Jorge Manuel Neves Coelho

XML Processing in Logic Programming

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2007

Jorge Manuel Neves Coelho

XML Processing in Logic Programming

Tese submetida à Faculdade de Ciências da Universidade do Porto para obtenção do grau de Doutor em Ciência de Computadores

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
2007

To my parents Ana Maria and Fernando...

Acknowledgments

This thesis was my work over the last three and a half years. This work was not possible without the support and dedication of my supervisor, Prof. Mário Florido. His expertise and assistance made it possible to successfully finish this thesis no matter the tight deadlines imposed.

I'd like to thank to the PRODEP III for the precious scholarship that liberated me from all other duties during three years. I'd also like to thank to all other funding sources that contributed to fund several participations in conferences, summer schools and research visits namely, LIACC, ISEP and Calouste Gulbenkian Foundation.

Also, I'd like to thank to many colleagues both at DCC-FCUP and at DEI-ISEP. I will not mention names because the list is endless. You know who you are.

Finally, I thank to my family, to my wife Sónia and to my parents Ana Maria and Fernando, what I am today I owe to them.

Abstract

The main goal of this thesis is to develop a new approach to XML processing based in Logic Programming by exploring the natural tree structure of data in the Logic Programming paradigm and new forms of unification.

In particular our aim is to focus on the following issues:

- Study an alternative form of unification more appropriate to XML processing.
- Study appropriate type systems suitable for both XML and to the alternative form of unification.
- Creation of a new programming language for XML processing based on the topics above and its use in the implementation of real-world applications.

XML processing in Logic Programming is usually done by traditional list processing or alternatively by translating documents into terms. In both approaches the programmer must process the whole document instead of focusing in the parts that matter. Also there is noway to use the schemas associated with the documents for verification of programs.

The contributions of this work are:

- A new approach to XML processing based in unification and pattern matching of terms with flexible arity. This led to a new programming language (here named XCentric).
- Definition of algorithms which use types (schemas) in the unification process in order to verify the correction of the results.
- Development of a simple API based on the concepts above to enable an easy access to parts of documents in depth and translation between different representations.
- Application of the language to website verification, where the simple and expressive programs that can be defined in XCentric can be used to impose constraints on websites and correct several errors.

Résumé

L'objectif principal de cette thèse est de développer une nouvelle approche du traitement du XML, fondée sur la Programmation Logique, en explorant la structure arborescente naturelle des données dans le paradigme de la Programmation Logique et de nouvelles formes d'unification. En particulier, notre but est de nous concentrer sur les questions suivantes:

- Étudier une forme alternative d'unification plus appropriée au traitement du XML.
- Étudier des systèmes de types adaptés à la fois au XML et à la forme d'unification alternative.
- Création d'un nouveau langage de programmation pour le traitement du XML basé sur les points précédents, et son utilisation dans l'implantation d'applications concrètes.

Le traitement du XML en Programmation Logique se fait habituellement par des traitements de listes traditionnels, ou alternativement en traduisant les documents en termes. Dans ces deux approches, le programmeur doit traiter le document entier au lieu de se concentrer sur les parties importantes. En outre, il n'y a aucun moyen d'utiliser les schémas associés aux documents pour la vérification de programmes.
Les contributions de ce travail sont:

- Une nouvelle approche au traitement du XML fondée sur l'unification et le filtrage de termes d'arité flexible. Cela a conduit à un nouveau langage de programmation (appelé XCentric).
- Définition d'algorithmes qui utilisent les (schémas de) types dans le processus d'unification afin de vérifier la correction des résultats.
- Développement d'une API simple basée sur les concepts précédents, qui autorise un accès facile aux parties des documents en profondeur et la traduction entre différentes représentations.
- Application du langage à la vérification de sites web, où les programmes simples et expressifs qui peuvent être définis dans XCentric peuvent être utilisés pour imposer des contraintes aux sites Web et pour corriger certaines erreurs.

Resumo

O principal objectivo desta tese é desenvolver uma nova abordagem ao processamento de XML no contexto da Programação em Lógica. Para isso pretende-se explorar novas formas de unificação aliadas ao tratamento árvores como uma estrutura natural em Programação em Lógica.

Em particular o nosso objectivo é focar a investigação nos seguintes temas:

- Estudo de formas alternativas de unificação mais adequadas ao processamento de XML.
- Estudo de sistemas de tipos adequados tanto para o XML como para a nova unificação.
- Criação de uma nova linguagem de programação, vocacionada para o procesamento e interrogação de documentos XML, baseada nos dois pontos anteriores e a sua aplicação a cenários reais.

O processamento de XML em Programação em Lógica é normalmente feito recorrendo ao processamento de listas ou alternativamente traduzindo os documentos para termos. Em ambas as abordagens, o programador tem que processar todo o documento em vez de se concentrar nas partes que realmente lhe interessam. Adicionalmente não há forma de tirar partido da informação de tipos associada aos documentos XML. As contribuições deste trabalho são:

- Uma nova abordagem ao processamento e interrogação de documentos XML baseado em unificação e encaixe de padrões sobre termos de aridade flexível. Este ponto deu origem a uma nova linguagem de programação (aqui chamada XCentric).
- Definição de algoritmos que usam os tipos associados aos documentos XML no processo de unificação para verificar a correcção de resultados.
- Desenvolvimento de uma API simples baseda nos conceitos expostos nos pontos anteriores para traduzir entre formatos e simplificar a interrogação de documentos em profundidade.
- Aplicação da linguagem à área de verificação de conteúdos em sítios web onde, os programas simples e expressivos definidos em XCentric são usados para impôr restricções sobre os conteúdos e corrigir erros.

Contents

Abstract **7**

Résumé **9**

Resumo **11**

List of Tables **19**

List of Figures **21**

1 Introduction **23**

1.1 XML Processing 24
1.2 XML in Logic Programming 25
1.3 Flexible Arity 26
1.4 Type-based Validation 27
1.5 Applications 28
1.6 Thesis in outline 28
1.7 Contributions 29

2 XML Processing **31**

2.1 Introduction 31
2.2 XML 31
2.2.1 The History of XML 32
2.2.2 Applications of XML 32
2.2.3 Contents of an XML document 33
2.2.3.1 Elements 33
2.2.3.2 Attributes 33
2.2.3.3 Entities 33

2.2.3.4 Comments 34
2.2.3.5 Processing Instructions 34
2.2.3.6 Declarations 34
2.2.3.7 Declaration of type of document 34
2.2.3.8 XML declaration 34
2.2.3.9 CDATA Declaration 35
2.2.4 Example 35
2.3 Document Type Definition 36
2.3.1 Content of a DTD 36
2.3.1.1 Elements 37
2.3.1.2 Attributes 38
2.3.1.3 Entities 40
2.3.1.4 Notations 40
2.3.1.5 Example 41
2.3.2 Limitations 41
2.4 XML Schema 42
2.4.1 Simple Types 43
2.4.1.1 Restrictions 43
2.4.1.2 Lists 45
2.4.2 Union 45
2.4.3 Complex Types 46
2.4.3.1 Concatenation 46
2.4.3.2 Choice 46
2.4.3.3 All 47
2.4.3.4 Any 47
2.4.3.5 Attribute 48
2.4.4 Other properties 48
2.4.5 Problems with XML Schema 49
2.5 Other Schema Languages 49
2.5.1 RelaxNG 49
2.5.2 Schematron 50
2.5.3 DSD2 50
2.6 XML Processing Languages 50
2.6.1 Untyped XML processing languages 51

2.6.1.1 DOM . . . 51
2.6.1.2 SAX . . . 51
2.6.1.3 XSLT . . . 52
2.6.1.4 Xcerpt . . . 53
2.6.2 Typed XML processing languages . . . 54
2.6.2.1 XDUCE . . . 54
2.6.2.2 XQuery . . . 55
2.6.2.3 Data Model . . . 55
2.6.2.4 Expressions . . . 56
2.6.2.5 Types . . . 58
2.6.3 Other Languages . . . 59
2.7 Discussion . . . 59

3 Constraint Solving **61**
3.1 Introduction . . . 61
3.2 Constraint Logic Programming . . . 61
3.2.1 CLP(X) . . . 61
3.2.2 CLP Languages . . . 62
3.2.2.1 CLP(R) . . . 63
3.2.2.2 Prolog III . . . 63
3.2.2.3 ECL^iPS^e . . . 63
3.3 First Order Unification . . . 65
3.3.1 Unification algorithm . . . 66
3.4 Flexible Arity Unification . . . 67
3.5 Unification of Flexible Arity Terms . . . 69
3.5.1 Projection . . . 70
3.5.2 Transformation . . . 70
3.6 Solving Quadratic Sequence Equations . . . 73
3.7 Discussion . . . 78

4 Constraint Based XML Processing **79**
4.1 Introduction . . . 79
4.2 XML Processing in Prolog . . . 79
4.3 Constraint Solving in CLP(Flex) . . . 81
4.4 XML Processing in CLP(Flex) . . . 82

4.4.1 XML as Terms with Flexible Arity Symbols 82
4.4.2 Using Constraints in CLP(Flex) 82
4.5 The Unification Algorithm 83
4.5.1 Correctness 88
4.5.2 Examples 93
4.6 Discussion 98

5 XCentric: Typed Unification based XML Processing 101
5.1 Introduction 101
5.2 XML processing in XCentric 101
5.2.1 Regular Expression Types 102
5.2.2 XML Schema support 103
5.2.2.1 Basic Types 103
5.2.2.2 Occurrences of sequences 103
5.2.2.3 Orderless sequences 103
5.2.3 Examples 104
5.3 Regular Types 109
5.4 Sequence Types 111
5.5 Dynamic Typing 111
5.5.1 Types as programs 111
5.5.2 Types in the unification process 112
5.6 Static Typing 113
5.6.1 Type intersection 114
5.6.2 Typed unification for terms with flexible arity 121
5.7 Pattern Matching 128
5.7.1 Incomplete terms in depth 128
5.8 Unification vs. Pattern Matching 131
5.9 Discussion 136

6 Case-Study: Website Content Verification 139
6.1 Introduction 139
6.2 XCentric for Website Content Verification 139
6.2.1 Examples 140
6.3 Extension with type and consistency checking 146
6.3.1 Verification Framework 147

6.3.1.1 Website Constraints 147
6.3.1.2 Static verification of action rules 151
6.3.1.3 Run time consistency checking 154
6.4 Discussion 156

7 Conclusions **157**
7.1 General Conclusions 157
7.2 Declarative XML Processing 157
7.3 Sequence Types 158
7.4 Further Work 159
7.4.1 Implementation 159
7.4.2 Relation with other forms of unification 160
7.4.3 New Type Language 160
7.4.4 Applications 160
7.5 Summary 161

Bibliography **162**

List of Tables

2.1 Pre-defined Entities 40
2.2 Simple Types 44
2.3 Facets of Simple Types 44

5.1 Result of tests with 0,5 KB file for addressbook transformation 132
5.2 Result of tests with 5 KB file for addressbook transformation 133
5.3 Result of tests with 15 KB file for addressbook transformation 133
5.4 Result of tests with 0,5 KB file for person transformation 136
5.5 Result of tests with 5 KB file for person transformation 136
5.6 Result of tests with 15 KB file for person transformation 136

List of Figures

2.1 Tree for an XML document 32

2.2 XML document 35

2.3 Example of a DTD 41

3.1 Rules for unification 66

3.2 Successful derivations of unifiers for $\{f(x, b, Y, f(X)) \stackrel{?}{=} f(a, X, f(b, Y)))\}$. . . 72

3.3 Automata generated for problem $f(X, a) \doteq f(a, X)$ 76

4.1 Transformation rules 87

5.1 Intersection of sequences of types 117

5.2 Success rules 124

5.3 Eliminate and split rules 125

5.4 Algorithm for sequence matching 128

6.1 Teacher's home page constraints 152

Chapter 1

Introduction

Since XML [W3C04c] became the standard for document data interchange and storage, building tools to deal with XML data at the programming language level became a relevant topic. Libraries for XML processing like SAX [Meg04] and DOM [W3C04a] are available for many programming languages and XML processing specific programming languages [MS99, CS00, May01, HP00, KK03, BS02a, W3C04e] were and are being developed. The aim of this thesis is the development of the basis of a new declarative programming paradigm for XML processing based on recent work on equality in an algebra of trees where each node has an arbitrary number of children [Kut02c].

The motivation for our work is the observation that XML data is usually described by trees and Logic Programming deals with trees naturally as terms. We strongly believe that XML processing can be performed naturally in Logic Programming. Our goal is to improve XML processing in the Logic Programming paradigm by exploring the link between the tree representation and unification.

With this goal in mind we studied alternative forms of unification, more appropriate for dealing with XML data than the traditional Robinson unification used in Prolog. Flexible arity term unification, due to its compact representation for terms is the core for our new language. Types, based in schemas for XML documents [W3C04b, W3C04f] are used to improve XML processing.

Regular types [Zob87] are a well studied type representation for logic programming that describe sets of terms, and are used here to represent schema types. In global terms, our main aims are to define and implement a new logic programming language for XML processing based in flexible arity terms unification and regular types.

To achieve our goal we make use of techniques arising from the following topics:

- Flexible arity terms: these particular terms are a compact representation for XML documents.

- Constraint Solving: based in the unification of flexible arity terms as the core of the

language.

- Regular Types: as a representation of XML DTDs and Schemas.

1.1 XML Processing

Having its first application in the publishing industry, XML rapidly expanded its boundaries to other areas. XML is widely used in data transfer and exchange [HLS04, SBm04, oJ04], document archives [oC02] and application integration [ebX04]. A lot of information is stored in XML documents. Every day new applications appear that store their data in XML files, from word processors or spread sheets applications [Cor02] to interface development applications [Moz06].

Due to the increasing popularity of XML, APIs and languages specialized in processing XML documents appeared. The most popular APIs are DOM [W3C04a] and SAX [Meg04] which can be used in languages like Java, Python or C++. In the specialized XML processing languages domain one can find two types of languages: typed and untyped. One example of an untyped language is XSLT [W3C01b] which is also one of the oldest proposals and probably still the most popular way to process XML data. More recently several new languages specialized in XML processing appeared, for example XDuce [HP00], CDuce [BCF03], Xtatic [GLPS05] and XQuery [W3C04e]. These new languages explore the use of types by translating DTDs and XML Schemas to a representation named Regular Expression Types. These types are then used at compile time for validation of programs.

The already mentioned, XQuery is inspired in [HP00] and is supposed to gather all of the successful research carried in the last years in the domain of XML processing languages. Some implementations are now available [Sof04, FSC+03, Rys01]. Although a promising language XQuery still has some open problems, such as:

- XQuery needs good tools for debugging, it is complex and for some simple operations, a graphical interface would be needed;

- The XQuery type system is complex [KCK+03]. It uses tree automata and only addresses a subset of XML Schema. For example properties like the minimum and maximum numbers of occurrences, keys and ids are not supported;

- Memoizing and caching techniques are needed and not implemented.

The reason for these problems is mainly the fact that XQuery is a new language. One advantage of adding XML processing to a well established language is that one benefits from all the research already done for that language. Considering all these issues we can, for sure, state that there is still room for new approaches to XML processing such as ours.

1.2 XML in Logic Programming

Previous work on the representation of XML using a term language inspired in logic programming was presented in [GH01], [BE00], [Bol00] and [BS02b]. In [BS02b] it was defined an untyped rule-based transformation language based on logic programming, with its own term syntax directed for querying graph-structured data. The kind of terms used to represent XML in our work follows the representation presented in [Bol00].

In [CF03] we introduced type inference for XML processing in Pure Prolog. As far as we know, our work was the first one dealing with statically type validation in the context of logic programming for XML processing.

XML processing in the most popular logic programming compilers, is mainly done by list processing. One of the most complete libraries for dealing with XML data comes with SWI-Prolog [Wie06]. We use an example to illustrate XML processing in SWI-Prolog: Let's suppose we have an XML document with a catalog of books like the following one:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <book number="1">
    <author>Donald Knuth</author>
    <name>Art of Computer Programming</name>
    <price>140</price>
    <year>1998</year>
  </book>
  ...
  <book number="500">
    <author>Simon Thompson</author>
    <name>Haskell: The Craft of Functional Programming (2nd Edition)</name>
    <price>41</price>
    <year>1999</year>
  </book>
</catalog>
```

To get all the books with two or more authors using SWI-Prolog [Wie06] we need the following code:

```
pbib([element(_,_,L)]):-
      pbib2(L).
pbib2([]).

pbib2([element('book',_,Cont)|Books]):-
      authors(Cont),!,pbib2(Books).
pbib2([_|Books]):-
      pbib2(Books).

authors([element('author',_,_),element('author',_,_)|R]):-
      write_name(R).
```

```
write_name([element('name',_,[N])]):-
    write(N),nl.
write_name([_|R]):-
    write_name(R).
```

In SWI-Prolog, XML documents are translated to lists of terms where each element is stored inside a term with root "element" and where the content is, the element name, the list of its attributes (if any) and its content. Processing this data is done using traditional list processing.

1.3 Flexible Arity

Languages with flexible arity symbols have been used in various areas: Xcerpt [BS02a] is a query and transformation language for XML which used terms with flexible arity function symbols as an abstraction of XML documents. It used a special notion of term (called *query terms*) as patterns specifying selection of XML terms much like Prolog goal atoms. The underlying mechanism of the query process was *simulation unification* [BS02b], used for solving inequations of the form $q \leq t$ where q is a query term and t a term representing XML data. This framework is technically quite different from ours, being more directed to query languages and less to XML processing. The Knowledge Interchange Format KIF [GF92] and the tool Ontolingua [FFR97] extend first order logic with variable arity function symbols and apply it to knowledge management. Feature terms [Smo92] can also be used to denote terms with flexible arity and have been used in logic programming, unification grammars and knowledge representation. Unification for flexible terms has, as particular instances, previous work on word unification [Jaf90, Sch93], equations over lists of atoms with a concatenation operator [Col90] and equations over free semigroups [Mak77]. Kutsia [Kut02c] defined a procedure for unification with sequence variables and flexible arity symbols applied to an extension of *Mathematica* for mathematical proving [BDJ$^+$00]. From all the previous frameworks we followed the work of Kutsia because it is the one that fits better in our initial goal, which was to define a highly declarative language for XML processing based on an extension of standard unification to denote the same objects denoted by XML: trees with an arbitrary number of leafs.

In this thesis we define and implement a new programming language [CF04, CF07b, CF07c] for dealing with the unification of flexible arity terms. As an example, the SWI-Prolog code described before can be simply done with the following query in our language (XML file is in variable *BibDoc* and =*= is a new operator for dealing with the new unification of terms with functors of arbitrary arity):

