately as they do not provide means for finding nor repairing bugs in the source code w.r.t. the intended program semantics that a programmer has in mind. So, in this case the diagnosis process is performed manually by analyzing data extrapolated from the execution of the program.

Oracle-guided methodologies work in a semi-automatic fashion, the analysis is carried out by means of an oracle (typically the user) being supposed to endow the debugger with error symptoms, as well as to correctly answer questions driven by proof trees aimed at locating the actual source of errors. The first approach of this kind was proposed by Shapiro in his enlightening paper [102], in which is explained an oracle-guided method to debug logic programs. This approach has had several refinements and was implemented not only for the logic programming paradigm. For instance, the functional logic programming language NUE-Prolog is endowed with a declarative debugger [88] which works in the style proposed by Shapiro [102] and a similar declarative debugger for the functional logic language Escher is proposed in [77]. Also for the functional programming paradigm, diagnosis systems have been developed, which are very close to the one proposed by Shapiro (e.g. [100, 88, 91]).

Finally, $T_{\mathcal{R}}$-based methodologies provide fully automatic methods for the declarative diagnosis, which are able to find out bugs for a given faulty program w.r.t. an intended semantics which is generally expressed by means of some kind of formal specification. They are based on the immediate consequence operator that is typically used to define program semantics in a fixpoint style. In the context of pure logic programming, [39] has defined a declarative framework for debugging which extends the methodology in [55, 102] to diagnosis w.r.t. computed answers. The framework does not require the determination of the symptoms in advance and is goal independent –it is driven by a set of most general atomic goals. The immediate consequences operator $T_{\mathcal{R}}$ allows to identify program bugs and has the advantage of giving a symptom–independent diagnosis method [39, 38]. In [13], a declarative diagnosis method w.r.t. computed answers has been developed which generalizes the ideas of [39] to the diagnosis of functional logic programs. The conditions which have been imposed on programs allow to define a framework for declarative debugging which works for programs with an eager (*call–by–value*) as well as lazy (*call–by–name*) semantics. A similar framework which exploits information given by an immediate consequence operator and which works for functional programs has been devised in [12].

Declarative diagnosis is only concerned with the localization of bugs in a faulty program. However another interesting problem, which is strictly related to debugging, is the (semi-)automated correction of detected errors. The problem can be formulated as follows.

> Let $\mathcal{R}$ be a wrong declarative program w.r.t. a given specification (which models the programmer's intentions) and $\mathcal{A}$ be a set of buggy rules which has been found by means of some kind of debugging methodology. Then, we aim at finding a set of new rules which can replace $\mathcal{A}$ in order to produce a corrected version of $\mathcal{R}$.

Surprisingly, such problem has not been systematically addressed by computer science researchers and this is witnessed by the lack of results in this area. In particular, to our knowledge there is no general framework which allows to integrate a debugging phase with an automatic bug correction. Some work in this direction was carried out by Shapiro in [102], nonetheless his method requires a strong interaction between the debugging system and the user, who has to provide example evidences, answering correctness questions, establishing equivalence classes among the rules, or manually correcting the buggy code. Consequently this framework results neither automatic nor easy to use. A more automatic approach to the correction of faulty programs has been investigated in the context of concurrent logic programming. In [2, 3], a framework for the diagnosis and the correction of Moded flat GHC programs [105] has been developed. This framework exploits strong mode/typing and constraint analysis in order to locate bugs; then, symbols which are likely sources of error are syntactically replaced by other program symbols, so that new slightly different programs (mutations) are produced. Finally, mutations are newly checked for correctness. This approach is essentially able to correct *near misses* (i.e wrong variable/constant occurrences), no mistakes involving predicate or function symbols can be repaired. So, the framework has a very limited correction power and, as a further drawback, it is designed for a too much specialized programming language.

Verifying and correcting data

The explosive development of the Internet and related information and communication technologies has brought into focus the problems of information overload: we live in an information-saturated environment, in which the management of the data is becoming a non trivial task.

In this scenario, the verification and the correction of the information assume a crucial role, which cannot be ignored. In particular, the increasing complexity of Web sites has turned the data verification problem into a challenging problem. As a matter of fact, it is far simpler to discover inconsistent information on the Web than to find a well-maintained site on the Internet. Web site design, construction and maintenance are phases which should be "engineered" in order to deliver consistent and trustable information.

In our opinion, systematic, formal approaches can bring many benefits to solve these problematics giving support for automated Web site verification.

Although the management of Web sites has received significant attention in recent years [31, 45, 53], few works address the semantic verification of Web sites. In [53], a declarative verification algorithm is developed which checks a particular class of integrity constraints concerning the Web site's structure, but not the contents of a given instance of the site. In [45], a methodology to verify some semantic constraints concerning the Web site contents is proposed, which consists of using inference rules and axioms of natural semantics. The framework XLINKIT [48, 89] allows one to check the consistency of distributed, heterogeneous documents as well as to fix the (possibly) inconsistent information. The specification language is a restricted form of first order

logic combined with Xpath expressions [108] where no functions are allowed. With respect to the correctness of web applications, a symbolic model-checking approach is formalized in [46] which constructs a finite-state model of the system in the model checker input language, and then checks the considered properties which are expressed in CTL logic. For a comprehensive survey about the general problem of checking constraints between multiple documents, we refer to [51, 47, 34].

We believe that an approach to Web site specification and verification, which is based on rewriting-like machinery can be fruitfully employed, since it can take advantage of the reasoning capabilities which are typical of the declarative programming world. Our idea is that term rewriting techniques can support in a natural way not only intuitive, high level Web site specification, but also efficient Web site verification techniques. As far as we know, rewriting-based techniques have not been explored in the context of Web site verification to date. We only know of two rewriting-based approaches for Web site processing, but they focus on transformation rather than verification issues: a term rewriting implementation is provided in [75] for (a fragment of) XSLT, the rule-based language designed by W3C for the transformation of XML documents, whereas rewrite rules are used in [24] to perform HTML transformations with the aim of improving Web applications by cleaning up syntax, reorganizing frames, or updating to new standards.

Contents of the thesis

This dissertation is divided in two parts. The former is mainly devoted to formalizing and developing some program correction methodologies, while the latter deals with the verification and the correction of Web sites, that is, collections of semistructured data (e.g., XHTML/XML documents). In the following, we briefly describe the contents of each part.

Part I: Automated Program Correction

This part presents some original methodologies for the automated correction of declarative programs. We propose a general correction scheme for correcting programs which exploits the *inductive learning* techniques. Inductive learning is concerned with the task of learning programs from positive and negative examples, generally in the form of ground evidences [87]. A challenging subfield of inductive learning is known as *inductive theory revision* which is close to program debugging under the *competent programmer* assumption of [102]. Actually, the relation between declarative debugging and inductive inference [87] has been investigated since Shapiro's influential work. In [102], the initial program is assumed to be written with the intention of being correct and, if it is not correct, then a close variant of it is. There exist several approaches to declarative diagnosis which may require ([102]) or not ([13]) an interaction with the user.

The automatic search for a new rule in an induction process can be performed either bottom-up (i.e. from an overly specific rule to a more general one) or top-

down (i.e. from an overly general rule to a more specific one). In [67, 68, 57, 85, 86], bottom–up frameworks for synthesizing correct programs in the logic and functional logic paradigm are presented. These methods induce declarative programs from sets of evidences (ground atoms, ground equations) which are respectively incorrect and correct w.r.t. the pursued program. Their methodology, however, is not particularly tailored for *theory revision*, and the uncontrolled application of the method produces much speculation in our framework. There are some reasons to prefer the top-down, or *backward reasoning* process to the bottom-up, or *forward reasoning* process [29]. On the one hand, it eliminates the need for navigating through all possible logical consequences of the program. On the other hand, it integrates inductive reasoning with the deductive process, so that the derived programs are guaranteed to be correct. Unfortunately, it is known that the deductive process alone is inadequate for coming up with the intended correction, and inductive generalization techniques are necessary [44, 99].

Our methodology is based on the combination, in a single framework, of a diagnoser which identifies those parts of the code containing errors, together with an inductive (as well as deductive) program learner which, once the bug has been located in the program, tries to repair it starting from examples which are essentially obtained as an outcome of the diagnoser. Thus, to fully profit from the advantages of top-down as well as bottom-up synthesis, a hybrid approach is taken which is able to infer program corrections that are hard, or even impossible, to obtain with a simpler, automatic deductive learner. The scheme will also be particularized to some well-known declarative programming paradigm: that is, the functional logic and the functional programming paradigm. Moreover, in order to test its effectiveness and practicability, the method has been implemented for the functional logic setting.

Part II: Web Site Verification

In this part, we formalize a framework for the automated verification of Web sites which can be used to specify integrity conditions for a given Web site, and then automatically check whether these conditions are fulfilled. First, we provide a rewriting-based, formal specification language which allows us to define syntactic as well as semantic properties of the Web site. Then, we formalize a verification technique which detects both incorrect/forbidden patterns as well as lack of information, that is, incomplete/missing Web pages, inside the Web site. Useful information is gathered during the verification process which can be used to repair the Web site. So, after a verification/debugging phase, one can also infer semi-automatically some possible corrections in order to fix the Web site.

Our rule specification language does offer the expressiveness and computational power of functions and is simpler than formalizations of XML schemata based on tree automata often used in the literature such as, e.g. the regular expression types [71].

The methodology is based on a novel rewriting-based technique, called *partial rewriting*, in which the traditional pattern matching mechanism is replaced by tree *simulation* [66], which is a suitable technique for recognizing patterns inside semistructured documents. Besides, the framework has been implemented in the prototype

VERDI (Verification and Rewriting for Debugging Internet sites) [11] —which is publicly available— in order to test its effectiveness and efficiency.

Thesis overview

The thesis is organized as follows.

Chapter 1 presents a generic, multiparadigm correction scheme [4], which can be instantiated to several declarative programming paradigms, e.g., the functional, logic, and functional logic programming paradigm. It is based on a hybrid inductive learning technique which can infer corrections starting from evidences of correct and wrong computations.

In Chapter 2, we introduce the basic notions and terminologies which are used throughout this dissertation.

In Chapter 3, we particularize the generic correction scheme of Chapter 1 to the functional logic programming paradigm. Specifically, we provide an inductive learning, example-guided correction methodology for *eager* as well as *lazy* functional logic languages. Besides, an experimental evaluation of the proposed methodology is presented. This chapter extends our previous work [6].

Chapter 4 describes an automated correction methodology for the correction of first-order functional programs (i.e., term rewriting systems), which can be seen as an instance of the scheme of Chapter 1. The method can be applied to a wide class of term rewriting systems which can also be non-confluent. This chapter describes the results we have presented in [7, 8].

In Chapter 5, we formalize a rewriting-like language for the specification of properties (e.g., integrity conditions) on Web sites and a verification technique which exploits a rewriting technology in order to detect those properties which are not fulfilled by a given input Web site. Our technique is able to detect incorrect as well as missing information and to provide suggestions to fix the site. The chapter describes the results we achieved in [10, 9]. The chapter also gives some information about the VERDI system [11], which implements the verification framework.

Then, in Chapter 6, some conclusions are drawn.

Finally, Appendix A describes *context sensitive rewriting* and how to apply it in order to prove the over-generality condition for the functional logic setting (see Chapter 3). Appendix B includes the proofs of the technical results which are presented in this thesis.

I

Automated Program Correction

1

A Multiparadigm Correction Scheme

Declarative programming is supported both by functional and by logic programming. However, each of these programming styles has different advantages w.r.t. practical applications. Functional languages provide sophisticated abstraction facilities, module systems and clean solutions for integrating I/O into declarative programming as well as for efficient program execution. Logic languages allow for computing with partial information and provide built-in search facilities which have strong applications for knowledge-based systems and operations research. However, recent results show that the advantages of these styles can be efficiently and usefully combined into a single language. Modern functional logic languages offer features from both styles.

All these declarative paradigms are now very mature, and each of them has applications to several different fields. There exist programming environments for each of these paradigms in order to assist the user in the process of design, development and debugging. However, to our knowledge there is no general framework which allows to integrate a debugging phase with an automatic correction of the bugs. We believe that such an integration can be quite important and draw new techniques and useful results for the process of synthesis of correct programs in all these paradigms.

Our goal: a generic, multiparadigm correction scheme

In this chapter, we outline a general correction scheme for synthesizing correct programs. Our methodology is based on the combination, in a single framework, of a diagnoser which identifies those parts of the code which contain errors, together with an inductive as well as deductive program learner which, once the bug has been located in the program, tries to repair it starting from evidence examples which are essentially obtained as an outcome of the diagnoser. Thus, to fully profit from the advantages of top-down as well as bottom-up synthesis, a hybrid approach is taken which is able to infer program corrections that are hard, if not at all impossible, to obtain with a simpler, automatic deductive learner.

Our correction scheme can be applied to every declarative paradigm for which declarative diagnosis methodologies as well as bottom-up and top-down inductive

learning techniques driven by examples have been developed. In the next chapters, we will see how it can be easily adapted in the case of functional logic and functional programming paradigms.

Structure of the chapter

Section 1.1 remarks how to infer some useful information to guide the correction process. In Section 1.2, we introduce the general problem of inductive learning and the bottom-up and top-down inductive synthesis techniques. Finally, Section 1.3 describes the generic, mutiparadigm scheme for the correction of declarative programs, which defines a hybrid inductive learning technique.

1.1 Exploiting debugger outcomes

In order to detect bugs, declarative diagnosis tools produce a lot of auxiliary information which can be fruitfully collected for driving the synthesis and correction processes. Particularly, they are able —as in the case of the functional logic debugger Buggy[15]— to automatically find out incorrectness and incompleteness symptoms which are responsible for bugs, that is, wrong computed values and missing values. These data can be used as example evidences in inductive learning processes in order to infer correct declarative programs as illustrated in Fig. 1.1. For instance the latest

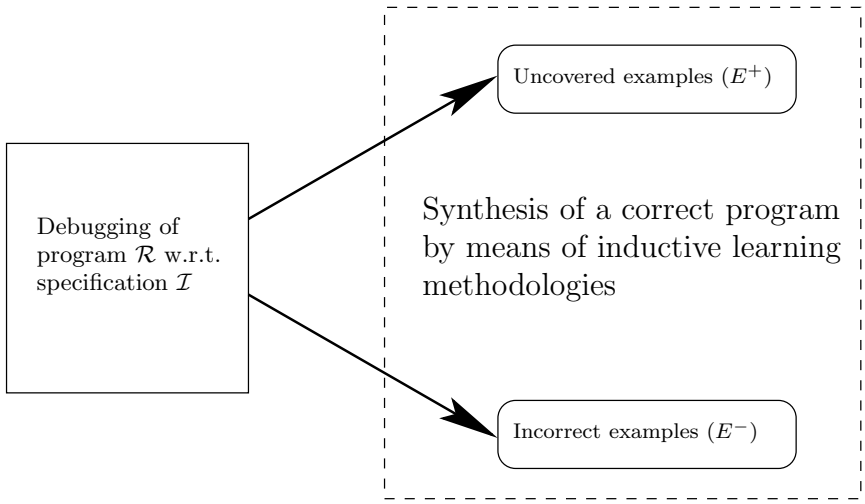

Figure 1.1: Automated generation of incorrectness and incompleteness symptoms.

release of the debugger Buggy, which is available at

`http://www.dsic.upv.es/users/elp/soft.html`,

is able to produce sets of uncovered (incorrect) equations, given a functional logic program $\mathcal{R}$ and a specification $\mathcal{I}$ of the intended semantics (also expressed by a functional logic program). Here we present an example.

Example 1.1.1 *Consider the functional logic program*

$$\mathcal{R} = \{sum(0, X) \to X, sum(s(X), Y) \to sum(X, Y), double(X) \to sum(X, X)\}$$

and the specification of the intended semantics

$$\begin{aligned}\mathcal{I} = \{\, & sum(0, X) \to X, sum(s(X), Y) \to s(sum(X, Y)), double(0) \to 0, \\ & double(s(X)) \to s(s(double(X)))\}.\end{aligned}$$

By executing BUGGY *on program $\mathcal{R}$ and specification $\mathcal{I}$, we discover that there exists an incorrect rule. Besides, the final outcome produces the following sets of uncovered and incorrect equations*

$$\begin{aligned} E^+ \;=\; & \{\, double(s^3(0)) = s^6(0), sum(s^3(0), s(0)) = s^4(0), \\ & sum(s^3(0), 0) = s^3(0), sum(s^2(0), 0) = s^2(0), \\ & sum(s^2(0), s(0)) = s^3(0), \\ & sum(s(0), s(0)) = s^2(0), sum(s(0), 0) = s(0), \\ & double(0) = 0, double(s^2(0)) = s^4(0), double(s(0)) = s^2(0)\} \end{aligned}$$

$$\begin{aligned} E^- \;=\; & \{\, sum(s(0), s(0)) = s(0), sum(s(0), 0) = 0, \\ & sum(s^2(0), 0) = 0, sum(s^2(0), s(0)) = s(0) \\ & double(s(0)) = s(0)\}. \end{aligned}$$

1.2 Inductive learning

Inductive learning allows us to infer theories from evidences and to synthesize knowledge from experience. Several approaches have been proposed in the literature and different techniques have been developed for each declarative programming paradigm. More formally, inductive learning allows us to solve the following problem. Consider

- a set of *positive* examples E^+, which models the pursued behavior of the program to be inferred;
- a set of *negative* examples E^-, which models the unwanted behavior of the program to be inferred;
- a background knowledge theory $\mathcal{B}$, i.e. a program which expresses the information we know *a priori*.

Then, the inductive learning goal is to synthesize a program $\mathcal{R}$ such that

- the positive examples are derivable in (or covered by) $\mathcal{R} \cup \mathcal{B}$, in symbols, $\mathcal{R} \cup \mathcal{B} \vdash E^+$;

- the negative examples are not derivable in (or covered by) $\mathcal{R} \cup \mathcal{B}$, in symbols, $\mathcal{R} \cup \mathcal{B} \not\vdash E^-$.

In order to synthesize the program $\mathcal{R}$, we can follow a *top-down* as well as a *bottom-up* approach.

The former concerns the refinement of an *overly general* program (that is, a program that derives all the positive examples), in order to get a specialized version $\mathcal{R}$ of the original program which does not cover any negative example and, still, covers the positive examples. In logic programming, this has been done by means of specialization techniques such as those illustrated in Boström's works [29, 30, 28]. These methodologies transform the original program into a "purified" version by using rule unfolding as well as rule deletion. An adaptation to functional and functional logic programs will be presented in the next chapters.

The latter method simply starts from evidences and tries to infer programs by using background theories and some kind of operator which reverses the concept of deduction. For instance, in [87], Muggleton et al. discuss a bottom-up technique for the induction of logic programs based on an *inverse resolution* operator which permits to navigate among all the possible hypotheses of a given set of evidences (i.e., ground atoms). A functional logic learner, which exploits the inversion of the *narrowing* relation, has also been presented in [67, 68, 57] and it will be described in detail later on.

1.3 Correction scheme

In this section, we describe a hybrid correction scheme [4], which is based on the top-down and the bottom-up synthesis techniques. Our methodology is general and it can be applied to each declarative programming paradigm, since bottom-up as well as top-down learners have been successfully studied and developed in several declarative paradigms, [67, 57, 87, 29, 28, 5].

The goal we plan to pursue is explained in the following. Let us consider a program $\mathcal{R}$ and a specification $\mathcal{I}$ of the *intended* semantics, that is, the semantics expressing the programmer's intentions. Let $\mathcal{R}' \subseteq \mathcal{R}$ be a set of rules such that each rule $r \in \mathcal{R}'$ is incorrect w.r.t. the given specification $\mathcal{I}$. We aim at synthesizing a set of rules which replace $\mathcal{R}'$ in order to produce a suitable correction.

More formally, we want

- to generate a pair (E^+, E^-) such that E^+ is a set of positive examples and E^- is a set of negative examples;
- to infer a set of rules $\mathcal{X}$ such that
  1. $\mathcal{R}^c \equiv (\mathcal{R} \setminus \mathcal{R}') \cup \mathcal{X}$ covers E^+;
  2. $\mathcal{R}^c$ does not cover E^-.

Program $\mathcal{R}^c$ is called the *correct* program. We will call $\mathcal{R}^- = \mathcal{R} \setminus \mathcal{R}'$ *diminished* program. Therefore, in our framework, the correction problem can be considered as

a particular instance of the inductive learning problem which has been illustrated in section 1.2.

Our method is mainly based on the top-down synthesis methodology, which allows to produce specializations of the original program which cover positive examples and fail on the negative ones. Anyway using this method is a little bit restrictive for the correction issue, since it is only able to correct overly-general program by purifying them from wrong answers. Top-down learning methods cannot infer missing answers. Hence, only wrong programs, which entirely cover E^+, have a chance to get a correction, as witnessed by the following example expressed in the functional programming paradigm.

Example 1.3.1 *Consider program* $\mathcal{R} = \{zero(0) \rightarrow 1, zero(s(X)) \rightarrow 0\}$ *and specification* $\mathcal{I} = \{zero(X) \rightarrow 0\}$. $\mathcal{R}$ *is incorrect w.r.t.* $\mathcal{I}$. *Suppose the following example sets have been generated*

$$E^+ = \{zero(0) = 0, zero(s(0)) = 0, zero(s(s(0))) = 0\}$$
$$E^- = \{zero(0) = 1\}.$$

Program $\mathcal{R}$ *cannot cover* E^+ *entirely, thus no specializations of* $\mathcal{R}$ *will be able to cover* E^+ *entirely and, therefore, no corrections of* $\mathcal{R}$ *can be obtained.*

Hence a sole top-down algorithm is not enough for our objectives, we have to improve it in order to achieve better corrections. A possibility concerns augmenting the original program with new rules, so that the entire set E^+ succeeds w.r.t. the augmented program, and thus the top-down algorithm can still be applied without information loss.

The augmentation method is based on the bottom-up synthesis technique. Basically the augmentation algorithm works in this way: first, the set E of all the positive examples that are not covered by $\mathcal{R}$ is computed, then —by using the bottom-up methodology— we calculate a program $\mathcal{A}$ such that every example in E succeeds. Finally the resulting program is added to the original program in order to obtain an overly general hypothesis $\mathcal{R}_{aug} \equiv \mathcal{R} \cup \mathcal{A}$ to which the top-down algorithm can be applied. The correction scheme is presented in Fig. 1.2.

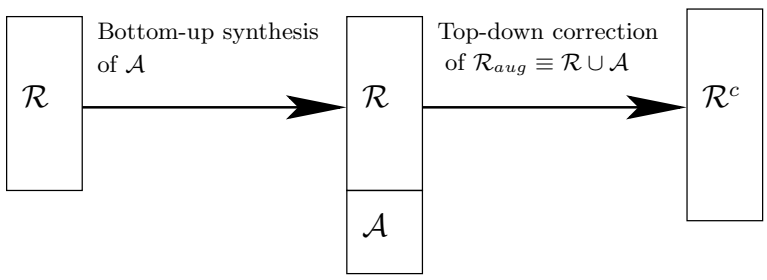

Figure 1.2: Hybrid correction scheme.

In the rest of this section, we present a possible hybrid correction algorithm, which could be tailored to different declarative paradigms.

We suppose that function *GenerateExampleSets*$(\mathcal{R}, \mathcal{I})$ is a generic method to produce example sets, it might be based on an automatic procedure which exploits debugger outcomes or it might simply interact with the user in a more traditional way (for instance, the user could provide example sets manually). Function *BU-Learner*$(E, \mathcal{B})$ is the function that computes, by means of the background theory $\mathcal{B}$, a program covering the example set E. This function implements a simplified bottom-up synthesis technique, as it does not deal with negative examples. Function *TD-Corrector*$(\mathcal{R}, E^+, E^-)$ is a top-down algorithm which specializes program $\mathcal{R}$ w.r.t. sets of positive and negative examples E^+ and E^-.

Finally, notice that in order to exploit all the available information during the bottom-up induction process, we set the background theory $\mathcal{B}$ to the diminished program $\mathcal{R}^-$. By doing so, also the rules in $\mathcal{R}^-$ could be used for synthesizing corrections, allowing us to achieve better results.

Algorithm 1 Hybrid Correction Algorithm.

procedure HYBRID-CORRECTOR$(\mathcal{R}, \mathcal{I})$
  $(E^+, E^-) \leftarrow$ GENERATEEXAMPLESETS$(\mathcal{R}, \mathcal{I})$
  $E \leftarrow \{e \mid \mathcal{R} \not\vdash e, e \in E^+\}$
  $\mathcal{B} \leftarrow \mathcal{R}^-$
  $\mathcal{A} \leftarrow$ BU-LEARNER$(E, \mathcal{B})$
  $\mathcal{R}_{aug} \leftarrow \mathcal{R} \cup \mathcal{A}$
  $\mathcal{R}^c \leftarrow$ TD-CORRECTOR$(\mathcal{R}_{aug}, E^+, E^-)$
end procedure

As the reader can figure out, the presented correction algorithm does not generally imply that a correction for the wrong program $\mathcal{R}$, which is correct w.r.t. the intended semantics, is obtained as the outcome of the corrector (that is, a program $\mathcal{R}^c$ such that $\mathcal{R}^c$ has the same semantics of $\mathcal{I}$, up to the extra auxiliary function symbols which might appear in $\mathcal{I}$). It might happen that the outcome program is correct w.r.t. E^+ and E^-, but it is not a correction w.r.t. $\mathcal{I}$. Several factors could influence the behaviour of the synthesis process. For instance the lack of good example patterns might not permit to drive the algorithm towards corrections. Besides, since the bottom-up method is generally guided by some heuristics, we do not always achieve the best inferred programs. Finally, top-down learners might lead to several specializations [30]. The resulting specializations are always correct w.r.t. finite example sets, but not always correct w.r.t. the specification of the intended semantics.

2

Preliminaries

In this chapter, we recall the basics of term rewriting systems and functional logic programming, we need in this thesis. For the sake of simplicity, definitions are given in the one-sorted case. The extension to many-sorted signatures is straightforward, see [97]. For more information about these topics, you can consult [22, 76] for term rewriting systems, and the extended surveys [62, 69] for functional logic programming.

Structure of the chapter

First of all, we give some basic definitions about terms, equations, and substitutions. Then, Section 2.2 introduces the term rewriting system framework, we use for defining first-order functional programs as well as functional logic programs. In Section 2.3, we describe the narrowing relation (and some variants of it), which is the evaluation mechanism used in functional logic programming. Finally, Section 2.4 concludes by giving some details about conditional term rewriting systems and conditional narrowing.

2.1 Foundations

2.1.1 Terms and equations

Let $\mathcal{V}$ denote a countably infinite set of variables and Σ be a set of function symbols, or *signature*, each of which has a fixed associated arity. By notation f/n, we denote a function symbol f of arity n. $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ denote the non-ground term algebra and the term algebra built over $\Sigma \cup \mathcal{V}$, and Σ, respectively.

$\tau(\Sigma)$ is usually called the *Herbrand universe* ($\mathcal{H}_\Sigma$) over Σ and it will be denoted by $\mathcal{H}$.

An *equation* $s = t$ is a pair of terms $s, t \in \tau(\Sigma, \mathcal{V})$ or $true$. $\mathcal{B}$ denotes the *Herbrand base*, namely the set of all ground equations which can be built with the elements of $\mathcal{H}$. Syntactic equality of terms is represented by $\equiv$. By abuse of notation, we will extend $\equiv$ to represent the identity relation between any syntactic object.

Terms are viewed as labelled trees in the usual way. We denote any sequence of equations $true, \ldots, true$ by $\top$.

Term *positions* $(p, q, \ldots)$ are represented by sequences of natural numbers, where Λ denotes the empty sequence. Given two positions p and q, we represent the concatenation of p and q by $p.q$. Positions are ordered w.r.t. the *prefix* ordering $\leq$, that is, $p \leq q$ iff there exists a position w such that $p.w = q$.

$O(t)$ denotes the set of positions of a term t and is inductively defined as follows

$$O(t) = \begin{cases} \Lambda & \text{if } t \in \mathcal{V} \\ \Lambda \cup \{i.p \mid 1 \leq i \leq n \wedge p \in O(t_i)\} & \text{if } t \equiv f(t_1, \ldots, t_n) \end{cases}$$

Moreover, given $S \subseteq \Sigma \cup \mathcal{V}$, $O_S(t)$ denotes the set of positions of a term t which are rooted by symbols in S. In particular, $O_\Sigma(t)$ represents the set of all non-variable positions in t, we will also denote it by $\overline{O}(t)$. $t_{|u}$ is the subterm at the position u of t. $t[r]_u$ is the term t with the subterm at the position u replaced by r. Let we denote the symbol labeling the root of a term t by $root(t)$. These notions extend to sequences of equations and/or terms in a natural way.

By $Var(s)$ we denote the set of variables occurring in the syntactic object s, while $[s]$ denotes the set of ground instances of s. The symbol $\tilde{}$ represents a finite sequence of symbols.

2.1.2 Substitutions and syntactic unification

A *substitution* $\sigma \equiv \{X_1/t_1, \ldots, X_n/t_n\}$ is a mapping from the set of variables $\mathcal{V}$ into the set of terms $\tau(\Sigma, \mathcal{V})$ such that

(*i*) $X_i \neq X_j$, whenever $i \neq j$;

(*ii*) $X_i\sigma = t_i$, $i = 1, ..n$;

(*iii*) $X\sigma = X$, for any $X \in \mathcal{V} \setminus \{X_1, \ldots, X_n\}$.

The set $Dom(\sigma) = \{X \in \mathcal{V} \mid X\sigma \neq X\}$ denotes the *domain* of the substitution σ. The *empty* substitution ϵ is the substitution such that $Dom(\epsilon) = \emptyset$.

We write $\theta_{\restriction s}$ to denote the restriction of the substitution θ to the set of variables in the syntactic object s.

Substitutions can be extended to morphisms over terms in $\tau(\Sigma, \mathcal{V})$, i.e., $f(\tilde{t})\sigma = f(\tilde{t\sigma})$, for each $f(\tilde{t}) \in \tau(\Sigma, \mathcal{V})$.

A substitution θ is *more general* than σ, which is denoted by $\theta \leq \sigma$, if $\sigma = \theta\gamma$ for some substitution γ. It is easy to show that $\leq$ is a preorder over substitutions. The preorder $\leq$ induces a preorder over terms (called *relative generality*), that is, $t \leq t'$ iff there exists a substitution σ such that $t\sigma \equiv t'$. $\leq$ for terms is extended to equations in the obvious way, i.e. $s = t \leq s' = t'$ iff there exists σ s.t. $\sigma(s) = \sigma(t) \equiv s' = t'$.

A *renaming* is a substitution ρ for which there exists the inverse ρ^{-1}, such that $\rho\rho^{-1} = \rho^{-1}\rho = \epsilon$. A term t' is a *variant* of a term t iff $t\rho = t'$ for some renaming ρ.

An equation set E is *unifiable*, if there exists θ such that, for all $s = t$ in E, we have $s\theta \equiv t\theta$, and θ is called a *unifier* of E. A *goal* is any finite sequence of equations.

We let $mgu(E)$ denote the most general unifier of an equation set E iff $\theta \equiv mgu(E)$ is a unifier of E and $\theta \leq \sigma$ for each substitution σ that is another unifier of E [80].

2.1.3 $\mathcal{V}$-Herbrand base and program semantics

In order to formulate the program semantics we will use later on, the usual Herbrand base is extended to the set of all (possibly) non-ground equations [49, 50]. $\mathcal{H}_\mathcal{V}$ denotes the *$\mathcal{V}$-Herbrand universe* which allows variables in its elements, and is defined as $\tau(\Sigma, \mathcal{V})/_\simeq$, where $\simeq$ is the equivalence relation induced by the preorder of relative generality $\leq$ over terms.

For the sake of simplicity, the elements of $\mathcal{H}_\mathcal{V}$ (equivalence classes) have the same representation as the elements of $\tau(\Sigma, \mathcal{V})$ and are also called terms. $\mathcal{B}_\mathcal{V}$ denotes the *$\mathcal{V}$-Herbrand base*, namely, the set of all equations $s = t$ modulo variance, where $s, t \in \mathcal{H}_\mathcal{V}$. A subset of $\mathcal{B}_\mathcal{V}$ is called *$\mathcal{V}$-Herbrand interpretation*. We assume that the equations in a $\mathcal{V}$-Herbrand interpretation are renamed apart. For us, a *program semantics* will be a suitable $\mathcal{V}$-Herbrand interpretation.

2.2 Programs as term rewriting systems

A *term rewriting system* (TRS for short) is a pair $(\Sigma, \mathcal{R})$, where $\mathcal{R}$ is a finite set of reduction (or rewrite) rules of the form $\lambda \to \rho$, λ, $\rho \in \tau(\Sigma, \mathcal{V})$, $\lambda \notin \mathcal{V}$ and $Var(\rho) \subseteq Var(\lambda)$. Term λ is called the *left-hand side* (lhs) of the rule and ρ is called the *right-hand side* (rhs). We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$ and call $\mathcal{R}$ the program.

Given a rule $\lambda \to \rho$, a rule $\lambda' \to \rho'$ is a *variant* of $\lambda \to \rho$ iff λ' is a variant of λ and ρ' is a variant of ρ via the same renaming. By $r \ll \mathcal{R}$, we denote that r is a new variant of a rule in $\mathcal{R}$ such that r contains only *fresh* ("standardized apart") variables .

Given a TRS $(\Sigma, \mathcal{R})$, we assume that the signature Σ is partitioned into two disjoint sets $\Sigma = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} = \{f \mid f(t_1, \ldots, t_n) \to r \in \mathcal{R}\}$ and $\mathcal{C} = \Sigma \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*. The elements of $\tau(\mathcal{C}, \mathcal{V})$ are called *constructor terms*, while elements in $\tau(\mathcal{C})$ are called *values*. A *pattern* is a term of the form $f(\bar{d})$ where $f/n \in \mathcal{D}$ and $\bar{d}$ is an n-tuple of constructor terms.

A TRS $\mathcal{R}$ is a *constructor system* (CS), if all lhs's of $\mathcal{R}$ are patterns. A TRS $\mathcal{R}$ is *left-linear* (LL), if no variable appears more than once in the lhs of any rule of $\mathcal{R}$.

A *rewrite step* is the application of a rewrite rule to an expression. A term s *rewrites* to a term t via $r \ll \mathcal{R}$, $s \to_r t$ (or $s \to_\mathcal{R} t$), if there exist $u \in O_\Sigma(s)$, $r \equiv \lambda \to \rho$, and substitution σ such that $s_{|u} \equiv \lambda\sigma$ and $t \equiv s[\rho\sigma]_u$. The subterm $s_{|u}$ of the term s is called *redex*.

We say that $\mathcal{S} := t_0 \to_{r_0} t_1 \to_{r_1} t_2 \ldots \to_{r_{n-1}} t_n$ is a *rewrite sequence* from term t_0 to term t_n. When no confusion can arise, we will omit any subscript (i.e. $s \to t$). We call the relation $\to$ *reduction* (or *rewriting*) relation. By means of $\to^+$ and $\to^*$, we denote the transitive closure, and the transitive and reflexive closure of the relation $\to$, respectively. We say that relation $\to$ is *terminating* iff there exists no infinite rewrite sequence $t_1 \to t_2 \to \ldots$. Besides, relation $\to$ is called *confluent*, iff, for all terms $s, t_1, t_2 \in \tau(\Sigma, \mathcal{V})$, $s \to^* t_1$ and $s \to^* t_2$ imply that there exists $t \in \tau(\Sigma, \mathcal{V})$ such

that $t_1 \rightarrow^* t$ and $t_2 \rightarrow^* t$.

A term s is a *normal form* (or an *irreducible form*) w.r.t. $\mathcal{R}$, if there is no term t such that $s \rightarrow_{\mathcal{R}} t$. t is the normal form of s w.r.t. $\mathcal{R}$, if $s \rightarrow^*_{\mathcal{R}} t$ and t is a normal form (in symbols $s \rightarrow^!_{\mathcal{R}} t$). We say that a TRS $\mathcal{R}$ is *terminating*, if relation $\rightarrow_R$ is terminating, while a TRS $\mathcal{R}$ is *confluent*, whenever $\rightarrow_R$ is confluent. A terminating and confluent TRS is called *canonical*.

A TRS $\mathcal{R}$ is *completely defined*, iff for each term $f(\tilde{t})$, where $f \in \mathcal{D}$, there exists a term s such that $f(\tilde{t}) \rightarrow_{\mathcal{R}} s$.

Term rewriting systems provide an adequate computational model we will employ in Chapter 4 in order to formalize a framework for the automated correction of first-order functional programs. In the sequel, we describe the *narrowing* relation, which is the needed ingredient to formalize the functional logic programming paradigm and, also, the program transformations we will use in the next chapters.

2.3 The narrowing relation

The standard operational semantics of functional logic programs is based on *narrowing* [52, 101], a combination of unification for parameter passing and reduction as evaluation mechanism which subsumes rewriting and SLD-resolution. Essentially, narrowing consists of the instantiation of goal variables, followed by a rewrite step on the instantiated goal. Hence, narrowing allows to evaluate function calls, which are not completely instantiated. We define a *goal* as a sequence of terms or equations.

More formally, the narrowing relation can be defined as follows.

Definition 2.3.1 (Narrowing) *Let $(\Sigma, \mathcal{R})$ be a TRS and g be a goal. The* narrowing relation *is defined as the smallest relation $\rightsquigarrow$ satisfying*

$$\frac{u \in O_\Sigma(g), (\lambda \rightarrow \rho) \ll \mathcal{R}, \sigma = mgu(\{g_{|u} = \lambda\})}{g \overset{\sigma,u}{\rightsquigarrow} (g[\rho]_u)\sigma}.$$

The subterm $g_{|u}$ of g is called narrowing redex.

Sometimes, when it is clear from the context, we disregard information about position and/or substitution from the narrowing relation $\overset{\sigma,u}{\rightsquigarrow}$.

A narrowing derivation from goal g to goal g' is defined as $g \overset{\theta}{\rightsquigarrow}^* g'$ iff there exist $\theta_1, \ldots, \theta_n$ such that $g \overset{\theta_1}{\rightsquigarrow} \ldots \overset{\theta_n}{\rightsquigarrow} g'$ and $\theta = \theta_1 \ldots \theta_n$. Let $\mathcal{R}_+$ be the TRS $\mathcal{R} \cup \{X = X \rightarrow true\}$. Rule $X = X \rightarrow true$ is added to a TRS $\mathcal{R}$ to treat syntactic unification as a narrowing step; that is, $(s_1 = t_1, \ldots, s_n = t_n) \overset{\sigma}{\rightsquigarrow} \top$ iff $\sigma = mgu(\{s_1 = t_1, \ldots, s_n = t_n\})$. Rule $X = X \rightarrow true$ is also called *standard equality* axiom.