```
?- catalog(_,book(name(N),author(_),author(_),_),_) =*= BibDoc.
```

Functional languages for XML processing (such as XDuce [HP00] and CDuce [BCF03]) relied on the notion of trees with an arbitrary number of leaf nodes. However they dealt

with pattern matching, not unification. Regular expression patterns are often ambiguous and thus presume a fixed matching strategy for making the matching deterministic. In contrast, our work just leaves ambiguous matching non-deterministic and examines every possible matching case by backtracking.

1.4 Type-based Validation

Since XML documents are generally trees and schema languages describe sets of trees, types in modern XML processing languages are usually translated to *tree automata* [Tha73]. These kind of automata provide a theoretical model with nice properties but can easily lead to exponential blow-ups. Subtyping is an essential feature of these type systems, since it provides a way to know if one schema is a subschema of another schema. Thus, we need to know if a tree automata contains another one (if the set of trees one describes is a subset of the trees the other describes). Subtyping is a very inefficient task [HVP00]: it is shown that in general the problem of subtyping is EXPTIME-complete.

In a recent work [DL03], it was presented a new class of tree automata, the Sheaves Automata (SA). These automata use Presburger's Constraints [DL03] to recognize documents valid with respect to a schema with interleaving (a feature not available in DTDs but present in XML Schema and RelaxNG [Spe01]: it consists of bags of elements without any specific order). Also here an important problem is subtyping: the class of languages accepted by non-deterministic SA is not closed under complementation (an essential feature for subtyping computation) and some recursive schemas can only be captured by nondeterministic SA.

Tree automata recognize languages defined by regular tree grammars that correspond to regular types [DZ92], a framework widely used as type language for logic programming [YS90, FD92, DZ92]. Here we follow the regular type approach since it is natural in the Logic Programming paradigm and simpler than the tree automata approach.

In [CF03] we presented a framework for type inference for XML processing based in Regular Types where we added compile-time checking applied to Pure Prolog (with classical Robinson unification).

Types are used in several languages for XML processing. The system presented in [WD03] uses types together with a novel form of unification (simulation unification [BS02b]) for a small subset of the logic XML query language Xcerpt [BS02a]. The previous work extended tree automata to deal directly with terms with functors of arbitrary arity. In our work we do not need such extension because the arbitrary number of arguments in our framework is denoted by the notion of *sequence* that is expressible using two symbols of fixed arity 2 and 0.

1.5 Applications

Website verification consists in verifying the content of data in websites. The programmer describes rules and the web pages in the website are verified against the rules. Since our language provides a way of writing short rules about XML data content it can be successfully applied to Website verification. To show this claim, we built VeriFlog [CF06b, CF07a], a tool based in our language but specialized in website content verification. Along with the ability to write rules and detect errors our tool can be used to infer new data from the website and automatically repair the web pages that don't obey to a given constraint . We accomplish these tasks by defining the notion of *Sets of Web Constraints* and introducing type declarations, domain specific compile time and run time checking and consistency verification of rules.

Website verification was addressed in several previous works. In [ABFR05a] the authors present a rewriting-based framework that uses simulation [HHK95] in a query language. In [ABFR05b], the authors present a semi-automatic methodology for repairing faulty websites by applying a set of concepts from Integrity Constraint [MT99] thus, very different from our type-based approach which uses static checking and run time validation. In [Des04] the author proposed the use of a simple pattern-matching-based language and its translation to Prolog as a framework for website verification. Our work is also based in Prolog but our syntax smoothly integrates with it, thus our framework inherits all the power of Prolog. We also provide a richer interface to semistructured data. In [vHvdM99] logic was proposed as the rule language for semantic verification. There the authors provide a mean for introducing rules in a graphical format. In contrast, our work provides a programming language and thus a richer and more flexible way to write rules. In [Kut05] the author proposed an algorithm for website verification similar to [BS02b] in expressiveness but based on a different theoretical approach. The idea was to extend sequence and non-sequence variable pattern matching with context variables, allowing a more flexible way to process semistructured data, but the author doesn't provide an implementation.

1.6 Thesis in outline

The rest of the thesis is organized as follows:

Chapter 2: Introduces the background related with XML by describing its constructs, the associated type schemas, APIs and languages specialized in XML processing.

Chapter 3: Introduces the background related with Constraint Solving and presents unification as a particular case of CLP(X). Flexible arity term unification is also introduced here.

Chapter 4: Introduces CLP(FLex), a constraint logic programing language in the domain

of terms of flexible arity which is an extension to Prolog and a general framework for XML processing based in constraint resolution. CLP(Flex) uses flexible arity term unification and the declarativeness and expressivity of this approach is illustrated by several examples.

Chapter 5: Introduces XCentric, a typed logic programming language for XML processing with flexible arity unification. Starts by presenting examples of the benefits of types both for validation and improved querying. Here we describe how to extend regular types [Zob90] to *regular expression types* which in turn describe schemas for XML. In this chapter we present two approaches for type checking: Dynamic Typing where types are translated to programs and used at run time, and Static Typing where we define algorithms for intersection of type sequences and extend the unification algorithm presented in chapter 4 to use types. We also introduce a pattern matching algorithm which is more efficient and can be used in the cases where unification is not necessary.

Chapter 6: Here we present a case-study of an application of XCentric. We implement the VeriFlog tool for verifying website content, impose constraints, infer data from web pages and automatically repair faulty websites.

Chapter 7: Here we present final remarks and future work.

1.7 Contributions

The work presented in this thesis resulted in the following contributions:

- In [CF04] and [CF05], CLP(Flex) and its application to XML processing was presented.
- In [CF06a] the algorithms for the typed version of CLP(Flex), named XCentric were presented.
- In [CF07b] and [CF07c] we presented XCentric.
- In [CF06b, CF07a] we presented VeriFLog, the application for website verification and data inference based on XCentric.
- In chapter 3, section 3.6 we present a complete and terminating procedure for the unification with sequence variables and flexible arity functions, for the case where sequence variables do not occur more than twice in the unification problem. The work presented in this section was made in collaboration with Temur Kutsia.

Chapter 2

XML Processing

2.1 Introduction

In this chapter we give a brief overview of several XML processing mechanism. We start with the description of XML, its history and constructs, then we describe several type languages for XML. We present some XML processing languages and describe their capabilities. The chapter ends with a discussion about the topics introduced.

2.2 XML

XML (eXtensible Markup Language) [W3C04d] is a metalanguage developed by W3C (World Wide Web Consortium) [W3C07] to describe markup languages. Markup languages are a useful tool to organize data under a structured format. Since there are no fixed elements, XML allows persons or organizations to create their own markup language for their specific domain. Some examples of this languages are, SMIL [W3C05a], MathML [W3C01a] and VML [W3C98].

XML documents are organized by markup which group data. Let us present an example of an XML document for an address book:

```
<addressbook>
        <record>
                <name>John</name>
                <address>New York</address>
                <email>john.ny@mailserver.com</email>
        </record>
        <record>
                <name>Sofia</name>
                <address>Rio de Janeiro</address>
                <phone>123456789</phone>
                <email>sofia.brasil@mail.br</email>
        </record>
```

```
</addressbook>
```

One important property of this kind of structure is that, in general, it can be seen as a tree. For the former example, the corresponding tree is presented in figure 2.1.

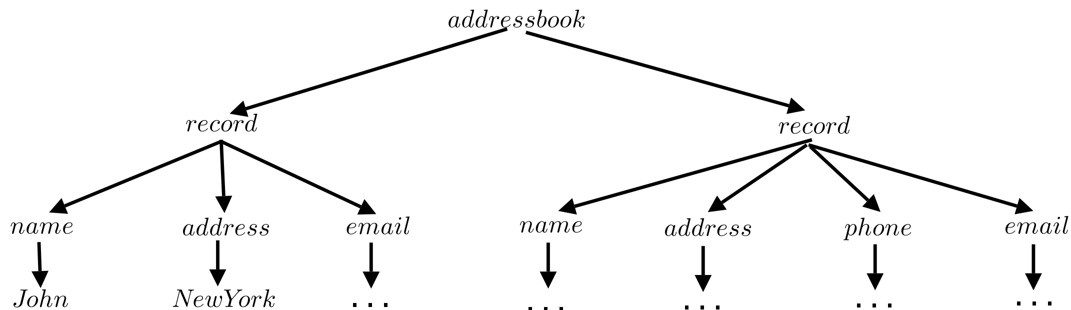

Figure 2.1: Tree for an XML document

2.2.1 The History of XML

The goal of markup languages is to provide notation to make explicit a given interpretation of a given text. For example, spaces, words, paragraphs and pontuation are all notation that allow a human, with knowledge about the language, to make an interpretation of a text. Since long ago markup is used in order to create an explicit structure for a document. SGML (Standard Generalized Markup Language) [W3C95] is an independent standard for the representation of text in electronic format, available since 1986. Since its complexity is high, its success was limited. With SGML as base, HTML (Hyper Text Markup Language) [W3C06a] was created. The goal of HTML was to be used as a markup language for the World Wide Web. In opposite to SGML, HTML was successful, so successful indeed that, although there is one group responsible for its development [W3C07], the developers of *browsers* (like Mozilla and MSIE) ended up adding non-standard extensions with the objective of improving HTML with their own vision and for commercial profit.

The raising pressure in order to simplify the too complex SGML and enhance HTML resulted in XML.

2.2.2 Applications of XML

XML offers an independent and powerful way of data interchange. We can divide applications in two main areas:

Information interchange between applications: data interchange between different Relational Database Systems, where XML is used to organize data from fields and tables

in an intermediate format and interchange between applications in the Web [W3C06b].

Document publication: since XML documents have only structured data stored in a format independent from publication, it is possible, using specific tools, to translate an XML documents to specific formats used for publication in different platforms. For example, with a browser we can bind an XML document to an XSLT [W3C99a] style sheet and produce the final XHTML [W3C06a].

2.2.3 Contents of an XML document

An XML document is made by specific markup which organize the data in a predefined way. In the next sessions we introduce the different kinds of markup.

2.2.3.1 Elements

Elements are the most common notation present in XML documents. Usually elements are refereed as *tags* and enclose data between two tags, one opening and one closing, for example: < element_name > . . . < \ element_name >. They can also occur as one single tag, for example, <element_name /> where all the data is described inside the tag. Next example shows the use of elements *name* and *img*:

```
<name> Jorge Coelho </name>
<img href=”/images/img.png” />
```

2.2.3.2 Attributes

Attributes are pairs (attribute,value) which occur inside the opening tag, for example:

```
<phone group=”private”> 123456789 </phone>
```

Attributes allow additional information inside a tag. In the former example, the tag for element *phone* contains an attribute *group* which has a value accordingly to the group to which the phone belongs to.

2.2.3.3 Entities

The idea of entities is to identify some data by a name. Whenever we want to use the data referenced by the entity it is enough to use the entity's name. It is possible to refer to a given piece of text by using an entity (used as an abbreviation) which is useful in the case that the text is repeated several times along the document. We can also use an entity to make reference to an external object like, for example a binary file.

2.2.3.4 Comments

Comments can be added to XML documents between <!- - and - ->. Comments can include any characters except –. One example is:

```
<!-- I'm a comment -->
```

2.2.3.5 Processing Instructions

Processing instructions is a special notation used to enclose content that should be processed by an specific application. Processing instructions are described between <? and ?> and start by the name of the application that should be used to process the content followed by that content. For example:

```
<? xml version="1.0" ?>
```

2.2.3.6 Declarations

Declarations can be added to XML documents enclosed between <! and > and can be processed by a pre-processor of the document.

2.2.3.7 Declaration of type of document

This is a specific declaration which declares the root element along with a set of rules that describe how the structure of the document should be. These rules can be described inside the declaration or in an external file. For example:

```
<!DOCTYPE addressbook SYSTEM "addressbook.dtd">
```

In this example it is said that the root element is *addressbook* and the set of rules is given in file *addressbook.dtd*.

2.2.3.8 XML declaration

An XML document can have at the beginning a special declaration which describes some information about its content. The possible information is:

- XML version.
- Character encoding.
- Declaration if external data should be or not processed.

For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

Here we are describing an XML document version 1.0. Character encoding is UTF-8 [IET03]. The *standalone* declaration says that no external data should be processed. In case we wanted to have an external schema for validation of the XML that should be checked, the last declaration should be set to “no”.

2.2.3.9 CDATA Declaration

CDATA declarations allows the inclusion of many characters, including < and >. For example:

```
<![CDATA [ (A>B)?A:B]]>
```

In this *CDATA* declaration a little piece of *C* code is included in the document.

2.2.4 Example

In figure 2.2 we show a small example of an XML document where some of the markup described before is used:

```
<? xml version="1.0" encoding="UTF-8" standalone="no"?>
<! DOCTYPE addressbook SYSTEM "addressbook.dtd" [
<! ENTITY signature "Jorge Manuel Neves Coelho" >
<! ENTITY pic1 SYSTEM "/images/jorge.png" NDATA Png>
<! ENTITY pic2 SYSTEM "/images/frank.png" NDATA Png> ]>

<addressbook>
        <name> &signature; </name>
        <country city="Porto"> Portugal </country>
        <email> jcoelho@ncc.up.pt </email>
        <phone>
                <landline> 123456789 </landline>
                <mobile> 987654321 </mobile>
        </phone>
        <pic file="pic1"> Jorge on the street </pic>
        <name> Frank Brown </name>
        <country city="London"> United Kingdom </country>
        <email> frank_brown@mailserver.uk </email>
        <pic file="pic2"> Frank's id photo </pic>
</addressbook>

<!-- End of document -->
```

Figure 2.2: XML document

First line tells the version, type of encoding and that external data should be processed. Then in the second line the root element is declared along with a validation schema (given in a DTD file). Then, three entities are described, the first one with an abbreviation and the following ones with binary objects. The remaining lines describe elements and at the end there is a comment.

2.3 Document Type Definition

(*Document Type Definition*) [W3C04d] is a schema language for describing the structure of an XML document. Several standards are described in DTDs, for example, SMIL, MathML, and XHTML. They are optional but very useful, for example, one can use a DTD to verify if a given element or attribute is valid, if its position is the correct one, etc.. DTD declarations can appear inside an XML document but usually they are described in a separate file. They can also appear both inside and as an external file at the same time. The reader should notice that declarations inside the document are processed first and thus have precedence. Whenever we declare DTD we use keyword *DOCTYPE*. The next example shows DTD declarations inside an XML file:

```
<! DOCTYPE addressbook [
<!-- Beginning of DTD -->
<!ENTITY ...>
...
<!-- End of DTD -->]>
```

If we want to declare the DTD as an external file we can do:

```
<! DOCTYPE addressbook SYSTEM "addressbook.dtd">
```

The keyword *SYSTEM* is used whenever the DTD is a local file, if on the other hand, the DTD is accessible by an url we can use the keyword *PUBLIC*:

```
<!DOCTYPE addressbook PUBLIC "-//Jorge Coelho//Addressbook//EN"
                              "http://www.mydomain.com/config/addressbook.dtd">
```

In this example the text following keyword *PUBLIC* starts by '-' (meaning that the DTD is not ISO neither is centrally assigned), followed by the author name a description of the content and the language (ISO 639). All this data is divided by //.

2.3.1 Content of a DTD

Inside a DTD we can describe elements, attributes, entities, and notations. All declarations are enclosed between <! and >. In this section we describe this declarations.

2.3.1.1 Elements

We can define one element using the keyword *ELEMENT*, arguments used are, the name of the element (can only start by a a-zA-Z, _ or : and can additionally contain numbers and characters "." and "-") and a declaration of its content. Possible content for an element is:

- ANY: content can be other elements or text.
- EMPTY: there is no content.
- #PCDATA: (Parsable Character Data), content is text.
- One or more sub elements, grouped with comas (or in a particular case by "|"). The order of sub elements is the order they must occur in the XML document.

Examples of valid elements are:

```
<!ELEMENT a ANY>
<!ELEMENT b EMPTY>
<!ELEMENT c (a,b)>
```

One valid document against this DTD is:

```
<c>
  <a> example text <b/> </a>
  <b/>
</c>
```

One invalid document is:

```
<c>
  <a> example text <b/> </a>
  <a> another example text </a>
  <b/>
</c>
```

The non-validity comes from the occurrence of two *a* elements which is not allowed by the DTD.

Whenever an element contains one or more sub elements (which can be mixed with text) it is possible to add more information. Thus we can declare if one or more elements are mandatory, if they are optional, if they occur one or more times or if they occur zero or more times. This is specified by adding one of the following characters after the element(s):

***** element(s) can occur zero or more times.

+ element(s) can occur one or more times.

? element(s) is(are) optional.

| elements separated by this character are optional in the sense that only one of them should occur.

For example, given the following DTD:

```
<!ELEMENT a (b*,c?)+>
<!ELEMENT b (#PCDATA | d)>
<!ELEMENT c (#PCDATA)>
<!ELEMENT d (#PCDATA)>
```

Element *a* may contain one or more groups of elements *b* and *c* where *b* must occur zero or more times and *c* is optional. Element *b* may contain text or a sub element *d*. One valid XML document against this DTD is:

```
<a>
  <b> text </b>
  <b> more text </b>
  <b> <d>  this is boring </d></b>
  <c> tired of text </c>
  <c> finally the text is over </c>
</a>
```

2.3.1.2 Attributes

Attributes can be defined using the keyword *ATTLIST*. Declaration of attributes is separated from element declaration. For example to declare attributes for an element *name* we can do:

```
<!ATTLIST name  attribute1      type_attribute1
                attribute2      type_attribute2
                ...             ...              >
```

Inside an *ATTLIST* declaration it is possible to define all the attributes for a given element, their type and also a pre-defined value. The same constraints in elements names apply to attributes names. Possible types are:

CDATA Attribute can include text. For example:

```
<costumer name="Donald Duck">
```

NMTOKEN Attribute can include one word only. Restrictions are the same as the ones for elements names seen on section 2.3.1.1, except for the first character.

NMTOKENS Several *NMTOKEN* separated by an empty space. For example:

```
<map coordinates="1 5 78 65">
```

ID Identification of an element in a unique form. All values for *ID* should be unique inside the document.

IDREF Contains the value of an *ID* present in the document.

IDREFS Several *ID*'s separated by white spaces.

ENTITY The attribute is an entity. For example:

```
<!ENTITY pic1 SYSTEM "/images/jorge.png" NDATA Png>
<!ELEMENT pic EMPTY>
<!ATTLIST pic file ENTITY ...>
...
<pic file="pic1" />
```

ENTITIES Several entities separated by white spaces.

Enumeration One or more names separated by "|". The attribute needs to have one of the values in the enumeration. For example:

```
...
<! ATTLIST person          ...
                           sex (male|female)
                           ...                    >
...
<person sex="female">
...
```

Notation Attributes can be binded to a value defined by a notation. Details on notations appear in section 2.3.1.4.

Attributes can have predefined values and it is possible to state if they are optional or not. Options follow:

- *#REQUIRED:* Attribute must occur in all the occurrences of the element which is associated with it.
- *#IMPLIED:* Attribute is optional.
- *#FIXED:* Attribute has a fixed value.
- *Value:* Attribute has the value presented between " and ".

For example:

```
<ATTLIST costumer
          cod          ID        #REQUIRED
          name         CDATA     #IMPLIED
          sex          (fem|mal)      "fem"
          comment      CDATA     #FIXED  "I'm a fixed comment">
```

Here we declare attributes for element *costumer*: *cod* which is an ID and is mandatory, *name* whose content is text and is optional, *sex* that may have one of two values *fem* or *mal* and by default has value *fem* and finally *comment* which is fixed and its value is "I'm a fixed comment".

Entity name	Text
<	<
>	>
&	&
'	’
"	”

Table 2.1: Pre-defined Entities

2.3.1.3 Entities

Entities are declared by the keyword *ENTITY*. There are three types of entities, *internal*, *external* and *parameter* entities. The description of these entities follows.

- *Internal Entities:* Internal entities are blocks of text that can be refereed by an abbreviation, for example:

```
<! ENTITY signature "Jorge Manuel Neves Coelho" >
```

  Whenever we need to insert the name “Jorge Manuel Neves Coelho”, we simply use &signature;. There are five pre-defined entities as described in table 2.1.

- *External Entities:* External entities bind a name to an external file. The file can be text of binary. References to external files are preceded by keyword *SYSTEM* or *PUBLIC*, for example:

```
<!ENTITY  ent1 SYSTEM "/ents/ent1.xml">
<!ENTITY  pic1 SYSTEM "/images/jorge.png" NDATA Png>
```

- *Parameter Entities:* Parameter entities are used exclusively in DTDs and are used as an abbreviation for declarations. For example, given the following DTD:

```
<!ELEMENT a (c|d|e)>
<!ELEMENT b (c|d|e)>
```

  We can create the following entity:

```
<!ENTITY % ab "(c|d|e)">
```

  And use it as an abbreviation:

```
<!ELEMENT a %ab;>
<!ELEMENT b %ab;>
```

2.3.1.4 Notations

Notations are declared using keyword *NOTATION*: they refer to external formats. For example:

```
...
        <!ELEMENT movie (#PCDATA)>
        <!ATTLIST movie file ENTITY #REQUIRED>
        <!ENTITY amelie SYSTEM "movies/amelie.mov" NDATA Mov>
        <!NOTATION Mov SYSTEM "movieplayer">
...
<movie file = "&amelie;"> Wonderful french movie </movie>
```

The notation is used to declare the application used to play the file “amelie.mov”.

2.3.1.5 Example

The DTD presented in figure 2.3 validates documents as the one presented in figure 2.2.

```
<!ELEMENT addressbook (name,country,email,phone?,pic)*>
<!ELEMENT name (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ATTLIST country city CDATA #REQUIRED>
<!ELEMENT email (#PCDATA)>
<!ELEMENT phone (landline?,mobile)>
<!ELEMENT landline (#PCDATA)>
<!ELEMENT mobile (#PCDATA)>
<!ELEMENT pic EMPTY>
<!ATTLIST pic file ENTITY #REQUIRED>
<!NOTATION Png SYSTEM "picview">
```

Figure 2.3: Example of a DTD

2.3.2 Limitations

DTDs have several limitations. Among them, we refer the following ones:

- Not being an XML dialect itself makes it more difficult to build parsers.
- *#PCDATA* is to generic, it would be better if we could define other base types.
- Attribute values cannot be constrained into different types.
- It is hard to reuse data in DTDs.
- DTDs do not support namespaces.

These and other limitations made the W3C make an effort to create a new schema language named XML Schema, that is the subject of the next section.

2.4 XML Schema

XML Schema [W3C04f] is a schema language developed by W3C whose goal was to overcome several limitations of DTDs. It is described in two parts, Part1: Structures, describing the core of XML Schema including declaration of elements and attributes and Part2: Datatypes, describing the builtin datatypes and their facets. Due to its complexity it is impossible to describe it in few pages (the first part of XML Schema description has more than 130 pages itself), thus we will focus on the core of the schema and address the interested reader to the schema description available at [W3C04f].

XML Schemas are based in four main constructs:

- *Simple Types*: simple constraints over values.
- *Complex Types*: defines attributes and sub-elements for a given element.
- *Element declaration*: binds an element to a type.
- *Attribute declaration*: binds an attribute to a simple type.

We start by presenting an example and explaining basic concepts:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:p="http://www.ncc.up.pt/persons"
           targetNamespace="http://www.ncc.up.pt/persons">

        <xs:element name="person" type="p:personType"/>

        <xs:attribute name="name" type="xs:string"/>
        <xs:attribute name="age" type="p:ageType"/>


        <xs:complexType name="personType">
                <xs:attribute ref="p:name" use="required"/>
                <xs:attribute ref="p:age" use="required"/>
        </xs:complexType>

        <xs:simpleType name="ageType">
                <xs:restriction base="xs:integer">
                        <xs:minInclusive value="0"/>
                        <xs:maxInclusive value="100"/>
                </xs:restriction>
        </xs:simpleType>
</xs:schema>
```

XML Schema is itself described in XML and supports namespaces. In the beginning of this schema we define a namespace for XML Schema constructs (xs:) and another for persons

(p:), the *targetNamespace* attribute indicates the namespace being described by the schema. Here we declare one *simple type*, containing a value for an *age* attribute that ranges from 0 to 100 and a *complex type* containing two attributes, *name* which is a string and *age* which is defined by the *simple type*. The schema declares one element (*person*) which is binded to the *complex type personType*. One valid document is:

```
<?xml version="1.0" encoding="UTF-8"?>
<person age="31" name="Jorge Coelho"/>
```

2.4.1 Simple Types

Simple types, also called datatypes, are constraints over strings. For example, declaring a type as *integer* means that the value can be any integer number.

Example 2.4.1 *The following element declaration assigns a simple type (date) to element named* birth*:*

```
<element name="birth" type="date"/>
```

Valid XML is:

```
<birth> 1975-08-15 </birth>
```

In table 2.2 we describe simple types. It is possible to extend these simple types creating new ones.

2.4.1.1 Restrictions

We now show how to extend types with restrictions.

Example 2.4.2 *The next simple type uses keyword* restriction *to create a new type for element* grade *where the accepted values are integers between 0 and 20.*

```
<simpleType name="grade">
 <restriction base="integer">
  <minInclusive value="0"/>
  <maxInclusive value="20"/>
 </restriction>
</simpleType>
```

Valid XML is:

```
<grade> 15 </grade>
```

Restrictions are a simple way to extend types, *minInclusive* and *maxInclusive* are facets that are used to constraint the type. Facets are described in table 2.3.

Other possible ways to extend simple types with restrictions are:

Type	**Example values**
string	any Unicode string
boolean	true, false, 0, 1
decimal	1.5
float	1.0122E29
double	42E970
duration	P0Y1M23DT09H15M
dateTime	2007-02-15T12:00:00Z
time	16:45:00-05:99
date	2007-02-01
gYearMonth	2007-02
gYear	2007
gMonthDay	–02-01
gDay	–01
gMonth	–02
hexBinary	567ac
base64Binary	SGVsbg8K
anyURI	http://www.ncc.up.pt/person/
QName	person, p:age
NOTATION	related to the declared notation in the schema

Table 2.2: Simple Types

Facet	**Constraining**
length	length of string or number of items
minLength	minimal length
maxLength	maximal length
pattern	regular expression
enumeration	enumeration value
whitespace	white space normalization
maxInclusive	inclusive upper bound (for ordered types)
maxExclusive	exclusive upper bound
minInclusive	inclusive lower bound
minExclusive	exclusive lower bound
totalDigits	maximum number of digits (for numeric types)
fractionDigits	maximum number of fractional digits

Table 2.3: Facets of Simple Types

- Enumerations - where a finite list of values is given.

Example 2.4.3 *In this example we constraint possible values for element* country *to, EN, PT and ES.*

```
<simpleType name="country">
 <restriction base="string">
  <enumeration value="PT"/>
  <enumeration value="ES"/>
  <enumeration value="EN"/>
 </restriction>
</simpleType>
```

Valid XML:

```
<country> PT </country>
```

- Patterns - where regular expressions can be used for types.

Example 2.4.4 *The next type allows only two uppercase characters for its value:*

```
<simpleType name="country">
 <restriction base="string">
  <pattern value="[A-Z][A-Z]"/>
 </restriction>
</simpleType>
```

Valid XML:

```
<country> PT </country>
```

2.4.1.2 Lists

A list of a type defines a whitespace separated sequence of values of that type.

Example 2.4.5 *The next type describes a list of integers:*

```
<simpleType name="grades">
 <list itemType="integer"/>
</simpleType>
```

Valid XML:

```
<grades> 10 15 20 17 </grades>
```

2.4.2 Union

A union makes it possible to declare alternative types.

Example 2.4.6 *The next type describes an element that accepts an integer or a string:*

```
<simpleType name="intstr">
 <union>
  <simpleType>
   <restriction base="string">
  </simpleType>
  <simpleType>
   <restriction base="integer">
  </simpleType>
 </union>
</simpleType>
```

The more common extensions to simple types were included in XML Schema and are: normalizedString, token, language, NMTOKEN, NMTOKENS, Name, NCName, ID, IDREF, IDREFS, ENTITY, ENTITIRS, integer, nonPositiveInteger, negativeInteger, nonNegativeInteger, unsignedLong, unsignedInt, unsignedShort, unsignedByte, positiveInteger, long, int, short and byte. These can be consulted in [W3C04f].

2.4.3 Complex Types

Complex types allow the description of sub-elements and attributes. To assign a complex type to an element one does:

```
<element name="addressbook" type="addr:addressbook_type"/>
```

We now describe the different constructs available for complex types.

2.4.3.1 Concatenation

Concatenation of element is made using a *sequence*.

Example 2.4.7 *Next complex type describes a sequence of elements for an addressbook.*

```
<complexType name="addressbook_type">
 <sequence>
  <element name="name" type="string"/>
  <element name="address" type="string"/>
  <element name="email" type="string"/>
  <element name="phone" type="string"/>
 </sequence>
</complexType>
```

2.4.3.2 Choice

Choice makes it possible to choose between a list of elements.

Example 2.4.8 *Next complex type describes a sequence of elements for an addressbook where a phone or email, but not both, occurs.*

```
<complexType name="addressbook_type">
 <sequence>
  <element name="name" type="string"/>
  <element name="address" type="string"/>
  <choice>
   <element name="email" type="string"/>
   <element name="phone" type="string"/>
  </choice>
 </sequence>
</complexType>
```

2.4.3.3 All

All allows an unordered sequence of elements to appear.

Example 2.4.9 *Next complex type describes a sequence of elements for an addressbook were order is irrelevant.*

```
<complexType name="addressbook_type">
 <all>
  <element name="name" type="string"/>
  <element name="address" type="string"/>
  <element name="email" type="string"/>
  <element name="phone" type="string"/>
 </all>
</complexType>
```

In order to avoid further complexity the *all* sequence may only contain elements (not *sequence*, *choice* or *all*).

2.4.3.4 Any

Any allows any element to occur in the place where it is declared.

Example 2.4.10 *Next complex type describes a sequence of elements for an addressbook were one element matches any of the elements available under namespace* contact *and may occur from one to 3 times.*

```
<complexType name="addressbook_type">
 <sequence>
  <element name="name" type="string"/>
  <element name="address" type="string"/>
  <any namespace="##contact" minOccurs="1" maxOccurs="3"
                                processContents="strict">
```

```
  </sequence>
</complexType>
```

The processContents = "strict" *forces checking the validity of the elements and their descendants.*

2.4.3.5 Attribute

Attributes may be declared in complex types.

Example 2.4.11 *Next complex type describes a sequence of elements for an addressbook were* contact *has content type string and an attribute of type* country_code*.*

```
<simpleType name="country_code">
 <restriction base="string">
  <pattern value="\+[0-9][0-9][0-9]"/>
 </restriction>
</simpleType>

<attribute name="country" type="country_code"/>

<complexType name="addressbook_type">
 <sequence>
  <element name="name" type="string"/>
  <element name="address" type="string"/>
  <element ref="contact"/>
 </sequence>
</complexType>

<complexType name="contact">
 <element name = "phone" type = "string"/>
 <attribute ref = "country" use ="required"/>
</complexType>
```

If use = "required" *is omitted then it assumes the default value which is "optional".*

2.4.4 Other properties

XML Schema has other features, some of them are:

Type Derivation: It is possible to derive new complex types by extending or restricting existing types.

Groups: It is possible to define groups of expressions in complex types and reuse them.

Nil values: It is possible to describe elements as possibly null.

Annotations: It is possible to introduce comments in the form of markup.

Modularization: Reuse and evolution is supported through schema modularization techniques that are available in XML Schema.

Keys and references: Uniqueness and referential constraints are available in XML Schema by using an extension to the DTD ID attribute.

2.4.5 Problems with XML Schema

Although the intention was to overcome the problems from DTDs, XML Schema introduced several new ones. Examples follow:

- XML Schema is too complex, a big and dense description makes difficult to understand how to work with it.
- There is no complete schema for XML Schema itself.
- Elements defaults cannot contain markup, only text.
- The *all* construct has a very strict type of content, restricting its applications.

2.5 Other Schema Languages

Although the most popular schema languages are DTDs and XML Schemas there are other schemas developed by different entities in order to overcome the drawbacks found in both DTDs and XML Schemas noticed in practical situations. We give a short overview of three of these schema languages: RelaxNG, Schematron and DSD2.

2.5.1 RelaxNG

RelaxNG [Spe01] was developed by the Organization for the Advancement of Structured Information Standards (OASIS) and is an ISO standard. The main goals are simplicity and expressiveness. Relax NG is based on grammars and has both an XML and non-XML syntax. XML-based schemas can be difficult to read and the non-XML syntax can be very useful in several contexts. We now describe briefly some of the properties of this schema language:

- Elements and Attributes are treated in an almost uniform manner not making the rigid separation that exists in DTDs and XML Schemas.
- Patterns consisting of familiar quantificators: any pattern may be conditioned as <oneOrMore>, <zeroOrMore>, or <optional>; these correspond to the DTD quantifiers +, *, and ? respectively.
- Allows sequences of patterns at the same level given using the <choice>, <group>, or <interleave> elements.

2.5.2 Schematron

Schematron [Jel04] is an ISO standard that is based in the use of tree patterns. The principle is simple, instead of using a grammar, Schematron makes assertions applied to specific context within the document. If the assertion fails, a message can be reported. The central ideas are:

- Assertions are used to specify the constraints that should be checked whitin a specific context of the XML document like, for example, a given element e should have an attribute a and if not report a given message.
- Rules specify contexts and which assertions to apply to the contexts.
- Patterns are used to group rules, in this way it is possible to have different rules applied to the same context but grouped by a specific pattern.

2.5.3 DSD2

The DSD2 (Document Structure Description 2.0) [Ml04] is a language developed by the University of Aarhus and AT&T Labs Research. The idea was to produce a powerful language but simpler than XML Schema. The central ideas are:

- A schema consists of a list of rules.
- Rules contain *declare* and *require* sections. In *declare* section sub-elements and attributes are declared for a given element. In *require* sections extra restrictions on contents, attributes and context are specified.
- Element and attribute content are described by regular expressions.
- Rules are described by boolean logic.

2.6 XML Processing Languages

Due to the increasing popularity of XML it is normal to see the rising of several languages specialized in processing XML documents. We can divide XML processing languages in two groups, untyped languages and typed languages. The former group is the oldest and includes languages like XSLT [W3C01b] and the APIs DOM [W3C04a] and SAX [Meg04]. The later group differs from the former by allowing validation of XML processing programs at compile time. Examples are XDuce [HP00], CDuce [BCF03] and Xtatic [GLPS05]. The type system of these languages uses the schemas for XML validation associated with the documents, as type declarations. Knowing which kind of data may be acceptable in documents improves program developing by providing error detection at compile time. In this section we describe

briefly some languages for XML processing focusing on one of the most promising, namely XQuery.

2.6.1 Untyped XML processing languages

We present briefly, the language and platform-independent SAX and DOM APIs, the XSLT (eXtensible Stylesheet Language Transformations) and the rule-based query language Xcerpt [BS02a]. Although without having compile time validation, XSLT and the DOM and SAX APIs proved to be useful in several domains and are the most popular tools for XML processing today.

2.6.1.1 DOM

The Document Object Module is an API for XML processing. XML is translated by a parser into a structure in memory organized in a hierarchy of nodes. Those nodes are structures that may contain information (like other nodes). Examples are:

- The Document
- Elements
- Attributes
- Text
- CDATA
- Processing Instructions
- Entities
- Notations

DOM provides means to describe nodes and the relationship between them using objects and methods. We can use a Document node to get its root element and an element node to get its child elements and attributes. DOM also allows adding, changing and removing nodes from the document. DOM is currently available for languages like JAVA [Mic03] and C++ [Pro04].

2.6.1.2 SAX

SAX (Simple API for XML) is an alternative to the memory consuming DOM approach. In SAX an XML tree is viewed as stream of events and not like a data structure. SAX events are:

- start and end of document.
- start and end of tag
- character data is found
- processing instruction is found

In a language with access to a SAX API actions are triggered whenever some of the events mentioned above take place. For example, in a document containing records of costumers, we can use the start of element event to get a costumer with a given name. SAX API has been implemented for languages such as Java [Meg04], Perl [Pro03] and Python [Fou03].