Given a goal g, a *successful* narrowing derivation for g is $g \overset{\theta}{\rightsquigarrow}^* \top$ in $\mathcal{R}_+$; $\theta_{\restriction e}$ is also called *computed answer substitution* of e.

When TRS are not terminating, it is frequent to consider the *strict* equality $\approx$ instead of standard equality, which gives to equality the weak meaning of identity on

finite objects (e.g., see [84]). The semantics of strict equality is given by the following set of confluent rules (strict equality axioms):

$$\begin{array}{rcll} c \approx c & \rightarrow & true & \forall\, c/0 \in \mathcal{C} \\ c(x_1,\ldots,x_n) \approx c(y_1,\ldots,y_n) & \rightarrow & (x_1 \approx y_1) \wedge \ldots \wedge (x_n \approx y_n) & \forall\, c/n \in \mathcal{C} \\ true \wedge true & \rightarrow & true & \end{array}$$

Narrowing is complete in the sense of functional programming (computation of normal forms) as well as logic programming (computation of answers) for interesting classes of TRSs. In the following, we briefly recall the notion of *completeness*. Any TRS (or, equivalently, equational Horn theory) $\mathcal{E}$ together with its equality axioms generates a congruence relation $=_\mathcal{E}$ over $\tau(\Sigma,\mathcal{V})$, which is called $\mathcal{E}$*-equality*, representing the smallest equational theory which is closed under the entailment relation $\models$. Given a set of equations E, we say that E is $\mathcal{E}$*-unifiable* iff there exists a substitution σ such that, for each $s = t \in E$, $s\sigma =_\mathcal{E} t\sigma$. The substitution σ is called $\mathcal{E}$*-unifier* (or simply *solution*) of E. Given a set of variables $W \subseteq \mathcal{V}$, the relation $=_\mathcal{E}$ is naturally extended to substitutions: $\sigma =_\mathcal{E} \theta[W]$ iff $X\sigma =_\mathcal{E} X\theta$, for each $X \in W$. A substitution σ' is an $\mathcal{E}$*-instance* of a substitution σ w.r.t. W (in symbols, $\sigma \leq_\mathcal{E} \sigma'[W]$) iff there exists a substitution γ such that $\sigma\gamma =_\mathcal{E} \sigma'[W]$. a set of $\mathcal{E}$-unifiers S of a set of equations E is *complete* iff, for each $\mathcal{E}$-unifier σ of E, $\sigma =_\mathcal{E} \theta\gamma[Var(E)]$, where $\theta \in S$.

A narrowing algorithm is *complete* iff it generates a complete set of $\mathcal{E}$-unifiers for each equations set provided as input. More formally, let g be a goal.

> if $\mathcal{E} \models g\sigma$, then there exists a successful narrowing derivation $g \overset{\theta}{\leadsto}{}^{*} \top$ such that $\theta \leq_\mathcal{E} \sigma[Var(g)]$.

It has been shown that narrowing is complete for the class of canonical TRSs [62, 69]. Due to the huge search space of narrowing, steadily improved strategies have been proposed. In the literature, an impressive variety of strategies has been developed ([62] cites 18 distinct narrowing strategies!), which are complete for distinct classes of TRSs. However, we can classify narrowing strategies in two broad classes:

- *eager* narrowing strategies, which give priority to the reduction of innermost narrowing redexes.
- *lazy* narrowing strategies, which give priority to the reduction of outermost narrowing redexes.

A *narrowing strategy* (or *position constraint*) φ is any well-defined criterion that obtains a smaller search space by permitting narrowing to reduce only some chosen positions. We denote by $\leadsto_\varphi$ the narrowing relation with strategy φ (see [62] for a survey on narrowing strategies). $I\!R_\varphi$ denotes the class of TRSs which satisfy the conditions for the completeness of the strategy φ.

In this dissertation, we will mainly consider two narrowing strategies: *leftmost-innermost* narrowing strategy ($\varphi = inn$) and *needed* narrowing strategy ($\varphi = needed$).

The former is an eager strategy, which —given a goal g— reduces the leftmost-innermost narrowing redex of g. More formally, an *innermost* narrowing redex of

a term t is a narrowing redex $t_{|u}$ for which there does not exist a position w such that $t_{|u.w}$ is a narrowing redex of t. So, strategy $inn(g)$ computes the position of the leftmost-innermost narrowing redex of g. Completeness for strategy $\varphi = inn$ is ensured for the class of confluent, terminating, completely defined CS.

The latter is a lazy strategy, which is optimal w.r.t. the length of the derivations and the number of computed solutions in inductively sequential (IS) programs, that is, programs such that all its defined functions have a *definitional tree*.

Roughly speaking, a definitional tree for a function symbol f is a tree whose leaves (*rule* nodes) contain all (and only) the rules used to define f and whose inner nodes (*branch* nodes) contain information to guide the (optimal) pattern matching during the evaluation of expressions. Each inner node contains a *pattern* and a variable position in this pattern (the *inductive position*) which is further refined in the patterns of its immediate children by using different constructor symbols. The pattern of the root node is simply $f(\tilde{X})$, where $\tilde{X}$ is a tuple of different variables. Informally, inductive sequentiality [19] amounts to the existence of discriminating left-hand sides, i.e. typical functional programs. A precise definition of this class of programs and the needed narrowing strategy is provided in [20].

For the completeness of needed narrowing, strict equality $\approx$ is considered, since inductively sequential TRS can be non-terminating. Therefore, equations will be of the form $s \approx t$. Moreover, whenever we consider "eager strategies" such as leftmost-innermost, we will use standard equality.

By notation $=_\varphi$, we will distinguish standard and strict equality according to the chosen narrowing strategy $\varphi \in \{needed, inn\}$.

2.4 Conditional programs and narrowing

The notions we presented in this chapter can be extended to *conditional* programs by taking advantage of the formalism of conditional term rewriting systems.

A *conditional term rewriting system* (CTRS for short) is a pair $(\Sigma, \mathcal{R})$, where $\mathcal{R}$ is a finite set of reduction (or rewrite) rule schemes of the form $(\lambda \to \rho \Leftarrow C)$, λ, $\rho \in \tau(\Sigma \cup V)$, $\lambda \notin \mathcal{V}$ and $Var(\rho) \subseteq Var(\lambda)$. Variables in C that do not occur in λ are called *extra-variables*. The condition C is a (possibly empty) sequence $e_1, \ldots, e_n$, $n \geq 0$, of equations which we handle as a set (conjunction) when we find it convenient. We will often write just $\mathcal{R}$ instead of $(\Sigma, \mathcal{R})$. If a rewrite rule has no condition, we write $\lambda \to \rho$.

Also for CTRSs, we assume that the signature Σ is partitioned into two disjoint sets $\Sigma = \mathcal{C} \uplus \mathcal{D}$, where $\mathcal{D} = \{f \mid (f(\bar{t}) \to r \Leftarrow C) \in \mathcal{R}\}$ and $\mathcal{C} = \Sigma \setminus \mathcal{D}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{D}$ are called *defined functions*. The elements of $\tau(\mathcal{C}, \mathcal{V})$ are *constructor* terms. A pattern is a term $f(l_1, \ldots, l_n)$ such that $f \in \mathcal{D}$ and $l_1, \ldots, l_n$ are constructor terms.

Given a CTRS $(\Sigma, \mathcal{R})$, a term s *conditionally* rewrites to a term t, in symbols $s \to_{\mathcal{R}} t$ or $s \to t$, iff there exist a rule $\lambda \to \rho \Leftarrow s_1 = t_1, \ldots, s_n = t_n$, a position $p \in O_\Sigma(s)$, a substitution σ such that $s_{|p} = \lambda\sigma$, $t = s[\rho\sigma]_p$ and, for each $i = 1, \ldots, n$, there exists a term q_i such that $s_i\sigma \to^*_{\mathcal{R}} q_i$ and $t_i\sigma \to^*_{\mathcal{R}} q_i$, where $\to^*_{\mathcal{R}}$ denotes the

transitive and reflexive closure of $\rightarrow_{\mathcal{R}}$.

The notions of normal form, terminating and confluent CTRS are trivial extensions of the unconditional case (for more details, see [76]). Moreover, a CTRS $\mathcal{R}$ is called *decreasing* iff for each conditional rule $\lambda \rightarrow \rho \Leftarrow t_1 = t'_1, \ldots, t_n = t'_n \in \mathcal{R}$, we have that $\forall \sigma, \sigma(t_i)$ and $\sigma(t'_i)$ are smaller than $\sigma(\lambda)$ w.r.t. some termination ordering [74].

Conditional narrowing

Narrowing can be lifted to CTRS's framework in an obvious way. Let $\mathcal{R}$ be a CTRS, then we define $\mathcal{R}_+$ as $\mathcal{R} \cup \{X = X \rightarrow true\}$.

Definition 2.4.1 (Conditional narrowing) *Let $(\Sigma, \mathcal{R})$ be a CTRS and g be a goal. The* conditional narrowing relation *is defined as the smallest relation $\rightsquigarrow$ satisfying*

$$\frac{u \in O_\Sigma(g), (\lambda \rightarrow \rho \Leftarrow C) \ll \mathcal{R}, \sigma = mgu(\{g_{|u} = \lambda\})}{g \overset{\sigma,u}{\rightsquigarrow} (C, g[\rho]_u)\sigma}.$$

We denote the transitive closure and the transitive and reflexive closure of $\rightsquigarrow$ by $\rightsquigarrow^+$ and $\rightsquigarrow^*$, respectively. A *successful* narrowing derivation for a goal g is of the form $g \overset{\theta}{\rightsquigarrow}^* \top$ in $\mathcal{R}_+$, and $\theta_{\lfloor g}$ is usually called *computed answer substitution*. We can also define conditional narrowing w.r.t. some strategy φ in a way similar to the one presented in Section 2.3. By abuse of notation, $\mathbb{I}\!R_\varphi$ denotes the class of CTRSs which satisfy the conditions for the completeness of conditional narrowing with strategy φ.

3

Correction of Functional Logic Programs

In this chapter, we provide a methodology for developing advanced debugging and correction tools for functional logic languages. Functional logic programming is now a mature paradigm and as such there exist modern environments which assist in the design, development and debugging of integrated programs. However, there is no theoretical foundation for integrating debugging and synthesis into a single unified framework.

In a previous work [13], a generic diagnosis method w.r.t. computed answers which generalizes the ideas of [39] to the diagnosis of functional logic programs has been proposed. The method works for eager (*call–by–value*) as well as for lazy (*call–by–name*) integrated languages. Given the intended specification $\mathcal{I}$ of a program $\mathcal{R}$, we can check the correctness of $\mathcal{R}$ w.r.t. $\mathcal{I}$ by a single step of a (continuous) immediate consequence operator which we associate to our programs. This specification $\mathcal{I}$ may be partial or complete, and can be expressed in several ways: for instance, by (another) functional logic program [13, 14], by an assertion language [37] or by equation sets (in the case when it is finite). Our methodology is based on abstract interpretation: we construct *over* and *under* specifications $\mathcal{I}^+$ and $\mathcal{I}^-$ to correctly over- (resp. under-) approximate the intended semantics $\mathcal{I}$. We then use these two sets respectively for the functions in the premises and the consequences of the immediate consequence operator, and by a simple static test we can determine whether some of the clauses are wrong. The debugging system Buggy [15] is an experimental implementation of the method which allows the user to specify the (concrete) semantics by means of a functional logic program. In [14], we also presented a preliminary correction algorithm based on the deductive synthesis methodology known as *example-guided unfolding* [29]. This methodology uses unfolding in order to discriminate positive from negative examples (resp. uncovered and incorrect equations) which are essentially obtained as an outcome by the diagnoser.

However, this pure deductive learner cannot be applied when the original wrong program is overspecialized (that is, it does not cover all the (positive) examples chosen to describe the pursued behavior).

In this chapter, we develop a new program corrector based on, and integrated with,

the declarative debugger of [13, 14], which integrates top–down as well as bottom–up synthesis techniques, following the correction scheme we presented in Chapter 1. Furthermore, our method is parametric w.r.t. the chosen bottom-up learner. As an instance of such parameter, we consider for the bottom-up part of the algorithm the functional logic inductive framework of [57, 68, 67]. Informally, our correction procedure works as follows. Starting from an overly general program (that is, a program which covers all positive examples as well as some negative ones), the top–down algorithm unfolds the program until a set of rules which only occur in the refutation of the negative examples is identified, and then they are removed from the program. When the original wrong program does not initially cover all positive examples, we first invoke the bottom–up procedure, which "generalizes" the program as to fulfil the applicability conditions. After introducing the new method we prove its correctness and completeness w.r.t. the considered example sets. Finally we present a prototypical implementation of our system.

The following example illustrates our method.

Example 3.0.2 *Let us consider the program:*

$$\mathcal{R} = \{\ od(\mathbf{0}) \to true, od(\mathbf{s}(\mathbf{X})) \to od(X), z(0) \to \mathbf{1}, z(s(X)) \to z(X)\ \}$$

which is wrong w.r.t. the following specification of the intended semantics (mistakes in $\mathcal{R}$ are marked in bold):

$$\mathcal{I} = \{\, ev(0) \to true, ev(s(s(X))) \to ev(X), od(s(X)) \to true \Leftarrow ev(X) = true,\\ z(X) \to 0\ \}.$$

By running the diagnosis system BUGGY*, we are able to isolate the wrong rules of $\mathcal{R}$ w.r.t. the given specification. By exploiting the debugger outcome as described later, the following positive and negative example sets are automatically produced (the user is allowed to fix the cardinality of the example sets by tuning some system parameters):*

$$E^+ = \{od(s^3(0)) = true, od(s(0)) = true, z(s^2(0)) = 0, z(s(0)) = 0, z(0) = 0\ \}$$
$$E^- = \{od(s^2(0)) = true, od(0) = true, z(0) = 1, z(s(0)) = 1, z(s^2(0)) = 1\ \}.$$

We observe that unfolding the rule $r \equiv od(s(X)) \to od(X)$ w.r.t. $\mathcal{R}$ results in replacing r by two new rules $r_1 \equiv od(s(0)) \to true$ and $r_2 \equiv od(s^2(X)) \to od(X)$. Now, by getting rid of rule $od(0) \to true$, we obtain a new recursive definition for function od covering the positive examples while no negative example can be proven, which corrects the bug on function od.

However, note that this approach cannot be used for correcting function z: unfolding the rules defining z does not contribute to prove the positive examples since the original program is overspecialized and unfolding can only specialize it further. Nevertheless, by generalizing function z as in the bottom-up inductive framework of [67], we get the new rule $z(X) \to 0$. Now, by eliminating rule $z(0) \to 1$, which does not contribute to any positive example, we obtain the final outcome

$$\mathcal{R}^c = \{\quad od(s(0)) \to true, od(s(s(X))) \to od(X), z(X) \to 0, z(s(X)) \to z(X)\ \}$$

which is correct w.r.t. the computed example sets.

Structure of the chapter

The chapter is organized as follows. In Section 3.1, we introduce some basics about the denotation of functional logic programs. Section 3.2 recalls the framework for the declarative debugging of functional logic programs defined in [14]. In Section 3.3, we present the basic, top-down automatic correction procedure. Section 3.4 integrates this algorithm with a bottom-up inductive learner which allows us to apply the correction methodology when the original program is overly specialized. In Section 3.5, we present an experimental evaluation of the method on a set of benchmarks.

3.1 Denotation of functional logic programs

In the following we recall two useful semantics for functional logic programs (we refer to [13] for details). Basically, the semantics we consider are suitable $\mathcal{V}$-interpretations contained in a given $\mathcal{V}$-Herbrand base.

Observe that, in non-strict languages, if the compositional character of meaning has to be preserved in presence of infinite data structures and partial functions, then non-normalizable terms (i.e., terms without a normal form), which may occur as subterms within normalizable expressions, also have to be assigned a denotation. Following [59, 84], we introduce a fresh constant symbol $\bot$ into Σ to represent the value of expressions which would otherwise be undefined.

Operational semantics

The operational success set semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ of a program $\mathcal{R}$ w.r.t. narrowing strategy φ is defined by considering the answers computed for "most general calls". Let $\Im^{\varphi}_{\mathcal{R}}$ denotes the set of identical equations $c(x_1,\ldots,x_n) =_{\varphi} c(x_1,\ldots,x_n)$, where c is a constructor symbol occurring in $\mathcal{R}$.

$$\mathcal{O}^{ca}_{\varphi}(\mathcal{R}) = \Im^{\varphi}_{\mathcal{R}} \cup \{(f(x_1,\ldots,x_n) = x_{n+1})\theta \mid (f(x_1,\ldots,x_n) =_{\varphi} x_{n+1}) \overset{\theta}{\leadsto}{}^{*}_{\varphi} \top \text{ s.t. } f/n \in \mathcal{D}, x_{n+1} \text{ and } x_i, i = 1,\ldots,n, \text{ are distinct variables}\}.$$

Fixpoint semantics

The (bottom-up) fixpoint semantics $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$, modeling computed answers w.r.t. a narrowing strategy φ, is defined as the least fixpoint

$$\mathcal{F}^{ca}_{\varphi}(\mathcal{R}) = T^{\varphi}_{\mathcal{R}} \uparrow \omega$$

of a parametric immediate consequence operator $T^{\varphi}_{\mathcal{R}} : 2^{\mathcal{B}_{\mathcal{V}}} \to 2^{\mathcal{B}_{\mathcal{V}}}$ which generalizes the ground immediate consequence operator of [69] in order to model computed answers.

The relationship between the operational and fixpoint semantics are established in [14] .

For the sake of clarity, let us summarize the relationship among the two different program denotations $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ and $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ introduced above. The compositional, fixpoint semantics $\mathcal{F}^{ca}_{\varphi}(\mathcal{R})$ which models successful as well as partial (nonterminating as well as intermediate computations, i.e. those equations $f(\bar{t}) = s$ where s "has not reached its value") is obtained by computing the least fixpoint of the immediate consequence operator $T^{\varphi}_{\mathcal{R}}$. On the other hand, the operational success set semantics $\mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ only catches successful derivations, that is, it models the observable computed answers.

3.2 Diagnosis of declarative programs

First we recall some basic definitions on the diagnosis of declarative programs [39].

Definition 3.2.1 *Let $\mathcal{I}_{ca}$ be the specification of the intended success set semantics for $\mathcal{R}$. An* incorrectness symptom *is an equation e such that $e \in \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$ and $e \notin \mathcal{I}_{ca}$. An* incompleteness symptom *is an equation e such that $e \in \mathcal{I}_{ca}$ and $e \notin \mathcal{O}^{ca}_{\varphi}(\mathcal{R})$.*

In case of errors, in order to determine the faulty rules, we make use of the following definitions. We need to consider a fixpoint intended semantics $\mathcal{I}_{\mathcal{F}}$, that models both successful and "in progress" computations. The relation between $\mathcal{I}_{\mathcal{F}}$ and the intended operational meaning is given by $\mathcal{I}_{ca} = \mathcal{I}_{\mathcal{F}} \setminus \mathit{inprogress}(\mathcal{I}_{\mathcal{F}})$, where, for an equation set S, $\mathit{inprogress}(S) = \{\lambda = \rho \in S \mid \bot$ occurs in ρ or ρ contains a defined function symbol of $\Sigma\}$.

Definition 3.2.2 *Let $\mathcal{I}_{\mathcal{F}}$ be the specification of the intended fixpoint semantics for $\mathcal{R}$. If there exists an equation $e \in T^{\varphi}_{\{r\}}(\mathcal{I}_{\mathcal{F}})$ and $e \notin \mathcal{I}_{\mathcal{F}}$, then the rule $r \in \mathcal{R}$ is incorrect on e. We also say that e is* incorrect. *Reciprocally, the equation e is uncovered if $e \in \mathcal{I}_{\mathcal{F}}$ and $e \notin T^{\varphi}_{\mathcal{R}}(\mathcal{I}_{\mathcal{F}})$.*

Since program denotations generally consist of an infinite number of equations, the conditions above for correctness and completeness of a program w.r.t. to a given specification cannot be effectively computed. In [14], an abstract diagnosis methodology based on the abstract interpretation theory [42] was proposed. Abstract diagnosis is a correct approximation of the diagnosis technique presented so far where the semantic domains and operators are replaced by abstract ones. First, we build a suitable abstract immediate consequences operator ($T^{\sharp\varphi}_{\mathcal{R}}$), which uses an abstraction of the program rules where all infinite computations have been removed and is also parametric w.r.t. the narrowing strategy. The approximation is done by using a loop-checker which replaces the calls which could be responsible for the infinite derivations by a fresh irreducible symbol $\sharp$. The fixpoint of $T^{\sharp\varphi}_{\mathcal{R}}$ correctly approximates the fixpoint semantics of $\mathcal{R}$ and can be computed finitely. The abstract diagnosis process is performed w.r.t. two abstract (finite) semantics $\mathcal{I}^{-}$ and $\mathcal{I}^{+}$ which under- and over-approximate the intended semantics $\mathcal{I}$.

3.3 Correction method

In this section, we present an inductive learning methodology which is able to repair a functional logic program containing buggy rules. As we explained in Chapter 1, the correction problem can be stated as follows. Let $\mathcal{R}$ be a CTRS, $\mathcal{I}$ the intended specification, $\mathcal{R}' \subseteq \mathcal{R}$ a set of incorrect rules w.r.t. $\mathcal{I}$, and $E = E^+ \cup E^-$ two disjoint (ground) example sets which model the pursued (not pursued) computational behaviour. We denote by $\mathcal{R} \vdash E$ the fact that the (ground) equation set E can be reduced to $\top$ by using the rules of $\mathcal{R}$. Then, we want to determine a set of rules $\mathcal{X}$ such that $\mathcal{R}^c = (\mathcal{R} \setminus \mathcal{R}') \cup \mathcal{X}$, $\mathcal{R}^c \vdash E^+$ and $\mathcal{R}^c \nvdash E^-$. Program $\mathcal{R}^c$ will be called *correct* program (w.r.t. E^+ and E^-). We will call $\mathcal{R}^- = \mathcal{R} \setminus \mathcal{R}'$ the *diminished* program. We note that $\mathcal{R} \vdash E$ can be checked, even in the case that $\mathcal{R}$ is not terminating, by using the "normalization via μ–normalization" method of [78] to compute, by levels, the 'maximal contexts' of the lhs's of the examples, and then comparing them with the ground constructor term in the corresponding rhs. By this technique, normal forms can be obtained by successively computing μ-normal forms and shifting computations to maximal non-replacing subterms when a μ-normal form has been obtained. The conditions for the completeness of this technique (*csr*–conditions) essentially amount to the termination of "context–sensitive rewriting" (*csr*) [79], which is much easier than the termination of rewriting. For more information, see Appendix A. A practical tool for proving termination of context sensitive rewriting is available at `http://www.dsic.upv.es/users/elp/slucas/muterm`.

In order to infer program corrections, our methodology is based on the hybrid, top-down as well as bottom-up approach, which we proposed in Chapter 1. We believe that the resulting blend of top-down and bottom-up synthesis is conceptually cleaner than more sophisticated, purely top-down or bottom-up ones and combines the advantages of both techniques.

First, we present the basic, top-down specialization method. We follow the deductive approach known as *example guided unfolding* [29], which is able to specialize an overly general program by applying unfolding and deletion of program rules until coming up with a correct program. The top-down correction process is "guided" by the examples, in the sense that transformation steps focus on discriminating positive from negative examples.

3.3.1 Automatic generation of positive and negative example sets

Let us present a simple method for automatically generating the example sets which exploits the debugger outcome so that the user does not need to provide error symptoms, evidences or other kind of information .

Consider the diminished program $\mathcal{R}^-$. Due to the absence of faulty rules in $\mathcal{R}^-$, $\mathcal{R}^-$ is already partially correct; however $\mathcal{R}^-$ might be incomplete, as there can be equations which are covered in $\mathcal{I}$, but not in $\mathcal{R}^-$.

By applying the diagnosis method presented in Section 3.2, we are able to find out the sets of *uncovered* and *incorrect* equations w.r.t. an abstraction of the intended

semantics, respectively E_1 and E_2. Considering equations in E_1 seems a sensible way for yielding *positive* examples (missing proofs which should be achieved by $\mathcal{R}$). Instead, set E_2 contains equations modeling erroneous behaviours, thus we can take them as *negative* examples.

Since E_1 and E_2 might contain non-ground equations, we find it useful to instantiate (a subset of) them in order to get ground positive/negative example sets E^+ and E^-. This allows us to perform some standard optimizations based on term rewriting which are very satisfactory in practice. On the other hand, since program $\mathcal{R}$ and specification $\mathcal{I}$ might use different auxiliary functions, we only consider ground examples of the form $l = d$ where l is a pattern and d is a constructor term. In this way, the inductive process becomes independent from the extra functions contained in $\mathcal{I}$, since we start synthesizing directly from data structures d. In order to achieve this, we normalize the term in the rhs of (the instantiated) examples w.r.t. $\mathcal{I}$. Finally, we disregard those examples which, after normalization, do not have a constructor term at the rhs.

3.3.2 Specialization operators

Definition 3.3.1 (Unfolding) *Let $\mathcal{R}$ be a CTRS and $r \equiv (\lambda \to \rho \Leftarrow C) \ll \mathcal{R}$ be a rule. Let $\{g \overset{\theta_i}{\leadsto}_\varphi (C'_i, \rho'_i = y)\}_{i=1}^n$ be the set of all one-step narrowing derivations with strategy φ that perform an effective narrowing step for the goal $g \equiv (C, \rho = y)$ in $\mathcal{R}$. Then, $Unf^\varphi_\mathcal{R}(r) = \{(\lambda\theta_i \to \rho'_i \Leftarrow C'_i) | i = 1 \ldots n\}$ (that is, the derived goal $(C'_i, \rho'_i = y)$ is different from g).*

Roughly speaking, *unfolding* a program $\mathcal{R}$ w.r.t. a rule r delivers a new specialized version of $\mathcal{R}$ in which the rule r is replaced by new rules obtained from r by performing a narrowing step on the rhs or the conditional part of r.

Definition 3.3.2 (Unfolding operator) *Let $\mathcal{R}$ be a CTRS, $r \equiv \lambda \to \rho \Leftarrow C$ be a rule in $\mathcal{R}$. The* Rule Unfolding *operator $\mathsf{U}^\varphi_r(\mathcal{R})$ on $\mathcal{R}$ w.r.t. r is defined by $\mathsf{U}^\varphi_r(\mathcal{R}) = \mathcal{R} \setminus \{r\} \cup Unf^\varphi_\mathcal{R}(r)$.*

As it has been proven in [17, 18], for $\varphi = inn, needed$, unfolding using strategy φ preserves the semantics (even for the observable computed answers) in $I\!R_\varphi$ programs. In the case when needed narrowing is considered, completeness is only guaranteed under the condition that expressions in the rule are not unfolded beyond their head normal form [18]. On the other hand, the absence of narrowable positions in the rule to be unfolded yields no specialization of r. We just get the removal of r from $\mathcal{R}$. In the sequel, we use the following notion of "unfoldable rule."

Definition 3.3.3 *Let $\mathcal{R}$ be a CTRS, r be a rule in $\mathcal{R}$. The rule r is* unfoldable *w.r.t. $\mathcal{R}$ if $\mathsf{U}^\varphi_r(\mathcal{R}) \neq \mathcal{R} \setminus \{r\}$. If $\varphi = needed$, we also require that r is not unfolded beyond its head normal form.*

For the sake of simplicity, in the following we omit φ whenever this does not compromise readability.

An *unfolding succession* $\mathcal{S}(\mathcal{R}) \equiv \mathcal{R}_0, \mathcal{R}_1, \ldots$ of program $\mathcal{R}$ is defined as follows: $\mathcal{R}_0 = \mathcal{R}$, $\mathcal{R}_{i+1} = \mathsf{U}_r(\mathcal{R}_i)$ where $r \in \mathcal{R}_i$ is unfoldable.

3.3.3 Top-down correction algorithm

Following [30], the algorithm in the next page works in two phases: the *unfolding phase* and the *deletion phase*. Roughly speaking, we first perform unfolding upon (arbitrarily selected) unfoldable rules, until we get a specialized version of the program $\mathcal{R}$ where no negative example can be proven by applying only rules used in proofs of positive examples. The following definition is auxiliary.

Definition 3.3.4 *Given* $\mathcal{D} : g \equiv g_1 \overset{r_1}{\rightarrow} g_2 \overset{r_2}{\rightarrow} \ldots \overset{r_n}{\rightarrow} g_n$, *the sequence* $\langle r_1, r_2, \ldots, r_n \rangle$ *is called the* rewriting rule sequence *of* $\mathcal{D}$. *The set* $\mathsf{OR}(\mathcal{D}) = \{r_1, r_2, \ldots, r_n\}$ *is called the set of* occurring rules *of* $\mathcal{D}$.

Given an equation e, let $\mathcal{D}^{\varphi}_{\mathcal{R}}(e)$ denote the successful rewrite sequence which proves e in program $\mathcal{R}$ (if it exists) by using a normalizing, deterministic, rewriting strategy for the class $I\!R_{\varphi}$. We also call $\mathcal{D}^{\varphi}_{\mathcal{R}}(e)$ *proof* of e.

The key idea of the algorithm is thus applying unfolding until we get a specialized program $\mathcal{R}_i$ satisfying that, for each $e^- \in E^-$ there exists a rule $r \in \mathsf{OR}(\mathcal{D}^{\varphi}_{\mathcal{R}_i}(e^-))$ such that, for each example $e^+ \in E^+$, $r \notin \mathsf{OR}(\mathcal{D}^{\varphi}_{\mathcal{R}_i}(e^+))$. Now, since the rules which only contribute to the proof of negative examples are useless, in the subsequent phase we just remove these rules from the specialized program $\mathcal{R}_i$. By *discriminable rule* of $\mathcal{R}_i$ we mean an unfoldable rule of $\mathcal{R}_i$ which occurs in the proof of, at least, one positive and one negative example.

Example 3.3.5 *Consider again the program* $\mathcal{R}$ *and specification* $\mathcal{I}$ *of example 3.0.2, with the example sets for learning function od. Since the rewriting proof for the negative example* $od(s^2(0)) = true \in E^-$ *uses the rule* $od(s(X)) \rightarrow od(X)$ *(either with* $\varphi = inn$ *or* $\varphi = needed$*), which is also used in the proofs of positive examples, we enter the main loop. By unfolding* $od(s(X)) \rightarrow od(X)$ *we get* $\mathcal{R}_1 = \{od(0) \rightarrow true, od(s(0)) \rightarrow true, od(s^2(X)) \rightarrow od(X)\}$. *Now we enter the deletion phase which purifies* $\mathcal{R}_1$ *by removing the rule* $od(0) \rightarrow true$ *that only occurs in the proof of a negative example, thus producing the expected correction.*

Example 3.3.5 allows us to clarify the differences between the preliminary correction algorithm in [14] and the one we propose here. The algorithm in [14] was based on unfolding the rules which incorrectly cover the negative examples. In our example, this could result in trying to unfold the rule $od(0) \rightarrow true$, which is fruitless, whereas the new correction procedure does consider any discriminable rule for unfolding, which is generally needed in order to achieve the desired specialization (correction).

Algorithm 2 The top-down correction algorithm.

procedure TD-CorrectorFL($\mathcal{R}, \mathcal{I}$)
  $(E^+, E^-) \leftarrow$GenerateExampleSets($\mathcal{R}, \mathcal{I}$)
  if $\mathcal{R} \not\models E^+$ **then** Halt
  end if
  $i \leftarrow 0$ ▷ Unfolding phase
  $\mathcal{R}_0 \leftarrow \mathcal{R}$
  while $\exists\, e^- \in E^-$ s.t. $\forall r(r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^-)) \Rightarrow \exists e^+ \in E^+$ s.t. $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^+)))$ **do**
    Select a discriminable rule $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^-))$ of $\mathcal{R}_i$
    $\mathcal{R}_{i+1} \leftarrow \mathsf{U}_r(\mathcal{R}_i)$
    $i \leftarrow i + 1$
    for all $e^- \in E^-$ **do** ▷ Deletion phase
      $\mathcal{R}_{i+1} \leftarrow \mathcal{R}_i \backslash \{r\}$, where $r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^-)) \wedge \forall e^+ \in E^+\ r \notin \mathsf{OR}(\mathcal{D}_{\mathcal{R}_i}(e^+))$
      $i = i + 1$
    end for
    $\mathcal{R}^c \leftarrow \mathcal{R}_i$
  end while
end procedure

3.3.4 Correctness of the algorithm

We prove the correctness of the top-down correction algorithm in two steps: first we show that, provided that $\mathcal{R}$ covers E^+, the unfolding phase produces a specialized version $\mathcal{R}'$ of $\mathcal{R}$ (still covering E^+) such that, for each negative example, there is a rule occurring in the corresponding proof which is not used in the proof of any of the positive examples. Next, we prove that the deletion phase yields a corrected version of $\mathcal{R}$ which covers E^+ and does not cover E^-.

The following proposition states our first result: by a suitable finite number of applications of the unfolding operator to a program in $\mathbb{R}_\varphi$, we get a specialized version such that, in any successful rewriting derivation of a negative example, there occurs a rule that is not applied in any successful rewriting derivation for the positive examples under the same strategy. A condition is necessary for proving this result: no negative/positive couple of the considered examples can have the same rewriting rule sequence, as shown in the following counterexample.

Example 3.3.6 *Consider the program* $\mathcal{R} = \{f(X) \to g(X), g(X) \to 0\}$ *with example sets* $E^+ = \{f(a) = 0\}$, $E^- = \{f(b) = 0\}$. *Then* $f(a) = 0$ *and* $f(b) = 0$ *are proven by using the same rewriting rule sequence (using any of the considered rewriting strategies). By applying the top-down algorithm, we unfold rule* $f(X) \to g(X)$, *which produces the outcome* $\mathcal{R}_1 = \{f(X) \to 0, g(X) \to 0\}$ *which cannot be purified (by using the rule deletion operator) as removing rule* $f(X) \to 0$ *in order to get rid of* E^- *would cause losing* E^+.

Proposition 3.3.7 *Let φ be a normalizing rewriting strategy for $\mathbb{R}_\varphi$ and $\mathcal{R}$ be a program in $\mathbb{R}_\varphi$. Let E^+ (resp. E^-) be a set of positive (resp. negative) examples. If there are no $e^+ \in E^+$ and $e^- \in E^-$ which can be proven in $\mathcal{R}$ by using the same rule sequence, then, for each unfolding succession $\mathcal{S}(\mathcal{R})$, there exists k such that $\forall e^- \in E^- \exists r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_k}(e^-))$ s.t. r is not discriminable*

We note that Proposition 3.3.7 holds for every unfolding succession of the original program; this implies that the rule to be unfolded at each unfolding step can be arbitrarily selected, provided that it is discriminable. Moreover, the termination of the unfolding phase is granted by the finite number k of applications of the unfolding operator that we need to obtain specialization $\mathcal{R}_k$.

After the unfolding phase, the refutation of every negative example contains a rule from $\mathcal{R}_k$ which does not occur in the proof of any positive example, thus we can safely remove this rule without jeopardizing completeness. Therefore, the deletion phase purifies $\mathcal{R}_k$ and yields correctness w.r.t. both positive and negative examples.

Theorem 3.3.8 (Correctness) *Let $\mathcal{R} \in \mathbb{R}_\varphi$ be a CTRS which satisfies the csr conditions, E^+ and E^- be two sets of examples such that $\mathcal{R} \vdash E^+$. If the rewriting rule sequences for $e^+ \in E^+$ and $e^- \in E^-$ are different, then the* TD-CorrectorFL *algorithm yields a correct specialization of $\mathcal{R}$ w.r.t. E^+ and E^-.*

As in other approaches for example-guided program correction, the above result does not generally imply that a correction for the wrong program $\mathcal{R}$ w.r.t. the intended semantics is obtained as the outcome of the top-down correction algorithm (that is, a program $\mathcal{R}$ with the same semantics of $\mathcal{I}$, up to the extra auxiliary function symbols which might appear in $\mathcal{I}$), under the conditions required for the correctness of the algorithm, but it might happen that the output program is only correct w.r.t. E^+ and E^-. Therefore, derived programs would be newly diagnosed for correctness at the end.

3.4 Improving the algorithm

In the following, we propose a bottom-up correction methodology which we smoothly combine with the deductive one in order to correct programs which do not fulfil the applicability condition (over–generality). Therefore, the methodology just consists of applying a bottom-up pre–processing to "generalize" the initial wrong program, before proceeding to the top-down correction.

A naïve solution for overcoming the restriction to overly general programs consists of restraining E^+ to the set E' of examples covered by the program, that is $E' = \{e \in E^+ \mid \mathcal{R} \vdash e\}$. Clearly, $\mathcal{R}$ is an overly general program w.r.t. E' and E^-. Unfortunately, this does not generally work, since E' might lack the examples necessary to drive the specialization process towards a correction of the faulty rules.

Let us propose a different methodology which is based on extending the original program with new rules so that the entire set E^+ succeeds w.r.t. the generalized program, and hence the top-down corrector can be effectively applied.

Our generalization method is based on a simplified version of the bottom-up technique for the inductive learning of functional logic programs developed by Ferri, Hernández and Ramírez [57] which is able to produce an intensional description (expressed by a functional logic program) of a set of ground examples. The algorithm is also able to introduce functions, defined as a background theory, in the inferred intensional description (see [57, 67] for details).

In the following we recall the definitions of *restricted generalization* and *inverse narrowing* which are the heart of the bottom-up procedure of [57, 67]. The former allows to generalize program rules, the latter is needed to introduce defined symbols in the right hand sides of the synthesized rules.

Definition 3.4.1 (Generalization operator) *The rule* $r' \equiv (s' \to t' \Leftarrow C')$ *is a* restricted generalization *of* $r \equiv (s \to t \Leftarrow C)$ *if there exists a substitution* θ *such that (i)* $\theta(r') \equiv r$*; (ii)* $Var(t') \subseteq Var(s')$*. The* generalization operator $\mathsf{RG}(r)$ *is defined as follows:*

$$\mathsf{RG}(r) = \{r' | r' \text{ is a restricted generalization of } r\}.$$

Definition 3.4.2 (Inverse narrowing operator) *The rule* $r \equiv s \to t \Leftarrow C$ reversely narrows *into* $r' \equiv s \to t' \Leftarrow C'$ *(in symbols* $r \overset{u,r'',\theta}{\hookleftarrow} r'$*) iff there exist a position* $u \in O(t)$ *and a rule* $r'' \equiv \lambda \to \rho \Leftarrow C''$ *such that (i)* $\theta = mgu(t_{|u}, \rho)$*; (ii)* $t' = (t[\lambda]_u)\theta$*; (iii)* $C' = (C'', C)\theta$*.*
The inverse narrowing *operator* $\mathsf{INV}(r, r'')$ *is given by:*

$$\mathsf{INV}(r, r'') = \{r' \mid r \overset{u,r'',\theta}{\hookleftarrow} r' \text{ and extra-variables in the rhs of } r' \text{ are instantiated to variables in the rhs of } r\}.$$

The extra instantiation of variables in the rhs of the derived rule is necessary since inverse narrowing considers the rules oriented reversely (that is, from right to left) and hence extra-variables (that is, variables which occur in the rhs of a rule but not in its lhs) might be introduced in the synthesized rules, which are not allowed.