2.6.1.3 XSLT

XSLT is part of XSL (eXtensible Stylesheet Language) [W3C99a]. The goal of XSL is to provide a way to publish documents based in style sheets. These style sheets provide for example, size of fonts and paragraph alignments. The transformation between the XML document and its publishing format is done by the XSLT. XSLT can be used to translate XML documents in other document, for example, another XML document or an XHTML document. The working method is simple, the programmer provides a style sheet with templates, XSLT processes a source document and each time it finds a part of the document that matches a given template it translates the source code into a new one accordingly to the template. For example given the following XML document:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<?xml-stylesheet type="text/xsl" href="./costumers.xsl"?>
<records>
   <costumer>
      <num> 1 </num>
      <name> John </name>
   </costumer>
   <costumer>
      <num> 2 </num>
      <name> Paul </name>
   </costumer>
</records>
```

We can use the following style sheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
   <body>
    <h1>Costumers</h1>
```

```
    <table border="1">
    <tr>
      <td align="left">Number</td>
      <td align="left">Name</td>
    </tr>
    <xsl:for-each select="records/costumer">
    <tr>
      <td><xsl:value-of select="num"/></td>
      <td><xsl:value-of select="name"/></td>
    </tr>
    </xsl:for-each>
    </table>
   </body>
  </html>
</xsl:template>
</xsl:stylesheet>
```

And translate the original document to HTML:

```
<html>
  <body>
    <h1>Costumers</h1>
    <table border="1">
    <tr>
      <td align="left">Number</td>
      <td align="left">Name</td>
    </tr>
    <tr>
      <td>1</td>
      <td>John</td>
    </tr>
    <tr>
      <td>2</td>
      <td>Paul</td>
    </tr>
    </table>
   </body>
</html>
```

In the style sheet we start by providing the version and namespace [W3C99b]. Every XSLT instruction is proceeded by *xsl:*. In this example, a table is created and for each costumer, the values of tags *name* and *num* are disposed inside table rows. XSLT uses XPath [W3C99c] as the language to describe the tags and attributes to match.

2.6.1.4 Xcerpt

Xcerpt is a deductive, rule-based query language for graph-structured data. It queries XML and also RDF. A rule is like a VIEW in database systems. In brief, this means that a

result (XML document) is created in case some query to one or several sources succeeds. For example, given the addressbook file presented in section 2.2 we can build the following query:

```
GOAL
  addressbook {
    all record { var NAME, var EMAIL }
  }
FROM
  in {
    resource { "file:addressbook.xml", "xml" },
    addressbook {{
            record {{
              var NAME -> name {{ }},
              var EMAIL -> email {{ }}
      }}
    }}
  }
END
```

In this example we query the addressbook file retrieving the names and emails in records.

Although appropriate for XML query, Xcerpt also focus on the Semantic Web as it can be seen in the latest extensions to the language [SW06, DW06]. It worths mentioning that types for a small subset of Xcerpt were studied in [WD03, WD06], extending tree automata to deal directly with terms with functors of arbitrary arity.

2.6.2 Typed XML processing languages

This group of languages has the novelty of providing type systems to improve program developing. They use information in DTDs and XML Schemas for compile-time and run-time type checking. We now present two examples of this group of languages, XDuce and XQuery.

2.6.2.1 XDUCE

XDuce (pronounced transduce) is a functional language specialized in XML processing. XDuce provides a type system based in regular expressions, that are a generalization of DTDs, and has regular expression pattern matching as its fundamental technique. For example, the following DTD:

```
<!ELEMENT records (name,address,phone?)*>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (#PCDATA))>
```

Corresponds to the following type definitions in XDuce:

```
type Records = records[(Name,Address,Phone?)*]
type Name = name[String]
type Adress = address[String]
type Phone = phone[String]
```

The following program translates documents valid accordingly to the previous DTD in new ones where only the records with phone are stored and where the address is ignored.

```
fun procRec : (Name,Address,Phone?)* -> (Name,Phone)* =

  name[n:String], address[m:String],
    phone[t:String],rest(Name,Address,Phone?)*
              -> name[n], phone[t], procRec(rest)

  | name[n:String], address[m:String],rest(Name,Address,Phone?)*
              -> procRec(rest)

| () -> ()
```

Types are declared at the head of the function. Following that, we have three rules each one for the three possible sequences of elements found in the document. The first one, matches a *record* with sub elements *name*, *address* and *phone*, the second one matches records without *phone* (and so they are ignored) and the third one matches the empty documents or the end of the sequence of records. This approach guarantees that the data processed is correct since pattern matching only succeeds when the data is associated to a given type. XDuce was the beginning of a new generation of XML processing languages.

2.6.2.2 XQuery

XQuery [W3C04e] is a functional-based language developed by W3C. XQuery integrates XPath 2.0 and all the information it processes is constrained by its data model [W3C04g]. We now describe briefly XQuery features, the data model, expressions and types.

2.6.2.3 Data Model

Every document XQuery processes is an instance of the data model. The data model is based in:

Simple types : strings, integers, date, etc.

Nodes : document, element, attribute, text, comment, processing-instruction and namespace.

Sequences : A sequence is an ordered list of simple values or nodes. . The comma operator can be used to concatenate two values or sequences. For example,

a,b is a sequence consisting of two chars

(1, (), (3,4)) is a sequence and the same as (1,3,4)

A sequence containing just a single value is the same as that value by itself.

2.6.2.4 Expressions

Expressions are the basic building blocks of XQuery. There are several expressions:

Literals : 1, 1.0 or “hello”.

Variables : In XQuery variables begin with the dollar sign for example let $a := 1 assigns the value 1 to variable $a.

Operators : For example, +,-,*,/ for arithmetic expressions, >,< and !=, for comparison, *intersect* and *except* for sequence manipulation. There are also operators for node comparison, order comparison and logic operations.

Path expressions : These are used to trace a path through the document structure to the nodes we want to extract. For example, if we have a document with costumer information and want to retrieve the age of costumer named “Paul” (provided that element *exists*) we can use the XPath expression:

```
document("costumers.xml")/*/costumer[name=''Paul'']/age
```

Element and Attribute Constructors : These consist in functions that create a value of a particular type. For example, the following XQuery program:

```
let $d = <p> This is a <b> test </b> for constructors </p>
```

is an element constructor, note that the keyword *let* assigns a value to a variable. It is possible to include XQuery expressions inside constructors by using curly braces. For example:

```
let $p = <b> test </b> return
let $d = <p> This is a {$p} for constructors </p>
```

Results in:

```
<p> This is a <b> test </b> for constructors </p>
```

FLOWR expressions : Pronounced flower, provide the already seen *Let* along with *For*, *Order-by*, *Where* and *Return*. The next XQuery code returns the names of all costumers older than 18 years ordered by age that were found in “costumers.xml”:

```
for $costumer in document('costumer.xml')/records/costumer
let $name := $costumer/name/text()
where $costumer/age > 18
order by $costumer/age ascending
return $name
```

Conditional expressions: A conditional expression provides a way of executing one of two expressions, depending on the value of a third expression. It is written in the familiar if . . . then . . . else. The following XQuery code returns costumer data if the costumers age is greater than 18 and the empty sequence otherwise:

```
if $costumer/age > 18
then $costumer
else ()
```

Quantified expressions. These allow testing of some condition to see whether it is true for *some* value in a sequence, or for *every* value in a sequence. The result of a quantified expression is always *true* or *false*. For example, the next XQuery code checks if some costumer in “costumer.xml” has less than 18 years old.

```
if some $costumer in document('costumer.xml')/records/costumer
satisfies $costumer/age < 18
then ``There are non-adult costumers''
else ``Only adult costumers''
```

Other way, using the *every* quantifier:

```
if every $costumer in document('costumer.xml')/records/costumer
satisfies $costumer/age > 18
then ``Only adult costumers''
else ``There are non-adult costumers''
```

User Defined Functions : XQuery allows the programmer to write its own functions. For example, the next user-defined function returns the message sent has its argument:

```
define function greetings(xsd:string $message)
  returns element {
                   {$message}
                  }
```

Built-ins : XQuery provides built-in functions for several purposes, for example, *compare, concat, substring, count, max, min, avg.* The next example computes the average age of the costumers:

```
let $avg_age := avg(document("costumers.xml")//costumer/age)
```

2.6.2.5 Types

XQuery provide a type-system based in XMLSchema (based, but not equal). The goal of the type system is to detect statically errors in the queries, infer the type of the result of valid queries and ensure statically that the result of a given query is of a given (expected) type. Whenever there is no information about types XQuery performs dynamic type checking.

Types can be assigned directly to values using for example, *xs:string*, *xs:integer* or with keywords like *element* and *attribute*, for example, *element of type costumer** accepts zero or more elements of type *costumer*. XQuery also provides keywords like *typeswitch* and *treat* useful for dynamic time validation. *Typeswitch* allows to execute different code depending on the run time type of an expression. *Treat* allows to ensure that an expression has an expected dynamic type at evaluation time. All the possible types can be found in [W3C04e]. XQuery also allows us to import XML Schemas. For example, consider the following simple schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns:c="http://www.ncc.up.pt/persons"
           targetNamespace="http://www.ncc.up.pt/costumers">

        <xs:element name="records" type ="recordType"/>
        <xs:element name="costumer" type="c:costumerType"/>
        <xs:element name="name" type="xs:string"/>
        <xs:element name="age" type="c:ageType"/>

        <xs:complexType name="costumerType">
         <sequence>
             <xs:element ref="c:name"/>
             <xs:element ref="c:age"/>
         </sequence>
        </xs:complexType>

        <xs:complexType name="recordType">
           <xs:element ref="c:costumer"
                       minOccurs="0" maxOccurs="unbounded"/>
        </xs:complexType>
</xs:schema>
```

Importing the schema:

```
import schema "costumers.xsd"
```

makes XQuery translates this schema to the following types:

```
element Records {Costumer*}
element Name {xs:string}
element Age {xs:integer}
```

now, it is possible to validate code with the keyword *validate*:

```
let $c =
  validate {
    <records>
      <costumer>
         <age> 25 </age>
         <name> John </name>
      </costumer>
    </records>
  }
```

From now on these elements are annotated with their types as it was declared by the schema. If we use this data in a context that doesn't insures type correctness XQuery outputs an error. Note that every element has type *anyType* (an universal type that is more general than every other) until a more specific one is found.

2.6.3 Other Languages

There are other approaches to XML processing based on the concepts originated by XDuce. CDuce is also a functional programming language specialized in XML processing, with some more features than XDuce. For example, CDuce has support for a richer type algebra with recursive types and arbitrary boolean combinations (union, intersection and complement). CDuce supports DTD typing and a subset of XML Schema. It also supports Namespaces. The authors of CDuce claim that it is faster than XSLT (see `http://www.cduce.org/bench.html` for benchmark results).

Xtatic [GLPS05] is an extension to C# tailored for native XML processing that aims to merge regular expression types with the C# object model, thus, mimicking XDuce techniques in a mainstream language.

Cω [BBC$^+$04] is a language that aims to bring XML processing and concurrency together.

2.7 Discussion

As we saw, XML documents are usually represented by trees. Since trees are naturally represented by terms, paradigms that deal with terms, like Logic Programming and Constraint Logic Programming, handle trees in a quite interesting and declarative way. There is thus an obvious link between these paradigms and XML processing. We will explore these connection in the following chapters.

Chapter 3

Constraint Solving

3.1 Introduction

In this thesis we assume that the reader is familiar with the logic programming paradigm (see [Llo87, SS94] for details). Here we briefly describe constraint solving focusing on the Constraint Logic Programming approach. We briefly introduce the ideas behind Constraint Logic Programming and explain how Unification is a particular case of this paradigm. Then we present the Flexible Arity Term unification which extends unification with terms of arbitrary arity. Finally we present a new terminating procedure for the unification with terms of arbitrary arity where sequence variables occur at most twice.

3.2 Constraint Logic Programming

Constraint Programming Languages, uses constraints as primitive elements of the programming language itself. Constraint Logic Programming Languages also use Horn clause programming as the main computational paradigm. In this section we focus on Constraint Logic Programming (CLP) as a general framework for reasoning about constraints.

3.2.1 CLP(X)

Constraint Logic Programming was presented by Jaffar and Lassez in [JL87]. There, the authors presented a class of programming languages based on constraint solving and the logic programming paradigm. The framework is named CLP(X) where X can be instantiated with a suitable domain of discourse, thus resulting for example in Prolog, Prolog-II, Prolog-III or other languages. More formally in CLP(X) languages we have:

1. Domain of computation: X (X can be Booleans, Rationals, Reals, Trees, etc.).
2. Syntax: definite clauses with constraints.

3. Operational interpretation based on:

   (a) SLD Resolution (such as Prolog).

   (b) Constraint Solving.

4. Declarative interpretation: the same as logic programming (Horn Clauses).

5. Output: set of constraints.

Note that the output is the set of all the constraints as a domain dependent logical answer to a query, while in logic programming the output are variable substitutions.

We now present some definitions.

Definition 3.2.1 (Structure) *A* structure *consists of a triple* (D, F, R) *consisting of domain* D*, a set* F *of operations of* D^n *to* D *and a set* R *of relations in* D^n*.*

Definition 3.2.2 (Terms) *A* term *is a constant* $c \in F$*, a variable or* $f(t_1, \ldots, t_n)$*, where* $f \in F$ *is n-ary and* $t_1, \ldots, t_n$ *are terms.*

Definition 3.2.3 (Atomic constraints) *An atomic constraint is of the form* $p(t_1, \ldots, t_n)$*, where* $p \in R$ *and* $t_1, \ldots, t_n$ *are terms.*

Definition 3.2.4 (Constraint) *A constraint is a possibly empty conjunction of atomic constraints.*

Definition 3.2.5 (CLP(X) Program) *Given symbols* $p_i, 0 \leq i \leq m$ *such that* $p_i \in F$ *and* $c_j, 0 \leq j \leq n$ *such that* $c_j \in R$ *and* $t_k, u_l, 0 \leq k \leq m, 0 \leq l \leq n$ *such that* t_k *and* u_l *are terms, a Constraint Logic Programming program consists of a finite set of rules of the form:*

$$p_0(t_0) : -c_1(u_1), \ldots, c_n(u_n), p_1(t_1), \ldots, p_m(t_m)$$

$p_0(t_0)$ *is the* head *and the conjunction* $c_1(u_1), \ldots, c_n(u_n), p_1(t_1), \ldots, p_m(t_m)$ *is the* body *of the rule.*

Definition 3.2.6 (Goal) *A* goal *is a constraint rule without the head.*

3.2.2 CLP Languages

In this section we present some examples of CLP languages, namely CLP(R), Prolog III and ECL^iPS^e.

3.2.2.1 CLP(R)

CLP(R) is an instance of CLP(X) defined by Jaffar and Lassez [JMSY92]. The domain of computation R is the algebraic structure consisting of uninterpreted functors over real numbers. Therefore CLP(R) is defined as:

Domain of computation: finite trees and reals.

Constraints: Linear equations, linear inequalities and non-linear equations (although those last are not solved but delayed until they become linear).

3.2.2.2 Prolog III

Prolog III was developed by Colmerauer [Col90], in Prolog III unification is replaced by the more general concept of Constraint Solving. Therefore Prolog III is defined as:

Domain of computation: infinite trees, rationals, booleans and lists.

Constraints: The Prolog III constraints are:

- Equalities and inequalities for finite trees;
- Linear equalities, inequalities and inequalities using the operations +,-,* and /;
- Logical formulae using the operators, $\neg, \vee, \wedge, \Rightarrow$ and $\subset$.
- Equalities and inequalities of lists.

3.2.2.3 ECL^iPS^e

ECL^iPS^e (Common Logic Programming System) [AW07] is a Prolog based system whose aim is to serve as a platform for integrating various Logic Programming extensions, in particular Constraint Logic Programming (CLP). ECL^iPS^e is defined by:

Domain of computation: finite domains of integers, finite sets of integers, integers, reals.

Constraints: The ECL^iPS^e contains:

- Membership, inclusion, intersection, union and disjointness of sets.
- Equality and inequality of general expressions.
- Membership on a finite domain
- Several constructors for reals and integers.

Example 3.2.1 *This example was taken from [Col90] and illustrates the use of constraints. The idea is that, given the definition of meal as consisting of an appetiser, a main meal and a dessert, and given facts about foods and their calorific values, we wish to know which meals are light (calorific value is 10).*

```
lightmeal(A,M,D):-
    appetiser(A,I),
    main(M,J),
    dessert(D,K),
    {I > 0, J > 0, K > 0, I+J+K = 10}.

main(M,I):-
    meat(M,I).

main(M,I):-
    fish(M,I).

appetiser(radishes,1).
appetiser(pate,6).

meat(beef,5).
meat(pork,7).

fish(sole,2).
fish(tuna,4).

dessert(fruit,2).
dessert(icecream,6).
```

The meaning of the first rule is: "provided the four conditions I > 0, J > 0, K > 0, I+J+K = 10" are satisfied, the triple H,M,D constitutes a light meal. The remaining of the program are easy to follow facts and rules about the foods and their calorific values. One answer to the question ?-lightmeal(A,M,D) is {A = radishes, M = beef, D = tuna}.

Example 3.2.2 *The next example is an ECL^iPS^e version of the "SEND + MONEY" puzzle. The "SEND + MONEY" is a classical "crypto-arithmetic" puzzle: the variables S, E, N, D, M, O, R, Y represent digits between 0 and 9, and the task is finding unique values for each variable such that SEND + MORE = MONEY. Also, S and M must be different from 0.*