The following definitions are helpful for discerning the overspecialized program rules. $Def_{\mathcal{R}}(f)$ is the set of rules in $\mathcal{R}$ needed to define the function f. This might be computed by constructing a functional dependency graph of the program $\mathcal{R}$ and by statically analyzing it. Given a set E of positive examples, $Res_f(E)$ denotes the restriction of E to the set of f-rooted examples (that is, examples of the form $f(\tilde{t}) = d$). We say that a function definition $Def_{\mathcal{R}}(f)$ is *overspecialized* w.r.t. the set of positive examples E^+, if there exists $e \in Res_f(E^+)$ which is not covered by $Def_{\mathcal{R}}(f)$. An incorrect rule belonging to an overspecialized function definition is called *overspecialized* rule.

Basically, the generalization algorithm works in this way: at an initial stage, we discover all function definitions which are overspecialized, by computing the subset of f-examples not provable in $\mathcal{R}$ (and hence not provable by the corresponding function definition). Then, overspecialized rules are deleted from $\mathcal{R}$. At this point, applying generalization and inverse narrowing operators, starting from the positive examples, we try to reconstruct the missing part of the code, that is, we synthesize a functional

logic program $\mathcal{A}$ such that $\mathcal{R} \cup \mathcal{A} \setminus \{r \in \mathcal{R} \mid r \text{ is overspecialized}\}$ allows us to derive the entire E^+. At the end, we get an overly general hypothesis to which the top-down corrector can be applied for repairing (incorrectness) bugs on the derived overly general faulty rules.

Formally, the bottom-up synthesis consists of the following steps. The algorithm firstly generates a set P_H (Program Hypothesis set) which consists of unary programs associated with the restricted generalizations of E^+, that is, $P_H = \{\{r\} \mid r \in \mathsf{RG}(s \rightarrow t), s = t \in E^+\}$. Then it enters a loop in which, by means of INV and RG operators, new programs in P_H are produced. The algorithm leaves the loop when an "optimal" solution, which covers E^+ entirely, has been found in P_H, or a maximal number of iterations is reached. In the latter case no solution might be found.

Due to the huge search space which that method generates, some heuristics must be implemented to guide the search. Following [67], *Minimum Description Length*[1] (MDL) and *Covering Factor*[2] criteria could be taken into consideration, so that inverse narrowing steps are only performed among the best programs and equations w.r.t. these criteria. Moreover, by means of MDL and Covering Factor, only the most concise programs are selected during the induction process. The notion of *Optimality* w.r.t. programs and equations could be defined as a linear combination of these two criteria. For a full discussion on this topic consult [67].

Algorithm GENERALIZE($\mathcal{R}, E^+$) takes as input a program $\mathcal{R}$ and an example set E^+ such that $\mathcal{R}$ just covers a proper subset E of E^+. It returns as output a generalized program $\mathcal{R}_{gen}$ in which the entire E^+ succeeds or—in the case the algorithm fails—a "No solution found" warning is delivered.

To exploit all the information at hand, we set the background theory $\mathcal{B}$ to the diminished program $\mathcal{R}^-$ [3] . By doing so, inverse narrowing steps among rules in $\mathcal{R}^-$ and rules in programs belonging to P_H are allowed.

The function *SelectBest(P_H)* returns the best program in P_H (w.r.t. the optimality criterion), variable *Opt* represents the desired optimality threshold for the inferred program.

Notice that the termination of the algorithm is not guaranteed by the sole upper bound on the number of iterations, since the inverse narrowing operator might yield non-terminating rules (e.g. $f(X,Y) \rightarrow f(Y,X)$), and hence the test $\mathcal{R} \vdash E^+$ could not terminate. This is the reason why a concrete implementation of the algorithm also needs an upper bound n on the narrowing steps. Programs in P_H, using more than n narrowing steps to prove some positive example, are disregarded.

Let us present a final example which illustrates the method. For the sake of clarity, we do not consider optimality issues. Also, we only pinpoint the relevant outcomes.

[1] $length(e) = 1 + n_v/2 + n_f$, where n_v and n_f are the number of variables and functors in the rhs of e.

[2] $CovF(E) = card(\{e \in E \mid \mathcal{R} \vdash e\})/card(E)$.

[3] Actually, we might consider only those function definitions in the diminished program which are necessary to reconstruct the faulty overly specialized part.

Algorithm 3 A bottom-up inductive learner for generalizing programs.

procedure GENERALIZE($\mathcal{R}, E^+$)
  $S \leftarrow \{\mathcal{R}' \mid \mathcal{R}' = Def_{\mathcal{R}}(f), f \in \mathcal{D}\}$
  $\mathcal{R}_{aux} \leftarrow \mathcal{R}$
  for all overspecialized $\mathcal{R}'$ in S **do**
    if $r \in \mathcal{R}' \wedge r$ is incorrect **then**
      $\mathcal{R}_{aux} \leftarrow \mathcal{R}_{aux} \setminus \{r\}$
    end if
  end for
  $P_H \leftarrow \{\{r\} \mid r \in \mathsf{RG}(s \rightarrow t), s = t \in E^+\}$
  $\mathcal{B} \leftarrow \mathcal{R}^-$
  $it \leftarrow 0$
  while $\neg$(SELECTBEST(P_H) $\vdash E^+ \wedge$ SELECTBEST(P_H) $> Opt$) $\wedge\ it < it_{\max}$ **do**
    SELECT the best $\mathcal{R}' \in P_H$ and $\mathcal{R}'' \in P_H \cup \{\mathcal{B}\}$
    SELECT the best $r' \in \mathcal{R}'$ and $r'' \in \mathcal{R}''$
    for all $r \in \mathsf{INV}(r', r'')$ **do**
      for all $rg \in \mathsf{RG}(r)$ **do**
        $P_H \leftarrow P_H \cup \{(\mathcal{R}' \cup \mathcal{R}'' \setminus \{r'\}) \cup \{rg\}\}$
      end for
    end for
    $it \leftarrow it + 1$
  end while
  $\mathcal{A} \leftarrow$ SELECTBEST(P_H)
  if $\mathcal{A} \vdash E^+ \wedge \mathcal{A} > Opt$ **then**
    $\mathcal{R}_{gen} \leftarrow \mathcal{R}_{aux} \cup \mathcal{A}$
  else
    OUTPUT('No solution found')
  end if
end procedure

Example 3.4.3 *Consider the following (wrong) program and the specification*

$$\mathcal{R} = \{ \quad playdice(X) \to double(winface(X)), dd(0) \to 0, \mathbf{dd(s(X))} \to \mathbf{dd(X)}), \\ winface(\mathbf{s(X)}) \to \mathbf{s}(winface(\mathbf{X})), winface(\mathbf{0}) \to \mathbf{0} \}$$

$$\mathcal{I} = \{ \quad playdice(X) \to dd(winface(X)), dd(X) \to sum(X, X), \\ sum(X, 0) \to X, sum(X, s(Y)) \to s(sum(X, Y)), \\ winface(s(0)) \to s(0), winface(s(s(0))) \to s(s(0)) \}.$$

Program rules marked in boldface are signaled as incorrect by the diagnosis system. The automated example generation procedure described in Section 3.3.1 produces the positive example set:

$$E^+ = \{ \quad playdice(s^2(0)) = s^4(0), playdice(s(0)) = s^2(0), dd(s^4(0)) = s^8(0), \\ dd(s^3(0)) = s^6(0), dd(s^2(0)) = s^4(0), dd(s(0)) = s^2(0) \\ dd(0) = 0, winface(s^2(0)) = s^2(0), winface(s(0)) = s(0) \}$$

The analysis of the functions dd and winface determines that the former one is overspecialized. Now, the generalization algorithm removes the rule $dd(s(X)) \to dd(X)$. *Note that rule* $dd(s(0)) \to s^2(0)$ *inversely narrows to rule* $dd(s(0)) \to s^2(dd(0))$ *by using rule* $dd(0) \to 0$. *The following restricted generalizations are computed:* $dd(s(0)) \to s^2(dd(0))$, $dd(s(X)) \to s^2(dd(0))$, $dd(s(X)) \to s^2(dd(X))$. *Now, when the third rule is added to the background knowledge, all examples in* E^+ *are covered. Thus, the following overly general definition is delivered:*

$$\mathcal{R} = \{ \quad playdice(X) \to dd(winface(X)), dd(0) \to 0, dd(s(X)) \to s(s(dd(X)))), \\ winface(\mathbf{s(X)}) \to \mathbf{s}(winface(\mathbf{X})), winface(\mathbf{0}) \to \mathbf{0} \}$$

which can finally be fed to the top-down corrector in order to repair the remaining wrong definition.

3.5 Automated correction system

A prototypical implementation of our correction methodology, written in SICStus Prolog, and a set of benchmarks are available at

`http://www.dimi.uniud.it/~demis/nobug/nobug.html`.

We have improved the preliminary debugging system BUGGY by adding new "correction" features. The new implementation, called NOBUG, is now able to compute sets of positive and negative examples by using the methodology described in Section 3.3.1. Our system includes a parser for the language, a module for computing the program approximation (based on loop-checking), an automatic debugger which allows the user to fix some parameters, such as the number n of iterations for approximating the fixpoint semantics. The module for generating the positive and negative example sets allows the user to provide a list of ground constructor terms for the

generation of the ground example sets. We have also developed a full implementation of the top-down correction method based on example-guided unfolding for the leftmost-innermost narrowing strategy. We are currently implementing the lazy version of the algorithm. The bottom-up synthesis method has not been integrated into the NOBUG system yet. Hence, in order to compute our benchmarks also for initially overspecialized programs, we used the inductive functional logic system FLIP[56].

3.5.1 Experimental evaluation

We have performed some tests by means of our top-down corrector and the bottom-up learner FLIP, in order to repair overly general as well as overspecialized functional logic programs. We have taken into account programs on several domains: naturals, lists and finite domains. In order to systematize the generation of the benchmarks, we have slightly modified correct programs in order to obtain wrong program mutations. We have introduced bugs in the left-hand sides, as well as the right-hand sides, of the program rules, and errors that affect the recursive definitions and the non-recursive ones. We were able to successfully repair incorrect mutations, achieving, in many cases, a correction w.r.t. both the example sets and the intended program semantics. We have noticed that small example sets generally suffice to get a satisfactory correction. In particular, all experiments required less than 20 positive examples and less than 10 negative examples.

A detailed description of our benchmarks, which includes example set generation, overly general transformation and a system execution, can be retrieved at the URL link mentioned above.

4

Correction of First-Order Functional Programs

The debugging support for functional languages in current systems is poor [106], and there are no general purpose, good semantics-based debugging tools available. Traditional debugging tools for functional programming languages consist of tracers which help to display the execution [35, 58, 93, 95] but which do not enforce program correctness adequately as they do not provide means for finding nor repairing bugs in the source code w.r.t. the intended program semantics. This is particularly dramatic for equational languages such as those in the OBJ family, which includes OBJ3, CafeOBJ and Maude.

Abstract diagnosis of (first-order) functional programs (that is, TRSs) [12] is a declarative diagnosis framework extending the methodology of [38], which relies on (an approximation of) the immediate consequence operator $T_{\mathcal{R}}$, to identify bugs in functional programs. Given the intended specification $\mathcal{I}$ of the semantics of a program $\mathcal{R}$, the debugger checks the correctness of $\mathcal{R}$ by a single step of the abstract immediate consequence operator $T^{\kappa}_{\mathcal{R}}$, where the abstraction function κ refers to the $depth(k)$ cut abstraction [38]. Then, by a simple static test, the system can determine all the rules which are wrong w.r.t. a particular abstract property. The framework is goal independent and does not require the determination of symptoms in advance. This is in contrast with traditional, semi-automatic debugging of functional programs [92, 94, 104], where the debugger tries to locate the node in an execution tree, which is ultimately responsible for a visible bug symptom, by making questions to the oracle (typically the user). When debugging real code, the questions are often textually large and may be difficult to answer.

In this chapter, we endow the functional debugging method of [12] with a bug-correction program synthesis methodology which, after diagnosing the buggy program, tries to correct the erroneous components of the wrong code automatically. In this framework, we will consider first-order functional programs which are modeled by term rewriting systems (TRSs).

As in Chapter 3, the method uses unfolding in order to discriminate positive from negative examples (resp. uncovered and incorrect equations) which are automatically produced as an outcome by the diagnoser. Informally, our correction procedure works

as follows. Starting from an *overly general* program (that is, a program which covers all the positive examples as well as some negative ones), the algorithm unfolds the program and deletes program rules until reaching a suitable specialization of the original program which still covers all the positive examples and does not cover any negative one. Both, the example generation and the top-down correction processes, exploit some properties of the abstract interpretation framework of [12] which they rely on. Let us emphasize that we do not require any demanding condition on the class of the programs which we consider. This is particularly convenient in this context, since it should be undesirable to require strong properties, such as termination or confluence, for a buggy program which is known to contain errors.

In Chapter 3, following the hybrid correction scheme which we illustrated at the beginning of this dissertation, we presented a general bottom-up inductive generalization methodology along with a top-down inductive correction method. The former transforms a given program into a program which is "overly general" w.r.t. an example set, so that the latter can be applied. Since this bottom-up, inductive learning methodology can also be employed in the functional setting, in this chapter, we will only focus on the formalization of the functional top-down corrector.

We would like to clarify the contributions of this correction method w.r.t. the one presented in the previous chapter, where a different unfolding-based technique was developed which applies to synthetizing multiparadigm, functional-logic programs from a set of positive and negative examples. First, the method for automatically generating the example sets is new. In Chapter 3, (abstract) non-ground examples were computed as the outcome of an abstract debugger based on the *loop-check* techniques of [16], whereas now we compute (concrete) ground examples after a *depth-k* abstract diagnosis phase [38] which is conceptually much simpler and allows us to compute the example sets more efficiently. Regarding the top-down correction algorithm, the one proposed in this chapter significantly improves the previous method. We have been able to devise an abstract technique for testing the "overgenerality" applicability condition, which saves us from requiring program termination or the slightly weaker condition of *μ-termination* (termination of context-sensitive rewriting [79]). Finally, we have been able to prove the correctness of the new algorithm for a much larger class of programs, since we do not even need confluence whereas the previous method applies only to inductively sequential programs or canonical, completely defined systems (depending on the lazy/eager narrowing strategy chosen, i.e., $\varphi = needed$ or $\varphi = inn$).

The debugging methodology which we consider can be very useful for a functional programmer who wants to debug a program w.r.t. a preliminary version which was written with no efficiency concern. Actually, in software development a specification may be seen as the starting point for the subsequent program development, and as the criterion for judging the correctness of the final software product. Therefore, a debugging tool which is able to locate bugs in the user's program and correct the wrong code becomes also important in this context. In general, it also happens that some parts of the software need to be improved during the software life cycle, e.g. for getting a better performance. Then the old programs (or large parts of them) can be usefully (and automatically) used as a specification of the new ones. For in-

stance, the executability of OBJ specifications supports prototype-driven incremental development methods [60].

Structure of the chapter

The rest of the chapter is organized as follows. Section 4.1 recalls the abstract diagnosis framework for functional programs of [12]. Section 4.2 formalizes the correction problem in this framework. Section 4.3 illustrates the example generation methodology. Section 4.4 presents the top-down correction method together with some examples and correctness results.

4.1 Denotation of functional programs

In this section we first recall the semantic framework introduced in [12]. We will provide a finite/angelic relational semantics [41], given in fixpoint style, which associates an input-output relation to a program, while intermediate computation steps are ignored. Then, we formulate an abstract semantics which approximates the evaluation semantics of the program.

4.1.1 Concrete semantics

The considered concrete domain $\mathbb{C}$ is the lattice of $\mathcal{V}$-Herbrand interpretations, i.e., the powerset of $\mathcal{B}_\mathcal{V}$ ordered by set inclusion.

In the sequel, a semantics for program $\mathcal{R}$ is a $\mathcal{V}$-Herbrand interpretation. Since in functional programming, programmers are generally concerned with computing values (ground constructor normal forms), the semantics which is usually considered is $Sem_{\mathsf{val}}(\mathcal{R}) := \{s = t \mid s \to^!_\mathcal{R} t, t \in \tau(\mathcal{C})\}$. Sometimes, we will call *proof* of equation $s = t$, a rewrite sequence from term s to value t.

Following [41], in order to formalize our evaluation semantics via fixpoint computation, we consider the following immediate consequence operator.

Definition 4.1.1 [12] *Let $\mathcal{I}$ be a Herbrand interpretation, $\mathcal{R}$ be a TRS. Then,*

$$T_\mathcal{R}(\mathcal{I}) = \{t = t \mid t \in \tau(\mathcal{C})\} \cup \{s = t \mid r = t \in \mathcal{I}, s \to_\mathcal{R} r\}.$$

The following proposition is immediate.

Proposition 4.1.2 [12] *Let $\mathcal{R}$ be a TRS. The $T_\mathcal{R}$ operator is continuous on $\mathbb{C}$.*

Definition 4.1.3 [12] *The least fixpoint semantics of a program $\mathcal{R}$ is defined as*

$$\mathcal{F}_{\mathsf{val}}(\mathcal{R}) = T_\mathcal{R} \uparrow \omega.$$

```
obj GAMESPEC is
  sorts Nat Reward .
  op 0 : -> Nat .
  op s : Nat -> Nat .
  op prize : -> Reward .
  op sorry-no-prize : -> Reward .
  op coinflip : Nat -> Reward .
  op win? : Nat -> Reward .
  var X : Nat .
  eq coinflip(X) = win?(X) .
  eq win?(s(s(X))) = sorry-no-prize .
  eq win?(s(0)) = prize .
  eq win?(0) = sorry-no-prize .
endo
```

Figure 4.1: Coin-flip specification.

Example 4.1.4 *Suppose you toss a coin after having chosen one of its faces. If the face revealed after the coin flip is the predicted one, you win a prize. The problem can be modeled by the specification $\mathcal{I}$ (written in OBJ syntax) depicted in Figure 4.1.*

Face values are expressed by natural numbers `0` *and* `s(0)`*; besides, specification $\mathcal{I}$ tells us that we win the prize at stake (expressed by the constructor prize), if the revealed face is* `s(0)`*, while we get no prize whenever the revealed face is equal to* `0`*.*

The associated least fixpoint semantics is

$$\begin{aligned}\mathcal{F}_{\mathsf{val}}(\mathcal{I}) \quad = \quad \{\ & \texttt{prize} = \texttt{prize}, \texttt{sorry-no-prize} = \texttt{sorry-no-prize},\\ & \texttt{win?(0)} = \texttt{sorry-no-prize}, \texttt{win?(s(0))} = \texttt{prize},\\ & \texttt{win?(s(s(X))} = \texttt{sorry-no-prize},\\ & \texttt{coinflip(0)} = \texttt{sorry-no-prize},\\ & \texttt{coinflip(s(0))} = \texttt{prize},\\ & \texttt{coinflip(s(s(X))} = \texttt{sorry-no-prize}\ \}.\end{aligned}$$

The following result establishes the equivalence between the (fixpoint) semantics computed by the $T_{\mathcal{R}}$ operator and the evaluation semantics $Sem_{\mathsf{val}}(\mathcal{R})$.

Theorem 4.1.5 (soundness and completeness) [12] *Let $\mathcal{R}$ be a TRS. Then,*

$$Sem_{\mathsf{val}}(\mathcal{R}) = \mathcal{F}_{\mathsf{val}}(\mathcal{R}).$$

Note that if $e \equiv (l = c)$ belongs to the (fixpoint) semantics S of $\mathcal{R}$, then for each proof $e \to e_1 \to e_2 \ldots e_n \to (c = c)$ of e in $\mathcal{R}$, e_i belongs to S, $i = 1, \ldots, n$. That is, the semantics models all partial computations of equations in $\mathcal{R}$.

4.1.2 Abstract semantics

Starting from the concrete fixpoint semantics of Definition 4.1.3, we give an abstract semantics which approximates the concrete one by means of abstract interpretation techniques. In particular, we will focus our attention on abstract interpretations achieved by means of a *depth*(k) cut [38], which allows to finitely approximate an infinite set of computed equations.

First of all we define a *term abstraction* as a function

$$/_k : (\tau(\Sigma, \mathcal{V}), \leq) \rightarrow (\mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}}), \leq)$$

which cuts terms having a depth greater than k. Terms are cut by replacing each subterm rooted at depth k with a new variable taken from the set $\hat{\mathcal{V}}$ (disjoint from $\mathcal{V}$). *depth*(k) terms represent all the terms which are obtained by instantiating the variables of $\hat{\mathcal{V}}$ with terms built over $\tau(\Sigma, \mathcal{V})$. Note that $/_k$ is finite. We denote by $T/_k$ the set of *depth*(k) terms $(\mathcal{T}(\Sigma, \mathcal{V} \cup \hat{\mathcal{V}})/_k)$. We choose as abstract domain $\mathbb{A}$ the set $\mathcal{P}(\{a = a' \mid a, a' \in T/_k\})$ ordered by the Smyth's extension of ordering $\leq$ to sets, i.e. $X \leq_S Y$ iff $\forall\, y \in Y\ \exists\, x \in X : x \leq y$. Thus, we can lift the term abstraction $/_k$ to a Galois Insertion of $\mathbb{A}$ into $\mathbb{C}$ by defining

$$\begin{array}{ll} \kappa(E) & := \{s/_k = t/_k \mid s = t \in E\} \\ \gamma(A) & := \{s = t \mid s/_k = t/_k \in A\} \end{array}$$

Now we can derive the optimal abstract version of $T_{\mathcal{R}}$ simply as $T^{\kappa}_{\mathcal{R}} := \kappa \circ T_{\mathcal{R}} \circ \gamma$ and define the abstract semantics of program $\mathcal{R}$ as the least fixpoint of this (obviously) continuous operator, i.e. $\mathcal{F}^{\kappa}_{\mathsf{val}}(\mathcal{R}) := T^{\kappa}_{\mathcal{R}} \uparrow \omega$. Since $/_k$ is finite, we are guaranteed to reach the fixpoint in a finite number of steps, that is, there exists a finite natural number h such that $T^{\kappa}_{\mathcal{R}} \uparrow \omega = T^{\kappa}_{\mathcal{R}} \uparrow h$. Abstract interpretation theory assures that $T^{\kappa}_{\mathcal{R}} \uparrow \omega$ is the best correct approximation of $Sem_{\mathsf{val}}(\mathcal{R})$, where correct means

$$\mathcal{F}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \leq_S \kappa(Sem_{\mathsf{val}}(\mathcal{R}))$$

and best means that it is the maximum w.r.t. $\leq_S$.

By the following proposition, we provide a simple and effective mechanism to compute the abstract fixpoint semantics.

Proposition 4.1.6 [12] *For $k > 0$, the operator $T^{\kappa}_{\mathcal{R}} : T/_k \times T/_k \rightarrow T/_k \times T/_k$ holds the property $\widetilde{T}^{\kappa}_{\mathcal{R}}(X) \leq_S T^{\kappa}_{\mathcal{R}}(X)$ w.r.t. the following operator:*

$$\widetilde{T}^{\kappa}_{\mathcal{R}}(X) = \kappa(B) \cup \{\sigma(u[l]_p)/_k = t \mid \begin{array}{l} u = t \in X, p \in O_{\Sigma \cup \mathcal{V}}(u), \\ l \rightarrow r \ll \mathcal{R}, \sigma = mgu(\{u|_p = r\})\} \end{array}$$

where $B = \{t = t \mid t \in \tau(\mathcal{C})\}$.

Definition 4.1.7 [12] *The abstract least fixpoint semantics of a program $\mathcal{R}$ is defined as $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) = \widetilde{T}^{\kappa}_{\mathcal{R}} \uparrow \omega$.*

Proposition 4.1.8 (Correctness) [12] *Let $\mathcal{R}$ be a TRS and $k > 0$.*

1. $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \leq_S \kappa(\mathcal{F}_{\mathsf{val}}(\mathcal{R})) \leq_S \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.
2. *For every $e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ such that $Var(e) \cap \hat{\mathcal{V}} = \emptyset$, $e \in \mathcal{F}_{\mathsf{val}}(\mathcal{R})$.*

Example 4.1.9 *Consider again the specification in Example 4.1.4. Its abstract least fixpoint semantics for $\kappa = 2$ (without considering symbol* `win?`*) becomes*

$$\begin{aligned}\widetilde{\mathcal{F}}^{2}_{\mathsf{val}}(\mathcal{I}) \quad = \quad \{\ & \mathtt{prize} = \mathtt{prize}, \mathtt{sorry\text{-}no\text{-}prize} = \mathtt{sorry\text{-}no\text{-}prize}, \\ & \mathtt{coinflip(0)} = \mathtt{sorry\text{-}no\text{-}prize}, \mathtt{coinflip(s(0))} = \mathtt{prize}, \\ & \mathtt{coinflip(s(s(\hat{X})))} = \mathtt{sorry\text{-}no\text{-}prize}\ \}.\end{aligned}$$

4.2 The correction problem

The problem of repairing a faulty functional program can be addressed by using inductive learning techniques guided by appropriate examples. Roughly speaking, given a wrong program and two example sets specifying positive (pursued) and negative (not pursued) computations respectively, our correction scheme aims at synthesizing a set of program rules that replaces the wrong ones in order to deliver a corrected program which is "consistent" w.r.t. the example sets [29].

More formally, we can reformulate the correction problem which we stated in Chapter 1 in an equivalent way as follows.

Let $\mathcal{R}$ be a TRS, $\mathcal{I}$ be a specification of the intended semantics of $\mathcal{R}$, E^+ and E^- be two finite sets of equations such that

1. $E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{I})$;
2. $E^- \cap (Sem_{\mathsf{val}}(\mathcal{R}) \setminus Sem_{\mathsf{val}}(\mathcal{I})) \neq \emptyset$.

The correction problem consists in constructing a TRS $\mathcal{R}^c$ satisfying the following requirements

1. $E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{R}^c)$;
2. $E^- \cap Sem_{\mathsf{val}}(\mathcal{R}^c) = \emptyset$.

Equations in E^+ (resp. E^-) are called *positive* (resp. *negative*) examples. The TRS $\mathcal{R}^c$ is called *correct* program w.r.t. E^+ and E^-. Note that, by construction, positive and negative example sets are disjoint, which permits to drive the correction process towards a discrimination between E^+ and E^-.

4.3 How to generate example sets automatically

Before giving a constructive method to derive a correct program, we present a simple methodology for automatically generating example sets, so that the user does not

need to provide error symptoms, evidences or other kind of information which would require a good knowledge of the program semantics.

In the following, we observe that we can easily compute "positive" equations, i.e. equations which appear in the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{I})$, since all equations in $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I})$ not containing variables in $\hat{\mathcal{V}}$ belong to the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{I})$, as stated in the following lemma.

Lemma 4.3.1 *Let $\mathcal{I}$ be a TRS and $E_P := \{e | e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \wedge Var(e) \cap \hat{\mathcal{V}} = \emptyset\}$. Then, $E_P \subseteq Sem_{\mathsf{val}}(\mathcal{I})$.*

Now, by exploiting the information in $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I})$ and $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$, we can also generate a set of "negative" equations which belong to the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{R})$ of the wrong program $\mathcal{R}$ but not to the concrete evaluation semantics $Sem_{\mathsf{val}}(\mathcal{I})$ of the specification $\mathcal{I}$.

Lemma 4.3.2 *Let $\mathcal{R}$ be a TRS, $\mathcal{I}$ be a specification of the intended semantics and $E_N := \{e | e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \wedge Var(e) \cap \hat{\mathcal{V}} = \emptyset \wedge \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \not\leq_S \{e\}\}$. Then,*

$$E_N \subseteq (Sem_{\mathsf{val}}(\mathcal{R}) \setminus Sem_{\mathsf{val}}(\mathcal{I})).$$

Starting from sets E_P and E_N, we construct the positive and negative example sets E^+ and E^- which we use for the correctness process, by considering the restriction of E_P and E_N to examples of the form $l = c$ where l is a pattern and c is a value. By considering these "data" examples, the inductive process becomes independent from the extra auxiliary functions which might appear in $\mathcal{I}$, since we start synthesizing directly from data structures.

The sets E^+ and E^- are defined as follows.

$$E^+ = \{l = c \mid \quad f(t_1, \ldots, t_n) = c \in E_P \wedge f(t_1, \ldots, t_n) \equiv l \text{ is a pattern} \wedge \\ \wedge\, c \in \tau(\mathcal{C}) \wedge f \in \Sigma_{\mathcal{R}}\}$$

$$E^- = \{l = c \mid \quad l = c \in E_N \wedge l \text{ is a pattern } \wedge c \in \tau(\mathcal{C})\}$$

where $\Sigma_{\mathcal{R}}$ is the signature of program $\mathcal{R}$.

In the sequel, the function which computes the sets E^+ and E^-, according to the above description, is called ExGen$(\mathcal{R}, \mathcal{I})$.

4.4 Example-guided unfolding

In this section we present a basic top-down correction method based on the so-called *example-guided unfolding* [29], which is able to specialize a program by applying unfolding and deletion of program rules until coming up with a correction. The top-down correction process is "guided" by the examples, in the sense that transformation steps focus on discriminating positive from negative examples. The accuracy of the correction improves as the number of positive and negative examples increase as it is common to the learning from examples approach.

In order to successfully apply the method, the semantics of the program to be specialized must include the positive example set E^+ (that is, $E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{R})$). Programs satisfying this condition are called *overly general* (w.r.t. E^+).

The over-generality condition is not generally decidable, as we do not impose program termination [6]. Fortunately, when we consider the abstract semantics framework of [12] we are able to ascertain a useful sufficient condition to decide whether a program is overly general, even if it does not terminate. The following proposition formalizes our method.

Proposition 4.4.1 *Let $\mathcal{R}$ be a TRS and E^+ be a set of positive examples. If, for each $e \in E^+$, there exists $e' \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ s.t.*

1. *$e' \leq e$;*
2. *$Var(e') \cap \hat{\mathcal{V}} = \emptyset$;*

then, $\mathcal{R}$ is overly general w.r.t. E^+.

Now, by exploiting Proposition 4.4.1, it is not difficult to figure out a procedure OverlyGeneral$(\mathcal{R}, E)$ testing this condition w.r.t. a program $\mathcal{R}$ and a set of examples E, e.g. a boolean function returning *true* if program $\mathcal{R}$ is overly general w.r.t. E and *false* otherwise.

4.4.1 The unfolding operator

Informally, *unfolding* a program $\mathcal{R}$ w.r.t. a rule r delivers a new specialized version of $\mathcal{R}$ in which the rule r is replaced with new rules obtained from r by performing a narrowing step on the rhs of r.

Definition 4.4.2 *Given two rules $r_1 \equiv \lambda_1 \to \rho_1$ and r_2, we define the* rule unfolding *of r_1 w.r.t. r_2 as*

$$\mathcal{U}_{r_2}(r_1) = \{\lambda_1\sigma \to \rho' \mid \rho_1 \overset{\sigma}{\leadsto}_{r_2} \rho'\}.$$

Definition 4.4.3 *Given a TRS $\mathcal{R}$ and a rule $r \ll \mathcal{R}$, we define the* program unfolding *of r w.r.t. $\mathcal{R}$ as follows*

$$\mathcal{U}_{\mathcal{R}}(r) = \left(\mathcal{R} \cup \bigcup_{r' \in \mathcal{R}} \mathcal{U}_{r'}(r)\right) \setminus \{r\}.$$

Note that, by Definition 4.4.3, for any TRS $\mathcal{R}$ and rule $r \ll \mathcal{R}$, r is never in $\mathcal{U}_{\mathcal{R}}(r)$.

Definition 4.4.4 *Let $\mathcal{R}$ be a TRS, r be a rule in $\mathcal{R}$. The rule r is* unfoldable *w.r.t. $\mathcal{R}$ if $\mathcal{U}_{\mathcal{R}}(r) \neq \mathcal{R} \setminus \{r\}$.*

Now, we are ready to prove that the "transformed" semantics, obtained after applying the unfolding operator to a given program $\mathcal{R}$, still contains the semantics of $\mathcal{R}$. In symbols, $Sem_{\mathsf{val}}(\mathcal{R}) \subseteq Sem_{\mathsf{val}}(\mathcal{U}_{\mathcal{R}}(r))$, where r is an unfoldable rule. We call this property *unfolding correctness*.

The following definitions are auxiliary.

Definition 4.4.5 *Let $t \in \tau(\Sigma, \mathcal{V})$ and $\perp$ be a symbol not in Σ. The* shell *of t, in symbols $shell(t)$, is defined as follows*

$$shell(t) = \begin{cases} f(shell(t_1), \ldots, shell(t_n)) & \text{if } t \equiv f(t_1, \ldots, t_n), \text{ where } f \in \mathcal{D} \\ \perp & \text{otherwise} \end{cases}$$

Definition 4.4.6 *Let $(\Sigma, \mathcal{R})$ be a TRS where $\Sigma = \mathcal{C} \cup \mathcal{D}$. $\mathcal{R}$ is* well-framed, *if for each $\lambda \to \rho \in \mathcal{R}$, $O_{\mathcal{D}}(shell(\rho)) = O_{\mathcal{D}}(\rho)$.*

The following theorem establishes the correctness of unfolding even for non-confluent programs, provided that they are well-framed. Note that well-framedness is much less demanding and easy to check.

Theorem 4.4.7 (unfolding correctness) *Let $\mathcal{R}$ be a well-framed left-linear CS, $r \ll \mathcal{R}$ be an unfoldable rule and $\mathcal{R}' = \mathcal{U}_{\mathcal{R}}(r)$. Let $e \equiv (l = c)$ be an equation such that $l \in \tau(\Sigma, \mathcal{V})$ and $c \in \tau(\mathcal{C})$. Then, $Sem_{\mathsf{val}}(\mathcal{R}) \subseteq Sem_{\mathsf{val}}(\mathcal{R}')$.*

Finally, some important program properties such as well-framedness and program termination are preserved through unfolding.

4.4.2 The top-down correction algorithm

Basically, the idea behind the basic correction algorithm is to eliminate rules from the program in order to get rid of the negative examples without losing the derivations for the positive ones. Clearly, this cannot be done by naïvely removing program rules, since sometimes a rule is used to prove both a positive and a negative example. So, before applying deletion, we need to specialize programs in order to ensure that the deletion phase only affects those program rules which are not necessary for proving the positive examples. This specialization process is carried out by means of the unfolding operator of Definition 4.4.3. Considering this operator for specialization purposes has important advantages. First, positive examples are not lost by repeatedly applying the unfolding operator, since unfolding preserves the proper semantics (see Theorem 4.4.7). Moreover, the nature of unfolding is to "compile" rewrite steps into the program, which allows us to shorten and distinguish the rewrite rules which occur in the proofs of the positive and negative examples.

Algorithm 4 formalizes the correction procedure, called TDCorrectorF, which takes as input a program $\mathcal{R}$ and a specification of the intended semantics $\mathcal{I}$, also expressed as a program. First, TDCorrectorF computes the example sets E^+ and E^- by means of ExGen, following the method presented in Section 4.3. Then, it checks whether program $\mathcal{R}$ is overly general following the scheme of Proposition

4.4.1, and finally it enters the main correction process. In order to ensure correctness of the algorithm, we require k to be greater than or equal to the maximum depth of the terms occurring in E^+.

This last phase consists of a main loop, in which we perform an unfolding step followed by a rule deletion until no negative example is covered (approximated) by the abstract semantics of the current transformed program $\mathcal{R}_n$. This amounts to saying that no negative example belongs to the concrete semantics of $\mathcal{R}_n$. We note that the **while** loop guard is easy to check, as the abstract semantics is finitely computable. Note the difference w.r.t. the algorithm in the previous chapter, where decidability is ensured by requiring both confluence and (μ-termination) of the program.

During the unfolding phase, we select a rule upon which performing a program unfolding step. In order to specialize the program w.r.t. the example sets, we pick up an unfoldable rule which occurs in some proof of a positive example by using a fair selection strategy. More precisely, we choose rules which appear first in proofs of positive examples. Those rules can be easily computed retrieving them from the fixpoint semantics $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}$ by a further step of the $\widetilde{T}^{\kappa}$ operator by means of the auxiliary function *first*, which is formally defined as follows.

Definition 4.4.8 *Let $\mathcal{R}$ be a TRS and E be an example set. Then, we define*

$$\mathit{first}(E) := \bigcup_{e \in E} \{r \mid e \in \widetilde{T}^{\kappa}_{\{r\}}(\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}))\}.$$

Once unfolding has been accomplished, we proceed to remove the "redundant" rules, that is, all the rules which are not needed to prove the positive example set E^+. This can be done by repeatedly testing the overgenerality of the specialized program w.r.t. E^+ and removing one rule at each iteration of the inner **for** loop. Roughly speaking, if program $\mathcal{R}_k \setminus \{r\}$ is overly general w.r.t. E^+, then rule r can be safely eliminated without losing E^+. Then, we can repeat the test on another rule.

Let us consider the coin-flip game to illustrate our algorithm.

Example 4.4.9 *The OBJ program $\mathcal{R}$ of Figure 4.2 is wrong w.r.t. the specification $\mathcal{I}$ of Example 4.1.4. Note that program $\mathcal{R}$ is non-confluent and computes both values* `prize` *and* `sorry-no-prize` *for any natural* $\mathtt{s}^n\mathtt{(0)}$, $n > 0$.