```
sendmore(Digits) :-
     Digits = [S,E,N,D,M,O,R,Y],
     Digits :: [0..9],
     alldifferent(Digits),
     S #\= 0,
     M #\= 0,
                  1000*S + 100*E + 10*N + D
                + 1000*M + 100*O + 10*R + E
     #= 10000*M + 1000*O + 100*N + 10*E + Y,
     labeling(Digits).
```

This program starts by creating a structure containing several elements each one of them corresponding to one of the letters in the problem. Then, this structure is binded to a list

of digits from 0 to 9, meaning that each of the letters represented in the structure may bind to one of these digits. Next, it is told that all the digits must be different and the equality restriction (#\ =) prevents S and M of being binded with 0. Finally an equality constraint imposes that the digits for the letters satisfy the equation SEND+MORE=MONEY. The labelling *predicate tries out all the possibilities for each variable.*

3.3 First Order Unification

In this section we describe briefly the concepts behind syntactic unification [Rob65], here refereed only as unification. Unification is an instance of CLP(X) where X corresponds to the Herbrand domain [Llo87].

Consider an alphabet consisting of the following sets: the set of variables, the set of constants, and the set of function symbols.

Definition 3.3.1 *The set of terms over the previous alphabet is the smallest set that satisfies the following conditions:*

1. *Constants and variables are terms.*
2. *Given the function symbol f and terms $t_1, \ldots, t_n$ ($n \geq 0$), then $f(t_1, \ldots, t_n)$ is a term.*

Notation 3.3.1 *Given a term t, $\mathcal{V}ar(t)$ returns the set of all free variables in t.*

Definition 3.3.2 *If t_1 and t_2 are terms such that neither t_1 nor t_2 is a sequence variable, then $t_1 \stackrel{?}{=} t_2$ is an equation.*

Definition 3.3.3 *A substitution is a finite set $\{x_1 := s_1, \ldots, x_n := s_n\}$ where $n \geq 0$, $x_1, \ldots, x_n$ are distinct variables and for all $1 \leq i \leq n, s_i$ is a term and $s_i \neq x_i$.*

Example 3.3.1 *The set $\{x := a\}$ is a substitution meaning that variable x is replaced by term a.*

Substitutions can be applied to terms, replacing all the occurrences of free variables in those terms by the term specified in the substitution.

Example 3.3.2 *Applying substitution $\sigma = \{x := a\}$ to term $f(x, b)$ is denoted by $f(x, b)\sigma$ and results in term $f(a, b)$*

Unification is the process of solving the satisfiability problem: given s and t, find a substitution σ such that $t\sigma = s\sigma$ (i.e. $t\sigma$ and $s\sigma$ are syntactically identical.

Example 3.3.3 *Given the equation, $f(x) \stackrel{?}{=} f(a)$ then $\sigma = \{x := a\}$.*

In the former example there is only one unifier for the equation but there are cases where several unifiers exist.

Example 3.3.4 *Given the equation* $x \stackrel{?}{=} f(y)$ *then* $\sigma = \{x := f(y)\}$ *and* $\sigma' = \{x := f(a), y := a\}$ *are both unifiers of the equation.*

Definition 3.3.4 *A substitution* σ *is* more general *than a substitution* σ' *if there is a substitution* δ *such that* $\sigma' = \delta\sigma$*. We also say that* σ' *is an* instance *of* σ*.*

Note that for example 3.3.4 the substitution $\sigma = \{x := f(y)\}$ is more general than $\sigma' = \{x := f(a), y := a\}$ since there is a $\delta = \{y := a\}$ such that $\sigma' = \delta\sigma$.

Definition 3.3.5 *A* unification problem *is a finite set of equations* $S = \{s_1 \stackrel{?}{=} t_1, \ldots, s_n = t_n\}$*. A* unifier *of* S *is a substitution* σ *such that* $\sigma s_1 \stackrel{?}{=} \sigma t_1, \ldots \sigma s_n \stackrel{?}{=} \sigma t_n$*.*

Definition 3.3.6 *Given a unification problem* S*, let* $\mathcal{U}(S)$ *be the set of unifiers for* S*, then the* most general unifier *is such that:*

- $\sigma \in \mathcal{U}(S)$.
- $\forall \sigma' \in \mathcal{U}(S)$, σ *is more general than* σ'.

Example 3.3.5 *It is easy to check that* $\sigma = \{x := y\}$ *is a most general unifier of* $x \stackrel{?}{=} y$*.*

3.3.1 Unification algorithm

The syntactic unification algorithm uses as its core the rewrite rules presented in figure 3.3.1.

Delete	$\{t \stackrel{?}{=} t\} \uplus S$	$\Longrightarrow$	S
Decompose	$\{f(t_1, \ldots, t_n) \stackrel{?}{=} f(s_1, \ldots, s_n)\} \uplus S$	$\Longrightarrow$	$\{t_1 \stackrel{?}{=} s_1, \ldots, t_n \stackrel{?}{=} s_n\} \cup S$
Orient	$\{t \stackrel{?}{=} x\} \uplus S$	$\Longrightarrow$	$\{x \stackrel{?}{=} t\} \cup S$
Eliminate	$\{x \stackrel{?}{=} t\} \uplus S$	$\Longrightarrow$	$\{x \stackrel{?}{=} t\} \cup \{x := t\}(S)$ if $x \notin \mathcal{V}ar(t)$.

Figure 3.1: Rules for unification

Note that the application of a substitution to S means applying to both sides of all the equations in S and that $\uplus$ denotes disjoint union.

Example 3.3.6 *Given the set unification problem* $\{x \stackrel{?}{=} f(a), g(x, x) \stackrel{?}{=} g(x, y)\}$ *one can apply the rules described for unification to obtain a most general unifier:*

$$\begin{array}{lll}
\{x \stackrel{?}{=} f(a), g(x,x) \stackrel{?}{=} g(x,y)\} & \Longrightarrow & \textit{Eliminate} \\
\{x \stackrel{?}{=} f(a), g(f(a),f(a)) \stackrel{?}{=} g(f(a),y)\} & \Longrightarrow & \textit{Decompose} \\
\{x \stackrel{?}{=} f(a), f(a) \stackrel{?}{=} f(a), f(a) \stackrel{?}{=} y\} & \Longrightarrow & \textit{Delete} \\
\{x \stackrel{?}{=} f(a), f(a) \stackrel{?}{=} y\} & \Longrightarrow & \textit{Orient} \\
\{x \stackrel{?}{=} f(a), y \stackrel{?}{=} f(a)\}. & &
\end{array}$$

Definition 3.3.7 *A unification problem* $S = \{x_1 \stackrel{?}{=} t_1, \ldots, x_n \stackrel{?}{=} t_n\}$ *is in* solved form *if the* x_i *are pairwise distinct variables, none of which occurs in any of the* t_i.

Definition 3.3.8 *The unification algorithm is defined by:*

$$\begin{array}{lll}
Unify(S) & = & \textit{while there is some } T \textit{ such that } S \Longrightarrow T, \textit{ replace } S \textit{ by } T. \\
& & \textit{If } S \textit{ is in solved form then return } S \textit{ else fail.}
\end{array}$$

Lemma 3.3.1 *Unify terminates on all inputs.*

Lemma 3.3.2 *If* S *is solvable then* $Unify(S)$ *does not fail.*

Proofs for these results and further details can be found in [BN98].

3.4 Flexible Arity Unification

Unification in a theory with sequence variables and flexible arity function symbols is a recent topic of research with applications in several relevant areas in computer science. In this chapter we present basic concepts related with unification with sequence variables and flexible arity function symbols.

We refer to this kind of unification shortly as unification with sequence variables and flexible arity symbols, underlying the importance of these two syntactic elements. Sequence variables are variables which can be instantiated by an arbitrary finite (possible empty) sequence of terms. Flexible arity function symbols can have arbitrary finite (possible empty) number of arguments.

Here we present this new kind of unification along the lines presented in [Kut02c]. Consider an alphabet consisting of the following sets: the set of individual variables, the set of sequence variables, the set of constants, and the set of flexible arity function symbols.

Definition 3.4.1 *The set of terms over the previous alphabet is the smallest set that satisfies the following conditions:*

1. *Constants, individual variables (here denoted by* $\mathcal{IV}$*) and sequence variables (here denoted by* $\mathcal{SV}$*) are terms.*

2. *Given the flexible arity function symbol* f *and terms* $t_1, \ldots, t_n$ $(n \geq 0)$*, then* $f(t_1, \ldots, t_n)$ *is a term.*

3. *If* f *is a fixed arity function symbol with arity* n*,* $n \geq 0$ *and* $t_1, \ldots, t_n$ *are terms such that for all* $1 \leq i \leq n$*,* t_i *does not contain sequence variables as subterms, then* $f(t_1, \ldots, t_n)$ *is a term.*

Definition 3.4.2 *A* sequence $\tilde{t}$*, is defined as follows:*

- ε *is the empty sequence.*
- $\ulcorner t_1, \tilde{t} \urcorner$ *is a sequence if* t_1 *is a term and* $\tilde{t}$ *is a sequence.*

Example 3.4.1 *Given the terms* $f(a)$*,* b *and* X*, then* $\tilde{t} = \ulcorner f(a), b, X \urcorner$ *is a sequence.*

Definition 3.4.3 (Equation) *If* t_1 *and* t_2 *are terms such that neither* t_1 *nor* t_2 *is a sequence variable, then* $t_1 \stackrel{?}{=} t_2$ *is an equation.*

Definition 3.4.4 (Substitution) *A substitution is a finite set* $\{x_1 := s_1, \ldots, x_n := s_n, X_1 := \ulcorner t^1_1, \ldots, t^1_{k_1} \urcorner, \ldots, X_m := \ulcorner t^1_m, \ldots, t^m_{k_m} \urcorner\}$ *where:*

- $n \geq 0, m \geq 0$ *and* $k_i \geq 0$ *for all* $1 \leq i \leq m$*,*
- $x_1, \ldots, x_n$ *are distinct individual variables,*
- $X_1, \ldots, X_m$ *are distinct sequence variables,*
- *for all* $1 \leq i \leq n, s_i$ *is a term,* s_i *is not a sequence variable and* $s_i \neq x_i$*,*
- *for all* $1 \leq i \leq m$*,* $\ulcorner t^i_1, \ldots, t^i_{k_i} \urcorner$ *is a sequence of terms and if* $k_i = 1$ *then* $t^i_{k_i} \neq X_i$*.*

Definition 3.4.5 (Instance) *Given a substitution* θ*, we define an instance of a term or equation with respect to* θ *recursively as follows:*

- $x\theta = \begin{cases} s, & \text{if } x := s \in \theta, \\ x, & \text{otherwise} \end{cases}$
- $X\theta = \begin{cases} \ulcorner s_1, \ldots, s_m \urcorner, & \text{if } X := \ulcorner s_1, \ldots, s_m \urcorner \in \theta, m \geq 0, \\ X, & \text{otherwise} \end{cases}$
- $f(s_1, \ldots, s_n)\theta = f(s_1\theta, \ldots, s_n\theta)$
- $(s_1 \stackrel{?}{=} s_2)\theta = s_1\theta \stackrel{?}{=} s_2\theta$.

We extend the notion of instance to sequences in a straightforward way: instance of a sequence is a sequence of instances. With this extension, definition of substitution composition is also straightforward.

Example 3.4.2 *Given the flexible arity sequence* $f(a, X, c, d)$ *and the set of substitutions* $\{X := \varepsilon\}$ *the resulting term is* $f(a, c, d)$

Example 3.4.3 *Given the flexible arity sequence* $f(a, X, c, d)$ *and the set of substitutions* $\{X := \ulcorner a, a \urcorner\}$ *the resulting term is* $f(a, a, a, c, d)$.

Example 3.4.4 *Given the two sets,* $\{X := \ulcorner f(x, Y), Y, f(X) \urcorner, Y := Z\}$ *and* $\{Y := \varepsilon, x := g(W), Z := Y, X := a\}$, *the composition of both results in* $\{X := \ulcorner f(g(W)), f(a) \urcorner, x := g(W), Z := Y\}$.

Definition 3.4.6 (Unification Problem, Unifier) *A unification problem is a set of equations* $\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$. *A substitution* θ *is called an unifier of an unification problem* $\{s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n\}$ *iff* $s_i\theta = t_i\theta$ *for all* $1 \leq i \leq n$.

Definition 3.4.7 (More General Substitution) *A substitution* θ *is more general than a substitution* σ *iff there exists a substitution* λ *such that* $X := \varepsilon \notin \lambda$ *for any sequence variable* X, *and* $\theta\lambda = \sigma$.

Note that $\{X := Y\}$ is more general than $\{X := \ulcorner a, Z \urcorner, Y := \ulcorner a, Z \urcorner\}$ but not more general than $\{X := \varepsilon, Y := \varepsilon\}$.

Definition 3.4.8 (Minimal Complete Set of Unifiers) *A* Minimal complete set of unifiers *of a unification problem* Γ, $mcu(\Gamma)$, *is a set of substitutions such that:*

1. *Every substitution in* $mcu(\Gamma)$ *is an unifier of* Γ.
2. *For any unifier* σ *of* Γ, *there is a* $\theta \in mcu(\Gamma)$ *such that* θ *is more general than* σ.
3. *For all* $\theta, \sigma \in mcu(\Gamma)$, *if* θ *is more general than* σ *then* $\theta = \sigma$.

Example 3.4.5 *A minimal complete set of unifiers of the unification problem*

$$\{f(g(a, X), g(Y, c)) \stackrel{?}{=} f(U, g(b, V))\}$$

is: $\{\{U := g(a, X),\ Y := b,\ V := c\}, \{X := \varepsilon,\ U := g(a),\ Y := b,\ V := c\}, \{U := g(a, X),\ Y := \ulcorner b, Y \urcorner,\ V := \ulcorner Y, c \urcorner\}, \{X := \varepsilon,\ U := g(a),\ Y := \ulcorner b, Y \urcorner,\ V := \ulcorner Y, c \urcorner\}\}$

3.5 Unification of Flexible Arity Terms

Here we recall the unification procedure presented in [Kut02c], to solve problems of the form $t_1 \stackrel{?}{=} t_2$ built over an alphabet which consists of sequence and individual variables, free flexible arity function symbols, free constants and free fixed arity function symbols. The unification procedure is divided in two parts, *Projection* and *Transformation*.

3.5.1 Projection

The idea of *Projection* is to eliminate sequence variables from a given unification problem.

Definition 3.5.1 *For an equation* $s \stackrel{?}{=} t$*, the set of* projecting substitutions *is defined as follows:*

$$\Pi(s \stackrel{?}{=} t) = \{\vartheta \mid \mathcal{D}om(\vartheta) \subseteq \mathcal{V}_S(\tilde{s} \stackrel{?}{=} t) \text{ and } X\vartheta = \varepsilon \text{ for all } X \in \mathcal{D}om(\vartheta)\}.$$

Example 3.5.1 *For* $f(X, a) \stackrel{?}{=} f(a, Y)$ *the set of projecting substitutions is* $\{\varepsilon, \{X \mapsto \varepsilon\}$, $\{Y \mapsto \varepsilon\}$, $\{X \mapsto \varepsilon,\ Y \mapsto \varepsilon\}\}$.

3.5.2 Transformation

We now present the transformation rules. Given an unification problem UP, each of the transformation rules for unification have one of the following forms:

- $UP \rightsquigarrow \bot$
- $UP \rightsquigarrow \langle\langle SUC_1, \sigma_1\rangle, \ldots, \langle SUC_n, \sigma_n\rangle\rangle$

Where each of the SUC_i is a either $\top$ or a new unification problem and each of the σ_i is a set of substitutions. The set of transformation rules is given in figure 3.5.2. Note that $\top$ stands for success and $\bot$ for failure.

In [Jaf90, Sch93] tree generation is used for solving word equations. Here, the idea is similar for the unification procedure with sequence variables and flexible arity symbols. Projection and transformation can be seen as single steps in a tree generation process. Each node of the tree is labeled either with a unification problem (non-terminal node),$\top$ or $\bot$ (terminal nodes). The edges of the tree are labeled by substitutions. The children of a non-terminal node are constructed in the following way: given a non-terminal node with unification problem UP, in case it is unifiable, we apply projection or transformation and get $\langle\langle SUC_1, \sigma_1\rangle, \ldots, \langle SUC_n, \sigma_n\rangle\rangle$. Thus, the node for the unification UP has n children and for each SUC_i its edge is σ_i.

Decidability of unification with sequence variables was shown in [Kut02b], and a minimal complete unification procedure was presented in [Kut02b, Kut02c]. It is a rule-based procedure that runs in a breadth-first manner, enumerates a minimal complete set of unifiers for a given problem, and terminates if this set is finite. The example below demonstrates how it works:

Example 3.5.2 *Figure 3.2 shows successful derivations for the unification problem* $\Gamma = f(x, b, Y, f(X)) \stackrel{?}{=} f(a, X, f(b, Y))$. $mcu(\Gamma) = \{\{x := a,\ X := \ulcorner b, X\urcorner, Y := X\},\ \{x := a,\ X := b, Y := \varepsilon\}\}$.

Success

(1) $t \stackrel{?}{=} t \rightsquigarrow \langle\langle \top, \varepsilon \rangle\rangle.$

(2) $x \stackrel{?}{=} t \rightsquigarrow \langle\langle \top, \{x \leftarrow t\} \rangle\rangle,$ if $x \notin vars(t).$

(3) $t \stackrel{?}{=} x \rightsquigarrow \langle\langle \top, \{x \leftarrow t\} \rangle\rangle,$ if $x \notin vars(t).$

Failure

(4) $c_1 \stackrel{?}{=} c_2 \rightsquigarrow \bot,$ if $c_1 \neq c_2.$

(5) $x \stackrel{?}{=} t \rightsquigarrow \bot,$ if $t \neq x$ and $x \in vars(t).$

(6) $t \stackrel{?}{=} x \rightsquigarrow \bot,$ if $t \neq x$ and $x \in vars(t).$

(7) $f_1(\tilde{t}) \stackrel{?}{=} f_2(\tilde{s}) \rightsquigarrow \bot,$ if $f_1 \neq f_2.$

(8) $f() \stackrel{?}{=} f(t_1, \tilde{s}) \rightsquigarrow \bot.$

(9) $f(t_1, \tilde{s}) \stackrel{?}{=} f() \rightsquigarrow \bot.$

(10) $f(X, \tilde{t}) \stackrel{?}{=} f(s_1, \tilde{s}) \rightsquigarrow \bot,$ if $s_1 \neq X$ and $X \in vars(s_1).$

(11) $f(s_1, \tilde{s}) \stackrel{?}{=} f(X, \tilde{t}) \rightsquigarrow \bot,$ if $s_1 \neq X$ and $X \in vars(s_1).$

(12) $f(t_1, \tilde{s}) \stackrel{?}{=} f(s_1, \tilde{t}) \rightsquigarrow \bot,$ if $s_1 \stackrel{?}{=} t_1 \rightsquigarrow \bot.$

Eliminate

(13) $f(t_1, \tilde{t}) \stackrel{?}{=} f(s_1, \tilde{s}) \rightsquigarrow \langle\langle g(\tilde{t}\sigma) \stackrel{?}{=} g(\tilde{s}\sigma) \rangle\rangle,$ if $s_1 \stackrel{?}{=} t_1 \rightsquigarrow \langle\langle \top, \sigma \rangle\rangle.$

(14) $f(X, \tilde{t}) \stackrel{?}{=} f(X, \tilde{s}) \rightsquigarrow \langle\langle g(\tilde{t}) \stackrel{?}{=} g(\tilde{s}) \rangle\rangle.$

(15) $f(X, \tilde{t}) \stackrel{?}{=} f(s_1, \tilde{s}) \rightsquigarrow$ if $s_1 \notin \mathcal{SV}$ and $X \notin vars(s_1),$
$\langle\langle g(t\tilde{\sigma_1}) \stackrel{?}{=} g(\tilde{s}\sigma_1), \sigma_1 \rangle,$ where $\sigma_1 = \{X \leftarrow s_1\},$
$\langle g(X, t\tilde{\sigma_2}) \stackrel{?}{=} g(\tilde{s}\sigma_2), \sigma_2 \rangle\rangle.$ $\sigma_2 = \{X \leftarrow \ulcorner s_1, X \urcorner\},$

(16) $f(s_1, \tilde{s}) \stackrel{?}{=} f(X, \tilde{t}) \rightsquigarrow$ if $s_1 \notin \mathcal{SV}$ and $X \notin vars(s_1),$
$\langle\langle g(\tilde{s}\sigma_1) \stackrel{?}{=} g(\tilde{t}\sigma_1), \sigma_1 \rangle,$ where $\sigma_1 = \{X \leftarrow s_1\},$
$\langle g(\tilde{s}\sigma_2) \stackrel{?}{=} g(X, \tilde{t}\sigma_2), \sigma_2 \rangle\rangle.$ $\sigma_2 = \{X \leftarrow \ulcorner s_1, X \urcorner\},$

(17) $f(X, \tilde{t}) \stackrel{?}{=} f(Y, \tilde{s}) \rightsquigarrow$ where
$\langle\langle g(\tilde{t}\sigma_1) \stackrel{?}{=} g(\tilde{s}\sigma_1), \sigma_1 \rangle,$ $\sigma_1 = \{X \leftarrow Y\},$
$\langle g(X, \tilde{t}\sigma_2) \stackrel{?}{=} g(\tilde{s}\sigma_2), \sigma_2 \rangle.$ $\sigma_2 = \{X \leftarrow \ulcorner Y, X \urcorner\},$
$\langle g(X, \tilde{t}\sigma_3) \stackrel{?}{=} g(Y, \tilde{s}\sigma_3), \sigma_3 \rangle\rangle.$ $\sigma_3 = \{Y \leftarrow \ulcorner X, Y \urcorner\}.$

Split

(18) $f(t_1, \tilde{t}) \stackrel{?}{=} f(s_1, \tilde{s}) \rightsquigarrow$ if $t_1, s_1 \notin \mathcal{IV} \cup \mathcal{SV}$ and ,
$\langle\langle f(r_1, t\tilde{\sigma_1}) \stackrel{?}{=} g(q_1, \tilde{s}\sigma_1), \sigma_1 \rangle, \ldots,,$ $t_1 \stackrel{?}{=} s_1 \rightsquigarrow \langle\langle r_1 \stackrel{?}{=} q_1, \sigma_1 \rangle, \ldots,$
$\langle\langle f(r_k, t\tilde{\sigma_k}) \stackrel{?}{=} g(q_k, \tilde{s}\sigma_k), \sigma_k \rangle, \ldots,,$ $\langle r_k \stackrel{?}{=} q_k, \sigma_k \rangle\rangle.$