By fixing $\kappa = 2$, we compute the following least fixpoint abstract semantics for $\mathcal{R}$.

$$\begin{aligned}\widetilde{\mathcal{F}}^{2}_{\mathsf{val}}(\mathcal{R}) \;=\; \{\; & \mathtt{prize} = \mathtt{prize},\\ & \mathtt{sorry\text{-}no\text{-}prize} = \mathtt{sorry\text{-}no\text{-}prize},\\ & \mathtt{coinflip(0)} = \mathtt{prize},\\ & \mathtt{coinflip(0)} = \mathtt{sorry\text{-}no\text{-}prize},\\ & \mathtt{coinflip(s(0))} = \mathtt{prize},\\ & \mathtt{coinflip(s(0))} = \mathtt{sorry\text{-}no\text{-}prize},\\ & \mathtt{coinflip(s(s(\hat{X})))} = \mathtt{prize},\\ & \mathtt{coinflip(s(s(\hat{X})))} = \mathtt{sorry\text{-}no\text{-}prize}\;\}.\end{aligned}$$

Algorithm 4 The top-down correction algorithm.

procedure TDCorrectorF($\mathcal{R}.\mathcal{I}$)
  $(E^+, E^-) \leftarrow$ ExGen$(\mathcal{R}, \mathcal{I})$
  if not OverlyGeneral$(\mathcal{R}, E^+)$ **then** Halt
  $k \leftarrow 0$; $\mathcal{R}_k \leftarrow \mathcal{R}$
  while $\exists\, e^- \in E^- : \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}_k) \leq_S \{e^-\}$ **do**
    $R \leftarrow \{r \in \mathcal{R}_k \mid r \text{ is unfoldable} \wedge r \in \mathit{first}(E^+)\}$
    if $R \neq \emptyset$ **then**
      Select(r, R)
      $\mathcal{R}_{k+1} \leftarrow \mathcal{U}_{\mathcal{R}_k}(r)$; $k \leftarrow k + 1$
    end if
    for each $r \in \mathcal{R}_k$ **do**
      if OverlyGeneral$(\mathcal{R}_k \setminus \{r\}, E^+)$ **then**
        $\mathcal{R}_k \leftarrow \mathcal{R}_k \setminus \{r\}$
      end if
    end for
  end while
  $\mathcal{R}^c \leftarrow \mathcal{R}_k$
end procedure

```
obj GAME is
    sorts Nat Reward .
    op 0 : -> Nat .
    op s : Nat -> Nat .
    op prize : -> Reward .
    op sorry-no-prize : -> Reward .
    op coinflip : Nat -> Reward .
    var X : Nat .
    eq coinflip(s(X)) = coinflip(X) .        (1)
    eq coinflip(0) = prize .                 (2)
    eq coinflip(0) = sorry-no-prize .        (3)
endo
```

Figure 4.2: Wrong coinflip OBJ program.

```
obj GAME is
   sorts Nat Reward .
   op 0 : -> Nat .
   op s : Nat -> Nat .
   op prize : -> Reward .
   op sorry-no-prize : -> Reward .
   op coinflip : Nat -> Reward .
   var X : Nat .
   eq coinflip(s(s(X))) = coinflip(X) .      (4)
   eq coinflip(s(0)) = prize .               (5)
   eq coinflip(s(0)) = sorry-no-prize .      (6)
   eq coinflip(0) = prize .                  (7)
   eq coinflip(0) = sorry-no-prize .         (8)
endo
```

Figure 4.3: Unfolded coinflip program.

Considering the abstract fixpoint semantics $\widetilde{\mathcal{F}}^2_{\mathsf{val}}(\mathcal{I})$ which is computed in Example 4.1.9 and following the methodology of Section 4.3, we obtain the example sets below:

$$E^+ = \{\texttt{coinflip(s(0))} = \texttt{prize}, \texttt{coinflip(0)} = \texttt{sorry-no-prize}\}$$
$$E^- = \{\texttt{coinflip(s(0))} = \texttt{sorry-no-prize}, \texttt{coinflip(0)} = \texttt{prize}\}.$$

Now, since program $\mathcal{R}$ fulfills the condition for overgenerality expressed by Proposition 4.4.1, the algorithm proceeds and enters the main loop. Here, program rule (1) *is unfolded, because* (1) *is unfoldable and belongs to first(E^+). So, the transformed program is the one depicted in Figure 4.3.*

Subsequently, a deletion phase is executed in order to check whether there are rules which are not needed to cover the positive example set E^+. The algorithm discovers that rules (4)*,* (6)*,* (7) *are not necessary, and therefore are removed producing the correct program which consists of rules* (5) *and* (8)*.*

Note that the above example cannot be repaired by using either the correction method of [6] or the method proposed in the previous chapter, since the wrong TRS is not confluent.

4.4.3 Correctness of algorithm TDCorrectorF

In this section, we prove the correctness of the top-down correction algorithm TDCorrectorF, i.e., we show that it produces a specialized version of $\mathcal{R}$ which is a correct program w.r.t. E^+ and E^-, provided that $\mathcal{R}$ is overly general w.r.t. E^+. A condition is necessary for establishing this result: no negative/positive couple of the considered examples must be proven by using the same sequence of rules.

Let us start by giving an auxiliary definition and some technical lemmata.

Definition 4.4.10 *Let $\mathcal{R}$ be a TRS and E be a set of examples. The* unfolding succession $\mathcal{US}(\mathcal{R}) \equiv \mathcal{R}_0, \mathcal{R}_1, \ldots$ *of program $\mathcal{R}$ w.r.t. E is defined as follows:*

$$\mathcal{R}_0 = \mathcal{R}, \quad \mathcal{R}_{i+1} = \begin{cases} \mathcal{U}_{\mathcal{R}_i}(r) & \text{where } r \in \mathcal{R}_i \text{ is unfoldable and } r \in \mathit{first}(E) \\ \mathcal{R}_i & \text{otherwise} \end{cases}$$

The next results state that we are always able to transform a program $\mathcal{R}$ into a program $\mathcal{R}'$ by a suitable number of unfolding steps, in such a way that any given proof of an example in $\mathcal{R}$ can be mimicked by a one-step proof in $\mathcal{R}'$. In the following we denote the length of a rewrite sequence $\mathcal{S}$ by $|\mathcal{S}|$.

Lemma 4.4.11 *Let $\mathcal{R}$ be a well-framed, left-linear CS, E be a set of examples and $r \in \mathit{first}(E)$ be an unfoldable rule such that $\mathcal{R}' = \mathcal{U}_{\mathcal{R}}(r)$. Let $t = c \in E$, where t is a pattern and c is a value. Then,*

1. *if $\mathcal{S}$ is a rewrite sequence from t to c in $\mathcal{R}$, then there exists a rewrite sequence $\mathcal{S}'$ from t to c in $\mathcal{R}'$;*
2. *if r occurs in $\mathcal{S}$, then $|\mathcal{S}'| < |\mathcal{S}|$.*

Lemma 4.4.12 *Let $\mathcal{R}$ be a well-framed, left-linear CS, E be an example set and $t = c \in E$, where t is a pattern and c is a value. Let $t \to_{r_1} \ldots \to_{r_n} c$, $n \geq 1$. Then, for each unfolding succession $\mathcal{US}(\mathcal{R})$ w.r.t. E, there exists $\mathcal{R}_k$ occurring in $\mathcal{US}(\mathcal{R})$ such that $t \to_{r^*} c$, $r^* \in \mathcal{R}_k$.*

The following result immediately derives from Claim (i) of Proposition 4.1.8 .

Lemma 4.4.13 *Let $\mathcal{R}$ be a TRS and e an equation. Then, if $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \not\leq_S \{e\}$, then $e \notin \mathit{Sem}_{\mathsf{val}}(\mathcal{R})$.*

The following definition is auxiliary. We say that the pair of positive an negative example sets (E^+, E^-) is *discriminable* in $\mathcal{R}$ [29], if there are no $e^+ \in E^+$ and $e^- \in E^-$ which can be proven by using the same sequence of rules of $\mathcal{R}$. This property can be checked by using standard tools for proving program termination.

Now we are ready to prove the partial correctness and the termination of the algorithm.

Theorem 4.4.14 *Let $\mathcal{R}$ be a well-framed, left-linear CS and $\mathcal{I}$ be a specification of the intended semantics of $\mathcal{R}$. Let E^+ and E^- be the example sets generated by* ExGen$(\mathcal{R}, \mathcal{I})$.

1. *If (E^+, E^-) is discriminable in $\mathcal{R}$, then the algorithm* TDCorrectorF$(\mathcal{R}, \mathcal{I})$ *terminates.*
2. *If $\mathcal{R}$ is overly general w.r.t. E^+ and* TDCorrectorF$(\mathcal{R}, \mathcal{I})$ *terminates, then the computed program $\mathcal{R}^c$ is correct w.r.t. E^+ and E^-.*

We note that well-framedness can be dispensed in exchange for requiring complete definedness (CD), i.e the property that functions are defined on all possible values of their arguments, as in [18] by simply delaying the deletion phase and performing a virtual deletion instead. This suffices to ensure that the CD property is preserved during the transformation.

II

Web Site Verification

5

A Language for Verifying Web sites

In this chapter, we first provide a rewriting-based, formal specification language which allows us to define conditions on both the structure and the contents of Web sites in a simple and concise way. For instance, it allows us to recognize erroneous information inside the Web site and, additionally, to enforce that some information is available at a given Web page, some links between pages do exist or even the existence of the Web pages themselves. In our formalism, web pages (XHTML/XML documents) are modeled as Herbrand terms, and, consequently, Web sites are finite sets of terms. Then, we formalize a verification technique in which a Web site is checked w.r.t. a given Web specification in order to detect incorrect data and incomplete and/or missing Web pages. Moreover, by analyzing the error symptoms gathered during the verification process, we are also able to

- exactly localize the incorrect/forbidden information;
- find out the missing information which should be provided to repair the Web site.

Since reasoning on the Web calls for formal methods specifically fitting the Web context, we combine a standard regular expression methodology with a novel, rewriting-based technique called *partial rewriting*, in which the traditional pattern matching mechanism is replaced by tree *simulation* [66] in order to provide a suitable mechanism for recognizing patterns inside semistructured documents. The notion of simulation has been already used before for dealing with semistructured data in a number of query and transformation languages [31, 33, 54, 40]. The reason is twofold: on the one hand, it provides a powerful method to extract information from semistructured data; on the other hand, there exist efficient algorithms for computing simulations [66]. To assess the feasibility and efficiency of our approach, we have implemented the preliminary prototype system GVERDI (VErification and Rewriting for Debugging Internet sites), which is based on the verification methodology that we propose and is publicly available online. Following the "tolerant" approach of XLINKIT [48, 89], we

do not force the immediate repairing of the Web site, but simply provide the diagnosis information that enables document owners to decide on further actions.

Structure of the chapter

Section 5.1 summarizes some preliminary definitions and terminologies. In Section 5.2, a term representation of Web sites is provided together with a simple method for translating XHTML/XML documents into Herbrand terms. Section 5.3 is devoted to formalizing the specification language, whereas Section 5.4 formalizes the *partial rewriting* mechanism, which is based on page simulation. Section 5.5 introduces our verification technique for detecting incorrect as well as missing/incomplete Web pages. Some notes regarding the implementation of the system GVERDI are given in Section 5.6.

5.1 Basic notions

In this section, we will recall some basic definitions which are needed for understanding the rest of the chapter. For the sake of clarity, we will redefine the notion of term (and, consequently, of some related notions), given in Chapter 2, by means of an equivalent formalization, which is more convenient in this context.

We call a finite set of symbols *alphabet*. Given the alphabet A, A^* denotes the set of all finite sequences of elements over A. Syntactic equality between objects is represented by $\equiv$.

By $\mathcal{V}$ we denote a countably infinite set of variables and Σ denotes a set of function symbols, or *signature*. We consider varyadic signatures as in [43] (i.e., signatures in which symbols have an unbounded arity, that is, they may be followed by an arbitrary number of arguments). $\tau(\Sigma, \mathcal{V})$ and $\tau(\Sigma)$ denote the *non-ground term algebra* and the *term algebra* built on $\Sigma \cup \mathcal{V}$ and Σ, respectively.

Terms are viewed as labelled trees in the following (non-standard) way: a term in $\tau(\Sigma)$ is a tree $(V, E, r, label)$, where V is a set of vertices, E is a set of edges (i.e. pairs of vertices), $r \in V$ is the *root* vertex and *label* is a *labeling* function $V \to \Sigma \cup \mathcal{V}$. Let us see a small example.

Example 5.1.1 *Consider the term $t \equiv f(h(a), X)$ in $\tau(\{f, h, a\}, \{X\})$, which is illustrated in Figure 5.1. Term t can be represented by the structure $(V, E, r, label)$, where*

$$V \equiv \{v_0, v_1, v_2, v_3\}, E \equiv \{(v_0, v_1), (v_0, v_2), (v_1, v_3)\}, r \equiv v_0$$

and function label is defined as follows: $label(v_0) = f$, $label(v_1) = h$, $label(v_2) = X$, $label(v_3) = a$.

Given two vertices $v, v' \in V$ of a term $t \equiv (V, E, r, label)$, by $v \geq v'$ we mean that v is a *descendant* of v' in t. By $t|_v$ we mean the subterm rooted at vertex v of t. $t[r]_v$ is the term t with the subterm rooted at vertex v replaced by term r.

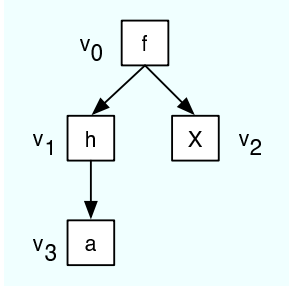

Figure 5.1: Term representation of $f(h(a), X)$.

We denote the *depth* of a vertex v in a term t, that is the number of edges between r and v in t, as $depth(t, v)$. A *substitution* $\sigma \equiv \{X_1/t_1, X_2/t_2, \ldots\}$ is a mapping from the set of variables $\mathcal{V}$ into the set of terms $\tau(\Sigma, \mathcal{V})$ satisfying the conditions given in Section 2.1.2. By $Var(s)$ we denote the set of variables occurring in the syntactic object s.

In the following, we consider marked terms. Given Σ and $\mathcal{V}$, we denote the *marked* version of Σ ($\mathcal{V}$, respectively) as $\underline{\Sigma}$ ($\underline{\mathcal{V}}$, respectively). A syntactic object $\underline{o} \in \underline{\Sigma} \cup \underline{\mathcal{V}}$ is called the *marked version* of $o \in \Sigma \cup \mathcal{V}$. Given a term $t \equiv (V, E, r, label) \in \tau(\Sigma, \mathcal{V})$, a *marking* for t is a (boolean) function $\mu\colon V \to \{yes, no\}$. The *empty* marking ε for t is a marking for t, such that $\varepsilon(v) = no$, for each $v \in V$. We define the *marked part* of a term t as

$$mark(t, \mu) \equiv (\{v \in V \mid \mu(v) = yes\}, \{(v_1, v_2) \in E \mid \mu(v_1) = \mu(v_2) = yes\}, r, label\ ).$$

A *valid* marking μ for a term $t \equiv (V, E, r, label)$ is the empty marking for t or a marking for t such that the two following conditions hold:

1. $\mu(r) = yes$;
2. $mark(t, \mu)$ is a term in $\tau(\Sigma, \mathcal{V})$.

Given a term $t \equiv (V, E, r, label)$ and a valid marking μ for t, by slightly abusing notation, a *marked* term $\mu(t)$ is a term in $\tau(\Sigma \cup \underline{\Sigma}, \mathcal{V} \cup \underline{\mathcal{V}})$ such that, for each vertex $v \in V$, the label associated with v in t is replaced by its marked version in $\mu(t)$, whenever $\mu(v) = yes$.

When no confusion can arise, we simply denote the marked term $\varepsilon(t)$ by t.

Example 5.1.2 *Consider again term $t \equiv (f(h(a), X))$ of Example 5.1.1. Let μ_1 be a marking for t defined as $\mu_1(v_0) = \mu_1(v_2) = \mu_1(v_3) = yes$, $\mu_1(v_1) = no$. Additionally, let μ_2 be a marking for t such that $\mu_2(v_0) = \mu_2(v_1) = yes$, $\mu_2(v_2) = \mu_2(v_3) = no$. Note that μ_1 is not a valid marking for t as the marked part of t is not a term in $\tau(\{f, h, a\}, \{X\})$ (see Figure 5.2 (a)) , whereas μ_2 is valid for t and $\mu_2(t) = \underline{f}(\underline{h}(a), X)$ is a marked term (see Figure 5.2 (b)).*

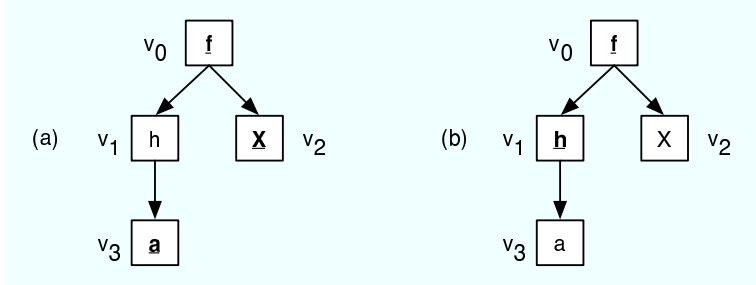

Figure 5.2: Examples of valid and non valid markings for the term $f(h(a), X)$.

In order to formalize our specification language, we will take advantage of the computational model, provided by term rewrting systems, which we explained in Section 2.2. Specifically, we will consider the class of the canonical TRSs. In this context, an equation $s = t$ *holds* in a canonical TRS R, if there exists $z \in \tau(\Sigma, \mathcal{V})$ such that $s \to_R^! z$ and $t \to_R^! z$.

Example 5.1.3 *Let R be the following canonical TRS*

$$\begin{aligned}
\mathtt{sum}(\mathtt{X}, \mathtt{0}) &\to \mathtt{X} \\
\mathtt{sum}(\mathtt{s}(\mathtt{X}), \mathtt{Y}) &\to \mathtt{s}(\mathtt{sum}(\mathtt{X}, \mathtt{Y})) \\
\\
\mathtt{append}(\mathtt{L}_1, [\,]) &\to \mathtt{L}_1 \\
\mathtt{append}([\mathtt{X}|\mathtt{L}_1], \mathtt{L}_2) &\to [\mathtt{X}|\mathtt{append}(\mathtt{L}_1, \mathtt{L}_2)] \\
\\
\leq (\mathtt{0}, \mathtt{X}) &\to \mathtt{true} \\
\leq (\mathtt{s}(\mathtt{X}), \mathtt{0}) &\to \mathtt{false} \\
\leq (\mathtt{s}(\mathtt{X}), \mathtt{s}(\mathtt{Y})) &\to \leq (\mathtt{X}, \mathtt{Y})
\end{aligned}$$

By means of TRS R we define three functions. The function sum *computes the sum of two natural numbers, the function* append *concatenates two lists, and the function $\leq$ defines the "less or equal to" relation between natural numbers. Natural numbers are represented in Peano's notation by means of the constant* 0 *and the unary operator* s*. By abuse, we will write $n \in \mathbb{N}$ as a shorthand for $\mathtt{s}^n(\mathtt{0})$. We use the standard notation for lists with [ ] being the empty list. Strings are viewed as lists of characters as usual.*

5.2 Denotation of Web sites

In our framework, a *Web page* is either an XML [107] or an XHTML [109] document, which we assume to be well-formed, as there are already programs and online services such as Tidy [90] which are able to validate and correct the XHTML/XML syntax, or Doctor HTML [73], which also performs link-checking.

Let us consider two alphabets T and $\mathcal{T}ag$. We denote the set T^* by $\mathcal{T}ext$. An object $\mathtt{t} \in \mathcal{T}ag$ is called *tag* element, while an element $\mathtt{w} \in \mathcal{T}ext$ is called *text* element. Since Web pages are provided with a tree-like structure, they can be straightforwardly translated into ordinary terms of a given term algebra $\tau(\mathcal{T}ext \cup \mathcal{T}ag)$ as shown in Figure 5.3. Note that XML/XHTML tag attributes can be considered as common tagged elements, and hence translated in the same way.

```
<members>
   <member status="professor">
        <name> mario </name>
        <surname> rossi </surname>
   </member>
   <member status="technician">
       <name> franca </name>
       <surname> bianchi </surname>
   </member>
   <member status="student">
       <name> giulio </name>
       <surname> verdi </surname>
   </member>
</members>
```

```
members(
  member(status(professor),
         name(mario),
         surname(rossi)
  ),
  member(status(technician),
         name(franca),
         surname(bianchi)
  )
  member(status(student),
         name(giulio),
         surname(verdi)
  )
)
```

Figure 5.3: An XML document and its corresponding encoding as a ground term $\mathtt{p}$.

A *marked* Web page is defined as $\mu(\mathtt{p})$, where $\mathtt{p} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag)$ and μ is a valid marking for $\mathtt{p}$. A *Web site* is a finite collection of marked Web pages $\{\varepsilon(\mathtt{p}_1) \ldots \varepsilon(\mathtt{p}_n)\}$. In the following, we will also consider terms of the non-ground term algebra $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$, which may contain variables. An element $\mathtt{s} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ is called *Web page template*. $\mu(\mathtt{s})$ is a *marked* Web page template, when $\mathtt{s} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ and μ is a valid marking for $\mathtt{s}$. In our methodology, (marked) Web page templates are used for specifying properties on Web sites as described in the following section.

Example 5.2.1 *In Figure 5.4, we present a Web site* $\mathtt{W}$ *of a research group, which contains information about group members affiliation, scientific publications, research projects, teaching and personal data.*

```
{(1) members(member(name(mario),surname(rossi),status(professor)),
             member(name(franca),surname(bianchi),status(technician)),
             member(name(giulio),surname(verdi),status(student)),
             member(name(mario),surname(rossi),status(professor))
            ),

  (2) hpage(fullname(mariorossi),phone(3333),status(professor),
            hobbies(hobby(reading),hobby(gardening))),

  (3) hpage(fullname(francabianchi),status(technician),phone(5555),
            links(link(url(www.google.com),urlname(google)),
            link(url(www.sexycalculus.com),urlname(FormalMethods))),

  (4) hpage(fullname(annagialli),status(professor),blink(phone(4444)),
            teaching(courselink(url(http://www.algebra.math),
            urlname(Algebra)))),

  (5) pubs(pub(name(mario),surname(rossi),title(blah1),year(2003)),
           pub(name(anna),surname(gialli),title(blah2),year(2002))),

  (6) projects(project(pname(A1),grant1(1000),grant2(200),
                       total(1200),cordinator(fullname(mariorossi))),
               project(pname(B1),grant1(800),grant2(300),
                       projectleader(surname(gialli),name(anna)),
                       total(1000)))}
```

Figure 5.4: An example of Web site

5.3 Web specification language

In the following, we present a term rewriting specification language, which is helpful to express properties about the content and the structure of a given Web site. Roughly speaking, a Web specification is a pair of finite sets of rules. The first set of rules describes constraints for detecting erroneous Web pages (*correctNess rules*) as well as for discovering incomplete/missing Web pages (*coMpleteness rules*). Diagnoses are carried out by running Web specifications on Web sites. The operational mechanism, formalized in Section 5.4, is based on a novel rewriting-based technique, which is able to extract partial structure from a term, and then rewrite it.

The second set of rules R contains the definition of some auxiliary functions which the user would like to provide, such as string processing, arithmetic, boolean operators, etc. It is formalized as a canonical term rewriting system which is handled by standard rewriting [76]. This implies that each input term t can be univocally reduced to an *irreducible form*.

Correctness rules allow us to recognize incorrect/forbidden patterns in the Web site and to locate the wrong Web pages containing these errors. We address the problem by following the nature of semistructured documents, that is, we combine a structured pattern search technique with a more standard text search, which is based on regular expression detection. For this purpose, we will consider an intuitive Unix-like regular expression syntax [96]. Our definition of correctness rule also embeds functions in order to check whether the values included in the semistructured documents are correctly computed.

The verification process is carried out by first extracting a partial structure of a (possibly incorrect) Web page, and then checking whether some constraints are fulfilled. These constraints are expressed by means of equations and regular expressions. Correctness rules are formalized as follows.

Definition 5.3.1 (Correctness rule) *Let (Σ, R) be a canonical TRS. A* correctness rule *has the following form $\mathtt{l} \rightharpoonup \mathtt{error} \mid \mathtt{C}$, where*

1. *$\mathtt{l} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ is a Web page template and $\mathtt{error} \notin (\mathcal{T}ext \cup \mathcal{T}ag \cup \Sigma)$ is a new fresh constant;*

2. *$\mathtt{C}$ is a (possibly empty) sequence*

$$X_1 \ \mathtt{in}\ \mathtt{rexp}_1, \ldots, X_n \ \mathtt{in}\ \mathtt{rexp}_n, \Gamma$$

with $Var(\mathtt{C}) \subseteq Var(\mathtt{l})$, $\mathtt{rexp}_i$ a regular expression over $(\mathcal{T}ext \cup \mathcal{T}ag)$, $i = 1, \ldots, n$, and Γ a sequence of equations over $\tau(\Sigma, \mathcal{V})$.

When $\mathtt{C}$ is empty, we simply write $\mathtt{l} \rightharpoonup \mathtt{error}$.

Given a correctness rule $r \equiv (\mathtt{l} \rightharpoonup \mathtt{error} \mid \mathtt{C})$, we call $\mathtt{l} \rightharpoonup \mathtt{error}$ the *unconditional part of* r and we denote it by r_u. For the sake of expressiveness, we also allow to write inequalities of the form $\mathtt{s} \neq \mathtt{t}$ in the conditional part of the correctness rules. Such inequalities are just syntactic sugar for $(\mathtt{s} = \mathtt{t}) = \mathtt{false}$.

Informally, the meaning of a correctness rule $\mathtt{l} \rightharpoonup \mathtt{error} \mid \mathtt{C}$ is the following. Whenever an instance $\mathtt{l}\sigma$ of $\mathtt{l}$ is recognized in some Web page $\mathtt{p}$, and

(i) each structured text $X_i\sigma$, $i = 1, \ldots, n$, is contained in the language of the corresponding regular expression $\mathtt{rexp_i}$;

(ii) each instantiated equality $(\mathtt{s} = \mathtt{t})\sigma$ in Γ holds in the canonical TRS R;

Web page $\mathtt{p}$ is signaled as an incorrect page.

Completeness rules of a Web specification formalize the requirement that some information must be included in all or some pages of the Web site. We use attributes $\langle \mathtt{A} \rangle$ and $\langle \mathtt{E} \rangle$ to distinguish "universal" from "existential" rules. Right-hand sides of completeness rules may contain functions, which are defined by the user via a canonical TRS.

Definition 5.3.2 (Completeness rule) *Let (Σ, R) be a canonical TRS. A* completeness rule *is either a* universal *rule of the form $\mathtt{l} \rightharpoonup \mu(\mathtt{r})\ \langle \mathtt{A} \rangle$ or an* existential *rule of the form $\mathtt{l} \rightharpoonup \mu(\mathtt{r})\ \langle \mathtt{E} \rangle$, where $\mathtt{l} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$, $\mathtt{r} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag \cup \Sigma, \mathcal{V})$, $Var(\mathtt{r}) \subseteq Var(\mathtt{l})$ and μ is a valid marking for $\mathtt{r}$.*

Intuitively, the interpretation of a universal rule $\mathtt{l} \rightharpoonup \mu(\mathtt{r})\ \langle \mathtt{A} \rangle$ (respectively, an existential rule $\mathtt{l} \rightharpoonup \mu(\mathtt{r})\ \langle \mathtt{E} \rangle$) w.r.t. a Web site W is as follows: if (an instance of) $\mathtt{l}$ is recognized in W, also (an instance of) the irreducible form of $\mathtt{r}$ must be recognized in *all* (respectively, *some*) of the Web pages which embed (an instance of) the marked part of $\mathtt{r}$. Somehow marking information provides the "scope" of the universal/existential quantifiers of the rule, since it allows to compute the subset of the Web site on which the quantifiers act.

Example 5.3.3 *Consider the Web site*

$$\{\mathtt{f}(\mathtt{h}(\mathtt{g}(\mathtt{a}), \mathtt{t})), \mathtt{h}(\mathtt{t}, \mathtt{s}(\mathtt{m}, \mathtt{n}), \mathtt{g}(\mathtt{p})), \mathtt{h}(\mathtt{n}, \mathtt{p}(\mathtt{r}, \mathtt{q})), \mathtt{w}(\mathtt{s}(\mathtt{m}, \mathtt{n}))\}$$

and the rule $r \equiv (\mathtt{w}(\mathtt{X}) \rightharpoonup \underline{\mathtt{h}}(\underline{\mathtt{t}}, \mathtt{X})\langle \mathtt{q} \rangle)$, $\mathtt{q} \in \{\mathtt{E}, \mathtt{A}\}$. Then, the marked part of the right-hand side of r is $\mathtt{h}(\mathtt{t})$.

Thus, the rule will be checked only on the set $\{\mathtt{f}(\mathtt{h}(\mathtt{g}(\mathtt{a}), \mathtt{t})), \mathtt{h}(\mathtt{t}, \mathtt{s}(\mathtt{m}, \mathtt{n}), \mathtt{g}(\mathtt{p}))\}$, which contains all and only the Web pages of W embedding the term $\mathtt{h}(\mathtt{t})$.

Formally, Web specifications are as follows.

Definition 5.3.4 (Web specification) *A* Web specification *is a pair (I, R), where $I \equiv I_{\mathtt{N}} \uplus I_{\mathtt{M}}$ is a finite set of rules such that $I_{\mathtt{N}}$ (respectively, $I_{\mathtt{M}}$) is a set of correctness (respectively, completeness) rules, and R is a canonical TRS.*

Given a set of completeness rules $I_{\mathtt{M}}$, we denote the set of all left-hand sides (right-hand sides without marks, respectively) of rules in $I_{\mathtt{M}}$ by $\mathtt{Lhs_M}$ ($\mathtt{Rhs_M}$, respectively). In symbols, $\mathtt{Lhs_M} = \{\mathtt{l} \mid \mathtt{l} \rightharpoonup \mu(\mathtt{r}) \in I_{\mathtt{M}}\}$ and $\mathtt{Rhs_M} = \{\mathtt{r} \mid \mathtt{l} \rightharpoonup \mu(\mathtt{r}) \in I_{\mathtt{M}}\}$.

Note that, by using the traditional encoding of boolean operations by means of rewrite rules [83, 65], and by introducing non-deterministic rewriting [72], it would

be also possible to extend our framework to a richer specification language providing non-confluent functions such as those that express disjunctive conditions (of the form that the presence of information of kind A requires information of kind B or C to be represented also). For the sake of simplicity, we do not deal with non-confluent TRSs in this work.

The following example illustrates the definition of a Web specification. Marks are introduced by the user to select those Web pages for which we want to check the specified integrity conditions.

Example 5.3.5 *Consider the Web specification which consists of the canonical TRS R of Example 5.1.3 joint with the built-in definition of function* `Nat(X)`*, which converts a string* `X` *to a natural number in Peano's notation, and the following completeness and correctness rules.*

$\mathtt{member(name(X),surname(Y))} \rightharpoonup \underline{\mathtt{hpage}}(\mathtt{fullname(append(X,Y)),status})\ \langle \mathtt{E} \rangle$
$\mathtt{hpage(status(professor))} \rightharpoonup \underline{\mathtt{hpage}}(\underline{\mathtt{status}}(\underline{\mathtt{professor}}),\mathtt{teaching}))\ \langle \mathtt{A} \rangle$
$\mathtt{pubs(pub(name(X),surname(Y)))} \rightharpoonup \underline{\mathtt{member}}(\mathtt{name(X),surname(Y)})\ \langle \mathtt{E} \rangle$
$\mathtt{courselink(url(X),urlname(Y))} \rightharpoonup \underline{\mathtt{cpage}}(\mathtt{title(Y)})\ \langle \mathtt{E} \rangle$
$\mathtt{hpage(X)} \rightharpoonup \mathtt{error} \mid \mathtt{X\ in\ [:TextTag:]*\ sex\ [:TextTag:]*}$
$\mathtt{blink(X)} \rightharpoonup \mathtt{error}$
$\mathtt{project(grant1(X),grant2(Y),total(Z))} \rightharpoonup \mathtt{error} \mid \mathtt{sum(Nat(X),Nat(Y))} \neq \mathtt{Nat(Z)}$
$\mathtt{pub(year(X))} \rightharpoonup \mathtt{error} \mid\ \mathtt{X\ in\ [0-9]*},\ \leq(\mathtt{Nat(X)},\mathtt{s}^{1999}(\mathtt{0})) = \mathtt{true}$
$\mathtt{members(member(name(X),surname(Y)),member(name(X),surname(Y)))} \rightharpoonup \mathtt{error}$

The given Web specification models some required properties for the Web site of Example 5.2.1.

The first four rules are completeness rules, while the remaining ones are correctness rules. The first rule formalizes the following property: if there is a Web page containing a member list, then for each member, a home page should exist which contains (at least) the full name and the status of this member. The full name is computed by appending the name and the surname strings by means of the standard `append` *function whose definition is given in TRS R. The marking information establishes that the property must be checked only on home pages (i.e., pages containing the tag "hpage"). The second rule states that, whenever a home page of a professor is recognized, that page must also include some teaching information. The rule is universal, since it must hold for each professor home page. Such home pages are selected by exploiting the mark given on the tag "status". The third rule specifies that, whenever there exists a Web page containing information about scientific publications, each author of a publication should be a member of the research group. In this case, we must check the property only in the Web page containing the group member list. The fourth rule formalizes that, for each link to a course, a page describing that course must exist. The verification process is carried out only on Web pages containing course information as described by marks.*

The fifth rule forbids sexual contents from being published in the home pages of the group members. Precisely, we check that the word `sex` *does not occur in any*

home page by using the regular expression `[:TextTag:]* sex [:TextTag:]*`*, which identifies the regular language of all the strings built over* $(\mathcal{T}ext \cup \mathcal{T}ag)$ *containing that word. The sixth rule is provided with the aim of improving accessibility for people with disabilities. It simply states that blinking text is forbidden in the whole Web site. The seventh rule states that, for each research project, the total project budget must be equal to the sum of the funds, which has been granted for the first and the second research periods. The eighth rule formalizes the condition that only recent publications are referred to in the Web site (5 last years). Finally, the last rule forbids repetitions of the same member entry in the group member list.*

The verification of both kinds of rules is mechanized by means of partial rewriting.

5.4 Partial rewriting

In order to mechanize the intended semantics of Web specification rules, we first devise a mechanism which is able to recognize the structure and the labeling of a given Web page template inside a particular page of the Web site. This is provided by page simulation. In this section we disregard the conditional part of correctness rules and the existential/universal quantification of completeness rules, since they do not play any role in the definition of partial rewriting. In other words, a rule is just a pair of (possibly) marked terms $\mathtt{l} \rightharpoonup \mu(\mathtt{r})$.

5.4.1 Page simulations

The notion of *page simulation* for Web pages allows us to analyze and extract the partial structure of the Web site which is subject to verification.

Roughly speaking, a Web page $\mathtt{p}_1$ is simulated by a Web page $\mathtt{p}_2$, if the tree-structure of $\mathtt{p}_1$ is "embedded" into the tree-structure of $\mathtt{p}_2$. In other words, a simulation of a Web page (i.e. a labelled tree) $\mathtt{p}_1$ in a Web page $\mathtt{p}_2$ can be seen as a relation among the nodes of $\mathtt{p}_1$ and the nodes of $\mathtt{p}_2$ which preserves the edges and the labelings. Before formalizing the idea, we illustrate it by means of a rather intuitive example.

Example 5.4.1 *Consider the following Web pages (called* $\mathtt{p}_1$ *and* $\mathtt{p}_2$*, respectively):*

```
hpage(name,surname,status(professor),teaching)
```

```
hpage(name(mario),surname(rossi),status(professor),
      teaching(course(logic1),course(logic2)),
      hobbies(hobby(reading),hobby(gardening)))
```

Looking at Figure 5.5, we observe that the structure of $\mathtt{p}_1$ *can be recognized inside the structure of* $\mathtt{p}_2$ *by considering the relation among nodes of* $\mathtt{p}_1$ *and nodes of* $\mathtt{p}_2$ *which is described by the dashed arrows in the figure. This relation essentially provides the so-called* simulation of $\mathtt{p}_1$ in $\mathtt{p}_2$*. Note that vice-versa does not hold: no relations can*

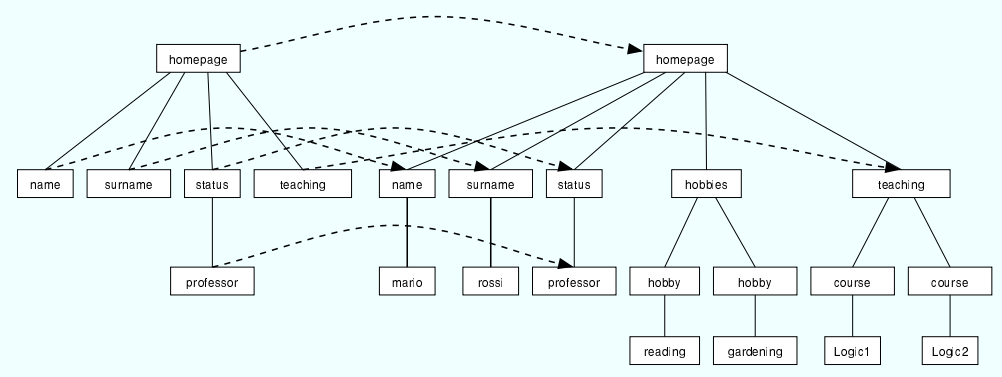

Figure 5.5: Page simulation between $\mathtt{p}_1$ and $\mathtt{p}_2$.

be found among nodes of $\mathtt{p}_2$ *and nodes of* $\mathtt{p}_1$*, which "embed" the structure of* $\mathtt{p}_2$ *into* $\mathtt{p}_1$*. In other words, there does not exist a simulation of* $\mathtt{p}_2$ *in* $\mathtt{p}_1$*.*

Simulations have been used in a number of works dealing with querying and transformation of semistructured data. For instance, [1, 54] propose some techniques based on simulation for analyzing semistructured data w.r.t. a given schema. The language Xcerpt [32, 31] is a (logic) query language for XML and semistructured documents which implements a sort of unification by exploiting the notion of graph simulation. Other approaches involving simulation, or closely related notions, have been employed to measure similarity among semistructured documents [25]. To keep our framework simple, we do not consider a semantic change/load for labels; this would require to introduce ontologies, which are outside the scope of the work.

In the following, we provide our notion of simulation which is a slight adaptation of the one given in [31] to consider Web page templates: we generalize the usual label relation to cope with the case when variables are used as labels, in the following definition.

Definition 5.4.2 *Let* $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2)$ *be two Web page templates in* $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$*. The* label *relation* $\sim \subseteq V_1 \times V_2$ *is defined as follows:*

$$v_1 \sim v_2 \quad \textit{iff} \quad label_1(v_1) = label_2(v_2) \text{ or } label_1(v_1) \in \mathcal{V}.$$

Definition 5.4.3 *Let* $\mathtt{s}_1 \equiv (r_1, V_1, E_1, label_1)$, $\mathtt{s}_2 \equiv (r_2, V_2, E_2, label_2)$ *be two Web page templates in* $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$ *and* $\sim \subseteq V_1 \times V_2$ *be the corresponding label relation. A* page simulation *of* $\mathtt{s}_1$ *in* $\mathtt{s}_2$ *w.r.t.* $\sim$ *is a relation* $\mathbf{S} \subseteq V_1 \times V_2$ *such that, for each* $v_1 \in V_1, v_2 \in V_2$

1. $r_1 \mathbf{S} r_2$;
2. $v_1 \mathbf{S} v_2 \Rightarrow v_1 \sim v_2$;

3. $v_1\ \mathbf{S}\ v_2 \wedge (v_1, v_1') \in E_1 \Rightarrow \exists\, v_2' \in V_2,\ v_1'\ \mathbf{S}\ v_2' \wedge (v_2, v_2') \in E_2$.

We define the *projection* of a simulation $\mathbf{S}$ of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t $\sim$ as $\pi(\mathbf{S}) = \{v_2 \mid (v_1, v_2) \in \mathbf{S}\}$.

Roughly speaking, Definition 5.4.3 ensures two degrees of similarity between Web page templates, not only w.r.t. the labelings but also w.r.t. the structures of the templates. On the one hand, Condition (2) of Definition 5.4.3 formalizes the similarity w.r.t labelings, that is, any pair of nodes (v, v') in a page simulation $\mathbf{S}$ of $\mathtt{s}_1$ in $\mathtt{s}_2$ have the same label, otherwise node v must be labelled by a variable, which somehow means that the label of v can be seen as a generalization of any concrete label of v'. Finally, Condition (1) and Condition (3) provide a relation between the tree structure of $\mathtt{s}_1$ and the tree structure of $\mathtt{s}_2$.

Note that simulations are just relations among nodes of two given Web page templates. For our purposes, we are interested in simulations which are injective mappings from nodes of a given Web page template to nodes of another Web page template. As it will be apparent later, those simulations allow us to project the structure of a Web page template into another one, thus performing a sort of "partial" pattern matching between templates, which will be exploited to formulate our verification technique. So, we first define a subclass of simulations called minimal simulations.

Definition 5.4.4 *Let* $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2)$ *be two Web page templates in* $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. *A page simulation* $\mathbf{S}$ *of* $\mathtt{s}_1$ *in* $\mathtt{s}_2$ *w.r.t.* $\sim$ *is* minimal *if there are no page simulations* $\mathbf{S}'$ *of* $\mathtt{s}_1$ *in* $\mathtt{s}_2$ *w.r.t.* $\sim$ *such that* $\mathbf{S}' \subseteq \mathbf{S}$.

It is straightforward to prove that minimal simulations are mappings.

Proposition 5.4.5 *Let* $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2)$ *be two Web page templates in* $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. *A minimal page simulation* $\mathbf{S}$ *of* $\mathtt{s}_1$ *in* $\mathtt{s}_2$ *w.r.t.* $\sim$ *is a mapping* $\mathbf{S} : V_1 \to V_2$.

Let us see an example which illustrates the notion of minimal simulation.

Example 5.4.6 *Let us consider the following Web page templates* $\mathtt{s}_1$ *and* $\mathtt{s}_2$*:*

```
hobbies(hobby(X))        hobbies(hobby(reading),hobby(gardening)).
```

In Figure 5.6(a), the dashed arrows represent a non-minimal simulation of $\mathtt{s}_1$ *in* $\mathtt{s}_2$*, while in Figures 5.6(b) and 5.6(c) two minimal simulations of* $\mathtt{s}_1$ *in* $\mathtt{s}_2$ *are depicted. Note that the last two simulations are mappings.*

However, minimal simulations do not guarantee that the tree structure of a given Web page template is recognized inside another template. Consider, for instance, the page simulation of $f(X, Y)$ in $f(a)$ depicted in Figure 5.7: it is minimal, but the tree structures of $f(X, Y)$ and $f(a)$ are distinct.

For this purpose, we need a one-to-one correspondence between edges of considered Web page templates. Therefore, we only consider minimal and *injective* simulations.

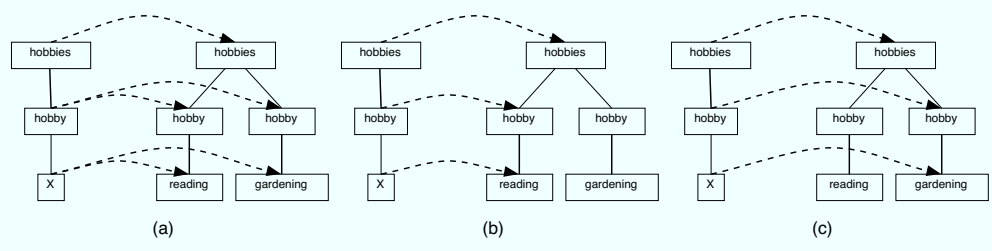

Figure 5.6: Non-minimal and minimal simulations.

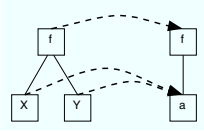

Figure 5.7: Minimal non-injective simulation

Given two Web page templates $\mathtt{s}$ and $\mathtt{t}$, we denote by $\mathtt{s} \cong \mathtt{t}$, the fact that there exists a minimal, injective simulation of $\mathtt{s}$ in $\mathtt{t}$ w.r.t. $\sim$.

It is not difficult to prove that minimal injective simulations are particular instances of Kruskal's *embeddings* [26] w.r.t. the relation $\sim$. In other words, a minimal injective page simulation of $\mathtt{s}_1$ in $\mathtt{s}_2$ w.r.t. $\sim$ exists iff $\mathtt{s}_1$ is embedded into $\mathtt{s}_2$ w.r.t. $\sim$, i.e., we are able to find out the structure and the labeling of $\mathtt{s}_1$ inside $\mathtt{s}_2$.

5.4.2 Rewriting Web page templates

Definition 5.4.7 *Let* $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2) \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. *We say that* $\mathtt{s}_2$ partially matches $\mathtt{s}_1$ *via substitution* σ *iff*

1. *there exists a minimal, injective page simulation* $\mathbf{S}$ *of* $\mathtt{s}_1$ *in* $\mathtt{s}_2$ *w.r.t.* $\sim$*;*
2. *for each* $(v, v') \in \mathbf{S}$ *such that* $label(v) = X \in \mathcal{V}$, $\sigma(X) = (\mathtt{s}_{2|v'})$.

In Definition 5.4.7, we consider only minimal, injective simulations between Web page templates $\mathtt{s}_1$ and $\mathtt{s}_2$ to easily compute a substitution σ such that there exists a simulation of $\mathtt{s}_1\sigma$ in $\mathtt{s}_2$ w.r.t. $\sim$; in other words, $\mathtt{s}_1\sigma$ is embedded into $\mathtt{s}_2$. It is worth noting that non-minimal simulations not always ensure the existence of such a substitution. This is the reason why the minimal simulations are also required to be injective. Let us see an example.

Example 5.4.8 *Consider again Example 5.4.6. We have that* $\mathtt{s}_2$ *partially matches* $\mathtt{s}_1$ *via* $\{\mathtt{X/reading}\}$ *(see Figure 5.6(b)) and* $\mathtt{s}_2$ *partially matches* $\mathtt{s}_1$ *via* $\{\mathtt{X/gardening}\}$ *(see Figure 5.6(c)). Note that performing partial matching by the non-minimal simulation of Figure 5.6(a) would produce* $\sigma \equiv \{\mathtt{X/reading}, \mathtt{X/gardening}\}$*, which is not a substitution.*

Now we are ready to define a partial rewrite relation between marked Web page templates, which includes a simplification stage using the user functions in R.

Definition 5.4.9 *Let R be a canonical TRS. Let $\mathtt{s} \equiv (V, E, r, label)$, $\mathtt{t} \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. Let μ_1 and μ_2 be two valid markings for $\mathtt{s}$ and $\mathtt{t}$, respectively. Then, $\mu_1(\mathtt{s})$* partially rewrites *to $\mu_2(\mathtt{t})$ via rule $r \equiv \mathtt{l} \rightharpoonup \mu(\mathtt{r})$ and substitution σ (in symbols, $\mu_1(\mathtt{s}) \rightharpoonup_r^\sigma \mu_2(\mathtt{t})$) iff there exists $v \in V$ such that*

1. *$\mathtt{s}_{|v}$ partially matches $\mathtt{l}$ via σ;*
2. *Let $\mathtt{r} \equiv (V_\mathtt{r}, E_\mathtt{r}, r, label_\mathtt{r})$ and $\mathtt{r}\sigma \equiv (V_{\mathtt{r}\sigma}, E_{\mathtt{r}\sigma}, r, label_{\mathtt{r}\sigma})$. For each $v \in V_{\mathtt{r}\sigma}$,*

$$\mu_2(v) = \begin{cases} \mu(v) & \text{if} \quad v \in (V_\mathtt{r} \cap V_{\mathtt{r}\sigma}) \\ \mu(v') & \text{if} \quad v \in (V_{\mathtt{r}\sigma} \setminus V_\mathtt{r}) \wedge (\exists\, v' \in V_\mathtt{r}, v \geq v', label_\mathtt{r}(v') \in Var(\mathtt{r})) \end{cases}$$

3. *$\mathtt{t}$ = Reduce($\mathtt{r}\sigma$,R), where function Reduce(x, R) computes, by standard term rewriting, the irreducible form of x in R ignoring the eventual marks for the functions in R.*

When rule r and substitution σ are understood, we simply write $\mu_1(\mathtt{s}) \rightharpoonup \mu_2(\mathtt{t})$.

It is worth noting that we provide a notion of partial rewriting in which the context of the selected reducible expression $\mathtt{s}_{|v}$ of the Web page template which is rewritten is disregarded after the rewrite step (see point (3) of Definition 5.4.9). Roughly speaking, given a Web specification rule $\mathtt{l} \rightharpoonup \mu(\mathtt{r})$, partial rewriting allows us to extract a subpart of a given Web page (template) $\mathtt{s}$, which partially matches $\mathtt{l}$, and to replace $\mathtt{s}$ by a reduced instance of $\mathtt{r}$; namely, *Reduce*$(\mathtt{r}\sigma, R)$ (see points (1) and (3) of Definition 5.4.9). Point (2) of Definition 5.4.9 establishes that rewritten templates inherit marks from the right-hand sides of the applied rules. More precisely,

- each vertex of $\mathtt{r}\sigma$, which is not affected by substitution σ, maintains the same marking of $\mathtt{r}$;
- each vertex, which belongs to a subterm of $\mathtt{r}\sigma$ replacing a variable $\underline{X}$ of $\mathtt{r}$, is marked *yes*;
- each vertex, which belongs to a subterm of $\mathtt{r}\sigma$ replacing a variable X of $\mathtt{r}$, is marked *no*.

A *partial* rewrite sequence is of the form $\mu_0(\mathtt{s}_0) \rightharpoonup_{r_0}^{\sigma_0} \mu_1(\mathtt{s}_1) \rightharpoonup_{r_1}^{\sigma_1} \ldots$ Moreover, we denote the transitive closure (resp., the transitive and reflexive closure) of $\rightharpoonup$ by $\rightharpoonup^+$ (resp., $\rightharpoonup^*$). With the notation $\mu_0(\mathtt{s}_0) \rightharpoonup^n \mu_1(\mathtt{s}_1)$ we denote a partial rewrite sequence of length n (that is, a partial rewrite sequence which is made up of n partial rewrite steps).

Example 5.4.10 *Consider the Web page $\mathtt{p}$ of Figure 5.3 and the first rule $\mathtt{r}_1$ of the Web specification of Example 5.3.5. Let us suppose that TRS R defines the standard*

function append *for concatenating strings. Then, Web page template* $\varepsilon(\mathtt{p})$ *partially rewrites to the following Web pages by applying* $\mathtt{r}_1$.

$$\varepsilon(\mathtt{p}) \rightharpoonup_{\mathtt{r}_1} \mathit{Reduce}(\underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{append}(\mathtt{mario},\mathtt{rossi})),\mathtt{status}),\mathtt{R}) = \underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{mariorossi}),\mathtt{status})$$

$$\varepsilon(\mathtt{p}) \rightharpoonup_{\mathtt{r}_1} \mathit{Reduce}(\underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{append}(\mathtt{franca},\mathtt{bianchi})),\mathtt{status}),\mathtt{R}) = \underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{francabianchi}),\mathtt{status})$$

$$\varepsilon(\mathtt{p}) \rightharpoonup_{\mathtt{r}_1} \mathit{Reduce}(\underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{append}(\mathtt{giulio},\mathtt{verdi})),\mathtt{status}),\mathtt{R}) = \underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{giulioverdi}),\mathtt{status})$$

Roughly speaking, markings in the right-hand sides of the rules allow us to find sets of Web pages, which might be incomplete or missing. Then, real buggy pages are detected inside these sets. We formalize the idea in the following section.

5.5 The verification framework

In the following, we show how simulation and partial rewriting can be applied to verify a given Web site w.r.t. a Web specification. As we have seen in Section 5.4.1, simulation allows us to identify the structure of a given Web page (possibly, a template) into another one. By taking advantage of this fact, we can develop a methodology, which is able to discover correctness as well as completeness errors in a given Web site w.r.t. a Web specification.

More precisely, our analysis allows us to discover the following kinds of errors:

- erroneous/forbidden information in the Web site (*correctness errors*);
- Web pages which are missing in a Web site or Web pages which are incomplete w.r.t a given Web specification (*completeness errors*).

5.5.1 Detecting correctness errors

In this section, we provide a simple mechanism based on partial rewriting which can detect erroneous or undesirable data included in a Web site. Our methodology allows us to precisely locate which part of a Web page does not fulfill the Web specification. We apply correctness rules to the Web pages of the Web site in order to discover incorrect patterns. More precisely, given a Web page p, we first try to recognize a given Web page template l into p by partially rewriting p via the (unconditional part of a) correctness rule $\mathtt{l} \rightharpoonup \mathtt{error} \mid \mathtt{C}$. Then, we analyze the values taken by the variables of C, which are obtained as a by-product of the partial rewrite step. If the structured text, which is bound to each variable in C, belongs to the language of the corresponding regular expression, and all the instantiated equations in C hold, the faulty Web page p and an incorrectness symptom are supplied to the user.

Definition 5.5.1 *Let W be a Web site, $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a Web specification. Given $\mathtt{p} \in W$, we say that* p *is* incorrect *w.r.t. $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$, if there exists a rule $r \equiv (\mathtt{l} \rightharpoonup \mathtt{error} \mid X_1$ `in` $\mathtt{rexp_1} \ldots X_n$ `in` $\mathtt{rexp_n}, \Gamma) \in I_{\mathtt{N}}$ such that*

1. $\mathtt{p} \rightharpoonup^{\sigma}_{r_u} \mathtt{error}$*, where r_u is the unconditional part of r;*
2. *for $i = 1, \ldots, n$, $X_i\sigma \in \mathcal{L}(\mathtt{rexp_i})$, where $\mathcal{L}(\mathtt{rexp_i})$ is the regular language described by* $\mathtt{rexp_i}$*;*
3. *every instantiated equation of Γ, $(\mathtt{s} = \mathtt{t})\sigma$, holds in R.*

We also say that $\mathtt{l}\sigma$ is an incorrectness symptom *for* p*.*

Let us see an example which illustrates the above definition.

Example 5.5.2 *Let $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be the Web specification of Example 5.3.5 and W be the Web site of Figure 5.4.*

Now, consider the correctness rule

$$r \equiv \mathtt{hpage(X)} \rightharpoonup \mathtt{error} \mid \mathtt{X}\ \mathtt{in}\ [\mathtt{:TextTag:}] * \mathtt{sex}\ [\mathtt{:TextTag:}] * \in I_{\mathtt{N}}.$$

Note that the only Web page in W which can yield a correctness error by using r is Web page (3)*, since* (3) *can be partially rewritten to* error *via r_u by means of the following substitution σ*

```
{X/links(link(url(www.google.com),urlname(google)),
         link(url(www.sexycalculus.com),urlname(FormalMethods)))}
```

and $\mathtt{X}\sigma$ belongs to $\mathcal{L}($`[:TextTag:]* sex [:TextTag:]*`*). The corresponding incorrectness symptom is*

```
hp(links(link(url(www.google.com),urlname(google)),
         link(url(www.sexycalculus.com),urlname(FormalMethods)))).
```

Web page (4) *is incorrect w.r.t $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$, as it rewrites to* error *by rule* $\mathtt{blink(X)} \rightharpoonup \mathtt{error}$ *and gives rise to the incorrectness symptom*

$$\mathtt{blink(phone(4444))}.$$

Moreover, we can also discover that the total budget of project B1 *is wrongly computed by applying rule*

$$\mathtt{project(grant1(X),\ grant2(Y),\ total(Z))} \rightharpoonup \mathtt{error} \mid \mathtt{sum(Nat(X), Nat(Y))} \neq \mathtt{Nat(Z)}.$$

Indeed, Web page (6) *is incorrect, since the equation*

$$\mathtt{sum(Nat(800), Nat(300))} \neq \mathtt{Nat(1000)}$$

hold in R. The associated incorrectness symptom is

$$\texttt{project}(\texttt{grant1}(800), \texttt{grant2}(300), \texttt{total}(1000)).$$

Finally, Rule

$$r' \equiv \texttt{members}(\texttt{member}(\texttt{name}(\texttt{X}), \texttt{surname}(\texttt{Y})), \texttt{member}(\texttt{name}(\texttt{X}), \texttt{surname}(\texttt{Y}))) \rightharpoonup \texttt{error}$$

detects that the entry for `Mario Rossi` *in Web page* `(1)` *is repeated twice, since it derives* `error` *by using* r' *via substitution* $\{\texttt{X}/\texttt{mario}, \texttt{Y}/\texttt{rossi}\}$. *The respective incorrectness symptom is:*

```
members(member(name(mario),surname(rossi)),
        member(name(mario),surname(rossi))).
```

The example above points out the usefulness of incorrectness symptoms: they allow us to precisely locate which erroneous piece of information must be modified by the user in order to repair the faulty Web site.

Algorithm 5 outlines a procedure for the detection of correctness errors, which takes as input a Web site W, a set of correctness rules $I_{\texttt{N}}$, and a canonical TRS R. Basically, the procedure repeatedly applies the test of Definition 5.5.1 for recognizing incorrect and forbidden patterns. More precisely, for every Web page in W, we verify whether (i) $\texttt{p}$ reduces to the constant `error` via the unconditional part of some correctness rule r, and (ii) the constraints in the condition of r are fulfilled. In the case that an error is found in a Web page $\texttt{p}$ by using a rule $\texttt{l} \rightharpoonup \texttt{r} \mid \texttt{C}$, the pair $(\texttt{p}, \texttt{l}\sigma)$ is returned, which consists of the wrong page together with the corresponding incorrectness symptom.

Algorithm 5 An algorithm for detecting correctness errors in a Web site.

```
1: procedure CORRECTNESS-ERRORS(W, I_N, R)
2:     for all p ∈ W do
3:         for all r ≡ (l ⇀ error | X_1 in rexp_1, ..., X_n in rexp_n,
                                    s_1 = t_1, ..., s_m = t_m) ∈ I_N do
4:             if (p ⇀^σ_{r_u} error) then
5:                 If (X_i σ ∈ L(rexp_i), i = 1, ..., n) and
                       ((s_j = t_j)σ, j = 1, ..., m, holds in R) then
6:                     output("Error: (p,lσ)") end if
7:             end if
8:         end for
9:     end for
10: end procedure
```

Proposition 5.5.3 *Let W be a Web site, and $(I_{\texttt{N}} \uplus I_{\texttt{M}}, R)$ be a Web specification. Then, the procedure* CORRECTNESS-ERRORS*$(W, I_{\texttt{N}}, R)$ terminates, and for each pair $(\texttt{p}, \texttt{l}\sigma)$, which is returned, $\texttt{p}$ is an incorrect Web page w.r.t. $(I_{\texttt{N}} \uplus I_{\texttt{M}}, R)$ and $\texttt{l}\sigma$ is the corresponding incorrectness symptom.*

5.5.2 Detecting completeness errors

Essentially, the main idea to diagnose completeness errors is to compute the set of all possible marked expressions that can be derived from W via the completeness rules of a Web specification $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ by means of partial rewriting. These marked terms can be thought of as requirements to be fulfilled by W. Then, we check whether the computed requirements are satisfied by W using simulation and marking/quantification information. In summary, the method works in two steps, as described below.

1. Compute the set of requirements $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ for W w.r.t. $I_{\mathtt{M}}$;
2. Check $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ in W.

Formally, a *requirement* is a pair $\langle \mu(\mathtt{e}), \mathtt{q} \rangle$, where $\mu(\mathtt{e})$ is a marked term and $\mathtt{q} \in \{\mathtt{A}, \mathtt{E}\}$. A requirement is called *universal* whenever $\mathtt{q} = \mathtt{A}$, while it is called *existential* when $\mathtt{q} = \mathtt{E}$. In order to formalize step 1, we define the following operator.

Definition 5.5.4 *Let* $\mathtt{T}$ *be a set of marked terms and* $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ *be a Web specification. Then, the* immediate completeness requirements *operator*

$$\mathcal{J}_{\mathtt{M}}(\overline{\mathtt{T}}) = \overline{\mathtt{T}} \cup \{\langle \mu_2(\mathtt{s}_2), \mathtt{q} \rangle \mid \exists \langle \mu_1(\mathtt{s}_1), \mathtt{q}_1 \rangle \in \overline{\mathtt{T}}, r \equiv (\mathtt{l} \rightharpoonup \mu(\mathtt{r})\ \langle \mathtt{q} \rangle) \in I_{\mathtt{M}} \text{ s.t. } \mu_1(\mathtt{s}_1) \rightharpoonup_r \mu_2(\mathtt{s}_2)\}$$

where $\overline{\mathtt{T}} = \{\langle \mathtt{s}, \_ \rangle \mid \mathtt{s} \in \mathtt{T}\}$.

The operator in Definition 5.5.4 computes all the requirements which are obtained by partially rewriting the marked expressions in $\overline{\mathtt{T}}$ using the completeness rules of $I_{\mathtt{M}}$, and returns the union of the resulting set and $\overline{\mathtt{T}}$. By repeatedly applying this operator, it is possible to compute all marked terms that can be derived from an initial Web site after an arbitrary number of partial rewriting steps. For this purpose, we formalize the *ordinal powers* of the operator $\mathcal{J}_{\mathtt{M}}$ w.r.t. a Web site W as follows: $\mathcal{J}_{\mathtt{M}} \uparrow^W 0 = \overline{W}$, $\mathcal{J}_{\mathtt{M}} \uparrow^W n = \mathcal{J}_{\mathtt{M}}(\mathcal{J}_{\mathtt{M}} \uparrow^W (n-1))$, $n > 0$.

It is immediate to prove that the operator $\mathcal{J}_{\mathtt{M}}$ is continuous on the lattice consisting of the powerset of the requirements ordered by set inclusion. This ensures that a least fixpoint of $\mathcal{J}_{\mathtt{M}}$ exists and can be reached after ω applications of $\mathcal{J}_{\mathtt{M}}$, that is, $\mathcal{J}_{\mathtt{M}} \uparrow^W \omega$ where ω is the first infinite ordinal. The least fixpoint of $\mathcal{J}_{\mathtt{M}}$ contains all the marked expressions derivable from W via $I_{\mathtt{M}}$ along with their quantification information.

Now, recalling the interpretation of the completeness rules of the Web site specification given in Section 5.3, marked terms derived by the application of a completeness rule must be recognized as (part of) some Web page in the Web site. Therefore, the expressions in the least fixpoint of $\mathcal{J}_{\mathtt{M}}$ provide information that must occur in W. Thus, since the pages in W trivially occur in the Web site W, we define the *set of requirements* for W w.r.t. $I_{\mathtt{M}}$ as

$$\mathtt{Req}_{\mathtt{M},\mathtt{W}} = \mathit{lfp}(\mathcal{J}_{\mathtt{M}}) \setminus \overline{W}$$

where $\mathit{lfp}(\mathcal{J}_{\mathtt{M}})$ is the least fixpoint of the operator $\mathcal{J}_{\mathtt{M}}$.

Example 5.5.5 *Consider the Web specification* $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ *of Example 5.3.5 and the Web site* W *of Figure 5.4. Then, the set of computed requirements* $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ *is*

$$\begin{array}{l}\{\ \langle \underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{mariorossi}), \mathtt{status}), \mathtt{E}\rangle,\\ \ \langle \underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{francabianchi}), \mathtt{status}), \mathtt{E}\rangle,\\ \ \langle \underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{giulioverdi}), \mathtt{status}), \mathtt{E}\rangle,\\ \ \langle \underline{\mathtt{hpage}}(\underline{\mathtt{status}}(\underline{\mathtt{professor}}), \mathtt{teaching}), \mathtt{A}\rangle,\\ \ \langle \underline{\mathtt{member}}(\mathtt{name}(\mathtt{mario}), \mathtt{surname}(\mathtt{rossi})), \mathtt{E}\rangle,\\ \ \langle \underline{\mathtt{member}}(\mathtt{name}(\mathtt{anna}), \mathtt{surname}(\mathtt{gialli})), \mathtt{E}\rangle,\\ \ \langle \underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{annagialli}), \mathtt{status}), \mathtt{E}\rangle,\\ \ \langle \underline{\mathtt{cpage}}(\mathtt{title}(\mathtt{Algebra})), \mathtt{E}\rangle\ \}\end{array}$$

Clearly, the fixpoint of $\mathcal{J}_{\mathtt{M}}$ (and hence $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$) for an arbitrary Web specification might be infinite. Consider for instance the following example.

Example 5.5.6 *Let* $W \equiv \{\mathtt{h}(\mathtt{g}(\mathtt{0}), \mathtt{f}(\mathtt{0}))\}$ *be a Web site and*

$$I_{\mathtt{M}} \equiv \{\mathtt{h}(\mathtt{g}(\mathtt{X})) \rightharpoonup \mathtt{h}(\mathtt{g}(\mathtt{g}(\mathtt{X})))\langle \mathtt{q}\rangle\}$$

be a set of completeness rules of a Web specification $\mathtt{S}$*. Then,*

$$\mathtt{Req}_{\mathtt{M},\mathtt{W}} = \{\langle \mathtt{h}(\mathtt{g}(\mathtt{g}(\mathtt{0}))), \mathtt{q}\rangle, \langle \mathtt{h}(\mathtt{g}(\mathtt{g}(\mathtt{g}(\mathtt{0})))), \mathtt{q}\rangle, \langle \mathtt{h}(\mathtt{g}(\mathtt{g}(\mathtt{g}(\mathtt{g}(\mathtt{0}))))), \mathtt{q}\rangle, \ldots\}$$

is an infinite set of requirements.

Fortunately, the computation of the set of requirements is finite for some interesting classes of Web specifications. Trivially, non-recursive specifications allow to reach $lfp(\mathcal{J}_{\mathtt{M}})$ after a finite number of applications of $\mathcal{J}_{\mathtt{M}}$, i.e., $lfp(\mathcal{J}_{\mathtt{M}}) = \mathcal{J}_{\mathtt{M}} \uparrow^{W} k$, $k \in \mathbb{N}$. However, non-recursive definitions are not expressive enough for verification purposes, since some relevant conditions about Web sites cannot be formalized without resorting to recursion; e.g., some properties stated in Example 5.3.5 cannot be formulated by using a non-recursive specification.

In the following, we ascertain two important classes of recursive Web specifications whose set of requirements is finite. Basically, the idea is to consider those specifications in which the computation of the least fixpoint only generates marked expressions whose size is bounded [10].

The following definition formalizes the class of the *bounded* Web specifications.

Definition 5.5.7 *A Web specification* $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ *is* bounded *iff, for each* $\mathtt{l} \equiv (V_1, E_1, r_1, label_1) \in \mathtt{Lhs}_{\mathtt{M}}$, $\mathtt{r} \equiv (V_2, E_2, r_2, label_2) \in \mathtt{Rhs}_{\mathtt{M}}$ *and each minimal injective simulation* $\mathbf{S}$ *of* $\mathtt{l}$ *in* $\mathtt{r}_{|v}$ *w.r.t.* $\sim$, $v \in V_2$, *the following properties hold*

1. *if* $v_2 \in \pi(\mathbf{S})$ *and* $label_2(v_2) \in Var(\mathtt{r}_{|v})$, *then for all* $v_1 \in V_1$ *s.t.* $label_1(v_1) \in Var(\mathtt{l})$, $depth(\mathtt{r}_{|v}, v_2) = depth(\mathtt{l}, v_1)$;
2. *for each* $\mathtt{r} \in \mathtt{Rhs}_{\mathtt{M}}$, $\mathtt{r}$ *does not contain any symbol of* Σ_R.

Roughly speaking, Definition 5.5.7 states that, whenever the left-hand side $\mathtt{l}$ of a rule is simulated by (a subterm of) the right-hand side $\mathtt{r}$ of a (possibly different) rule, then no variables in the recognized substructure of $\mathtt{r}$ must be located at positions which are deeper than all the positions of the variables in $\mathtt{l}$. Moreover, bounded Web specifications do not allow function calls in the completeness rules. Let us see an example.

Example 5.5.8 *Consider again the set of completeness rules $I_{\mathtt{M}}$ of Web specification $\mathtt{S}$ in Example 5.5.6. The left-hand side of the rule $\mathtt{h(g(X))} \rightharpoonup \mathtt{h(g(g(X)))\langle q\rangle}$ is simulated by its own right-hand side. Moreover, variable $\mathtt{X}$ in the right-hand side is located at depth 3, while the unique variable in the left-hand side is at depth 2. Thus, $\mathtt{S}$ is not bounded.*

Now, take into account the Web specification $\mathtt{S}' \equiv (I'_{\mathtt{N}} \uplus I'_{\mathtt{M}}, \emptyset)$, whose set of completeness rules is

$$I'_{\mathtt{M}} \equiv \{\mathtt{m(n(X))} \rightharpoonup \mathtt{h(n(X),s(s(X)))\langle q'\rangle}, \mathtt{h(n(X))} \rightharpoonup \mathtt{m(n(X),t)\langle q''\rangle}\}, \quad \mathtt{q'}, \mathtt{q''} \in \{\mathtt{A}, \mathtt{E}\}.$$

Then, $\mathtt{m(n(X))}$ is simulated by $\mathtt{m(n(X),t)}$ and there is also a simulation of $\mathtt{h(n(X))}$ in the term $\mathtt{h(n(X),s(s(X)))}$. In both cases, variables occurring in the substructures of the right-hand sides which are recognized by simulation and variables of the respective left-hand sides are located at the same depth. Therefore, the Web specification $\mathtt{S}'$ is bounded.

For bounded Web specifications, the least fixpoint of the operator $\mathcal{J}_{\mathtt{M}}$ is finite as stated by the next proposition. This provides an effective method for computing the set of requirements $\mathtt{Req}_{\mathtt{M,W}}$ for this class of specifications.

Proposition 5.5.9 *Let $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded Web specification and W be a Web site. Then, there exists $k \in \mathbb{N}$ such that $lfp(\mathcal{J}_{\mathtt{M}}) = \mathcal{J}_{\mathtt{M}} \uparrow^{W} k$.*

Let us now introduce a more general class of Web specifications in which defined functions can be invoked in the rhs's of the completeness rules. In order to keep the size of the derived terms bounded, the key idea is to ensure that function calls do not generate infinite data structures.

Given a Web specification $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$, we say that the completeness rule $(\mathtt{l} \rightharpoonup \mu(\mathtt{r})\langle\mathtt{q}\rangle) \in I_{\mathtt{M}}$ is *function-dependent* w.r.t. R iff $\mathtt{r}$ contains a function call $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \in \tau(\Sigma_R, \mathcal{V})$. Otherwise we say that rule r is *function-independent* w.r.t. R. Given a function-dependent rule r, we obtain the *function-independent version* of r w.r.t. R, and we denote it by r^*, by replacing all the function calls in the right-hand side of r with fresh variables not occurring in the rule.

Example 5.5.10 *Consider the function-dependent rule of Example 5.3.5*

$\mathtt{member(name(X),surname(Y))} \rightharpoonup \underline{\mathtt{hpage}}\mathtt{(fullname(append(X,Y)),status)}\ \langle\mathtt{E}\rangle.$

The function-independent version of the considered rule is

$\mathtt{member(name(X), surname(Y))} \rightharpoonup \underline{\mathtt{hpage}}\mathtt{(fullname(Z),status)}\ \langle \mathtt{E} \rangle$.

Definition 5.5.11 *A Web specification* $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ *is* bounded* *iff*

1. $(I_{\mathtt{N}} \uplus \{r \in I_{\mathtt{M}} \mid$ *r is function-independent w.r.t.* $R\}, R)$ *is bounded;*
2. *for each function-dependent rule* $r \in I_{\mathtt{M}}$ *w.r.t.* R *and* $\mathtt{l} \in \mathtt{Lhs}_{\mathtt{M}}$
   (a) *there exists no minimal, injective simulation of* $\mathtt{l}$ *in any subterm of the right-hand side of* r^* *w.r.t.* $\sim$*;*
   (b) *there exists no minimal, injective simulation of any subterm of the right-hand side of* r^* *in* $\mathtt{l}$ *w.r.t.* $\sim$*;*
3. $\{\mathtt{f} \mid \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n}) \rightharpoonup \mu(\mathtt{r}) \in \mathtt{I_m}\} \cap \Sigma_R = \emptyset$.

By Definition 5.5.11, the rhs's of function-dependent completeness rules do not introduce terms which can be partially rewritten in a subsequent step. This suffices to ensure the finiteness of all partial rewriting sequences. Now, it is immediate to prove the finiteness of the least fixpoint of the $\mathcal{J}_{\mathtt{M}}$ operator for bounded* Web specifications.

The following proposition generalizes Proposition 5.5.9 for bounded* Web specifications.

Proposition 5.5.12 *Let* $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ *be a bounded** *Web specification and* W *be a Web site. Then, there exists* $k \in \mathbb{N}$ *such that* $lfp(\mathcal{J}_{\mathtt{M}}) = \mathcal{J}_{\mathtt{M}} \uparrow^W k$.

Let us see an example.

Example 5.5.13 *Let us consider the Web specification of Example 5.3.5. Then, we can easily check that it is bounded***. Moreover, the least fixpoint of* $\mathcal{J}_{\mathtt{M}}$ *is finite as witnessed by Example 5.5.5.*

Example 5.5.14 *Consider now the following Web specification.*

$$\begin{array}{lcl} I_{\mathtt{N}} & = & \emptyset \\ I_{\mathtt{M}} & = & \{\mathtt{h(c(X))} \rightharpoonup \mathtt{h(g(X))}\} \\ R & = & \{\mathtt{g(0)} \to \mathtt{c(c(0))}, \mathtt{g(c(X))} \to \mathtt{c(g(X))}\} \end{array}$$

The function-independent version of $\mathtt{h(c(X))} \rightharpoonup \mathtt{h(g(X))}$ *is* $\mathtt{h(c(X))} \rightharpoonup \mathtt{h(Z)}$*, and there exists a minimal, injective simulation of* $\mathtt{h(Z)}$ *in* $\mathtt{h(c(X))}$ *w.r.t.* $\sim$*. Therefore the considered Web specification is not bounded***. Actually, we can generate the following infinite partial rewrite sequence:*

$$\mathtt{h(c(0))} \rightharpoonup \mathtt{h(c(c(0)))} \rightharpoonup \mathtt{h(c(c(c(0))))} \ldots$$

Now, we are ready to formalize step 2, that is, checking the computed completeness requirements in a given Web site. To accomplish this task, we first use simulation for checking whether (the marked part of) a requirement is embedded into some Web page of the considered site, and then consider the quantification attributes in order to diagnose completeness errors.

Definition 5.5.15 *Let W be a Web site, $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a Web specification and $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ be the set of requirements for W w.r.t. $I_{\mathtt{M}}$. Let $\langle \mu(\mathtt{e}), \mathtt{q} \rangle \in \mathtt{Req}_{\mathtt{M},\mathtt{W}}$. The* test set *w.r.t. $\langle \mu(\mathtt{e}), \mathtt{q} \rangle$ is defined as*

$$\mathtt{TEST}_{\langle \mu(\mathtt{e}),\mathtt{q} \rangle} = \{\mathtt{p} \equiv (V, E, r, label) \in W \mid \exists \text{ a minimal injective simulation of } mark(\mathtt{e}, \mu) \text{ in } \mathtt{p}_{|v} \text{ w.r.t. } \sim, \text{ with } v \in V\}.$$

This definition allows us to compute a subset of the Web site containing all the Web pages which simulate the marked part of a given requirement. These Web pages might be incomplete w.r.t. the Web specification, since they may not contain the considered requirement. Let us see an example.

Example 5.5.16 *Let us consider the completeness rule r*

$$\mathtt{hpage(status(professor))} \rightharpoonup \underline{\mathtt{hpage}}(\underline{\mathtt{status}}(\underline{\mathtt{professor}}),\mathtt{teaching})$$

and the Web site W of Figure 5.4. Rule r allows us to check whether Web pages of the professors contain some teaching information. In order to do this, we use the marking information in the rhs of r to select the professor Web pages. Let us consider the requirement $\langle \mu_1(\mathtt{e}_1), \mathtt{A} \rangle \equiv \langle \underline{\mathtt{hpage}}(\underline{\mathtt{status}}(\underline{\mathtt{professor}}), \mathtt{teaching}), \mathtt{A} \rangle$, which can be derived from W by means of r. By applying Definition 5.5.15, we get the following test set $\mathtt{TEST}_{\langle \mu_1(\mathtt{e}_1),\mathtt{A} \rangle}$

```
{(2) hpage(fullname(mariorossi),phone(3333),status(professor),
           hobbies(hobby(reading),hobby(gardening))),
 (4) hpage(fullname(annagialli),status(professor),
           blink(phone(4444)),teaching(courselink(
                                       url(http://www.algebra.org),
                                       urlname(Algebra))))}
```

which contains the two professor Web pages where the computed completeness requirement must be checked.

In the following, we consider completeness errors which refer to incomplete and/or missing Web pages. We distinguish two cases: the former allows us to discover whether a universal requirement is not fulfilled by a given Web site, while the latter recognizes unsatisfied existential requirements. In both cases, our analysis provides the missing/incomplete Web pages which are associated with those requirements.

Definition 5.5.17 *Let W be a Web site, $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a Web specification and $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ be the set of requirements for W w.r.t. $I_{\mathtt{M}}$. Let $\mathtt{re} \equiv \langle \mu(\mathtt{e}), \mathtt{A} \rangle \in \mathtt{Req}_{\mathtt{M},\mathtt{W}}$ be a universal requirement. Then, $\mathtt{re}$ is not* satisfied *in W if one of the following conditions hold:*

1. $\mathtt{TEST}_{\langle \mu(\mathtt{e}),\mathtt{A} \rangle} = \emptyset$*;*
2. *there exists $\mathtt{p} \equiv (V, E, r, label) \in \mathtt{TEST}_{\langle \mu(\mathtt{e}),\mathtt{A} \rangle}$ s.t. no minimal, injective simulation of $\mathtt{e}$ in $\mathtt{p}_{|v}$ w.r.t. $\sim$, with $v \in V$, exists.*

Vice versa, a universal requirement re is satisfied whenever it is possible to recognize re inside any Web page of the corresponding test set $\mathtt{TEST}_{\mathtt{re}}$. Let us clarify Definition 5.5.17 by an example.