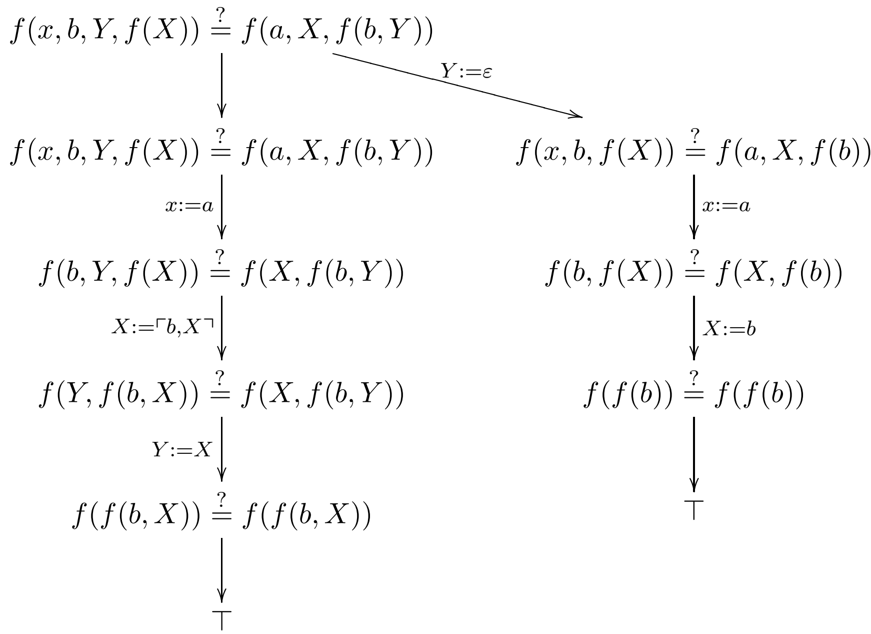

Figure 3.2: Successful derivations of unifiers for $\{f(x,b,Y,f(X)) \stackrel{?}{=} f(a,X,f(b,Y))\}$.

Example 3.5.3 *Given the unification problem* $\Gamma = \{f(X,a) \stackrel{?}{=} f(a,X)\}$, *the procedure enumerates the infinite minimal complete set of unifiers:* $mcu(\Gamma) = \{\{X := \varepsilon\}, \{X := a\}, \{X := \ulcorner a,a \urcorner\}, \{X := \ulcorner a,a,a \urcorner\}, \ldots\}$.

In [Kut02c] it is also shown that for a given unification problem U_p, the procedure terminates in the following cases:

- If no sequence variables occur in U_p the problem is reduced to Robinson unification.
- If for $U_p = t_1 \stackrel{?}{=} t_2$ one of t_1 or t_2 is ground (the problem is reduced to a matching problem).
- With cycle-checking if no individual or sequence variable occurs more than twice.
- With cycle-checking if for $U_p = t_1 \stackrel{?}{=} t_2$, where sequence variables occur only as arguments of t_1 or t_2 and occur at most twice.
- With sequence variable occurrence checking (by sequence variable occurrence checking we mean checking if for a problem $f(X) \stackrel{?}{=} f(t_1,\ldots,t_n), n > 1$, sequence variable X occurs in $f(t_1,\ldots,t_n)$) if all sequence variables occur only as the last elements of the terms they belong to.

We refer to [Kut02c] for details and proofs related to these notions.

3.6 Solving Quadratic Sequence Equations

In this section we present original work made in collaboration with Temur Kutsia.

In [Kut02c] it is shown that this unification problem is decidable.

Although decidable, unification with sequence variables and flexible arity symbols in the Siekmann unification hierarchy [Sie78] is infinitary: for any unification problem there exists a minimal complete set of unifiers which may be infinite for some problems.

In [Kut02c, Kut02a, KM04, Kut04] were defined unification procedures for this problem. As the unification problem is infinitary, previously defined complete procedures were non-terminating: for example, the equation $f(a, X) \doteq f(X, a)$ has infinitely many solutions $X \doteq \varepsilon$, $X \doteq a$, $X \doteq \ulcorner a, a \urcorner$, $X \doteq \ulcorner a, a, a \urcorner$, ...

Here we present a complete and terminating procedure, for the case where sequence variables don't occur more than twice in the unification problem.

Definition 3.6.1 *For a unification problem* Γ*, the set of* projecting substitutions *is defined as follows:*

$$\Pi(\Gamma) = \{\vartheta \mid \mathcal{D}om(\vartheta) \subseteq \mathcal{V}_S(\Gamma) \text{ and } X\vartheta = \varepsilon \text{ for all } X \in \mathcal{D}om(\vartheta)\}.$$

Example 3.6.1 *For* $\{f(\overline{x}, a) \doteq^? f(a, \overline{y})\}$ *the following set is the set of projecting substitutions:* $\{\varepsilon, \{\overline{x} \mapsto \varepsilon\}, \{\overline{y} \mapsto \varepsilon\}, \{\overline{x} \mapsto \varepsilon, \overline{y} \mapsto \varepsilon\}\}$.

Transformation rules:

T: **Trivial**

$\{s \doteq^? s\} \cup \Gamma \Longrightarrow_\varepsilon \Gamma.$

S: **Solve**

$\{x \doteq^? t\} \cup \Gamma \Longrightarrow_\sigma \Gamma,$ if $x \notin \mathcal{V}_I(t)$ and $\sigma = \{x \mapsto t\}$.

O1: **Orient 1**

$\{s \doteq^? x\} \cup \Gamma \Longrightarrow_\varepsilon \{x \doteq^? s\} \cup \Gamma$ if $s \notin \mathcal{V}_I$.

O2: **Orient 2**

$\{f(s_1, \tilde{s}) \doteq^? f(X, \tilde{t})\} \cup \Gamma \Longrightarrow_\varepsilon \{X \doteq^? s_1\} \cup \Gamma$ if $s \notin \mathcal{V}_S$.

FS: **First Argument Simplification**

$\{f(t_1, \tilde{s}) \doteq^? f(s_1, \tilde{t})\} \Longrightarrow_\varepsilon \{t_1 \doteq^? s_1, f(\tilde{s}) \doteq^? f(\tilde{t})\}$

SVE1: **Sequence Variable Elimination**

$\{f(X, \tilde{s}) \doteq^? f(X, \tilde{t})\} \cup \Gamma \Longrightarrow_\varepsilon \{f(\tilde{s}) \doteq^? f(\tilde{t})\} \cup \Gamma$

SVE2: **Sequence Variable Elimination**

$\{f(X, \tilde{s}) \doteq^? f(t, \tilde{t})\} \cup \Gamma \Longrightarrow_\sigma \{f(\tilde{s})\sigma \doteq^? f(\tilde{t})\sigma\} \cup \Gamma\sigma$ where $\sigma = \{X \mapsto t\}$, $X \notin \mathcal{V}_S(t)$.

W1: **Widening 1**

$$\{f(X,\tilde{s}) \doteq^? f(t,\tilde{t})\} \cup \Gamma \Longrightarrow_\sigma \{f(X,\tilde{s}\sigma) \doteq^? f(\tilde{t})\sigma\} \cup \Gamma\sigma,$$

where $\sigma = \{X \mapsto \ulcorner t, X \urcorner\}$, $X \notin \mathcal{V}_S(t)$.

W2: **Widening 2**

$$\{f(X,\tilde{s}) \doteq^? f(Y,\tilde{t}) \Longrightarrow_\sigma \{f(\tilde{s})\sigma \doteq^? f(Y,\tilde{t}\sigma) \cup \Gamma\sigma\}, \qquad \text{where } \sigma = \{Y \mapsto \ulcorner X, Y \urcorner\}.$$

The substitution ϑ used to make a step in the transformation rules is called a *local substitution.* Sometimes we may use the rule name abbreviation as subscripts and write, for instance, $\Gamma;\sigma \Longrightarrow_{\mathsf{W2}} \Delta;\sigma$ to indicate that Γ was transformed to Δ by W2. A *derivation* is a sequence $\Gamma_1;\sigma_1 \Longrightarrow \Gamma_2;\sigma_2 \Longrightarrow \cdots$ of transformations. A *selection strategy* $\mathcal{S}$ is a function which given a derivation $\Gamma_1;\sigma_1 \Longrightarrow \cdots \Longrightarrow \Gamma_n;\sigma_n$ returns an equation, called a *selected equation,* from Γ_n. A derivation is via a selection strategy $\mathcal{S}$ if in the derivation all choices of selected equations, being transformed by the transformation rules, are performed according to $\mathcal{S}$.

Proposition 3.6.1 *The transformation rules presented here are the same as the ones presented in [Kut02c] up to the following changes:*

- *The success rules are the same as the first three rules presented in this version.* T *and* S *are exactly the first and second ones presented in [Kut02c] and the combination of* O1 *and* S *is the same as the third success rule in [Kut02c].*
- *The failure rules are not needed here since we assume that any problem that doesn't match any rule fails.*
- *For the rules in Eliminate:*
  - *The first rule in Eliminate is the same as the* FAS.
  - *The second rule is the same as the* SVE1.
  - *The third rule is the same as combining* SVE2 *and* W1.
  - *The forth rule is the same as combining* O2*,* SVE2 *and* W1.
  - *The fifth rule is the same as combining* SVE2*,* W1 *and* W2.

A unification problem is called *quadratic* if no variable occurs in it more than twice. In this section we describe an algorithm that returns a finite representation of the solution set for quadratic problems. The finite representation is a finite automaton defining a language of substitutions. We now describe the algorithm in detail:

Note that it is easy to observe that if E_2 is an equation obtained from a quadratic equation E_1 by a rule in $\mathfrak{R}$, then E_2 is also quadratic.

We recall the definition of nondeterministic finite automata:

Definition 3.6.2 *An NFA is a tuple:*

$$A = (Q, \Sigma, P, S, F)$$

where:

1. *Q is a finite set of states.*
2. *Σ is a finite set of input symbols.*
3. *S, a member Q, is the start state.*
4. *F, a subset of Q, is the set of final states.*
5. *P is the set of transitions defined by δ, where δ is a function that takes a state in Q and an input symbol in Σ as arguments and returns a subset of Q.*

Algorithm 1 (Solving Quadratic Sequence Unification)
Input: A quadratic sequence unification problem Γ and a selection strategy $\mathcal{S}$.
Output: A finite automaton (Q, Σ, P, S, F).

1 **function** SolveQuadratic($\Gamma, \mathcal{S}$)
2 $\quad \Sigma := \Pi(\Gamma)$
3 $\quad Q := \{\Gamma\vartheta \mid \vartheta \in \Sigma\}$
4 $\quad P := \{\delta(\Gamma, \vartheta) = \Gamma\vartheta \mid \vartheta \in \Sigma\}$
5 $\quad S := \Gamma$
6 $\quad F := \top$
7 $\quad U := \{(\Gamma\vartheta, \{\Gamma\}) \mid \vartheta \in \Sigma \setminus \{\varepsilon\}\} \cup \{(\Gamma, \emptyset)\}$
8 $\quad$ **while** $U \neq \emptyset$ **do**
9 $\quad\quad$ Select a problem-history pair (Δ, h) from U.
10 $\quad\quad U := U \setminus \{\Delta, h\}$
11 $\quad\quad$ **for** each Δ' that can obtained from Δ by a rule R via $\mathcal{S}$:
12 $\quad\quad\quad\quad \langle \Delta;\ \varepsilon \rangle \Longrightarrow_{\mathsf{R}} \langle \Delta';\ \sigma \rangle$ **do**
13 $\quad\quad\quad \Sigma := \Sigma \cup \{\sigma\}$
14 $\quad\quad\quad Q := Q \cup \{\Delta'\}$
15 $\quad\quad\quad P := P \cup \{\delta(\Delta, \sigma) = \Delta'\}$
16 $\quad\quad\quad$ **if** $\mathsf{R} \in \{\mathsf{W1}, \mathsf{W2}\}$ **then**
17 $\quad\quad\quad\quad h' := h \cup \{\Delta\}$
18 $\quad\quad\quad\quad$ **if** $\Delta' \notin h'$ modulo symmetry of $\doteq$ **then** $U := U \cup \{(\Delta', h')\}$
19 $\quad\quad\quad$ **else**
20 $\quad\quad\quad\quad$ **if** $\mathsf{R} \in \{\mathsf{O1}, \mathsf{O2}\}$ **then** $h' := h$ **else** $h' := \emptyset$
21 $\quad\quad\quad\quad U := U \cup \{(\Delta', h')\}$
22 $\quad$ **return** (Q, Σ, P, S, F)

Example 3.6.2 *Given the unification problem $f(X, a) \doteq f(a, X)$, projection results in:*

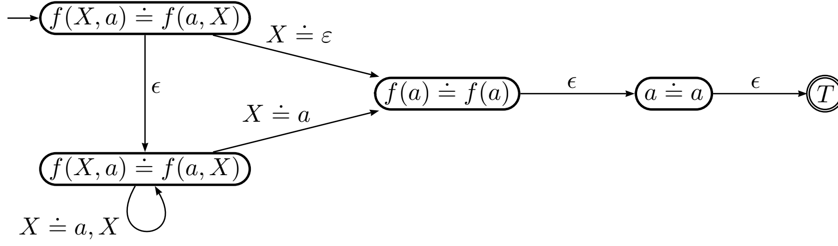

Figure 3.3: Automata generated for problem $f(X,a) \doteq f(a,X)$

- $\Sigma = \{\varepsilon, \{X \mapsto \varepsilon\}\}$
- $Q = \{f(a) = f(a), f(X,a) = f(a,X)\}$
- *P is defined by the following transitions:*
  - $\delta(f(X,a) \doteq f(a,X), \epsilon) \doteq (f(X,a) \doteq f(a,X))$
  - $\delta(f(X,a) \doteq f(a,X), X \doteq \varepsilon) \doteq (f(a) \doteq f(a))$
  - $\delta(f(X,a) \doteq f(a,X), X \doteq \ulcorner a, X \urcorner) \doteq (f(X,a) \doteq f(a,X))$
  - $\delta(f(X,a) \doteq f(a,X), X \doteq a) \doteq (f(a) \doteq f(a))$
  - $\delta(f(a) \doteq f(a), \epsilon) \doteq (a \doteq a)$
  - $\delta(a \doteq a, \epsilon) \doteq T\}$
- $S = \{f(X,a) \doteq f(a,X)\}$
- $F = T$

The generated automata is shown in figure 3.3.

We follow the definition presented in [HMU07] for $\hat{\delta}$:

Definition 3.6.3 *Given the transition function δ and the input string $w = xa$ where a is the last symbol of w, we define the extended transition function $\hat{\delta}$ as:*

$$\begin{array}{rcl} \hat{\delta}(q,\epsilon) & = & q \\ \hat{\delta}(q,w) & = & \delta(\hat{\delta}(q,x),a) \end{array}$$

Theorem 3.6.1 *Let e be a quadratic equation and let Sol be the solution set for e given by the transformation rules presented in [Kut02c]. Let p be a subpath in the solution tree for e leading to $\top$ and let $Solvequadratic(e) = \mathcal{A}$ be the automaton generated for the solution set of e. Then, there is a state U_p in $\mathcal{A}$ such that,*

$$\hat{\delta}(U_p, p) = \top \Leftrightarrow Up \Longrightarrow_\sigma \top, \textit{ where } p = \sigma.$$

Proof 3.6.1 *We divide the proof in two steps. First we will prove that $Up \Longrightarrow_\sigma \top \Rightarrow \hat{\delta}(U_p, p) = \top$. We use induction on the length of the path.*

Base cases: *It follows noting that the success rules are the same as the first three rules presented in this version of the algorithm.* Trivial *and* Solve *are exactly the first and second ones presented in [Kut02c] and the combination of* Orient *and* Solve *is the same as the third success rule in [Kut02c].*

Induction step: *Given* U_{p_n} *such that* $U_{p_n} \Longrightarrow_\sigma \top$ *with* $\sigma = \{\sigma_n, \ldots, \sigma_1\}$ *we have that* $\hat{\delta}(U_{p_n}, \sigma) = \top$*. Let's now prove that,* $U_{p_{n+1}} \Longrightarrow_{\sigma'} \top$*, where* $\sigma' = \{\sigma_{n+1}, \sigma_n, \ldots, \sigma_1\}$*. Since* $U_{p_{n+1}} \Longrightarrow_\sigma \top$ *is* $U_{p_{n+1}} \Longrightarrow_{\sigma_{n+1}} U_{p_n} \Longrightarrow_{\sigma_n} \ldots U_{p_1} \Longrightarrow_{\sigma_1} \top$ *and since the transformation rules in [Kut02c] and the new ones are the same by proposition 3.6.1 follows that* $\hat{\delta}(U_{p_{n+1}}, \sigma_{n+1}) = U_p$ *and thus, by definition of* $\hat{\delta}, \hat{\delta}(\delta(U_{p_{n+1}}, \sigma_{n+1}),\ \{\sigma_n, \ldots, \sigma_1\})$ $= \hat{\delta}(U_p, \{\sigma_n, \ldots, \sigma_1\})$*. The result follows from the induction hypothesis.*

Now we will prove that $\hat{\delta}(U_p, \sigma) = \top \Rightarrow Up \Longrightarrow_\sigma \top$*. We use induction on the length of* σ*.*

Base cases: *The first three rules in the new algorithm correspond to the three success rules presented in [Kut02c].*

Induction step: *Given that* $\hat{\delta}(U_p, \sigma) = \top \Rightarrow Up \Longrightarrow_\sigma \top$ *where* $\sigma = \{\sigma_n, \ldots, \sigma_1\}$ *we will prove that the result also holds for,* $\hat{\delta}(U_{p_{n+1}}, \sigma') = \top \Rightarrow U_{p_{n+1}} \Longrightarrow'_\sigma \top$ *where* $\sigma' = \{\sigma_{n+1}, \ldots, \sigma_1\}$*. Since* $\hat{\delta}(U_{p_{n+1}}, \sigma') = \hat{\delta}(\delta(U_{p_n}, \sigma_{n+1}), \{\sigma_n, \ldots, \sigma_1\})$ *and by direct observation that the original transformation rules and the new ones are the same up to notation rewriting (proposition 3.6.1) it is easy to conclude that* $\hat{\delta}(\delta(U_{p_{n+1}}, \sigma_{n+1}), \{\sigma_n, \ldots, \sigma_1\})$ $\Leftrightarrow \hat{\delta}(U_{p_n}, \{\sigma_n, \ldots, \sigma_1\})$ *and by the induction hypothesis, the result holds.*

Theorem 3.6.2 (Termination) *The algorithm* SolveQuadratic *terminates on any input.*

Proof 3.6.2 *This can be proved by associating with each equation-history pair* (U_p, h) *a complexity measure that is a tuple of natural numbers* $< n_1, n_2, n_3, n_4, n_5 >$*, where:*

- n_1 *is the sum of the sizes of equations in* Up*.*
- n_2 *is 1 if* Up *has a form* $\{s = x\} \cup \Gamma$*, where* s *is not an ind. variable. otherwise* $n_2 = 0$*.*
- n_3 *is 1 if* Up *has a form* $\{s = X\} \cup \Gamma$*, where* s *is not a seq. variable. otherwise* $n_3 = 0$*.*
- n_4 *is defined like this:*
  - *Let* $S(e) = \{eq | size(eq) = size(e) \text{ and } signature(eq) = signature(e)\}$ *be the factor set of the set of all equations with respect to the equivalence of equations (we say two equations are equivalent if they are variants of each other i.e. they differ only by variable names). The signature of* e *is the set of function symbols that occur in* e*. Note that* $S(e)$ *is finite.*
  - *For a pair* (U_p, h) *we always have* $h \subseteq \{S(e) | e \in Up\}$*. Therefore, let* n_4 *be* $|\{S(e) | e \in Up\} \setminus h|$*.*

- n_5 *is the sum of the depth_terms*(e) *for each equation* e *in* U_p *where depth_terms*(e) *is the sum of the depth of all subterms of* e*.*

Every rule strictly decreases the complexity measure:

T,S,SVE1,SVE2: *strictly decrease* n_1 *and leave all the others unchanged.*

O1: *strictly decreases* n_2 *and leaves all the others unchanged.*

O2: *strictly decreases* n_3 *and leaves all the others unchanged.*

FS: *strictly decreases* n_5 *and leave all the others unchanged.*

W1, W2: *If* (Up', h') *is obtained from* (Up, h) *by* $W1$ *or* $W2$*, we will have either* n_1 *decreasing, when the equation is found in the history and deleted from* U_p *or, when the equation is not in the history, it is added and* $|\{S(e')|e' \in Up'\} \backslash h'| < |\{S(e)|e \in Up\} \backslash h|$ *and thus* n_4 *decreases.*

3.7 Discussion

Here we presented CLP and introduced unification with sequence variables and flexible arity terms. One common mistake is to confuse this kind of unification with word unification [Sch93]. Note that word unification is a particular case of unification with sequence variables and flexible arity symbols: with only constants, sequence variables, and one flexible arity symbol. In flexible arity unification, one has as many flexible arity symbols as one wish and can have arbitrarily nested terms (that's impossible in word unification).

In [Kut04] it was shown that syntactic sequence unification can be considered as a special case of order-sorted higher-order E-unification.

We also added a new contribution to this field of research with a work made in collaboration with Temur Kutsia where we created a new terminating procedure for the case where no sequence variable occurs more than twice.

In the next chapter we will introduce a new CLP language based on flexible arity unification and directed to XML processing.

Chapter 4

Constraint Based XML Processing

4.1 Introduction

Here we present a general framework for XML processing based on constraint resolution. We define CLP(Flex), a constraint logic programing language in the domain of terms of flexible arity, and show its adequacy for XML processing. The idea behind CLP(Flex) is to extend Prolog with terms with flexible arity symbols and sequence variables. The novel features of this approach are the use of *flexible arity function symbols* and a corresponding mechanism for a *non-standard unification* in a theory with flexible arity symbols and variables which can be instantiated by an arbitrary finite sequence of terms. Since XML documents are denoted by terms with flexible arity symbols, applying CLP(Flex) to XML processing results in a substantial degree of flexibility in programming. For example, being able to extract sequences where a specific element has to exist is a standard procedure in every standard manipulation language for XML, but it can be rather tedious in logic programming, if the document is translated to a term with fixed arity. Our notion of unification of terms with function symbols of flexible arity overcomes this difficulty giving a quite declarative way of dealing with this cases in logic programming.

In this chapter we describe the syntax of CLP(Flex), its semantics, present briefly how traditional XML processing is done in Prolog, show how constraint solving is used and then present the unification algorithm with examples.

This work was partially presented in [CF04] and [CF05]. The web page of CLP(Flex) is:

`http://www.ncc.up.pt/~jcoelho/CLP(Flex).html`.

4.2 XML Processing in Prolog

We use a simple example to show how XML processing is usually implemented in Prolog and then show how to do it using CLP(Flex).

Example 4.2.1 *Let's suppose we have an XML document with a catalog of books like as following one:*

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <book number="1">
    <author>Donald Knuth</author>
    <name>Art of Computer Programming</name>
    <price>140</price>
    <year>1998</year>
  </book>
  ...
  <book number="500">
    <author>Simon Thompson</author>
    <name>Haskell: The Craft of Functional
                  Programming (2nd Edition)</name>
    <price>41</price>
    <year>1999</year>
  </book>
</catalog>
```