Example 5.5.18 *Consider the Web site W of Figure 5.4 and the universal requirement*

$$\langle \mu_1(\mathtt{e}_1), \mathtt{A} \rangle \equiv \langle (\underline{\mathtt{hpage}}(\underline{\mathtt{status}}(\underline{\mathtt{professor}}), \mathtt{teaching})), \mathtt{A} \rangle$$

belonging to the set of requirements $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ of Example 5.5.5. The requirement simply states that all *professor's home pages must contain teaching information.*

The test set associated with $\langle \mu_1(\mathtt{e}_1), \mathtt{A} \rangle$, i.e. $\mathtt{TEST}_{\langle \mu_1(\mathtt{e}_1), \mathtt{A} \rangle}$, was computed in Example 5.5.16 and contains all the professor's Web pages of the considered site. Now, by applying Definition 5.5.17, we detect that $\langle \mu_1(\mathtt{e}_1), \mathtt{A} \rangle$ is not satisfied by W, since there does not exist any minimal, injective simulation of $\mathtt{e}_1$ in (a subterm of) Web page (2) *w.r.t. $\sim$. In fact, Web page* (2) *lacks teaching information.*

Finally, an existential requirement re is fulfilled if it is recognized inside (at least) a Web page which belongs to the test set $\mathtt{TEST}_{\mathtt{re}}$.

Definition 5.5.19 *Let W be a Web site, $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a Web specification and $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ be the set of requirements for W w.r.t. $I_{\mathtt{M}}$. Let $\mathtt{re} \equiv \langle \mu(\mathtt{e}), \mathtt{E} \rangle \in \mathtt{Req}_{\mathtt{M},\mathtt{W}}$ be an existential requirement. Then,* re *is not* satisfied *in W if one of the following conditions hold:*

1. $\mathtt{TEST}_{\langle \mu(\mathtt{e}), \mathtt{E} \rangle} = \emptyset$;
2. *for each $\mathtt{p} \equiv (V, E, r, label) \in \mathtt{TEST}_{\langle \mu(\mathtt{e}), \mathtt{E} \rangle}$ no minimal, injective simulation of $\mathtt{e}$ in $\mathtt{p}_{|v}$ w.r.t. $\sim$, with $v \in V$, exists.*

Example 5.5.20 *Consider the Web site W of Figure 5.4 and the existential requirement*

$$\langle \mu_2(\mathtt{e}_2), \mathtt{E} \rangle \equiv \langle \underline{\mathtt{cpage}}(\mathtt{title}(\mathtt{Algebra})), \mathtt{E} \rangle$$

belonging to the set of requirements $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ of Example 5.5.5. Since $\mathtt{TEST}_{\langle \mu_2(\mathtt{e}_2), \mathtt{E} \rangle}$ is empty, by Definition 5.5.19, re *is not satisfied in W. This implies that a Web page containing some information about the Algebra course should be provided.*

Consider now the following existential requirements in $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$

$$\langle \mu_3(\mathtt{e}_3), \mathtt{E} \rangle \equiv \langle \underline{\mathtt{member}}(\mathtt{name}(\mathtt{anna}), \mathtt{surname}(\mathtt{gialli})), \mathtt{E} \rangle$$
$$\langle \mu_4(\mathtt{e}_4), \mathtt{E} \rangle \equiv \langle \underline{\mathtt{hpage}}(\mathtt{fullname}(\mathtt{giulioverdi}), \mathtt{status}), \mathtt{E} \rangle.$$

We have that $\mathtt{TEST}_{\langle \mu_3(\mathtt{e}_3), \mathtt{E} \rangle}$ just includes Web page (1) *of W, while $\mathtt{TEST}_{\langle \mu_4(\mathtt{e}_4), \mathtt{E} \rangle}$ contains Web pages* (2), (3) *and* (4). *For both requirements, there is no Web page $\mathtt{p}$ in the test sets such that a minimal, injective simulation of $\mathtt{e}_3$ (respectively, $\mathtt{e}_4$) in (a subterm of) $\mathtt{p}$ w.r.t $\sim$ exists. Therefore, the Web site W does not meet the given requirements. More precisely, the former unsatisfied requirement states that an entry for* Anna Gialli *must be introduced in the group member list, and the latter detects that the home page of* Giulio Verdi *is missing.*

The requirements which are not fulfilled can be considered as *incompleteness symptoms*. This allows us not only to locate bugs and inconsistencies w.r.t. a given specification, but also to easily repair them by comparing incomplete pages to unsatisfied requirements, since the latter ones provide the missing information which is needed to complete the erroneous Web pages.

An algorithm for detecting incomplete information can be defined as follows. First of all, we generate the requirements to be checked by computing the fixpoint of the operator $\mathcal{J}_{\mathtt{M}}$. Next, for each (universal/existential) requirement, the corresponding test set is produced and then checked in order to analyze which requirements are not fulfilled. The analysis is based on Definition 5.5.17 (resp., Definition 5.5.19) which formalizes the notion of universal (resp., existential) requirement satisfiability. Finally, incompleteness symptoms are returned, which are needed to help the user to locate the missing/incomplete pages, whenever a requirement is not verified.

The basic procedure is sketched in Algorithm 6, which takes as input a Web site W, a set of completeness rules $I_{\mathtt{M}}$, and a canonical TRS R.

Algorithm 6 An algorithm for detecting completeness errors.

```
1: procedure COMPLETENESS-ERRORS(W, I_M, R)
2:     Req_{M,W} ← lfp(J_M) \ W
3:     for all ⟨μ(e), q⟩ ∈ Req_{M,W} do
4:         TEST_{⟨μ(e),q⟩} ← {p ≡ (V, E, r, label) ∈ W | mark(e, μ) ≅ p_|v, v ∈ V}
5:         if TEST_{⟨μ(e),q⟩} = ∅ then
6:             output("Error: A Web page containing e must occur in Web site W!")
7:         end if
8:         case q
9:             q = A :
10:                if ∃ p ≡ (V, E, r, label) ∈ TEST_{⟨μ(e),A⟩} s.t. e ≇ p_|v , with v ∈ V then
11:                    output("Error: e must occur in all Web pages of TEST_{⟨μ(e),A⟩}!")
12:                end if
13:            q = E :
14:                if ∀p ≡ (V, E, r, label) ∈ TEST_{⟨μ(e),E⟩}, e ≇ p_|v, with v ∈ V then
15:                    output("Error: e must occur in at least one Web page of
                           TEST_{⟨μ(e),E⟩}!")
16:                end if
17:        end case
18:    end for
19: end procedure
```

Proposition 5.5.21 *Let W be a Web site, $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded* Web specification, and $\mathtt{Req}_{\mathtt{M,W}}$ be the set of requirements for W w.r.t. $I_{\mathtt{M}}$. Then, the procedure* COMPLETENESS-ERRORS*$(W, I_{\mathtt{M}}, R)$ terminates. Moreover, for each error message regarding term $\mathtt{e}$, which is returned by the procedure, there exists $\langle \mu(\mathtt{e}), \mathtt{q} \rangle \in \mathtt{Req}_{\mathtt{M,W}}$, with $\mathtt{q} \in \{\mathtt{A}, \mathtt{E}\}$, which is not satisfied in W.*

5.6 Implementation

The basic methodology presented so far has been implemented in the prototype system GVerdi (VErification and Rewriting for Debugging Internet sites) which has been written in Haskell (GHC v6.2.2) and is publicly available together with a set of tests at `http://www.dimi.uniud.it/~demis/#software`. A short system description can be found in [11, 23].

The implementation consists of approximately 1100 lines of source code. It includes a parser for semistructured expressions (i.e. XML/XHTML documents) and Web specifications, and several modules implementing the partial rewriting mechanism, the verification technique, and the graphical user interface. A snapshot of the running system is shown in Figure 5.8.

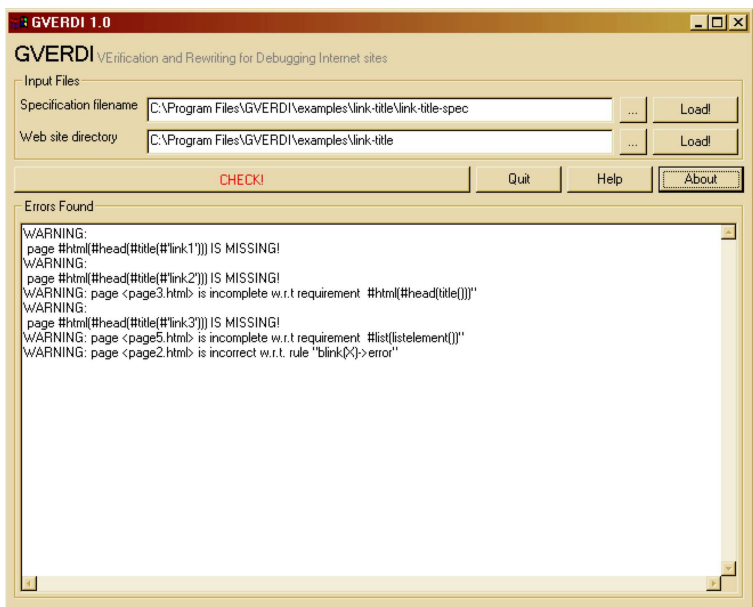

Figure 5.8: A screenshot of the system.

The system allows the user to load a Web site together with a Web specification. Additionally, he/she can inspect the loaded data and finally check the Web pages w.r.t. the Web site specification. We have tested the system on several XHTML Web sites and XML data collections which can be found at the URL address mentioned above. For instance, we checked the Web site of the Computational Logic Group of the University of Udine which is available at the URL `http://www.dimi.uniud.it/clg`. It contains about 20 Web pages concerning publications, people, and projects of the group. In each considered test case, we have been able to detect the errors (i.e. missing and incomplete Web pages) efficiently.

Conclusions

In this thesis we have discussed how verification and correction techniques can be applied to programs as well as to (semi-structured) data. We have presented the following new contributions. First, a generic correction scheme has been developed, then we particularized it to the functional logic and functional settings. Moreover, we have formalized a rewriting-based language in order to specify formal properties of Web sites and to subsequently check them automatically.

Automated Program Correction

The correction scheme we proposed allows one the automatic correction of the bugs by means of a new inductive learning methodology which is driven by examples. We have followed an hybrid, *deductive* (top-down) as well as *inductive* (bottom-up) approach, which is able to infer program corrections that are hard, or not possible at all, to obtain with a simple (pure deductive or inductive) program learner. The resulting blend of top-down and bottom-up synthesis is conceptually cleaner than more sophisticated, purely top down or bottom-up ones and combines the advantages of both techniques.

Diagnosis tools work in combination with this methodology in order to provide the needed examples (i.e., missing or wrong values) to drive the process towards a suitable correction. The effectiveness of the correction process depends directly on the available diagnosis and learning tools.

Correction of functional logic programs

Following the generic scheme of Chapter 1, we have described a new methodology for synthesizing (partially) correct functional logic programs which complements the diagnosis method we developed previously in [13, 14]. Our methodology is based on the combination, in a single framework, of a diagnoser [13, 14] which identifies those parts of the code containing errors, together with an hybrid program learner which, once the bug has been located in the program, tries to repair it starting from examples (uncovered as well as incorrect equations) which are essentially obtained as an outcome of the diagnoser. The method we have described works for both lazy and eager functional logic languages; more precisely, our framework is parametric w.r.t. the needed narrowing and the leftmost-innermost narrowing strategy.

We have tested a number of example programs on the available implementation of the system NoBug and we have presented an experimental evaluation.

We would like to note that our methodology could also be usefully combined within more traditional debugging environments for functional logic programs where the user

would be typically asked to provide the error symptoms (examples) manually.

Finally, we want to emphasize that this framework supersedes the preliminary approach of [14]. In [14], recursive definitions were difficult, and sometimes impossible, to repair, and no automated correction is provided for overspecialized programs either, whereas the new methodology overcomes both drawbacks.

Correction of (first-order) functional programs

Also, we have formalized a methodology for synthesizing (partially) correct (first-order) functional programs written in OBJ style, which endows the diagnosis method, developed previously in [12], with some correction capabilities. Specifications of the intended semantics, expressed as programs, are used to carry out the diagnosis as well as the correction. This is not only a common practice in logic as well as equational (or term rewriting) languages, but also in functional programming. For example, in QuickCheck [36], formal specifications are used to describe properties of Haskell programs (written as Haskell programs too) which are automatically tested.

As we did for the functional logic programming paradigm in Chapter 3, we particularized the generic scheme of Chapter 1 (in this case, to the functional setting).

This method is not comparable to our previous work [6] as it is lower-cost and it works for a much wider class of TRSs. In particular, it is able to repair non-confluent programs. This is not only theoretically more challenging, but also convenient in our framework, where it is not reasonable to expect that confluence holds for an erroneous program (even if program confluence was in the programmer's intention).

Automated Web Site Verification

Conceiving and maintaining Web sites is a difficult task. In the second part of this dissertation, we provided a rewriting-based, formal specification language which can be used to impose properties both on the structure (syntactic properties) and on the contents (semantic properties) of Web sites. Some XML schema languages such as XLINKIT [48] and Schematron [103] can express some of the semantic constraints we consider; however, our specification language is richer and can be appreciated much by users who prefer to avoid the encumbrances of DTDs and XML rule languages. The computation mechanism underlying our language is based on a novel rewriting-like technique, called *partial rewriting*, in which the traditional pattern matching mechanism is replaced by tree *simulation* [66]. In our methodology, Web sites are automatically checked w.r.t. a given Web specification in order to detect incorrect, incomplete and missing Web pages. Moreover, by analyzing the error symptoms, we are also able to find out the missing information needed to repair the Web site. We have also briefly discussed some implementation details of the preliminary system GVERDI, a prototypical implementation of the verification framework which can help Web administrators to design, check and maintain their Web sites.

Let us conclude by mentioning some directions for future work. We are currently extending our framework in order to provide a method for synthesizing the marking

information semi-automatically (currently, marks are provided by the user). In order to consider semi-structured documents containing cycles, a graph rewriting extension of our framework is also envisaged.

A

Context Sensitive Rewriting and Over-Generality

A.1 Deciding over-generality by context sensitive rewriting

Theorem 3.3.8 makes the assumption that all the programs $\mathcal{R}$ to be repaired are overly general w.r.t. the finite set of positive examples E^+ (i.e., $\mathcal{R} \vdash E^+$). Generally this condition is undecidable. Anyway, if we restrict ourself to consider examples whose right-hand sides are constructor terms, the predicate $\mathcal{R} \vdash E^+$ becomes decidable by imposing very simple constraints on the form of our programs. In the sequel, we will describe an algorithm for checking whether an example $t = c$, $\tau(\Sigma)$ and $c \in \tau(\mathcal{C})$, is covered by a program $\mathcal{R}$. This will immediately supply a procedure to decide on the over-generality of a program. Initially, we will deal with TRSs, then we will extend our algorithm to CTRSs. The methodology is based on the notion of *Context Sensitive Rewriting* (CSR)[78], which is a restraint (μ-rewriting) of the rewriting relation. We briefly recall some definitions regarding CSR in the following section.

A.1.1 Context Sensitive Rewriting

Given a signature Σ, we define a mapping $\mu : \Sigma \to \mathcal{P}(\mathbb{N})$, called *replacing map*, such that for each k-ary symbol $f \in \Sigma$, $\mu(f) \subseteq \{1, 2, \ldots, k\}$. We will use it to discriminate argument positions on which replacements (reductions) are allowed, that is, given a function call $f(t_1, \ldots, t_n)$, replacements are allowed on term t_i if and only if $i \in \mu(f)$. When $f \in \Sigma$ is a constant symbol, then $\mu(f) = \emptyset$. These restrictions are inductively raised to arbitrary positions of terms in the obvious way. By $O^\mu(t)$ we express the set of all the μ-*replacing* positions of a term t. Moreover, let $O_s(t)$ denote the set of all the positions of subterm s in term t.

In the CSR framework, we only rewrite replacing redexes; in other words,

> a term t μ-*rewrites* to a term s, in symbols $t \hookrightarrow_\mu s$, if and only if $t \xrightarrow{p}_{\mathcal{R}} s$ and p is a non-variable position in $O^\mu(t)$.

A term t is a *μ-normal form*, if there does not exist a term s such that $t \hookrightarrow_\mu s$. We say that a program $\mathcal{R}$ is *μ-terminating*, if $\hookrightarrow_\mu$ relation is terminating (that is, there are no infinite chain $t_1 \hookrightarrow_\mu t_2 \hookrightarrow_\mu \ldots$).

The *canonical* replacing map $\mu_{\mathcal{R}}^{can}$ is the most restrictive replacing map ensuring that the non-variable subterms of the left-hand sides of the rules of $\mathcal{R}$ are replacing [78]. Note that $\mu_{\mathcal{R}}^{can}$ can be automatically associated to a TRS $\mathcal{R}$ by means of a very simple calculus. Let $L(\mathcal{R})$ be the set of all the left-hand sides of a TRS $\mathcal{R}$, then, for each $f \in \Sigma$,

$$i \in \mu_{\mathcal{R}}^{can}(f) \text{ iff } \exists \lambda \in L(\mathcal{R}), p \in O_\Sigma(\lambda), (root(\lambda_{|p}) = f \wedge p.i \in O_\Sigma(\lambda)).$$

Example A.1.1 *Consider the program*

$$\begin{aligned} \mathcal{R} \equiv \quad & \{\, from(N) \rightarrow [N|from(s(N))], first(0, L) \rightarrow [\,], \\ & first(s(N), [X|L]) = [X|first(N, L)] \,\} \end{aligned}$$

Then, we have that $\mu_{\mathcal{R}}^{can}(s) = \mu_{\mathcal{R}}^{can}([\cdot|\cdot]) = \mu_{\mathcal{R}}^{can}(from) = \emptyset$ *and* $\mu_{\mathcal{R}}^{can}(first) = \{1, 2\}$.

Let $M_{\mathcal{R}}$ be the set of all the replacing maps for a CTRS $\mathcal{R}$. Given $M_{\mathcal{R}}$, we can define an *ordering* $\sqsubseteq$ on it as follows: $\mu \sqsubseteq \mu'$ if and only if for every $f \in \Sigma$, $\mu(f) \subseteq \mu'(f)$. The ordering $\sqsubseteq$ forms a complete lattice in $M_{\mathcal{R}}$. Let $CM_{\mathcal{R}} = \{\mu | \mu_{\mathcal{R}}^{can} \sqsubseteq \mu\}$.

A *context* is a term $C \in \tau(\Sigma \cup \{\Box\}, \mathcal{V})$ with zero or more "holes" $\Box$ (i.e., a new fresh constant symbol). We write $C[\,]$ to denote an arbitrary context with an arbitrary number of holes, $C[t_1, \ldots, t_n]$ denotes the term obtained by filling the holes of $C[\,]$ with terms $t_1, \ldots, t_n$. The *maximal replacing context* $MRC^\mu(t)$ of a term t is a context corresponding to the maximal prefix of t whose positions are μ-replacing in t. Hence, $\{p | p \notin O^\mu(MRC^\mu(t))\} = O_\Box(MRC^\mu(t))$.

A.1.2 Testing $\mathcal{R} \vdash E^+$

The procedure we outlined in Algorithm 7 slightly modifies the Lucas' algorithm of normalization via μ-normalization [79], in which the normalization of a term t proceeds by first obtaining a μ-normal form of t (namely, $\mu\mathsf{NF}(t)$) and then recursively normalizing the non-replacing subterms of $\mu\mathsf{NF}(t)$. It has been shown in [79] that this normalization method is correct and complete w.r.t. the set of all the normal forms of a term, whereas we consider left-linear TRSs and replacement maps in $CM_{\mathcal{R}}$. Indeed, under these conditions, each μ-normal form of a term t has got a maximal replacing context $MRC^\mu(\mu\mathsf{NF}(t))$ which is a 'rigid' term, that is, it cannot be rewritten any longer. We say that it represents a *stable* part of the final normal form. Hence, by recursively calculating μ-normal forms of subterms at non-replaceable positions, we calculate parts of the final normal form of a given term (if it exists).

We modify the methodology mentioned above in order to deal with equations and to establish if they are covered by a program. Given as input an equation of the form $t = c$, where c is a constructor term and hence a normal form, we prove $t = c$ by computing the normal form of t and comparing it to c by means of the normalization

via μ-normalization process. The methodology works for left-linear, confluent and μ-terminating TRS, where $\mu \in CM_{\mathcal{R}}$.

Roughly speaking, our procedure tries to compute a normal form of the left-hand side of the equation by finding stable parts (i.e. maximal replacing contexts of μ-normal forms) of the final normal form level by level and comparing them to the corresponding subterms in the right-hand side. The procedure returns *true* if and only if the normal form of the left-hand side is equal to the right-hand side of the equation. This suffices to prove the considered equation w.r.t a given TRS, provided that each term has got at most one normal form. In fact, a problem might arise, whenever a term can be reduced to several normal forms. Consider the equation $t = c$, where t has two distinct normal forms c, c' and suppose that t is normalized to c'. Then, the procedure would return *false*, although $t = c$ holds. Thus, to ensure the uniqueness of the normal forms and the correctness of the procedure, we only consider *confluent* TRSs.

Note that confluence does not imply the uniqueness of the μ-normal forms, however it preserves the uniqueness of the maximal replacing contexts of the μ-normal forms as proven in [78]. In short, this means that we are free to μ-normalize a term as we like, this will always result in computing the same maximal replacing context, which is what really matters.

The procedure works as follows. At each recursion call, we first compute the μ-normal form $\mu\mathsf{NF}(t, \mathcal{R})$ w.r.t. a given TRS $\mathcal{R}$, then we compare the maximal replacing context of $\mu\mathsf{NF}(t, \mathcal{R})$ to the maximal replacing context of c. If the test fails, the normal form of t (if it exists) cannot be equal to c, since there is a stable prefix of $\mu\mathsf{NF}(t, \mathcal{R})$ (namely, $MRC^{\mu}(\mu\mathsf{NF}(t, \mathcal{R}))$), which differs from the corresponding prefix of c. So, the procedure returns *false*. Otherwise, we know that $MRC^{\mu}(\mu\mathsf{NF}(t.\mathcal{R}))$ is a stable prefix of $\mu\mathsf{NF}(t, \mathcal{R})$ which is equal to the corresponding prefix of c (namely, $MRC^{\mu}(c)$). Here, we distinguish two cases.

1. There are no $\Box$ symbols in $MRC^{\mu}(\mu\mathsf{NF}(t, \mathcal{R}))$. Thus,

$$\mu\mathsf{NF}(t, \mathcal{R}) \equiv MRC^{\mu}(\mu\mathsf{NF}(t, \mathcal{R})) \equiv c$$

   by the fact that $\mathcal{R}$ is confluent, and therefore the procedure returns *true*.

2. There are $\Box$ symbols in $MRC^{\mu}(\mu\mathsf{NF}(t, \mathcal{R}))$. Since $MRC^{\mu}(\mu\mathsf{NF}(t, \mathcal{R}))$ is a stable prefix of $\mu\mathsf{NF}(t, \mathcal{R})$ which is equal to the corresponding prefix of c, we only need to recursively extend the check to all the subterms of $\mu\mathsf{NF}(t, \mathcal{R})$, which are rooted at each non-replaceable position.

In order to make the procedure effectively computable, we require that μ-normal forms are computed in finite time. A sufficient condition to guarantee it is considering μ-terminating TRSs. In this way, we can ensure that no infinite μ-rewriting sequences $t_1 \hookrightarrow_{\mu} t_2 \hookrightarrow_{\mu} \cdots$ can occur, therefore a μ-normal form is always computable in a finite number of μ-rewrite steps. Due to the syntactical restrictions imposed by CSR, the notion of μ-termination is weaker than standard termination, i.e., many TRSs which are μ-terminating are not terminating. This implies that it is generally much easier

proving μ-termination of a given TRS than its termination. Some methods to tackle this problem have been described in [79] and a practical tool can be found at the URL address already mentioned at the beginning of Section 3.3.

Example A.1.2 *Consider again the TRS of Example A.1.1 and the replacing map μ such that*

$$\mu(s) = \mu([\cdot|\cdot]) = \mu(from) = \{1\}, \mu(first) = \{1, 2\}.$$

Note that program $\mathcal{R}$ is left-linear, confluent and μ-terminating. Besides, we have that $\mu_{\mathcal{R}}^{can} \sqsubseteq \mu$.

Now, given the equation $(t = c) \equiv (first(s(0), from(0)) = [0])$, we apply Algorithm 7, tracing the computation step by step.

First of all, a μ-normal form of t is calculated:

$$\mu\mathsf{NF}(t, \mathcal{R}) = [0|first(0, from(s(0)))].$$

And, therefore, $MRC^{\mu}(\mu\mathsf{NF}(t, \mathcal{R})) \equiv [0|\Box]$. Since $MRC^{\mu}(\mu\mathsf{NF}(t, \mathcal{R})) \equiv MRC^{\mu}(c)$, we recursively call the procedure on the non-replaceable subterms of $\mu\mathsf{NF}(t, \mathcal{R})$. In this case, we have just one recursive call, as there is only one $\Box$ in $MRC^{\mu}(\mu\mathsf{NF}(t, \mathcal{R}))$. By executing the recursive call, we first compute a μ-normal form of the term

$$\mu\mathsf{NF}(t, \mathcal{R})_{|2} = first(0, from(s(0))),$$

obtaining $\mu\mathsf{NF}(\mu\mathsf{NF}(t, \mathcal{R})_{|2}, \mathcal{R}) \equiv [\ ]$. Then, by comparing the maximal replacing contexts of $\mu\mathsf{NF}(\mu\mathsf{NF}(t, \mathcal{R})_{|2}, \mathcal{R})$ and $c_{|2}$, we discover that

$$MRC^{\mu}(\mu\mathsf{NF}(\mu\mathsf{NF}(t, \mathcal{R})_{|2}, \mathcal{R})) \equiv [\ ] \equiv MRC^{\mu}(c_{|2}).$$

Hence, the recursive call returns true and the procedure terminates delivering the same value.

Consider now the equation $(t' = c') \equiv (from(0) = [0])$. Equation $t' = c'$ is not covered by $\mathcal{R}$, since the left-hand side of the equation does not admit a finite normal form. Therefore, the procedure should return false. Again, let us trace the execution of the procedure.

First, the μ-normal form of t' becomes $\mu\mathsf{NF}(t', \mathcal{R}) = [0|from(s(0))]$. Next, the comparison between $MRC^{\mu}(\mu\mathsf{NF}(t', \mathcal{R}))$ and $MRC^{\mu}(c')$ succeeds, since

$$MRC^{\mu}(\mu\mathsf{NF}(t', \mathcal{R})) \equiv [0|\Box] \equiv MRC^{\mu}(c').$$

Therefore, another recursive call is performed on equation $\mu\mathsf{NF}(t', \mathcal{R})_{|2} = c'_{|2}$. At this stage, we note that

$$MRC^{\mu}(\mu\mathsf{NF}(\mu\mathsf{NF}(t', \mathcal{R})_{|2}, \mathcal{R})) \equiv [s(0)|\Box] \not\equiv [\ ] \equiv MRC^{\mu}(c'_{|2}).$$

So, the call returns false and the algorithm terminates returning false as well.

Algorithm 7 An algorithm to test whether $\mathcal{R} \vdash t = c$.

```
1: function CHECK(t = c, R)
2:     t = μNF(t, R)
3:     if MRC^μ(t) ≢ MRC^μ(c) then
4:         return false
5:     else
6:         if O_□(MRC^μ(t)) = ∅ then
7:             return true
8:         else                ▷ t ≡ MRC^μ[t_1, ..., t_n] where t_i = t_|p_i, p_i ∈ O_□(MRC^μ(t))
9:             return ⋀_{i=1}^{n} check(t_i = c_|p_i)
10:        end if
11:    end if
12: end function
```

To summarize, algorithm 7 (namely, CHECK) allows to prove an equation of the form $t = c$ w.r.t. a given TRS $\mathcal{R}$. Therefeore, in order to check whether a TRS is overly general w.r.t. the sets of positive example E^+, we just have to execute the algorithm on every example belonging to E^+. In other words, we have

$$\mathcal{R} \vdash E^+ \quad \text{iff} \quad \bigwedge_{e \in E^+} \textsc{Check}(e) = \mathit{true}.$$

A.1.3 Extending the decision algorithm to CTRSs

Conditional rewriting is recognized as being much harder than unconditional rewriting. Thus, establishing if a CTRS is overly general may be a very complicated task which cannot be carried out by naïvely lifting Algorithm 7 to the CTRS' class. In the sequel of this section, we will consider *normal* CTRSs, which are defined in the following definition.

Definition A.1.3 *A CTRS $\mathcal{R}$ is a* normal *CTRS if each rule does not contain extra-variables and it is of the form $\lambda \to \rho \Leftarrow t_1 = c_1, \ldots, t_n = c_n$, where $c_i \in \tau(\mathcal{C})$, $i = 1, \ldots, n$.*

Instead of designing an algorithm suitable for normal CTRSs, we will decide the over-generality property of a given normal CTRS by reusing all the machinery we have already set up; roughly speaking, this means transforming the considered conditional program into an 'equivalent' unconditional one and then applying the previous procedure. For this purpose, we will use the unraveling transformation originally introduced by Marchiori [81], which maps CTRSs into TRSs.

Definition A.1.4 (Unraveling) *Given a rule of a normal CTRS $r \equiv \lambda \to \rho \Leftarrow t_1 = c_1, \ldots, t_n = c_n$ with $n \geq 1$, $Var(\lambda) = \{X_1, \ldots, X_p\}$ and u_r a fresh defined symbol of*

arity $n + p$ associated with rule r, we define the unraveling *of r (i.e.* $\mathsf{unravel}(r)$*) as follows*

$$\lambda \to u_r(t_1, \ldots, t_n, X_1, \ldots, X_p)$$
$$u_r(s_1, \ldots s_n, X_1, \ldots, X_p) \to \rho.$$

Given a normal CTRS $\mathcal{R}$, the unraveling *of $\mathcal{R}$ is* $\mathsf{unravel}(\mathcal{R}) = \bigcup_{r \in \mathcal{R}} \mathsf{unravel}(r)$.

By Definition A.1.4 it is immediate to see that the unraveling transformation preserves the left-linearity property, that is, if $\mathcal{R}$ is a left-linear normal CTRS [1], then $\mathsf{unravel}(\mathcal{R})$ is a left-linear TRS.

Another relevant property regarding the unraveling transformation is stated by the theorem below.

Theorem A.1.5 *[81] Let $\mathcal{R}$ be a left-linear normal CTRS, t and s be two terms. Then, $t \to^*_{\mathcal{R}} s$ iff $t \to^*_{\mathsf{unravel}(\mathcal{R})} s$.*

Notice that Theorem A.1.5 allows us to directly derive the preservation of the confluence property. In other words, if a left-linear normal CTRS is confluent, then the unravelled TRS is confluent.

Now we are ready to prove the following theorem.

Theorem A.1.6 *Let $\mathcal{R}$ be a left-linear, confluent, normal CTRS and $\mathcal{R}' = \mathsf{unravel}(\mathcal{R})$. Let $\mu \in CM_{\mathcal{R}'}$ and E be a finite set of ground examples. If $\mathcal{R}'$ is μ-terminating, then predicate $\mathcal{R} \vdash E$ is decidable.*

Proof. As unraveling preserves confluence and left-linearity, we have that $\mathcal{R}'$ is a left-linear and confluent TRS. Moreover, $\mathcal{R}'$ is μ-terminating and $\mu \in CM_{\mathcal{R}'}$. Thus, by applying Algorithm 7 to every example in E we can decide whether $\mathcal{R}' \vdash E$. At this point, by using Theorem A.1.5, we can show that $\mathcal{R} \vdash E$ iff $\mathcal{R}' \vdash E$. Thus, predicate $\mathcal{R} \vdash E$ is decidable as well. □

Finally, checking whether a program is overly-general w.r.t. a set of positive examples E^+ (that is, $\mathcal{R} \vdash E^+$) is trivially decidable by Theorem A.1.6.

[1] Left-linearity for normal CTRSs is defined as for TRSs.

B

Some technicalities

B.1 Proofs of the technical results of Chapter 3

Our unfolding methodology considers the leftmost-innermost narrowing strategy ($\varphi = inn$) and the needed narrowing strategy ($\varphi = needed$). In the case when $\varphi = inn$ is considered, the *normalizing* rewriting strategy which we employ for reducing the examples is leftmost-innermost rewriting, whereas outermost needed rewriting is applied in the case when $\varphi = needed$. By abuse of notation, we denote the normalizing rewriting strategy which is employed also by φ.

We use the following notion of conditional rewriting without evaluation of conditions, which slightly adapts the original definition by Bockmayr and Werner [27] to consider a rewriting strategy.

Definition B.1.1 *Let $\mathcal{R} \in I\!R_\varphi$ be a program and g, g' be two goals. We say that g* rewrites *to g' at position p via r w.r.t φ, in symbols $g \xrightarrow{\sigma,r,p}_\varphi g'$, if there exist a position $p \in \varphi(g)$, a variant of a rule $r \equiv \lambda \longrightarrow \rho \Leftarrow C \ll \mathcal{R}$ and a substitution σ such that $g_{|p} = \lambda\sigma$ and $g' = (C\sigma, g[r\sigma]_p)$. We omit some of the labels in $\xrightarrow{\sigma,r,p}_\varphi$ whenever this is clear from the context.*

We denote the *length* of the rewriting sequence $\mathcal{D}$ by $|\mathcal{D}|$. $\mathcal{D}^\varphi_\mathcal{R}(g)$ denotes the successful rewriting sequence which proves g in $\mathcal{R}$ by using the normalizing rewriting strategy φ.

Proposition B.1.2 *Let $\mathcal{R} \in I\!R_\varphi$ and g, g' be two* ground *goals. Then,*

$$g \equiv g_0 \longrightarrow_\varphi g_1 \longrightarrow_\varphi \ldots \longrightarrow_\varphi g_n \equiv g' \quad \textit{iff} \quad g \equiv g_0 \rightsquigarrow_\varphi g_1 \rightsquigarrow_\varphi \ldots \rightsquigarrow_\varphi g_n \equiv g'.$$

Proof. By induction on the length of the derivations. □

Definition B.1.3 *Let $\mathcal{R}$ be a CTRS and $r \equiv \lambda \to \rho \Leftarrow C, r' \ll \mathcal{R}$ two variants of rules in $\mathcal{R}$. Let $p \in \overline{O}(C, \rho = y)$. If $(C, \rho = y) \overset{\theta,r',p}{\rightsquigarrow}_\varphi (C', \rho' = y)$ is a conditional*

narrowing step, then we define the rule unfolding *of r w.r.t. r' at position p (the position p is referred to $\overline{O}(C, \rho = y)$) as*

$$unfold(r, r', p) = \{(\lambda\theta \to \rho' \Leftarrow C')\}.$$

Given a substitution $\theta = \{X_1/t_1, \ldots, X_n/t_n\}$, we denote the equational representation of θ by $\hat{\theta} = \{X_1 = t_1, \ldots, X_n = t_n\}$. In order to prove some auxiliary results, we use to the commutative parallel composition operator $\Uparrow$ [98], which is defined as $\theta_1 \Uparrow \theta_2 = mgu(\theta_1 \cup \theta_2)$. An interesting property of the operator $\Uparrow$ is the following.

Proposition B.1.4 *[98] Let θ_1 and θ_2 be two substitutions. Then,*

$$\theta_1 \Uparrow \theta_2 = \theta_1 mgu(\hat{\theta_2}\theta_1) = \theta_2 mgu(\hat{\theta_1}\theta_2).$$

The following result holds (c.f. Lemma 3.2.8 in [82]).

Lemma B.1.5 (Contraction) *[82] Let $\mathcal{R}$ be a CTRS, g_0 a goal, and $r, r' \ll \mathcal{R}$. Then,*

$$g_0 \overset{q,r,\theta}{\leadsto} g \overset{p',r',\theta'}{\leadsto} g' \quad \textit{iff} \quad g_0 \overset{q,r'',\theta''}{\leadsto} g''$$

with p' a non-variable position in the rhs or the condition of r, $r'' \in unfold(r, r', p)$, $g' = g''$ and $\theta\theta' = \theta''_{\restriction g_0}$

Proof. ($\Rightarrow$) Let $r \equiv (\lambda \to \rho \Leftarrow C)$ and $r' \equiv (\lambda' \to \rho' \Leftarrow C')$. W.l.o.g., we assume that p is a non-variable position in C and, therefore, $p = p'$ (the case where p is a non-variable position of ρ is analogous).
Since $g_0 \overset{q,r,\theta}{\leadsto} g \overset{p',r',\theta'}{\leadsto} g'$, we have that $\theta = mgu(\{g_{0|_q} = \lambda\})$, $g = (C, g_0[\rho]_q)\theta$, $\theta' = mgu(\{C\theta_{|_p} = \lambda'\})$, and $g' = (C', C[\rho']_p, g_0[\rho]_q)\theta\theta'$. Now, let us consider $\sigma = mgu(\{C_{|_p} = \lambda'\}$. We have that

$$\begin{aligned}
\theta\theta' &= \\
\theta mgu\{C\theta_{|_p} = \lambda'\} &= \text{(since } Dom(\theta) \cap Var(r') = \emptyset) \\
\theta mgu(\widehat{mgu}(\{C_{|_p} = \lambda'\})\theta) &= \\
\theta mgu(\hat{\sigma}\theta) &= \text{(by Proposition B.1.4)} \\
\theta \Uparrow \sigma &= \text{(by Proposition B.1.4)} \\
\sigma mgu(\hat{\theta}\sigma) &= \\
\sigma mgu(\widehat{mgu}(\{g_{0|_q} = \lambda\})\sigma) &= \text{(since } Dom(\sigma) \cap Var(g_0) = \emptyset) \\
\sigma mgu(\{g_{0|_q} = \lambda\sigma\}). &
\end{aligned}$$

Besides, as $\theta\theta' \neq fail$, $\sigma \neq fail$. Thus, there exists a rule $r'' = (\lambda \to \rho \Leftarrow C', C[\rho']_p)\sigma \in unfold(r, r', p)$. Now, since $mgu(\{g_{0|_q} = \lambda\sigma\}) \neq fail$, the following narrowing step is enabled: $g_0 \overset{q,r'',\sigma''}{\leadsto} g''$, where $\theta'' = mgu(\{g_{0|_q} = \lambda\sigma\}$ and $g'' = (C'\sigma, C[\rho']_p\sigma, g_0[\rho\sigma]_q)\theta''$. Finally, by $\theta\theta' = \sigma\theta''$ and $Dom(\sigma) \cap Var(g_0) = \emptyset$), we have that $g' = g''$ and $\theta\theta' = \theta''_{\restriction g_0}$.
($\Leftarrow$) This case is similar to the previous one. We just need to exploit again the equivalence between $\theta\theta'$ and $\sigma\theta''$. □

Now, we are able to demonstrate the following proposition, which essentially states that the length of the rewriting proofs for the considered examples are shortened by unfolding.