To get all the books with two or more authors using SWI-Prolog [Wie06] (which has a quite good library for processing XML in Prolog) we need the following code:

```
pbib([element(_,_,L)]):-
      pbib2(L).
pbib2([]).

pbib2([element('book',_,Cont)|Books]):-
      authors(Cont),!,pbib2(Books).
pbib2([_|Books]):-
      pbib2(Books).

authors([element('author',_,_),
             element('author',_,_)|R]):-
      write_name(R).

write_name([element('name',_,[N])]):-
      write(N),nl.
write_name([_|R]):-
      write_name(R).
```

Doing the same in CLP(Flex) where XML file is in variable BibDoc is much simpler. The following query is enough:

```
?- catalog(_,book(name(N),author(_),author(_),_),_) =*= BibDoc.
```

The simplicity and declarativeness of the second solution speaks by itself and we believe this can be a good alternative to the classical model.

4.3 Constraint Solving in CLP(Flex)

Terms in CLP(Flex) follow definition 3.4.1 from chapter 3. In CLP(Flex) we extend the domain of discourse of Prolog (trees over uninterpreted functors) with finite sequences of trees. Equality is the only relation between trees. Equality between trees is defined in the standard way: two trees are equal if and only if their root functor is the same and their corresponding subtrees, if any, are equal. CLP(Flex) programs have a syntax similar to Prolog extended with the new constraint $=*=$. The operational model of CLP(Flex) is the same of Prolog.

Definition 4.3.1 *If t_1 and t_2 are terms then $t_1 = t_2$ (standard Prolog unification) and $t_1 =*= t_2$ (unification of terms with flexible arity symbols) are constraints.*

Definition 4.3.2 *A constraint $t_1 = * = t_2$ or $t_1 = t_2$ is* solvable *if and only if there is an assignment of sequences or ground terms, respectively, to variables therein such that the constraint evaluates to* true*, i.e. such that and after that assignment the terms become equal.*

Remark 4.3.1 *In what follows, to avoid further formality, we shall assume that the domain of interpretation of variables is predetermined by the context where they occur. Variables occurring in a constraint of the form $t_1 =*= t_2$ are interpreted in the domain of sequences of trees, otherwise they are standard Prolog variables. In CLP(Flex) programs, therefore, each predicate symbol, functor and variable is used in a consistent way with respect to its domain of interpretation.*

Constraints of the form $t_1 =*= t_2$ are solved by a non-standard unification that calculates the corresponding minimal complete set of unifiers. This non-standard unification is based on Kutsia algorithm presented in chapter 3, section 3.5. As motivation we present some examples of unification:

Example 4.3.1 *Given the terms $a(X, b, Y)$ and $a(a, b, b, b)$ where X and Y are sequence variables, $a(X, b, Y) =*= a(a, b, b, b)$ gives three results:*

1. *$X = a$ and $Y = \ulcorner b, b \urcorner$*
2. *$X = \ulcorner a, b \urcorner$ and $Y = b$*
3. *$X = \ulcorner a, b, b \urcorner$ and $Y = \varepsilon$*

Recall that the notation $\ulcorner\urcorner$ is used only for readability reasons.

Example 4.3.2 *Given the terms $a(b, X)$ and $a(Y, d)$ where X and Y are sequence variables, $a(b, X) =*= a(Y, d)$ gives two possible solutions:*

1. *$X = d$ and $Y = b$*
2. *$X = \ulcorner N, d \urcorner$ and $Y = \ulcorner b, N \urcorner$ where N is a new sequence variable.*

4.4 XML Processing in CLP(Flex)

In CLP(Flex) there are some auxiliary predicates specific for XML processing. Through the following examples we will use the builtin predicates *xml2pro* and *pro2xml* which respectively convert XML files into terms and vice-versa. We will also use the predicate *newdoc(Root,Args,Doc)* where *Doc* is a term with functor *Root* and arguments *Args* (this predicate is similar to =.. in Prolog).

4.4.1 XML as Terms with Flexible Arity Symbols

An XML document is translated to a term with flexible arity function symbol. This term has a main functor (the root tag) and zero or more arguments. Although our actual implementation translates attributes to a list of pairs, since attributes do not play a relevant role in this work we will omit them in the examples, for the sake of simplicity. Consider the following simple XML file, describing an address book:

```
<addressbook>
        <record>
                <name>John</name>
                <address>New York</address>
                <email>john.ny@mailserver.com</email>
        </record>
        <record>
                <name>Sofia</name>
                <address>Rio de Janeiro</address>
                <phone>123456789</phone>
                <email>sofia.brasil@mail.br</email>
        </record>
</addressbook>
```

The equivalent term is:

```
addressbook(record(
               name('John'),
                address('New York'),
                email('john.ny@mailserver.com')),
             record(
                name('Sofia'),
                 address('Rio de Janeiro'),
                 phone('123456789'),
                 email('sofia.brasil@mail.br')))
```

4.4.2 Using Constraints in CLP(Flex)

One application of CLP(Flex) constraint solving is XML processing. With non-standard unification it is easy to handle parts of XML files. In this particular case, parts of terms representing XML documents.

Example 4.4.1 *Suppose that the term Doc is the CLP(Flex) representation of the document "addressbook.xml". If we want to gather the names of the people living in New York we can simply solve the following constraint:*

$$Doc = * = addressbook(_, record(name(N), address('New\ York'), _), _).$$

All the solutions can then be found by backtracking.

4.5 The Unification Algorithm

Unification of flexible arity terms is the constraint-solving method for constraints of the form =*= in CLP(Flex). The unification algorithm, as presented in chapter 3 section 3.5, consists of two main steps, *Projection* and *Transformation*. The first step, *Projection* is where some variables are erased from the sequence. This is needed to obtain solutions where those variables are instantiated by the empty sequence. The second step, *Transformation* is defined by a set of rules where the non-standard unification is translated to standard Prolog unification.

Definition 4.5.1 *Given terms t_1 and t_2, let V be the set of variables of T_1 and T_2 and A be a subset of V.* Projection *eliminates all variables of A in t_1 and t_2.*

Example 4.5.1 *Let $t_1 = f(b, Y, f(X))$ and $t_2 = f(X, f(b, Y))$ then $V = \{X, Y\}$ and let A be one element of the set $\{\{\}, \{X\}, \{Y\}, \{X, Y\}\}$. In the projection step we obtain the following cases (corresponding to $A = \{\}$, $A = \{X\}$, $A = \{Y\}$ and $A = \{X, Y\}$):*

- $t_1 = f(b, Y, f(X)), t_2 = f(X, f(b, Y))$
- $t_1 = f(b, Y, f), t_2 = f(f(b, Y))$
- $t_1 = f(b, f(X)), t_2 = f(X, f(b))$
- $t_1 = f(b, f), t_2 = f(f(b))$

Our version of Kutsia algorithm 3.5 uses a special kind of terms with fixed arity 2, here called, *sequence terms* for the implementation of sequences of arguments.

Definition 4.5.2 *A* sequence term, *$\bar{t}$ is defined as follows:*

- *ε is a* sequence term.
- *$seq(t, \bar{s})$ is a* sequence term *if t is a term and $\bar{s}$ is a* sequence term.

Definition 4.5.3 *Given a sequence term $\bar{t}$, the length of $\bar{t}$ is:*

$$\begin{array}{lcl} length(\varepsilon) & = & 0 \\ length(seq(t_1,\bar{t})) & = & 1 + length(\bar{t}) \end{array}$$

Definition 4.5.4 *A sequence term in* normal form *is defined as:*

- *ε is in normal form*
- *$seq(t_1, \bar{t})$ is in normal form if t_1 is not of the form $seq(s_1, \bar{s})$ or ε and $\bar{t}$ is in normal form.*

Example 4.5.2 *Given the function symbol f, the variable X and the constants a and b:*

$$seq(f(seq(a, \varepsilon)), seq(b, seq(X, \varepsilon)))$$

is a sequence term in normal form.

Note that sequence terms are lists and sequence terms in normal form are flat lists. We introduced this different notation because sequence terms are going to play a key role in our implementation of the algorithm and it is important to distinguish them from standard Prolog lists. Sequence terms in normal form correspond trivially to the definition of sequence presented in definition 3.4.2. In fact sequence terms in normal form are an implementation of this definition. Thus, in our implementation, a term $f(t_1, t_2, \ldots, t_n)$, where f has flexible arity, is internally represented as $f(seq(t_1, seq(t_2, \ldots, seq(t_n, \varepsilon) \ldots)))$, that is, arguments of functions of flexible arity are always represented as elements of a sequence term.

We now define a normalization function to reduce sequence terms to their normal form.

Definition 4.5.5 *Given the sequence terms $\bar{t}$ and $\bar{s}$, we define sequence term* concatenation *as $\bar{t} + +\bar{s}$, where the $++$ operator is defined as follows:*

$$\begin{array}{lclcl} \varepsilon & ++ & \bar{t} & = & \bar{t} \\ seq(t_1,\bar{t}) & ++ & \bar{s} & = & seq(t_1,\bar{t}++\bar{s}) \end{array}$$

Definition 4.5.6 *Given a sequence term, we define sequence term* normalization *as:*

$$\begin{array}{lcl} normalize(\varepsilon) & = & \varepsilon \\ normalize(t) & = & seq(t, \varepsilon), \text{ if } t \text{ is a constant or variable.} \\ normalize(t) & = & seq(f(normalize(t_1)),\varepsilon), \text{ if } t = f(t_1). \\ normalize(seq(t_1,\bar{t})) & = & normalize(t_1) \; ++ \; normalize(\bar{t}) \end{array}$$

Lemma 4.5.1 *The normalization procedure always terminates yielding a sequence in normal form.*

Proof 4.5.1 *This can be easily proved by induction on the length of the sequence.*
The base case is trivial:

$$normalize(\varepsilon) = \varepsilon$$
$$normalize(t) = seq(t,\varepsilon) \quad \textit{if } t \textit{ is a constant or variable.}$$

The induction hypothesis can be directly applied for the third case. By definition normalize $(seq(t, X)) = normalize(t){+}{+}normailze(X)$ *and by the induction hypothesis both normalize* (t) *and normailze* (X) *terminate.* $normalize(t)$ ++ $normailze(X)$ *also terminates by the termination of* ++ *(which can be proved by induction on the length of the first argument of* ++*).*

Transformation rules are defined in figure 4.1. We consider that upper case letters (X,Y,...) stand for sequence variables, lower case letters (s,t,...) for terms and overlined lower case letters ($\bar{t}, \bar{s}$) for sequence terms. These rules implement Kutsia algorithm using standard Prolog unification. Note that rules 6, 7, 8 and 9 are non-deterministic: for example rule 6 states that in order to solve $seq(X, \bar{t}) = * = seq(s_1, \bar{s})$ we can solve $\bar{t} = * = \bar{s}$ with $X = s_1$ or we can solve $normalize(seq(X_1, \bar{t})) = * = normalize(\bar{s})$ with $X = seq(s_1, seq(X_1, \varepsilon))$. At the end the solutions given by the algorithm are normalized by the *normalize* function. When none of the rules is applicable the algorithm fails. Note that in rules of the form $H \Longrightarrow B_1, \ldots, B_n$ the operator "," means the logical conjunction and the fixed computation rule used in Prolog is assumed: in $B_1, \ldots, B_n$ always select the leftmost equation first. Both Kutsia's algorithm and the new one we present here, define a tree of solutions like the one presented in figure 3.2. The difference is that in Kutsia's approach a breath-first strategy is used to get all the solutions and in our approach a depth-first strategy with backtracking (such as in Prolog) is used.