Proposition B.1.6 *Let $\mathcal{R} \in I\!R_\varphi$, $\varphi \in \{inn, needed\}$, $\mathcal{R}' = \mathsf{U}_r^\varphi(\mathcal{R})$, $r \ll \mathcal{R}$, and g be a ground goal. Then, we have*

1. *if $g \to_\varphi^* \top$ in $\mathcal{R}$ then also $g \to_\varphi^* \top$ in $\mathcal{R}'$*
2. *if $r \in \mathsf{OR}(\mathcal{D}_\mathcal{R}^\varphi(g))$, then $|\mathcal{D}_{\mathcal{R}'}^\varphi(g)| < |\mathcal{D}_\mathcal{R}^\varphi(g)|$.*

Proof. We prove (1) and (2) by induction on the length n of $\mathcal{D}_\mathcal{R}(g)$.

$\mathbf{n = 0}$. Since $g \equiv \top$, claims (1) and (2) hold trivially.

$\mathbf{n > 0}$. $\mathcal{D}_\mathcal{R}^\varphi(g)$ contains at least one redex in a position $q \in \varphi(g)$ which is reduced via $r_1 \equiv \lambda_1 \to \rho_1 \Leftarrow C_1 \ll \mathcal{R}$. So, we have

$$\mathcal{D}_\mathcal{R}^\varphi(g) \equiv g \overset{\sigma_1, r_1, q}{\longrightarrow_\varphi} (C_1\sigma_1, g[\rho_1\sigma_1]_q) \longrightarrow_\varphi^* true.$$

Here we consider two cases.

Case $\mathbf{r_1} \in \mathcal{R}'$. Since r_1 belongs to both programs, $\mathcal{R}$ and $\mathcal{R}'$, we have that the first reduction step of $\mathcal{D}_\mathcal{R}^\varphi(g)$ is also a reduction step w.r.t. $\mathcal{R}'$. Moreover, by the induction hypothesis, $(C_1\sigma_1, g[\rho_1\sigma_1]_q) \longrightarrow_\varphi^* true$ is a successful rewriting sequence in $\mathcal{R}'$. Thus, (1) holds.

To prove (2), we first observe that $r_1 \not\equiv r$ by Definition 3.3.2 (unfolding). By induction hypothesis, if $r \in \mathsf{OR}(\mathcal{D}_\mathcal{R}^\varphi(C_1\sigma_1, g[\rho_1\sigma_1]_q))$, then $|\mathcal{D}_{\mathcal{R}'}^\varphi(C_1\sigma_1, g[\rho_1\sigma_1]_q)| < |\mathcal{D}_\mathcal{R}^\varphi(C_1\sigma_1, g[\rho_1\sigma_1]_q)|$, and claim (2) follows.

Case $\mathbf{r_1} \notin \mathcal{R}'$. By Definition 3.3.2, we have that $r_1 \equiv r$. Since $r \equiv r_1$ is unfoldable, there is at least one defined function symbol in the rhs or the condition of r_1. Therefore, the derivation $\mathcal{D}_\mathcal{R}^\varphi(g)$ has the form

$$\mathcal{D}_\mathcal{R}^\varphi(g) \equiv g \overset{\sigma_1, r, q}{\longrightarrow_\varphi} (C_1\sigma_1, g[\rho_1\sigma_1]_q) \overset{\sigma_2, r_2, p}{\longrightarrow_\varphi} g'' \longrightarrow_\varphi^* true$$

with $r_2 \ll \mathcal{R}$. Since g is ground, by Proposition B.1.2, the sequence $\mathcal{D}_\mathcal{R}^\varphi(g)$ is a narrowing derivation for g in $\mathcal{R}$ as well. By applying Lemma B.1.5 to

$$g \overset{r, q}{\leadsto_\varphi} (C_1\sigma_1, g[\rho_1\sigma_1]_q) \overset{r_2, p}{\leadsto_\varphi} g''$$

we know that

$$g \overset{r^*, q}{\leadsto_\varphi} g'',$$

with $r^* \in \mathcal{R}' = \mathsf{U}_r^\varphi(\mathcal{R})$. By applying Proposition B.1.2 in the opposite sense of direction, we get

$$g \overset{q, r^*}{\longrightarrow_\varphi} g'',$$

which is a one-step rewriting sequence in $\mathcal{R}'$ w.r.t. φ, since $q \in \varphi(g)$. Now, by the induction hypothesis, a successful rewriting sequence for g'' in $\mathcal{R}'$ w.r.t. φ does exist. Hence, the sequence $\mathcal{D}^{\varphi}_{\mathcal{R}'}(g) \equiv g \longrightarrow^{*}_{\varphi} \mathit{true}$ is proven, which demonstrates (1).

Let us give the proof for (2). First, remember that the rules r_1 and r_2 of $\mathcal{R}$ used in $\mathcal{D}^{\varphi}_{\mathcal{R}}(g)$ are replaced by the rule $r^* \in \mathcal{R}'$ in $\mathcal{D}^{\varphi}_{\mathcal{R}'}(g)$. Now, if $r \equiv r_1$ occurs in $\mathsf{OR}(\mathcal{D}^{\varphi}_{\mathcal{R}'}(C_1\sigma_1, g[\rho_1\sigma_1]_q))$, by induction hypothesis,

$$|\mathcal{D}^{\varphi}_{\mathcal{R}'}(C_1\sigma_1, g[\rho_1\sigma_1]_q)| < |\mathcal{D}^{\varphi}_{\mathcal{R}}(C_1\sigma_1, g[\rho_1\sigma_1]_q)|,$$

hence $|\mathcal{D}^{\varphi}_{\mathcal{R}'}(g)| < |\mathcal{D}^{\varphi}_{\mathcal{R}}(g)|$. If $r \notin \mathsf{OR}(\mathcal{D}^{\varphi}_{\mathcal{R}'}(C_1\sigma_1, g[\rho_1\sigma_1]_q))$, then by Definition 3.3.2, each rule in $\mathsf{OR}(\mathcal{D}^{\varphi}_{\mathcal{R}'}(C_1\sigma_1, g[\rho_1\sigma_1]_q)) \subset \mathcal{R}$. Then,

$$|\mathcal{D}^{\varphi}_{\mathcal{R}'}(C_1\sigma_1, g[\rho_1\sigma_1]_q)| = |\mathcal{D}^{\varphi}_{\mathcal{R}}(C_1\sigma_1, g[\rho_1\sigma_1]_q)|,$$

hence $|\mathcal{D}^{\varphi}_{\mathcal{R}'}(g)| < |\mathcal{D}^{\varphi}_{\mathcal{R}}(g)|$.

□

Now, we are ready to proceed with the proofs of the correctness of Algorithm 2 which is described in Chapter 3.

Proposition 3.3.7 *Let φ be a normalizing rewriting strategy for $\mathbb{R}_{\varphi}$ and $\mathcal{R}$ be a program in $\mathbb{R}_{\varphi}$. Let E^+ (resp. E^-) be a set of positive (resp. negative) examples. If there are no $e^+ \in E^+$ and $e^- \in E^-$ which can be proven in $\mathcal{R}$ by using the same rule sequence, then, for each unfolding succession $\mathcal{S}(\mathcal{R})$, there exists a natural k, such that*

$$\forall e^- \in E^- \exists r \in \mathsf{OR}(\mathcal{D}_{\mathcal{R}_k}(e^-)) \textit{ s.t. } r \textit{ is not discriminable.}$$

Proof. W.l.o.g., we consider those successful rewrite sequences for the example e of the form $e \longrightarrow^{*}_{\varphi} c_1 = c_1, \ldots c_n = c_n \longrightarrow^{*}_{\varphi} \top$, with $c_i \in \tau(\mathcal{C})$; that is, program rules are first applied and then the equality rules are applied to reduce the ground constructor equation set $c_1 = c_1, \ldots c_n = c_n$ to $\top$. We also assume as many fresh constants available as required to derive the program correction. The key idea for the proof is in the following fact (which holds by Proposition B.1.6).

> At each unfolding step involving a discriminable rule, the length of the proof of, at least, one positive example decreases.

Therefore, by a finite number k of unfolding steps, we get program $\mathcal{R}_k$ where each $e^+ \in E^+$ succeeds by using just one rule r of $\mathcal{R}_k$. This amounts to saying that there is no defined symbol in the rhs or the condition of r.
Now, consider a negative example e^- and the corresponding proof $\mathcal{D}^{\varphi}_{\mathcal{R}_k}(e^-)$. In order to prove the claim, we distinguish two cases:

$|\mathcal{D}^{\varphi}_{\mathcal{R}_k}(\mathbf{e}^-)| > \mathbf{1}$. In this case, there exists one rule $r \in \mathcal{R}_k$ occurring in $\mathcal{D}_{\mathcal{R}_k}(e^-)$, where the rhs or the condition of r contains at least one defined function symbol. Hence, r cannot occur in the proof of any positive example and, thus, the claim follows.

$|\mathcal{D}^{\varphi}_{\mathcal{R}_k}(\mathbf{e}^-)| = \mathbf{1}$. Let $r \in \mathcal{R}_k$ be the rule used to prove e^-. By contradiction, suppose that there exists a positive example $e^+ \in E^+$, whose proof $\mathcal{D}^{\varphi}_{\mathcal{R}_k}(e^+)$ uses the very same rule r. Since $\mathcal{R}_k$ derives by unfolding from $\mathcal{R}$, then by repeatedly applying the Contraction Lemma B.1.5 and by Proposition B.1.2, the application of the rule $r \in \mathcal{R}_k$ can be mimicked in $\mathcal{R}$ by applying a rule sequence $\langle r_1, \ldots, r_n \rangle$ of $\mathcal{R}$. This means that examples e^+ and e^- can be proven in $\mathcal{R}$ by using the same rules sequence, which leads to a contradiction.

□

Theorem 3.3.8 [Correctness] *Let $\mathcal{R} \in I\!R_{\varphi}$ which satisfies the csr conditions, E^+ and E^- be two sets of examples such that $\mathcal{R} \vdash E^+$. If the rewriting rule sequences for $e^+ \in E^+$ and $e^- \in E^-$ are different, then the* TD-CorrectorFL *algorithm yields a correct specialization of $\mathcal{R}$ w.r.t. E^+ and E^-.*

Proof. Assuming that $\mathcal{R}$ satisfies the *csr* conditions [79], then each example $l = c \in E^+$, with ρ ground constructor term and l ground pattern, can be checked by finitely comparing c with the "maximal context" of l (normalization via μ–normalization) (See appendix A.1). In the case when this test succeeds, we know that there exists a (finite) rewriting sequence which proves e in $\mathcal{R}$ using the normalizing strategy φ for the class $I\!R_{\varphi}$. On the other hand, since the semantics of $\mathcal{R}$ is preserved by unfolding w.r.t. φ under the conditions of Definition 3.3.3, program $\mathcal{R}_k$ does cover E^+ also. According to Proposition 3.3.7, every negative example e^- succeeds in $\mathcal{R}_k$ by means of (at least) one rule which is not used in the proof of any positive example. Subsequently, the Deletion phase removes all, and only, those rules $r \in \mathcal{R}_k$ which do not appear in the proof of any positive example. Hence, a specialized program is computed which does not cover E^- while still succeeds on the whole E^+, which gives the desired result. □

B.2 Proofs of the technical results of Chapter 4

Lemma 4.3.1 *Let $\mathcal{I}$ be a TRS and $E_P := \{e | e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \wedge Var(e) \cap \hat{\mathcal{V}} = \emptyset\}$. Then, $E_P \subseteq Sem_{\mathsf{val}}(\mathcal{I})$.*

Proof. Let $e \in E_P$. Then, $Var(e) \cap \hat{\mathcal{V}} = \emptyset$ and $e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I})$. By Claim (2) of Proposition 4.1.8, $e \in \mathcal{F}_{\mathsf{val}}(\mathcal{I})$. By Theorem 4.1.5, $e \in Sem_{\mathsf{val}}(\mathcal{I})$. □

Lemma 4.3.2 *Let $\mathcal{R}$ be a TRS, $\mathcal{I}$ be a specification of the intended semantics and $E_N := \{e | e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \wedge Var(e) \cap \hat{\mathcal{V}} = \emptyset \wedge \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \not\leq_S \{e\}\}$. Then, $E_N \subseteq (Sem_{\mathsf{val}}(\mathcal{R}) \setminus Sem_{\mathsf{val}}(\mathcal{I}))$.*

Proof. Let $e \in E_N$. Then, $e \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ and $Var(e) \cap \hat{\mathcal{V}} = \emptyset$. By Claim (2) of Proposition 4.1.8, $e \in \mathcal{F}_{\mathsf{val}}(\mathcal{R})$, and thus by Theorem 4.1.5, $e \in Sem_{\mathsf{val}}(\mathcal{R})$. Now, we proceed by contradiction. Suppose that $e \in Sem_{\mathsf{val}}(\mathcal{I})$. Since $e \in E_N$, $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \not\leq_S \{e\}$. This implies that, for each $e' \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I})$, $e' \not\leq e$. By Theorem 4.1.5, $e \in \mathcal{F}_{\mathsf{val}}(\mathcal{I})$. By Claim (1) of Proposition 4.1.8, there exists $e' \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I})$ such that $e' \leq e$, which contradicts the fact that $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{I}) \not\leq_S \{e\}$. Hence, $e \notin Sem_{\mathsf{val}}(\mathcal{I})$. Finally, $e \in (Sem_{\mathsf{val}}(\mathcal{R}) \setminus Sem_{\mathsf{val}}(\mathcal{I}))$. □

Proposition 4.4.1 *Let $\mathcal{R}$ be a TRS and E^+ be a set of positive examples. If, for each $e \in E^+$, there exists $e' \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ s.t.*

1. *$e' \leq e$;*
2. *$Var(e') \cap \hat{\mathcal{V}} = \emptyset$;*

then, $\mathcal{R}$ is overly general w.r.t. E^+.

Proof. Let $e \equiv (l = c) \in E^+$. Then, there exists $e' \equiv (l' = c) \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$. Since $Var(e') \cap \hat{\mathcal{V}} = \emptyset$ holds, we can apply Proposition 4.1.8 to e'. So, $e' \in \mathcal{F}_{\mathsf{val}}(\mathcal{R})$. Thus, $e' \in Sem_{\mathsf{val}}(\mathcal{R})$ (that is, $l' \rightarrow^{!}_{\mathcal{R}} c$). Since $e' \leq e$, there exists σ such that $e'\sigma \equiv e$. This implies that $l'\sigma = l$. By the stability of the rewriting relation and the fact that c is a ground normal form, $l \equiv l'\sigma \rightarrow^{!}_{\mathcal{R}} c\sigma \equiv c$, therefore $e \in Sem_{\mathsf{val}}(\mathcal{R})$. This proves that $\mathcal{R}$ is overly-general w.r.t. E^+. □

The following auxiliary lemma is instrumental for the proof of Theorem 4.4.7.

Lemma B.2.1 *Let $r_1 \equiv \lambda_1 \rightarrow \rho_1$ and r_2 be two rules. Let $t_0, t_1, t_2 \in \tau(\Sigma \cup \mathcal{V})$. Then, $t_0 \rightarrow_{r_1,p_1} t_1 \rightarrow_{r_2,p_2} t_2$, where $p_2 \in O_\Sigma(t_1)$ corresponds to some position $p \in O_\Sigma(\rho_1)$, iff $t_0 \rightarrow_{r',p_1} t_2$, where $r' \in \mathcal{U}_{r_2}(r_1)$.*

Proof. It follows immediately from Lemma 3.2.8 in [82]. □

Theorem 4.4.7 *Let $\mathcal{R}$ be a well-framed left-linear CS, $r \ll \mathcal{R}$ be an unfoldable rule and $\mathcal{R}' = \mathcal{U}_{\mathcal{R}}(r)$. Let $e \equiv (l = c)$ be an equation such that $l \in \tau(\Sigma, \mathcal{V})$ and $c \in \tau(\mathcal{C})$. Then, if $e \in Sem_{\mathsf{val}}(\mathcal{R})$, then $e \in Sem_{\mathsf{val}}(\mathcal{R}')$.*

Proof. The proof proceeds by induction following a proof scheme similar to the one proposed in [82]. □

Lemma 4.4.11 *Let $\mathcal{R}$ be a well-framed left-linear CS, E be a set of examples and $r \in first(E)$ be an unfoldable rule such that $\mathcal{R}' = \mathcal{U}_{\mathcal{R}}(r)$. Let $t = c \in E$, where t is a pattern and c is a value. Then,*

1. *if $\mathcal{S}$ is a rewrite sequence from t to c in $\mathcal{R}$, then there exists a rewrite sequence $\mathcal{S}'$ from t to c in $\mathcal{R}'$;*

2. if r occurs in $\mathcal{S}$, then $|\mathcal{S}'| < |\mathcal{S}|$.

Proof. Claim (1) directly follows from Theorem 4.4.7. Claim (2) is based on the following observation. Consider a rewrite sequence for $t = c$ in $\mathcal{R}$

$$\mathcal{S} \equiv t \to_{r_1,q_1} t[\rho_1\sigma_1]_{q_1} \to_{r_2,q_2} t_2 \ldots \to_{r_n,q_n} c,$$

where $(r_1 \equiv \lambda_1 \to \rho_1) \in \mathit{first}(E)$ and r_1 is unfoldable. Since term t is a pattern, Lemma B.2.1 applies to the first two rewrite steps of $\mathcal{S}$. Therefore, we can mimic the first two rewrite steps of $\mathcal{S}$ by a single rewrite step $t \to_{r^*,q_1} t_2$ using a rule $r^* \in \mathcal{U}_{r_2}(r_1) \subseteq \mathcal{U}_R(r_1) \equiv \mathcal{R}'$, which demonstrates that a derivation $\mathcal{S}'$ in $\mathcal{R}'$ shorter than $\mathcal{S}$ does exist. □

Lemma 4.4.12 *Let $\mathcal{R}$ be a well-framed left-linear CS, E be an example set and $t = c \in E$, where t is a pattern and c is a value. Let $t \to_{r_1} \ldots \to_{r_n} c$, $n \geq 1$. Then, for each unfolding succession $\mathcal{US}(\mathcal{R})$ w.r.t. E, there exists $\mathcal{R}_k$ occurring in $\mathcal{US}(\mathcal{R})$ such that $t \to_{r^*} c$, $r^* \in \mathcal{R}_k$.*

Proof. We demonstrate the claim by induction on the length n of the rewrite sequence from t to c.

Case $n = 1$**.** We have that $t \to_{r_1} c$ in $\mathcal{R}$. For each unfolding succession $\mathcal{US}(\mathcal{R})$ w.r.t. E, $\mathcal{R}_0 \equiv \mathcal{R}$. Then, by taking $k = 0$ the claim is proven.

Case $n > 1$**.** Let $\mathcal{S} \equiv t \to_{r_1} t_1 \ldots \to_{r_n} c$ in $\mathcal{R}$, where $n > 1$. Consider an arbitrary unfolding succession $\mathcal{US}(\mathcal{R}) \equiv \mathcal{R}_0, \mathcal{R}_1, \ldots$ w.r.t. E. Moreover, $r_1 \in \mathit{first}(E)$. By the fact that t is a pattern and $\mathcal{R}$ is well-framed, r_1 is unfoldable. Therefore, there exists an $\mathcal{R}_{k'}$, $k' \geq 0$, in $\mathcal{US}(\mathcal{R})$ such that $\mathcal{R}_{k'+1} \equiv \mathcal{U}_{\mathcal{R}_{k'}}(r_1)$. By repeatedly applying Lemma 4.4.11, we have that there exists a rewrite sequence $\mathcal{S}'$ in $\mathcal{R}_{k'+1}$, which proves $t = c$ and $|\mathcal{S}'| < |\mathcal{S}|$. By inductive hypothesis, for each $\mathcal{US}(\mathcal{R}_{k'+1})$ w.r.t. E, there exists a $\mathcal{R}_{k''}$ occurring in $\mathcal{US}(\mathcal{R}_{k'+1})$ such that $t \to_{r^*} c$, $r^* \in \mathcal{R}_{k''}$. Thus, by choosing program $\mathcal{R}_{k'+1+k''}$ in $\mathcal{US}(\mathcal{R})$, we get the desired result.

□

Lemma 4.4.13 *Let $\mathcal{R}$ be a TRS and e an equation. Then, if $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \not\leq_S \{e\}$, then $e \notin \mathit{Sem}_{\mathsf{val}}(\mathcal{R})$.*

Proof. Let us proceed by contradiction. Suppose $e \in \mathit{Sem}_{\mathsf{val}}(\mathcal{R})$, then by Theorem 4.1.5, $e \in \mathcal{F}_{\mathsf{val}}(\mathcal{R})$. By Claim (1) of Proposition 4.1.8, there exists $e' \in \widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R})$ such that $e' \leq e$, which contradicts the hypothesis $\widetilde{\mathcal{F}}^{\kappa}_{\mathsf{val}}(\mathcal{R}) \not\leq_S \{e\}$. Thus, $e \notin \mathit{Sem}_{\mathsf{val}}(\mathcal{R})$. □

Theorem 4.4.14 *Let $\mathcal{R}$ be a well-framed left-linear CS and $\mathcal{I}$ be a specification of the intended semantics of $\mathcal{R}$. Let E^+ and E^- be the example sets generated by* ExGen$(\mathcal{R},\mathcal{I})$.

1. *If (E^+,E^-) is discriminable in $\mathcal{R}$, then the algorithm* TDCorrectorF$(\mathcal{R},\mathcal{I})$ *terminates.*
2. *If $\mathcal{R}$ is overly general w.r.t. E^+ and* TDCorrectorF$(\mathcal{R},\mathcal{I})$ *terminates, then the computed program $\mathcal{R}^c$ is correct w.r.t. E^+ and E^-.*

Proof.

Claim (i). First of all, the termination of the procedure ExGen$(\mathcal{R},\mathcal{I})$, which is used to generate the example sets, trivially holds, since abstract semantics is finitely computed. Then, note that the overgenerality test is computable as well. Therefore, if program $\mathcal{R}$ is not overly general w.r.t. E^+, the algorithm halts without delivering a correction. Otherwise, we have to demonstrate termination of the **while** loop for an overly-general program w.r.t E^+. It suffices to prove that program rules cannot be infinitely unfolded. As the considered programs are well-framed left-linear CS, Lemma 4.4.12 holds. Since the rule selection strategy is fair and by applying Lemma 4.4.12, in the worst case after a finite number n of unfolding steps, we get program $\mathcal{R}_n$, where at least a rewrite sequence for every $e^+ \in E^+$ consist of a unique step. This amounts to saying that, in every one-step rewrite sequence of $\mathcal{R}_n$ proving a positive example, the applied rule contains no defined symbols in their rhs's (that is, the applied rule is not unfoldable any further) and is of the form $\lambda \to c$, where λ is a pattern and c belongs to $\tau(\mathcal{C},\mathcal{V})$.

Let us consider the iteration of the **while** loop in which we obtain such a program $\mathcal{R}_n$. At this point, no rule of $\mathcal{R}_n$—which is used to prove some positive example—is unfoldable, so no more unfolding steps are possible. Now, a refinement of program $\mathcal{R}_n$ is computed after removing some rules by means of the OverlyGeneral procedure. W.l.o.g., we assume that the considered k for the *depth-k* abstraction is greater than or equal to the maximum depth of the terms occurring in E^+. Consequently, $(\lambda = c) \in \widetilde{\mathcal{F}}^k_{\mathsf{val}}(\mathcal{R}_n)$ for each $\lambda \to c \in \mathcal{R}_n$ which is used to prove the positive examples; and therefore, by Proposition 4.4.1, the program $\mathcal{R}_{E^+} = \{\lambda \to c \in \mathcal{R}_n | \lambda \to c$ is used to prove some $e^+ \in E^+\}$ is overly general w.r.t. E^+.

Now, we show that, after the deletion phase, no negative example can be proven in $\mathcal{R}_n$ and thus the **while** loop exits, hence the algorithm terminates. Let us consider a negative example $e^- \in E^-$. Let $\mathcal{S}_{e^-}$ be a rewrite sequence in $\mathcal{R}_n$ for e^-. In order to prove the claim we distinguish two cases.

Case $|\mathcal{S}_{e^-}| > 1$**.** In this case, there exists one rule $r \in \mathcal{R}_k$ occurring in $\mathcal{S}_{e^-}$, where the rhs of r contains at least one defined function symbol. Since $\mathcal{R}_n \setminus \{r\} \supseteq \mathcal{R}_{E^+}$, $\mathcal{R}_n \setminus \{r\}$ is overly general w.r.t. E^+. Thus, rule r is not necessary to prove any positive example in $\mathcal{R}_n$ and can be safely deleted.

Case $|\mathcal{S}_{e^-}| = 1$**.** If the unique rule r occurring in $\mathcal{S}_{e^-}$ is never used to prove some positive example; then, $\mathcal{R}_n \setminus \{r\} \supseteq \mathcal{R}_{E^+}$ is overly general w.r.t. E^+ and rule r is deleted.

On the other hand, it cannot happen that rule r appears in a proof of a positive example. Let us demonstrate it by contradiction. Let $r \in \mathcal{R}_n$ be the unique rule used in $\mathcal{S}_{e^-}$. Now, suppose that r is also used to prove a positive example, i.e., there exists a positive example $e^+ \in E^+$, which is proven by a rewrite sequence $\mathcal{S}_{e^+}$ of $\mathcal{R}_n$ using the same rule r. Since $\mathcal{R}_n$ derives by unfolding from $\mathcal{R}$, by repeatedly applying Lemma B.2.1, the application of rule $r \in \mathcal{R}_n$ can be mimicked in $\mathcal{R}$ by applying a sequence of rules $\langle r_1, \ldots, r_n \rangle$ of $\mathcal{R}$. This means that examples e^+ and e^- can be proven in $\mathcal{R}$ by using the same sequence of rules (i.e. (E^+, E^-) is not discriminable in $\mathcal{R}$), which leads to a contradiction.

Therefore, after having computed the specialization $\mathcal{R}_n$ by unfolding, all the rules which are not needed to prove E^+ are removed, which implies the termination of the **while** loop and hence the termination of the TDCorrectorF algorithm, which delivers a program called $\mathcal{R}^c$.

Claim (ii). Since the algorithm TDCorrectorF$(\mathcal{R}, \mathcal{I})$ terminates and the program $\mathcal{R}$ is overly general w.r.t. E^+, the algorithm delivers a program $\mathcal{R}^c$ (see proof of Claim (i)).

Now we show that $\mathcal{R}^c$ is correct w.r.t. E^+ and E^-. By Lemma 4.4.13, when the **while** loop is exited, we obtain a specialization $\mathcal{R}^c \equiv \mathcal{R}_k$ such that $E^- \cap Sem_{\mathsf{val}}(\mathcal{R}_k) = \emptyset$. Moreover, we have the following facts:

- the original program $\mathcal{R} \equiv \mathcal{R}_0$ is overly general w.r.t. E^+;
- if $E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{R}_k)$, then $E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{R}_{k+1})$ (cf. Theorem 4.4.7);
- rule deletion is only performed if the resulting program specialization is overly general w.r.t. E^+.

Therefore, we have that $E^+ \subseteq Sem_{\mathsf{val}}(\mathcal{R}^c)$ and $E^- \cap Sem_{\mathsf{val}}(\mathcal{R}^c) = \emptyset$, when the procedure terminates. Hence, $\mathcal{R}^c$ is correct w.r.t. E^+ and E^-.

□

B.3 Proofs of the technical results of Chapter 5

Proposition 5.4.5 *Let* $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2)$ *be two Web page templates in* $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$*. A minimal page simulation* $\mathbf{S}$ *of* $\mathtt{s}_1$ *in* $\mathtt{s}_2$ *w.r.t.* $\sim$ *is a mapping* $\mathbf{S} : V_1 \to V_2$ *.* **Proof.** Let $\mathtt{s}_1 \equiv (V_1, E_1, r_1, label_1)$, $\mathtt{s}_2 \equiv (V_2, E_2, r_2, label_2) \in \tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$. By Conditions 1 and 3 of Definition 5.4.3,

and by using the fact that $\mathtt{s}_1$ has an underlying tree structure (in particular, it is a connected graph), we have that

$$\forall\, v_1 \in V_1,\ \exists\, v_2 \in V_2 \text{ such that } (v_1, v_2) \in \mathbf{S}.$$

Moreover, the minimality of $\mathbf{S}$ ensures that

$$\forall\, v_1 \in V_1,\ \exists \text{ a } \textit{unique}\ v_2 \in V_2 \text{ such that } (v_1, v_2) \in \mathbf{S}$$

which implies that $\mathbf{S}$ is a mapping from V_1 to V_2. □

Proposition 5.5.3 *Let W be a Web site, and $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a Web specification. Then, the procedure* CORRECTNESS-ERRORS*$(W, I_{\mathtt{N}}, R)$ terminates, and for each pair $(\mathtt{p}, \mathtt{l}\sigma)$, which is returned, $\mathtt{p}$ is an incorrect Web page w.r.t. $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ and $\mathtt{l}\sigma$ is the corresponding incorrectness symptom.* **Proof.** First of all, let us prove the termination of the algorithm. The outer loop (lines 2–9) is executed $|W|$ times. The inner loop (lines 3–8) is executed $|I_{\mathtt{N}}|$ times. As for the partial rewrite step $\mathtt{p} \rightharpoonup^{\sigma}_{r_u} \mathtt{error}$ in line 4, it terminates, because it is just an application of the simulation algorithm [66], which terminates. The evaluation of the condition in line 5 terminates, since:

1. the problem of evaluating $(\mathtt{X}_{\mathtt{i}}\sigma \in \mathcal{L}(\mathtt{rexp}_{\mathtt{i}}), i = 1, \ldots, n)$ boils down to the membership problem in regular languages, which is decidable (see [70]), so there exists an effective procedure which tests $(\mathtt{X}_{\mathtt{i}}\sigma \in \mathcal{L}(\mathtt{rexp}_{\mathtt{i}}), i = 1, \ldots, n)$ in finite time.

2. evaluating the condition $((\mathtt{s}_{\mathtt{j}} = \mathtt{t}_{\mathtt{j}})\sigma, j = 1, \ldots, m,$ holds in $R)$ trivially terminates, since R is canonical (in particular, it is terminating).

The output command in line 6 clearly terminates. Summing up, we execute the block of terminating instructions, which is included in lines 4–7, $|W| * |I_{\mathtt{N}}|$ times, so the whole procedure terminates.
Let us prove the partial correctness of the algorithm, that is, if the procedure terminates producing the outcome $(\mathtt{p}, \mathtt{l}\sigma)$, then $\mathtt{p} \in W$ is incorrect w.r.t. $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ and $\mathtt{l}\sigma$ is an incorrect symptom for $\mathtt{p}$.
Let $(\mathtt{p}, \mathtt{l}\sigma)$ be an output of the procedure. This implies that (i) $\mathtt{p} \rightharpoonup^{\sigma}_{r_u} \mathtt{error}$ for some $r \equiv (\mathtt{l} \rightharpoonup \mathtt{error} \mid \mathtt{X}_1 \,\mathtt{in}\, \mathtt{rexp}_1, \ldots, \mathtt{X}_{\mathtt{n}} \,\mathtt{in}\, \mathtt{rexp}_{\mathtt{n}}, \mathtt{s}_1 = \mathtt{t}_1, \ldots, \mathtt{s}_{\mathtt{m}} = \mathtt{t}_{\mathtt{m}}) \in I_{\mathtt{N}}$ (see line 4); (ii) for each $i = 1, \ldots, n$, $(\mathtt{X}_{\mathtt{i}}\sigma \in \mathcal{L}(\mathtt{rexp}_{\mathtt{i}}), i = 1, \ldots, n)$ (see line 5); and (iii) for each $j = 1, \ldots, m$, $(\mathtt{s}_{\mathtt{j}} = \mathtt{t}_{\mathtt{j}})\sigma$ holds in R (see line 5). Now, by simply applying Definition 5.5.1 , we get that $\mathtt{p} \in W$ is incorrect w.r.t. $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ and $\mathtt{l}\sigma$ is an incorrect symptom for $\mathtt{p}$. □

In order to prove Proposition 5.5.9, we need the following auxiliary definitions and results.

Definition B.3.1 *Given a term $t \in \tau(\Sigma, \mathcal{V})$, the* height *of t, $height(t)$, is defined as follows.*

$$height(t) = \begin{cases} 0 & \text{if } t \equiv X \in \mathcal{V} \text{ or } t \equiv c \in \tau(\Sigma, \mathcal{V}) \\ 1 + \max\{height(t_i) | i = 1, \ldots, n\} & \text{if } t \equiv f(t_1, \ldots, t_n) \in \tau(\Sigma, \mathcal{V}) \end{cases}$$

We can lift the notion of height to substitutions in the following way.

Definition B.3.2 *Given a substitution $\sigma = \{X_1/t_1, \ldots, X_n/t_n\}$, the* height *of σ, $height(\sigma)$, is defined as follows.*

$$height(\sigma) = \max\{height(t_i) \mid i = 1, \ldots, n\}$$

Now we can prove two useful technical results.

Proposition B.3.3 *Let $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded Web specification and*

$$\mu_0(\mathtt{s}_0) \rightarrow_{r_0}^{\sigma_0} \mu_1(\mathtt{s}_1) \rightarrow_{r_1}^{\sigma_1} \mu_2(\mathtt{s}_2) \rightarrow_{r_2}^{\sigma_2} \ldots$$

be a partial rewrite sequence, where $r_i \in I_{\mathtt{M}}$, $i = 0, 1, 2, \ldots$ Then, for each $\mathtt{s}_i$, $i = 1, 2, \ldots$,

$$\mathtt{s}_i \equiv \mathtt{r}_{i-1}\sigma_{i-1}$$

where $\mathtt{r}_{i-1}$ is the right-hand side of the rule r_{i-1}.

Proof. It directly comes from Definition 5.4.9 and by the fact that $Reduce(\mathtt{r}\sigma, R) = \mathtt{r}\sigma$ when considering a bounded Web specification, since no function of R can appear in the right-hand sides of the completeness rules. □

In other words, Proposition B.3.3 states that each term occurring in a partial rewrite sequence is an instance of the right-hand side of some completeness rule.

Proposition B.3.4 *Let $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded Web specification and*

$$\mu_0(\mathtt{s}_0) \rightarrow_{r_0}^{\sigma_0} \mu_1(\mathtt{s}_1) \rightarrow_{r_1}^{\sigma_1} \mu_2(\mathtt{s}_2) \rightarrow_{r_2}^{\sigma_2} \ldots$$

be a partial rewrite sequence, where $r_i \in I_{\mathtt{M}}$, $i = 0, 1, 2, \ldots$ Then, for each $\mathtt{s}_i$, $i = 1, 2, \ldots$,

$$height(\mathtt{s}_i) \leq height(\mathtt{r}_{i-1}) + height(\sigma_{i-1})$$

where $\mathtt{r}_{i-1}$ is the right-hand side of the rule r_{i-1}.

Proof. It is a direct consequence of Proposition B.3.3, Definition B.3.1, and Definition B.3.2. □

Lemma B.3.5 *Let $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded Web specification and*

$$\mu_0(\mathtt{s}_0) \rightharpoonup_{r_0}^{\sigma_0} \mu_1(\mathtt{s}_1) \rightharpoonup_{r_1}^{\sigma_1} \mu_2(\mathtt{s}_2) \rightharpoonup_{r_2}^{\sigma_2} \dots$$

be a (possibly infinite) partial rewrite sequence, where $r_i \in I_{\mathtt{M}}$, $i = 0, 1, 2, \dots$ Then, for each $\mathtt{s}_i$, $i = 1, 2, \dots$,

$$\mathit{height}(\mathtt{s}_i) \leq \mathit{height}(\mathtt{r}_{i-1}) + \mathit{height}(\sigma_0)$$

where $\mathtt{r}_{i-1}$ is the right-hand side of rule r_{i-1}.

Proof. By contradiction, let $\mathtt{s}_{k'}$ be the first term appearing in the partial rewrite sequence such that

$$\mathit{height}(\mathtt{s}_{k'}) > \mathit{height}(\mathtt{r}_{k'-1}) + \mathit{height}(\sigma_0),$$

where $\mathtt{r}_{k'-1}$ is the right-hand side of the rule $r_{k'-1}$. Clearly, $k' > 1$, as the claim trivially holds for $k' = 1$. Moreover, for each $j = 1, \dots k' - 1$, we have

$$\mathit{height}(\mathtt{s}_j) \leq \mathit{height}(\mathtt{r}_{j-1}) + \mathit{height}(\sigma_0).$$

Now, let us focus on the partial rewrite step

$$\mathtt{s}_{k'-1} \rightharpoonup_{r_{k'-1}}^{\sigma_{k'-1}} \mathtt{s}_{k'}.$$

By Proposition B.3.3, $\mathtt{s}_{k'-1} \equiv \mathtt{r}_{k'-2}\sigma_{k'-2}$. So, we have

$$\mathit{height}(\mathtt{s}_{k'-1}) \leq \mathit{height}(\mathtt{r}_{k'-2}) + \mathit{height}(\sigma_{k'-2})$$

And, also $\mathit{height}(\sigma_{k'-2}) \leq \mathit{height}(\sigma_0)$.
Again, by Proposition B.3.3, $\mathtt{s}_{k'} \equiv \mathtt{r}_{k'-1}\sigma_{k'-1}$. And, hence,

$$\mathit{height}(\mathtt{s}_{k'}) \leq \mathit{height}(\mathtt{r}_{k'-1}) + \mathit{height}(\sigma_{k'-1}) \quad \text{(by Proposition B.3.4).}$$

Here, we distinguish two cases: since $\mathtt{s}_{k'-1} \equiv \mathtt{r}_{k'-2}\sigma_{k'-2}$, the partial rewrite step $\mathtt{s}_{k'-1} \rightharpoonup_{r_{k'-1}}^{\sigma_{k'-1}} \mathtt{s}_{k'}$ can be given on a vertex in $\mathtt{r}_{k'-1}$ (Case 1) or on a vertex belonging to a term in $\{t \mid X/t \in \sigma_{k'-2})\}$ (Case 2).

Case 1. Since the Web specification is bounded, no variables in the substructure of $\mathtt{r}_{k'-1}$ which is simulated by the left-hand side of rule $r_{k'-1}$ can be located at positions which are deeper than all the positions of the variables in the considered left-hand side. So, we must have $\mathit{height}(\sigma_{k'-1}) = \mathit{height}(\sigma_{k'-2})$. Therefore,

$$\begin{aligned} \mathit{height}(\mathtt{s}_{k'}) &\leq \mathit{height}(\mathtt{r}_{k'-1}) + \mathit{height}(\sigma_{k'-1}) \\ &= \mathit{height}(\mathtt{r}_{k'-1}) + \mathit{height}(\sigma_{k'-2}) \\ &\leq \mathit{height}(\mathtt{r}_{k'-1}) + \mathit{height}(\sigma_0) \end{aligned}$$

which is a contradiction.

Case 2. Trivially, $height(\sigma_{k'-1}) \leq height(\sigma_{k'-2})$. Hence,

$$\begin{array}{rcl} height(\mathtt{s}_{k'}) & \leq & height(\mathtt{r}_{k'-1}) + height(\sigma_{k'-1}) \\ & \leq & height(\mathtt{r}_{k'-1}) + height(\sigma_{k'-2}) \leq height(\mathtt{r}_{k'-1}) + height(\sigma_0) \end{array}$$

which also leads to a contradiction.

Thus, there cannot exist a term $\mathtt{s}_{k'}$ such that $height(\mathtt{s}_{k'}) > height(\mathtt{r}_{k'-1}) + height(\sigma_0)$ and the claim is proven. □

Proposition B.3.6 *Let W be a Web site, $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a Web specification. Then,*

$$\{\mu(\mathtt{s}) \mid \varepsilon(\mathtt{p}) \rightharpoonup^m \mu(\mathtt{s}), 0 \leq m \leq k, \varepsilon(\mathtt{p}) \in W\} = \{\mu(\mathtt{s}) \mid \langle \mu(\mathtt{s}), \mathtt{q} \rangle \in \mathcal{J}_{\mathtt{M}} \uparrow^W k\}$$

Proof.
($\supseteq$) Simple induction on the ordinal power k.

($\subseteq$) Simple induction on the length m of the rewrite sequence. □

Corollary B.3.7 *Let W be a Web site, $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a Web specification. Then,*

$$\{\mu(\mathtt{s}) \mid \varepsilon(\mathtt{p}) \rightharpoonup^* \mu(\mathtt{s}), \varepsilon(\mathtt{p}) \in W\} = \{\mu(\mathtt{s}) \mid \langle \mu(\mathtt{s}), \mathtt{q} \rangle \in lfp(\mathcal{J}_{\mathtt{M}})\}$$

Now, we are ready to prove Proposition 5.5.9.

Proposition 5.5.9 *Let $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded Web specification and W be a Web site. Then, there exists $k \in \mathbb{N}$ such that $lfp(\mathcal{J}_{\mathtt{M}}) = \mathcal{J}_{\mathtt{M}} \uparrow^W k$.*

Proof. First, let us show that each $\mu(\mathtt{s})$, such that $\langle \mu(\mathtt{s}), \mathtt{q} \rangle \in lfp(\mathcal{J}_{\mathtt{M}})$, has a bounded height.
By Corollary B.3.7, $\{\mu(\mathtt{s}) \mid \varepsilon(\mathtt{p}) \rightharpoonup^* \mu(\mathtt{s}), \varepsilon(\mathtt{p}) \in W\} = \{\mu(\mathtt{s}) \mid \langle \mu(\mathtt{s}), \mathtt{q} \rangle \in lfp(\mathcal{J}_{\mathtt{M}})\}$. Thus, for each $\mu(\mathtt{s})$, such that $\langle \mu(\mathtt{s}), \mathtt{q} \rangle \in lfp(\mathcal{J}_{\mathtt{M}})$, there exists a Web page $\varepsilon(\mathtt{p}) \in W$ such that $\varepsilon(\mathtt{p}) \rightharpoonup^* \mu(\mathtt{s})$. Here, we distinguish two cases.

Case ($\varepsilon(\mathtt{p}) \rightharpoonup^0 \mu(\mathtt{s})$)**.** Let $H_W = \max\{height(\mathtt{p}) \mid \varepsilon(\mathtt{p}) \in W)\}$. In this case, $\mu(\mathtt{s}) \equiv \varepsilon(\mathtt{p})$, hence

$$height(\mu(\mathtt{s})) = height(\varepsilon(\mathtt{p})) \leq H_W.$$

Case ($\varepsilon(\mathtt{p}) \rightharpoonup^+ \mu(\mathtt{s})$)**.** Let

$$\begin{array}{rcl} H_I & = & \max\{height(\mathtt{r}) \mid \mathtt{r} \in \mathtt{Rhs}_{\mathtt{M}}\} \\ H_S & = & \max\{height(\sigma') \mid \varepsilon(\mathtt{p}) \rightharpoonup_{r'}^{\sigma'} \mu(\mathtt{s}), \varepsilon(\mathtt{p}) \in W, r' \in I_{\mathtt{M}}\}. \end{array}$$

Now, we have $\varepsilon(\mathtt{p}) \rightharpoonup_{r_0}^{\sigma_0} \mu(\mathtt{s}_1) \rightharpoonup^* \mu(\mathtt{s})$. By Lemma B.3.5, there exists $\mathtt{r} \in \mathtt{Rhs}_{\mathtt{M}}$ such that

$$height(\mathtt{s}) \leq height(\mathtt{r}) + height(\sigma_0) \leq H_I + H_S.$$

Thus, for each $\mu(\mathtt{s})$, such that $\langle \mu(\mathtt{s}), \mathtt{q} \rangle \in \mathit{lfp}(\mathcal{J}_{\mathtt{M}})$, we get

$$\mathit{height}(\mu(\mathtt{s})) \leq \max\{H_W, H_I + H_S\}.$$

Since we have only a finite number of marked terms whose height is less than or equal to $\max\{H_W, H_I + H_S\}$, $\mathit{lfp}(\mathcal{J}_{\mathtt{M}})$ must be a finite set. Finally, since the operator $\mathcal{J}_{\mathtt{M}}$ is continuous (in particular, monotonic) and $\mathit{lfp}(\mathcal{J}_{\mathtt{M}})$ is a finite set, then there exists a natural number k such that $\mathit{lfp}(\mathcal{J}_{\mathtt{M}}) = \mathcal{J}_{\mathtt{M}} \uparrow^W k$. □

In order to prove Proposition 5.5.12, we first give some auxiliary results.

Lemma B.3.8 *Let* $\mathtt{s}_1$ *and* $\mathtt{s}_2$ *be two Web page templates in* $\tau(\mathcal{T}ext \cup \mathcal{T}ag, \mathcal{V})$*. Then,* $\mathtt{s}_1 \cong \mathtt{s}_2$ *and* $\mathtt{s}_2 \cong \mathtt{s}_1$ *iff there exists a substitution* σ

$$\mathtt{s}_2\sigma \textit{ partially matches } \mathtt{s}_1 \textit{ via some substitution } \sigma'.$$

Proof.

(⇒) Let us assume that

$$\mathtt{s}_1 \cong \mathtt{s}_2 \text{ or } \mathtt{s}_2 \cong \mathtt{s}_1.$$

Case $\mathtt{s}_1 \cong \mathtt{s}_2$**.** By Definition 5.4.3, we also have $\mathtt{s}_1 \cong \mathtt{s}_2\sigma$, for every σ. Directly, by Definition 5.4.7, there exists a substitution σ' s.t. $\mathtt{s}_2\sigma$ partially matches $\mathtt{s}_1$ via σ'.

Case $\mathtt{s}_2 \cong \mathtt{s}_1$**.** $\mathtt{s}_1$ partially matches $\mathtt{s}_2$ via σ''. By Definition 5.4.3, we also have $\mathtt{s}_2\sigma'' \cong \mathtt{s}_1$. Indeed, $\mathtt{s}_2\sigma''$ is embedded into $\mathtt{s}_1$. Therefore, $\mathtt{s}_1 \cong \mathtt{s}_2\sigma''$, and $\mathtt{s}_2\sigma''$ partially matches $\mathtt{s}_1$ via the empty substitution ϵ. So, if we take $\sigma' = \epsilon$ and $\sigma = \sigma''$, then there exists σ and σ' such that $\mathtt{s}_2\sigma$ partially matches $\mathtt{s}_1$ via σ'.

(⇐) Since there exists a substitution σ s.t. $\mathtt{s}_2\sigma$ partially matches $\mathtt{s}_1$ via some substitution σ', we have that $\mathtt{s}_1 \cong \mathtt{s}_2\sigma$. Here we can distinguish two cases: (1) there is a minimal, injective simulation of $\mathtt{s}_1$ in a subterm of $\mathtt{s}_2$ w.r.t. $\sim$ and hence $\mathtt{s}_1 \cong \mathtt{s}_2$; (2) there is a minimal, injective simulation of $\mathtt{s}_1$ in $\mathtt{s}_2\sigma$ w.r.t. $\sim$, which also involves some terms occurring in σ. In this case, we have that, for some $X/t \in \sigma$, t is simulated by some subterm of $\mathtt{s}_1$. It is not difficult to see that, by Definition 5.4.2 and Definition 5.4.3, there always exists a minimal, injective simulation between a variable and an arbitrary term. Exploiting this fact, we can state that each variable in $\mathtt{s}_2$, which is replaced by some term in $\mathtt{s}_2\sigma$, simulates the corresponding subterm in $\mathtt{s}_1$, thus $\mathtt{s}_2 \cong \mathtt{s}_1$.

□

Lemma B.3.9 *Let $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded* Web specification and $r \equiv (\mathtt{l} \rightharpoonup \mathtt{r}\langle \mathtt{q} \rangle) \in I_{\mathtt{M}}$ be a function-dependent rule w.r.t. R. Let $\mu_1(\mathtt{s}_1)$ and $\mu_2(\mathtt{s}_2)$ be two marked Web page templates. If $\mu_1(\mathtt{s}_1) \rightharpoonup_r^{\sigma} \mu_2(\mathtt{s}_2)$, then there do not exist a Web page template $\mu_3(\mathtt{s}_3)$ and substitution σ' s.t. $\mu_2(\mathtt{s}_2) \rightharpoonup_{r'}^{\sigma'} \mu_3(\mathtt{s}_3)$, with $r' \in I_{\mathtt{M}}$.*

Proof. Let us consider $\mu_1(\mathtt{s}_1) \rightharpoonup_r^{\sigma} \mu_2(\mathtt{s}_2)$. Then, by Definition 5.4.9,

$$\mathtt{s}_2 = \mathit{Reduce}(\mathtt{r}\sigma, R).$$

We consider two cases.

Case $\mathtt{s}_2 \in \tau(\Sigma_R, \mathcal{V})$**.** $\mathtt{s}_2 = \mathit{Reduce}(\mathtt{r}\sigma, R)$ is just an irreducible form computed by the canonical TRS R. No partial rewriting steps can be applied to $\mathtt{s}_2$, because (i) $\{\mathtt{f} \mid \mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \rightharpoonup \mu(\mathtt{r}) \in \mathtt{I}_m\} \cap \Sigma_R = \emptyset$, since $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ is bounded*; (ii) the function calls cannot be applied to arguments containing symbols in $\{\mathtt{f} \mid \mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \rightharpoonup \mu(\mathtt{r}) \in \mathtt{I}_m\}$.

Case $\mathtt{s}_2 \in \tau(\mathcal{T}ext \cup \mathcal{T}ag \cup \Sigma_R, \mathcal{V}) \setminus \tau(\Sigma_R, \mathcal{V})$**.** $\mathtt{s}_2 = \mathit{Reduce}(\mathtt{r}\sigma, R) = \mathtt{r}^*\sigma^*$, where $\mathtt{r}^*$ is the right-hand side of the function-independent version of r w.r.t. R, and σ^* is a suitable substitution. Since $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ is bounded*, condition 2 of Definition 5.5.11 holds. Thus, by applying Lemma B.3.8, we can ensure that the subterms that occur at positions coming from $\mathtt{r}^*$ cannot be partially rewritten in a subsequent step. Moreover, for each $X/t \in \sigma^*$, t does not contain any subterm which can be partially rewritten, because (i) $\{\mathtt{f} \mid \mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \rightharpoonup \mu(\mathtt{r}) \in \mathtt{I}_m\} \cap \Sigma_R = \emptyset$, since $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ is bounded*; (ii) the function calls are never applied to arguments containing symbols in $\{\mathtt{f} \mid \mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \rightharpoonup \mu(\mathtt{r}) \in \mathtt{I}_m\}$. Therefore there are no subterms in $\mathtt{s}_2 = \mathtt{r}^*\sigma^*$ which can be partially rewritten any longer.

□

Lemma B.3.10 *Let $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded* Web specification and*

$$\mu_0(\mathtt{s}_0) \rightharpoonup_{r_0}^{\sigma_0} \mu_1(\mathtt{s}_1) \rightharpoonup_{r_1}^{\sigma_1} \mu_2(\mathtt{s}_2) \rightharpoonup_{r_2}^{\sigma_2} \ldots$$

be a (possibly infinite) partial rewrite sequence, where $r_i \in I_{\mathtt{M}}$, $i = 0, 1, 2, \ldots$ Then, there exists $k \in \mathbb{N}$ s.t. for each $\mathtt{s}_i$, $i = 0, 1, 2, \ldots$,

$$\mathit{height}(\mathtt{s}_i) \leq k.$$

Proof. Let us consider the partial rewrite sequence

$$\mu_0(\mathtt{s}_0) \rightharpoonup_{r_0}^{\sigma_0} \mu_1(\mathtt{s}_1) \rightharpoonup_{r_1}^{\sigma_1} \mu_2(\mathtt{s}_2) \rightharpoonup_{r_2}^{\sigma_2} \ldots$$

We distinguish two cases.

Case 1. First consider that the partial rewrite sequence only contains function-independent rules. Since, $(I_{\mathtt{N}} \uplus \{r \in I_{\mathtt{M}} \mid \text{ r is function-independent w.r.t. R}\}, R)$ is bounded, we can simply apply Lemma B.3.5. So,

$$height(\mathtt{s}_i) \leq height(\mathtt{r}_{i-1}) + height(\sigma_0)$$

where $\mathtt{r}_{i-1}$ is the right-hand side of rule r_{i-1}, $i = 1, 2, \ldots$

Let $k' = \max\{height(\mathtt{r}) \mid \mathtt{r} \in \mathtt{Rhs}_{\mathtt{M}}\}$. By taking $k = \max\{height(\mathtt{s}_0), k' + height(\sigma_0)\}$, we prove the claim.

Case 2. Now, assume that function-dependent completeness rules are also used in the sequence. Let us suppose that $\mu_j(\mathtt{s}_j) \rightharpoonup_{r_j}^{\sigma_j} \mu_{j+1}(\mathtt{s}_{j+1})$ is the first partial rewrite step of the sequence in which a function-dependent rule r_i occurs. By Lemma B.3.9, $\mu_{j+1}(\mathtt{s}_{j+1})$ cannot be partially rewritten any longer, therefore the sequence must be finite. So, by inspecting the heights of every term appearing in the sequence, we can find the one with the maximum height h. Finally, it is enough to choose $k = h$ and the claim in proven.

□

Proposition 5.5.12 *Let $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded* Web specification and W be a Web site. Then, there exists $k \in \mathbb{N}$ such that $lfp(\mathcal{J}_{\mathtt{M}}) = \mathcal{J}_{\mathtt{M}} \uparrow^W k$.* **Proof.** Similarly to the proof of Proposition 5.5.9, by Corollary B.3.7, $\{\mu(\mathtt{s}) \mid \varepsilon(\mathtt{p}) \rightharpoonup^* \mu(\mathtt{s}), \varepsilon(\mathtt{p}) \in W\} = \{\mu(\mathtt{s}) \mid \langle \mu(\mathtt{s}), \mathtt{q}\rangle \in lfp(\mathcal{J}_{\mathtt{M}})\}$.
Thus, for each $\mu(\mathtt{s})$, such that $\langle \mu(\mathtt{s}), \mathtt{q}\rangle \in lfp(\mathcal{J}_{\mathtt{M}})$, there exists a partial rewrite sequence $\varepsilon(\mathtt{p}) \rightharpoonup^* \mu(\mathtt{s})$, where $\varepsilon(\mathtt{p}) \in W$. As $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ is bounded*, every marked term appearing in the partial rewrite sequence $\varepsilon(\mathtt{p}) \rightharpoonup^* \mu(\mathtt{s})$ has a bounded height by Lemma B.3.10 and hence $lfp(\mathcal{J}_{\mathtt{M}})$ is a finite set.
Finally, since the operator $\mathcal{J}_{\mathtt{M}}$ is continuous (in particular, monotonic) and the set $lfp(\mathcal{J}_{\mathtt{M}})$ is finite, then there exists a natural number k such that $lfp(\mathcal{J}_{\mathtt{M}}) = \mathcal{J}_{\mathtt{M}} \uparrow^W k$. □

Proposition 5.5.21 *Let W be a Web site, $(I_{\mathtt{N}} \uplus I_{\mathtt{M}}, R)$ be a bounded* Web specification, and $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ be the set of requirements for W w.r.t. $I_{\mathtt{M}}$. Then, the procedure* Completeness-errors*$(W, I_{\mathtt{M}}, R)$ terminates. Moreover, for each error message regarding term $\mathtt{e}$, which is returned by the procedure, there exists $\langle \mu(\mathtt{e}), \mathtt{q}\rangle \in \mathtt{Req}_{\mathtt{M},\mathtt{W}}$, with $\mathtt{q} \in \{\mathtt{A}, \mathtt{E}\}$, which is not satisfied in W.*

Proof. First of all, let us prove the termination of procedure Completeness-errors$(W, I_{\mathtt{M}}, R)$. Line 2 computes the sets of requirements $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$. Since W is a bounded* Web specification, by Proposition 5.5.12, $lfp(\mathcal{J}_{\mathtt{M}})$ is a finite set which is computed in finite time. Therefore, $\mathtt{Req}_{\mathtt{M},\mathtt{W}} \leftarrow lfp(\mathcal{J}_{\mathtt{M}}) \setminus W$ is also computed in finite time.
The loop within lines 3–18 is executed $|\mathtt{Req}_{\mathtt{M},\mathtt{W}}|$ times. It simply takes every single requirement belonging to $\mathtt{Req}_{\mathtt{M},\mathtt{W}}$ and analyzes it in order to discover whether it is satisfied.

The assignment in line 4 computes the test set $\texttt{TEST}_{\langle \mu(\texttt{e}),\texttt{q}\rangle}$ which terminates by Definition 5.5.15: actually, it is just an application of the simulation algorithm [66] to a finite set of (marked) terms, i.e., the Web site W.
The emptiness test in lines 5–7 trivially terminates as well as the **case** statement in lines 8–17, which includes some checks based on the simulation algorithm [66].
Now, let us prove the partial correctness of the algorithm: that is, if an error message regarding $\texttt{e}$ (incompleteness symptom) is returned by the procedure, then $\langle \mu(\texttt{e}), \texttt{q}\rangle$, where $\texttt{q} \in \{\texttt{A}, \texttt{E}\}$, is not satisfied in W.
If an error message regarding $\texttt{e}$ (incompleteness symptom) is returned by the procedure, then it refers to a requirement $\langle \mu(\texttt{e}), \texttt{q}\rangle$. Let us consider the iteration of the loop in lines 3–18 which checks $\langle \mu(\texttt{e}), \texttt{q}\rangle$. Note that an error message can be risen in three cases. Whenever

1. $\texttt{TEST}_{\langle \mu(\texttt{e}),\texttt{q}\rangle} = \emptyset$;

2. there exists $\texttt{p} \equiv (V, E, r, \mathit{label}) \in \texttt{TEST}_{\langle \mu(\texttt{e}),\texttt{A}\rangle}$ s.t. $\texttt{e} \ncong \texttt{p}_{|v}$, with $v \in V$;

3. for each $\texttt{p} \equiv (V, E, r, \mathit{label}) \in \texttt{TEST}_{\langle \mu(\texttt{e}),\texttt{E}\rangle}$, $\texttt{e} \ncong \texttt{p}_{|v}$, with $v \in V$.

In case 1, requirement $\langle \mu(\texttt{e}), \texttt{A}\rangle$ is not satisfied in W directly by Definition 5.5.17 (see point 1), while requirement $\langle \mu(\texttt{e}), \texttt{E}\rangle$ is not satisfied in W by Definition 5.5.19 (see point 1)).
In case 2, requirement $\langle \mu(\texttt{e}), \texttt{q}\rangle$, where $\texttt{q} = \texttt{A}$, is not satisfied in W by Definition 5.5.17 (see point 2).
In case 3, requirement $\langle \mu(\texttt{e}), \texttt{q}\rangle$, where $\texttt{q} = \texttt{E}$, is not satisfied in W by Definition 5.5.19 (see point 2). □

Bibliography

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[2] Y. Ajiro and K. Ueda. Kima — an Automated Error Correction System for Concurrent Logic Programs. In *Automated Software Engineering*, volume 19, pages 67–94, 2002.

[3] Y. Ajiro, K. Ueda, and K. Cho. Error Correcting Source Code. In *Proc. of Int'l Conf. on Principle and Practice of Constraint Programming*, pages 40–54. Springer LNCS 1520, 1998.

[4] M. Alpuente, D. Ballis, F. Correa, and M. Falaschi. A Multi Paradigm Automatic Correction Scheme. In *Proc. of 11th Int'l Workshop on Functional and (Constraint) Logic Programming*, pages 157–170, 2002.

[5] M. Alpuente, D. Ballis, F. Correa, and M. Falaschi. The System for Automatic Correction NOBUG. Technical report, UPV, 2002. Available at URL: `http://www.dsic.upv.es/users/elp/papers.html`.

[6] M. Alpuente, D. Ballis, F. Correa, and M. Falaschi. Correction of Functional Logic Programs. In PierPaolo Degano, editor, *12th European Symposium on Programming, ESOP 2003*, volume 2618 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2003.

[7] M. Alpuente, D. Ballis, S. Escobar, M. Falaschi, and S. Lucas. Abstract Correction of Functional Programs. In G. Vidal, editor, *Proc. of the 12th Int'l Workshop on Functional and (Constraint) Logic Programming (WFLP'03)*, volume 86(4) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[8] M. Alpuente, D. Ballis, S. Escobar, M. Falaschi, and S. Lucas. Abstract Correction of `OBJ`-like Programs. In F. Buccafurri, editor, *Proc. 2003 Joint Conf. on Declarative Programming (AGP'03)*, pages 422–433, 2003.

[9] M. Alpuente, D. Ballis, and M. Falaschi. Automated Verification of Web Sites Using Partial Rewriting. In *1st Int'l Symposium on Leveraging Applications of Formal Methods (ISoLA'04)*, pages 81—88, 2004.

[10] M. Alpuente, D. Ballis, and M. Falaschi. A Rewriting-based Framework for Web Sites Verification. In *Proc. of 1st Int'l Workshop on Ruled-Based Programming (RULE'04)*, volume 124(1). ENTCS, Elsevier, 2004.

[11] M. Alpuente, D. Ballis, and M. Falaschi. VERDI: An Automated Tool for Web Sites Verification. In J. J. Alferes and J. Leite, editors, *Proc. of the 9th European Conference on Logics in Artificial Intelligence (JELIA'04)*, volume 3229 of *Lecture Notes in Computer Science*, pages 726–729. Springer, 2004.

[12] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract Diagnosis of Functional Programs. In M. Leuschel, editor, *International Workshop on Logic Based Program Development and Transformation (LOPSTR'02)*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2003.

[13] M. Alpuente, F. Correa, and M. Falaschi. Declarative Debugging of Funtional Logic Programs. In B. Gramlich and S. Lucas, editors, *Proc. of the International Workshop on Reduction Strategies in Rewriting and Programming (WRS 2001)*, volume 57 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[14] M. Alpuente, F. Correa, and M. Falaschi. Debugging Scheme of Functional Logic Programs. In M. Hanus, editor, *Proc. of International Workshop on Functional and (Constraint) Logic Programming, WFLP'01*, volume 64 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[15] M. Alpuente, F. Correa, M. Falaschi, and S. Marson. The Debugging System BUGGY. Technical report, UPV, 2001. Available at URL: `http://www.dsic.upv.es/users/elp/soft.html`.

[16] M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatisfiability for Equational Logic Programming. *Journal of Logic Programming*, 22(3):221–252, 1995.

[17] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Safe folding/unfolding with conditional narrowing. In H. Heering M. Hanus and K. Meinke, editors, *Proc. of the International Conference on Algebraic and Logic Programming, ALP'97, Southampton (England)*, pages 1–15. Springer LNCS 1298, 1997.

[18] M. Alpuente, M. Falaschi, G. Moreno, and G. Vidal. Rules + Strategies for Transforming Lazy Functional Logic Programs. *Theoretical Computer Science*, 311(1-3):479–525, 2004.

[19] S. Antoy. Definitional trees. In *Proc. of the 3rd Int'l Conference on Algebraic and Logic Programming, ALP'92*, pages 143–157. Springer LNCS 632, 1992.

[20] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. In *Proc. 21st ACM Symp. on Principles of Programming Languages, Portland*, pages 268–279, New York, 1994. ACM Press.

[21] P. Arenas and A. Gil. A debugging model for lazy functional languages. Technical Report DIA 94/6, Universidad Complutense de Madrid, April 1994.

[22] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[23] D. Ballis and J. García Vivó. A Rule-based System for Web Site Verification. In *Proc. of 1st Int'l Workshop on Automated Specification and Verification of Web Sites (WWV'05)*. ENTCS, Elsevier, 2005. To appear.

[24] I. D. Baxter, F. Ricca, and P. Tonella. Web Application Transformations based on Rewrite Rules. *Information and Software Technology*, 44(13), 2002.

[25] E. Bertino, M. Mesiti, and G. Guerrin. A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its Applications. *Information Systems*, 29(1):23–46, 2004.

[26] M. Bezem. *TeReSe, Term Rewriting Systems*, chapter Mathematical background (Appendix A). Cambridge University Press, 2003.

[27] A. Bockmayr and A. Werner. LSE Narrowing for Decreasing Conditional Term Rewrite Systems. In *Conditional Term Rewriting Systems, CTRS'94, Jerusalem*. Springer LNCS 968, 1995.

[28] H. Bostrom. Theory-Guided Induction of Logic Programs by Inference of Regular Languages, 1996.

[29] H. Bostrom and P. Idestam-Alquist. Induction of Logic Programs by Example–guided Unfolding. *Journal of Logic Programming*, 40:159–183, 1999.

[30] Henrik Bostrom. Specialization of Recursive Predicates. In *European Conference on Machine Learning (ECML'95)*, pages 92–106, 1995.

[31] F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proc. of the Int'l Conference on Logic Programming (ICLP'02)*, volume 2401 of *LNCS*. Springer-Verlag, 2002.

[32] F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. Technical report, 2002. Available at: `http://www.xcerpt.org`.

[33] P. Buneman, S. B. Davidson, G. G. Hillebrand, and D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. In *Proc. of ACM SIGMOD Int'l Conference on Management of Data (ICMD'96)*, 1996.

[34] L. Capra, W. Emmerich, A. Finkelstein, and C. Nentwich. XLINKIT: a Consistency Checking and Smart Link Generation Service. *ACM Transactions on Internet Technology*, 2(2):151–185, 2002.

[35] O. Chitil, C. Runciman, and M. Wallace. Freja, Hat and Hood - A Comparative Evaluation of Three Systems for Tracing and Debugging Lazy Functional Programs. In *Proc. of IFL 2000*, volume 2011 of *Lecture Notes in Computer Science*, pages 176–193. Springer, 2001.

[36] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. *Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP'00)*, 35(9):268–279, 2000.

[37] M. Comini, R. Gori, and G. Levi. Assertion Based Inductive Verification Methods for Logic Programs. In A. K. Seda, editor, *Proceedings of MFCSIT'2000*, volume 40 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[38] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract Diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

[39] M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In John W. Lloyd, editor, *Proceedings of the 1995 Int'l Symposium on Logic Programming*, pages 275–287. The MIT Press, 1995.

[40] A. Cortesi, A. Dovier, E. Quintarelli, and L. Tanca. Operational and Abstract Semantics of a Graphical Query Language. *Theoretical Computer Science*, 275:521–560, 2002.

[41] P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.

[42] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of Fourth ACM Symp. on Principles of Programming Languages*, pages 238–252, 1977.

[43] N. Dershowitz and D. Plaisted. Rewriting. *Handbook of Automated Reasoning*, 1:535–610, 2001.

[44] N. Dershowitz and U. Reddy. Deductive and Inductive Synthesis of Equational Programs. *Journal of Symbolic Computation*, 15:467–494, 1993.

[45] T. Despeyroux and B. Trousse. Semantic Verification of Web Sites Using Natural Semantics. In *Proc. of 6th Conference on Content-Based Multimedia Information Access (RIAO'00)*, 2000.

[46] E. Di Sciascio, F. M. Donini, M. Mongiello, and G. Piscitelli. Web Applications Design and Maintenance Using Symbolic Model Checking. In *Proc. 7th European Conf. on Software Maintenance and Reengineering*, page 63. IEEE Computer Society, 2003.

[47] S. M. Easterbrook, B. Nuseibeh, and A. Russo. Leveraging Inconsistency in Software Development. *IEEE Computer*, 33(4):24–29, 2000.

[48] E. Ellmer, W. Emmerich, A. Finkelstein, and C. Nentwich. Flexible Consistency Checking. *ACM Transaction on Software Engineering*, 12(1):28–63, 2003.

[49] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[50] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 103(1):86–113, 1993.

[51] W. Fan and L. Libkin. On XML Integrity Constraints in the Presence of DTDs. *J. ACM*, 49(3):368–406, 2002.

[52] M. Fay. First Order Unification in an Equational Theory. In *Proc of 4th Int'l Conf. on Automated Deduction*, pages 161–167, 1979.

[53] M. Fernandez, D. Florescu, A. Levy, and D. Suciu. Verifying Integrity Constraints on Web Sites. In *Proc. of Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, volume 2, pages 614–619. Morgan Kaufmann, 1999.

[54] M. F. Fernandez and D. Suciu. Optimizing Regular Path Expressions Using Graph Schemas. In *Proc. of Int'l Conference on Data Engineering (ICDE'98)*, pages 14–23, 1998.

[55] G. Ferrand. Error Diagnosis in Logic Programming, and Adaptation of E.Y.Shapiro's Method. *Journal of Logic Programming*, 4(3):177–198, 1987.

[56] C. Ferri, J. Hernández, and M.J. Ramírez. The FLIP System Homepage. 2000. Available at `http://www.dsic.upv.es/users/elp/soft.ht ml`.

[57] C. Ferri, J. Hernández, and M.J. Ramírez. Incremental Learning of Functional Logic Programs. In *Proc. FLOPS'01*, pages 233–247. LNCS 2024, 2001.

[58] A. Gill. Debugging Haskell by Observing Intermediate Data Structures. In Graham Hutton, editor, *2000 ACM SIGPLAN Haskell Workshop*, volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

[59] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. *Journal of Computer and System Sciences*, 42:363–377, 1991.

[60] J. A. Goguen and G. Malcom. *Software Engineering with OBJ*. Kluwer Academic Publishers, Boston, 2000.

[61] M. Hanus. Compiling Logic Programs with Equality. In *Proc. of 2nd Int'l Workshop on Programming Language Implementation and Logic Programming*, pages 387–401. Springer LNCS 456, 1990.

[62] M. Hanus. The Integration of Functions into Logic Programming: From Theory to Practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[63] M. Hanus and B. Josephs. A debugging Model for Functional Logic Programs. In M. Bruynooghe and J. Penjam, editors, *Proc. of 5th Int'l Symp. on Programming Language Implementation and Logic Programming*, volume 714 of *Lecture Notes in Computer Science*, pages 28–43. Springer, 1993.

[64] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In *Proc. ILPS'95 Workshop on Visions for the Future of Logic Programming*, pages 95–107, 1995.

[65] M. Hanus (ed.). Curry: An Integrated Functional Logic Language. Available at URL: `http://www-i2.informatik.rwth-aachen.de/~hanus/curry`, 1999.

[66] M. R. Henzinger, T. A. Henzinger, and P. W. Kopke. Computing Simulations on Finite and Infinite Graphs. In *IEEE Symposium on Foundations of Computer Science*, pages 453–462, 1995.

[67] J. Hernández and M.J. Ramírez. Inverse Narrowing for the Induction of Functional Logic Programs. In *Proc. Joint Conference on Declarative Programming, APPIA–GULP–PRODE '98*, pages 379–393, 1998.

[68] J. Hernández and M.J. Ramírez. A Strong Complete Schema for Inductive Functional Logic Programming. In *Proc. ILP'99*, pages 116–127. LNAI1634, 1999.

[69] S. Hölldobler. *Foundations of Equational Logic Programming.* Springer LNAI 353, 1989.

[70] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, MA, 1979.

[71] H. Hosoya and B. Pierce. Regular Expressions Pattern Matching for XML. In *Proc. of 25th ACM SIGPLAN-SIGACT Int'l Symp. POPL*, pages 67–80. ACM, 2001.

[72] H. Hussmann. Unification in Conditional-Equational Theories. In *Proc. of ALP'88*, pages 31–40. Springer LNCS 343, 1988.

[73] Imagiware Inc. Doctor HTML: Quality Assessment for the Web. Available at: `http://www.doctor-html.com/RxHTML/`.

[74] S. Kaplan. Simplifying Conditional Term Rewriting Systems: Unification, Termination and Confluence. *Journal of Symbolic Computation*, 4:295–334, 1987.

[75] C. Kirchner, Z. Qian, P. K. Singh, and J. Stuber. Xemantics: a Rewriting Calculus-Based Semantics of XSLT. Rapport de recherche A01-R-386, LORIA, 2001.

[76] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[77] J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-013, Computer Science Department, University of Bristol, 1995.

[78] S. Lucas. Context-Sensitive Rewriting Strategies. *Information and Computation*, 2002. To appear.

[79] S. Lucas. Termination of (Canonical) Context-Sensitive Rewriting. In *Proc. of RTA'02*, volume 2378 of *Lecture Notes in Computer Science*, pages 296–310. Springer, 2002.

[80] M.J. Maher. Equivalences of Logic Programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, Los Altos, Ca., 1988.

[81] M. Marchiori. Unraveling and Ultraproperties. In M. Rodríguez-Artalejo and M. Hanus, editors, *Proc. of Sixth Int'l Conf. on Algebraic and Logic Programming, ALP'96*, pages 107–121. Springer LNCS 1139, 1996.

[82] G. Moreno. *Rules and Strategies for Transforming Functional Logic Programs*. PhD thesis, Departamento de Sistemas Informáticos y Computación, Universidad Politécnica de Valencia, May 2000.

[83] J.J. Moreno-Navarro and M. Rodriguez-Artalejo. BABEL: A Functional and Logic Programming Language based on a constructor discipline and narrowing. In I. Grabowski, P. Lescanne, and W. Wechler, editors, *Proc. of the Int'l Conf. on Algebraic and Logic Programming*, pages 223–232. Springer LNCS 343, 1988.

[84] J.J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. *Journal of Logic Programming*, 12(3):191–224, 1992.

[85] S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8(3):295–318, 1991.

[86] S. Muggleton. Inductive Logic Programming: Issues, Results and the LLL Challenge. *Artificial Intelligence*, 114(1-2):283–296, 1999.

[87] S. Muggleton and L. de Raedt. Inductive Logic Programming: Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[88] L. Naish and T. Barbour. Towards a Portable Lazy Functional Declarative Debugger. *Australian Computer Science Communications*, 18(1):401–408, 1996.

[89] C. Nentwich, W. Emmerich, and A. Finkelstein. Consistency Management with Repair Actions. In *Proc. of the 25th International Conference on Software Engineering (ICSE'03)*. IEEE Computer Society, 2003.

[90] S. Nesbit. HTML Tidy: Keeping it clean, 2000. Available at `http://www.webreview.com/2000/06-16/webauthors/06-16-00-3.shtml`.

[91] H. Nilsson. A Declarative Approach to Debugging for Lazy Functional Languages. Licentiate Thesis, September 1994.

[92] H. Nilsson. Tracing Piece by Piece: Affordable Debugging for Lazy Functional Languages. In *Proceedings of the 1999 ACM SIGPLAN Int'l Conf. on Functional Programming*, pages 36–47. ACM Press, 1999.

[93] H. Nilsson. How to Look Busy While Being as Lazy as Ever. *Journal of Functional Programming*, 11(6):629–671, 2001.

[94] H. Nilsson and P. Fritzson. Algoritmic Debugging for Lazy Functional Languages. *Journal of Functional Programming*, 4(1):337–370, July 1994.

[95] J. T. O'Donell and C. V. Hall. Debugging in Applicative Languages. *Lisp and Symbolic Computation*, 1(2):113–145, 1988.

[96] The Open Group. Unix Regular Expressions. Available at: `http://www.opengroup.org/onlinepubs/7908799/xbd/re.html`.

[97] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Berlin, 1988.

[98] C. Palamidessi. Algebraic Properties of Idempotent Substitutions. In M.S. Paterson, editor, *Proc. of 17th Int'l Colloquium on Automata, Languages and Programming*, pages 386–399. Springer LNCS 443, 1990.

[99] A. Pettorossi and M. Proietti. Transformation of Logic Programs. In *Handbook of Logic in Artificial Intelligence*, volume 5, pages 697–787. Oxford University Press, 1998.

[100] B. Pope. Buddha: A Declarative Debugger for Haskell. Technical report, Dept. of Computer Science, University of Melbourne, Australia. Honours Thesis, 1998.

[101] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In *Proc. of Second IEEE Int'l Symp. on Logic Programming*, pages 138–151. IEEE, New York, 1985.

[102] E. Y. Shaphiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Massachusetts, 1982. ACM Distinguished Dissertation.

[103] Academisa Sinica Computing Centre. The Schematron: an XML Structure Validation Language using Pattern in Trees. Available at: `http://xml.ascc.net/resource/schematron/schematron.html`.

[104] J. Sparud and H. Nilsson. The Architecture of a Debugger for Lazy Functional Languages. In M. Ducassé, editor, *Proceedings of AADEBUG'95*, Saint-Malo, France, May 1995.

[105] Kazunori Ueda and Masao Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, 13(1):3–43, 1994.

[106] P. Wadler. Functional Programming: An Angry Half-Dozen. *ACM SIGPLAN Notices*, 33(2):25–30, 1998.

[107] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0, second edition, 1999. Available at: `http://www.w3.org`.

[108] World Wide Web Consortium (W3C). XML Path Language (XPath), 1999. Available at: `http://www.w3.org`.

[109] World Wide Web Consortium (W3C). Extensible HyperText Markup Language (XHTML), 2000. Available at: `http://www.w3.org`.