Note that depth-search is incomplete. Consider the following example:

Example 4.5.3 *Given the unification problem* $f(X, a, Y, b) = * = f(a, X, b, Y)$ *the depth first strategy is unable to find all the solutions. The results using a depth-first strategy are:*

- $X = \varepsilon, Y = \varepsilon$
- $X = a, Y = \varepsilon$
- $X = \ulcorner a, a \urcorner, Y = \varepsilon$
- $X = \ulcorner a, a, a \urcorner, Y = \varepsilon$
- $X = \ulcorner a, a, a, a \urcorner, Y = \varepsilon$
- ...

Solutions like $Y = b$*,* $Y = \ulcorner b, b \urcorner$*,* $Y = \ulcorner b, b, b \urcorner$*, ... are not found with this search strategy.*

Although depth-search is an incomplete search strategy it has several advantages:

- It is more efficient since space complexity is lower.

- We use Prolog as our base language and Prolog uses depth-first. Naturally we rely on the same strategy.
- Flexible arity terms unification in the XML processing context does not typically output an infinite number of solutions since documents are finite, thus depth-first is the better strategy for this specific case.

Kutsia showed in [Kut02c] that his algorithm terminated if it had a cycle check, (i.e. it stopped with failure if a unification problem gave rise to a similar unification problem) and if each sequence variable does not occur more than twice in a given unification problem. We also have the same restriction in the number of occurrences of a variable but we don't need to implement the cycle check since we use Prolog backtracking.

For the sake of simplicity, the following examples are presented in sequence notation, alternatively to the sequence term notation.

Example 4.5.4 *Given* $\bar{t} = f(X, b, Y)$ *and* $\bar{s} = f(c, c, b, b, b, b)$ *the projection step leads to the following transformation cases:*

- $f(X, b, Y) = * = f(c, c, b, b, b, b)$
- $f(b, Y) = * = f(c, c, b, b, b, b)$
- $f(X, b) = * = f(c, c, b, b, b, b)$
- $f(b) = * = f(c, c, b, b, b, b)$

Using the transformation rules we can see that only the first and third unifications succeed. For $f(X, b, Y) = * = f(c, c, b, b, b, b)$ *we have the following answer substitutions:*

- $X = \ulcorner c, c \urcorner$ *and* $Y = \ulcorner b, b, b \urcorner$
- $X = \ulcorner c, c, b \urcorner$ *and* $Y = \ulcorner b, b \urcorner$
- $X = \ulcorner c, c, b, b \urcorner$ *and* $Y = b$

And for $f(X, b) = * = f(c, c, b, b, b, b)$ *we have:*

- $X = \ulcorner c, c, b, b, b \urcorner$
- $Y = \varepsilon$

Example 4.5.5 *Given* $\bar{t} = f(b, Y, f(X))$ *and* $\bar{s} = f(X, f(b, Y))$ *we have the following equations after projection:*

- $f(b, Y, f(X)) = * = f(X, f(b, Y))$

Success					
(1)	t	$=*=$	s	$\Longrightarrow$	True, if $t == s$ [1]
(2)	X	$=*=$	t	$\Longrightarrow$	$X = t$ if X does not occur in t.
(3)	t	$=*=$	X	$\Longrightarrow$	$X = t$ if X does not occur in t.
Eliminate					
(4)	$f(\bar{t})$	$=*=$	$f(\bar{s})$	$\Longrightarrow$	$\bar{t} =*= \bar{s}$
(5)	$seq(t_1, \bar{t})$	$=*=$	$seq(s_1, \bar{s})$	$\Longrightarrow$	$t_1 =*= s_1$, $\bar{t} =*= \bar{s}$
(6)	$seq(X, \bar{t})$	$=*=$	$seq(s_1, \bar{s})$	$\Longrightarrow$	$X = s_1$, if X does not occur in s_1, $\bar{t} =*= \bar{s}$.
				$\Longrightarrow$	$X = seq(s_1, seq(X_1, \varepsilon))$, if X does not occur in s_1, $normalize(seq(X_1, \bar{t})) =*= normalize(\bar{s})$.
(7)	$seq(t_1, \bar{t})$	$=*=$	$seq(X, \bar{s})$	$\Longrightarrow$	$X = t_1$, if X does not occur in t_1, $\bar{t} =*= \bar{s}$.
				$\Longrightarrow$	$X = seq(t_1, seq(X_1, \varepsilon))$, if X does not occur in t_1. $normalize(\bar{t}) =*= normalize(seq(X_1, \bar{s}))$,
(8)	$seq(X, \bar{t})$	$=*=$	$seq(Y, \bar{s})$	$\Longrightarrow$	$X = Y$ $\bar{t} =*= \bar{s}$.
				$\Longrightarrow$	$X = seq(Y, seq(X_1, \varepsilon)$, where $normalize(seq(X_1, \bar{t})) =*= normalize(\bar{s})$, X, Y and X_1 are distinct variables.
				$\Longrightarrow$	$Y = seq(X, seq(Y_1, \varepsilon))$, where $normalize(\bar{t}) =*= normalize(seq(Y_1, \bar{s}))$, X, Y and Y_1 are distinct variables.
Split					
(9)	$seq(t_1, \bar{t})$	$=*=$	$seq(s_1, \bar{s})$	$\Longrightarrow$	if $t_1 =*= s_1 \Longrightarrow r_1 =*= q_1$ then $normalize(seq(r_1, \bar{t}))$ $=*=$ $normalize(seq(q_1, \bar{s}))$ $\vdots$
				$\Longrightarrow$	if $t_1 =*= s_1 \Longrightarrow r_w =*= q_w$, $normalize(seq(r_w, \bar{t}_n))$ $=*=$ $normalize(seq(q_w, \bar{s}))$, where t_1 and s_1 are compound terms.

Figure 4.1: Transformation rules

- $f(b, Y, f) = * = f(f(b, Y))$
- $f(b, f(X)) = * = f(X, f(b))$
- $f(b, f) = * = f(f(b))$

The first and third cases succeed with solution $X = \ulcorner b, Y \urcorner$ *and* $X = b, Y = \varepsilon$ *respectively.*

Example 4.5.6 *In some cases we can have an infinite set of solutions for the unification of two given terms. For example when we solve* $f(X, a) = * = f(a, X)$ *the solutions are:*

- $X = a$
- $X = \ulcorner a, a \urcorner$
- $X = \ulcorner a, a, a \urcorner$
- $X = \ulcorner a, a, a, a \urcorner$
- $\ldots$

In the previous example Kutsia algorithm with the cycle check fails immediately after detecting that it is repeating the unification problem. In the first case we have $X = a$ and in the second $X = a, X_1$ leading to the new problem $X_1, a = * = a, X_1$ which is exactly the same as the original problem.

Note that the transformation rules in figure 4.1 deal with sequence terms. This increases the expressiveness of CLP(Flex) enabling the explicit use of sequence terms in XML processing.

4.5.1 Correctness

We now prove the correctness of our implementation of Kutsia algorithm. In [Kut02c] Kutsia proved the correctness of his algorithm with respect to a given semantics for the non-standard unification. We show that our implementation of Kutsia algorithm is correct, i.e, both give the same set of solutions for a given equation. Before presenting soundness and completeness results we introduce some useful definitions.

Definition 4.5.7 *Given a sequence term* $\bar{t}$. $\mathcal{T}$ *translates* $\bar{t}$ *into a sequence* $\tilde{t}$ *(as defined by Kutsia in [Kut02c]).* $\mathcal{T}$ *is defined as:*

$$
\begin{array}{rcll}
\mathcal{T}(\varepsilon) & = & \varepsilon & \\
\mathcal{T}(t) & = & t & \textit{if } t \textit{ is a constant or variable} \\
\mathcal{T}(f(\bar{t})) & = & f(\mathcal{T}(\bar{t})) & \textit{if } f \textit{ is function symbol and } f \neq seq \\
\mathcal{T}(seq(A, B)) & = & \mathcal{T}(A), \mathcal{T}(B) &
\end{array}
$$

[1]== denotes syntactic equality (in opposite with = which denotes standard unification)

Definition 4.5.8 *Given a sequence $\tilde{t}$. $\mathcal{T}^{-1}$ translates $\tilde{t}$ into a sequence term $\bar{t}$. $\mathcal{T}^{-1}$ is defined as:*

$$\begin{array}{rcl} \mathcal{T}^{-1}(\varepsilon) & = & \varepsilon \\ \mathcal{T}^{-1}(t) & = & t, \text{ if } t \text{ is a constant or variable} \\ \mathcal{T}^{-1}(f(\tilde{t})) & = & f(\mathcal{T}^{-1}(\tilde{t})) \\ \mathcal{T}^{-1}(t_1, \ldots, t_n) & = & seq(\mathcal{T}^{-1}(t_1), seq(\mathcal{T}^{-1}(t_2), \ldots, seq(\mathcal{T}^{-1}(t_n), \varepsilon) \ldots) \end{array}$$

In the formalization of the unification algorithm, Kutsia aggregates arguments using a dummy function symbol, where we use a sequence. When we have $seq(t_1, seq(t_2, \ldots, seq(t_n, \varepsilon) \ldots)$, Kutsia has $g(t_1, \ldots, t_n)$, where g is a new dummy function symbol. The next function relates both notations.

Definition 4.5.9 *$\mathcal{K}$ translates sequence terms into Kutsia original notation and is defined as follows:*

$$\begin{array}{lcll} \mathcal{K}(seq(t_1, \bar{t_2})) & = & g(\mathcal{T}(seq(t_1, \bar{t_2}))) & \text{where } g \text{ is a new function symbol} \\ \mathcal{K}(True) & = & \top & \\ \mathcal{K}(False) & = & \bot & \\ \mathcal{K}(X = t) & = & \{X \leftarrow \mathcal{K}(t)\} & \\ \mathcal{K}(\bar{t} = * = \bar{s}) & = & \mathcal{K}(\bar{t}) \stackrel{?}{=} \mathcal{K}(\bar{s}) & \\ \mathcal{K}(t_1 \wedge \ldots \wedge t_n) & = & \{\mathcal{K}(t_1), \ldots, \mathcal{K}(t_n)\} & \\ \mathcal{K}(t) & = & \mathcal{T}(t) & \end{array}$$

Definition 4.5.10 *$\mathcal{K}^{-1}$ translates sequence terms into Kutsia original notation and is defined as follows:*

$$\begin{array}{lcl} \mathcal{K}^{-1}(g(t_1, \ldots, t_n)) & = & seq(\mathcal{T}^{-1}(t_1), \ldots, seq(\mathcal{T}^{-1}(t_n), \varepsilon) \ldots) \\ \mathcal{K}^{-1}(\top) & = & True \\ \mathcal{K}^{-1}(\bot) & = & False \\ \mathcal{K}^{-1}(\{X \leftarrow \tilde{t}\}) & = & X = \mathcal{K}^{-1}(\tilde{t}) \\ \mathcal{K}^{-1}(\tilde{t} \stackrel{?}{=} \tilde{s}) & = & \mathcal{K}^{-1}(\tilde{t}) = * = \mathcal{K}^{-1}(\tilde{s}) \\ \mathcal{K}^{-1}(\{t_1, \ldots, t_n\}) & = & \{\mathcal{K}^{-1}(t_1), \ldots, \mathcal{K}^{-1}(t_n)\} \\ \mathcal{K}^{-1}(\tilde{t}) & = & \mathcal{T}^{-1}(\tilde{t}) \end{array}$$

Theorem 4.5.1 (Soundness) *Let $\bar{t}$ and $\bar{s}$ be sequence terms. For the transformation rules in figure 4.1 let $\Longrightarrow^i$ mean i steps of transformation. Let*

$$\bar{t} = * = \bar{s} \Longrightarrow^i SU \wedge FA$$

where SU is a set of elements of the form $t_1 = t_2$, FA is of the form $\bar{t'} = * = \bar{s'}$ and $i \in \{1, 2\}$then,

$$\mathcal{K}(\bar{t}) \stackrel{?}{=} \mathcal{K}(\bar{s}) \rightsquigarrow (\mathcal{K}(SU), \mathcal{K}(FA))$$

for a transformation rule in Kutsia algorithm.

Proof 4.5.2 *The Failure rules are omitted in our version since all the cases that do not match any of the rules fail and thus the final result is the same. Also note that in our version we introduce new variables in rules 7 and 8 while Kutsia's version re-uses the variable being replaced. Since Kutsia replaces all the occurrences of the re-used variable by the new sequence, the effect is the same than creating a new variable. Also, we use only sequence variables where Kutsia uses individual variables. The reason is that individual variables are sequence variables with sequences of one element. Finally note that our version of the algorithm may need two steps to achieve the same result of Kutsia's. This happens in eliminate and split rules where each of Kutsia's rules is equivalent to applying the 4th rule of our algorithm followed by one of the following rules of eliminate. The reason for this division is that we don't use dummy functors, instead we have a fixed functor (seq) for sequences and thus we can deal with a sequence directly without the need of introducing a new functor in each step. In the 4th rule we discard the functor and in the following rules we deal with the sequence directly while Kutsia discards the functor and adds a new dummy functor for the rest of the sequence in one step only.*

The proof follows case by case:

Rule 1 $t = * = s \Longrightarrow True => \mathcal{K}(t) \stackrel{?}{=} \mathcal{K}(s) \rightsquigarrow (\mathcal{K}(\varepsilon), \mathcal{K}(True)) => \mathcal{K}(t) \stackrel{?}{=} \mathcal{K}(s) \rightsquigarrow (\varepsilon, \top)$. *Thus the first rule of our algorithm corresponds to the first rule of Kutsia's algorithm.*

Rule 2 $X = * = t \Longrightarrow X = t => \mathcal{K}(X) \stackrel{?}{=} \mathcal{K}(t) \rightsquigarrow (\mathcal{K}(X = t), \mathcal{K}(True)) => \mathcal{K}(X) \stackrel{?}{=} \mathcal{K}(t) \rightsquigarrow (\{X \leftarrow t\}, \top)$. *Note that Kutsia uses an individual variable and we use a sequence variable, but our sequence variable unifies one element only. Thus the second rule of our algorithm corresponds to the second rule of Kutsia's algorithm.*

Rule 3 *This case is similar to the one above. Thus the third rule of our algorithm corresponds to the third rule of Kutsia's algorithm.*

Rule 4 and 5 $f(seq(t_1, \bar{t})) = * = f(seq(s_1, \bar{s})) \Longrightarrow$ *by 4 and then* $seq(t_1, \bar{t}) = * = seq(s_1, \bar{s}) \Longrightarrow$ *by 5,* $t_1 = * = s_1 \wedge \bar{t} = * = \bar{s} \Longrightarrow \delta \wedge \bar{t}' = * = \bar{s}'$ *where* $\delta = (X_1 = e_1 \wedge \ldots X_n = e_n)$ *is the conjunction resulting from the successful unification of* $t_1 = * = s_1$ *and* $\bar{t}'$ *and* $\bar{s}'$ *result from* $\bar{t}$ *and* $\bar{s}$ *respectively by applying each of the substitutions on the conjunction* δ. *Then,* $\mathcal{K}(f(seq(t_1, \bar{t}))) \stackrel{?}{=} \mathcal{K}(f(seq(s_1, \bar{s}))) \rightsquigarrow (\mathcal{K}(\delta), \mathcal{K}\ (\bar{t}' = * = \bar{s}'))$ *is the same as* $f(t_1, \tilde{t}) \stackrel{?}{=} f(s_1, \tilde{s}) \rightsquigarrow (\delta', \tilde{t} \stackrel{?}{=} \tilde{s})$. *Thus, rules 4 and 5 of our algorithm correspond to rule 13 of Kutsia's algorithm.*

Rule 4 and 5 $f(seq(X, \bar{t})) = * = f(seq(X, \bar{s})) \Longrightarrow$ *by 4 and then* $seq(X, \bar{t}) = * = seq(X, \bar{s}) \Longrightarrow$ *by 5,* $X = * = X \wedge \bar{t} = * = \bar{s} \Longrightarrow \bar{t} = * = \bar{s}$ *since* $X = * = X$ *obviously succeeds.*

Then, $\mathcal{K}(f(seq(X,\bar{t}))) \stackrel{?}{=} \mathcal{K}(f(seq(X,\bar{s}))) \rightsquigarrow (\mathcal{K}(\varepsilon), \mathcal{K}(\bar{t} = * = \bar{s}))$ *is the same as* $f(X,\tilde{t}) \stackrel{?}{=} f(X,\tilde{s}) \rightsquigarrow (\varepsilon, \tilde{t} \stackrel{?}{=} \tilde{s})$. *Thus, rules 4 and 5 of our algorithm correspond to rule 14 of Kutsia's algorithm.*

Rule 4 and 6 $f(seq(X,\bar{t})) = * = f(seq(s_1,\bar{s})) \Longrightarrow$ *by 4 and then* $seq(X,\bar{t}) = * = seq(s_1,\bar{s})$ $\Longrightarrow$ *by 6:*

- $X = * = s_1 \wedge normalize(\bar{t}) = * = normalize(\bar{s}) \Longrightarrow X = s_1 \wedge \bar{t}' = * = \bar{s}'$ *where* $\bar{t}'$ *and* $\bar{s}'$ *result from* $\bar{t}$ *and* $\bar{s}$ *respectively by applying the substitutions* $X = s_1$ *and normalize. Then,* $\mathcal{K}(f(seq(X,\bar{t}))) \stackrel{?}{=} \mathcal{K}(f(seq(s_1,\bar{s}))) \rightsquigarrow (\mathcal{K}(X = s_1), \mathcal{K}(\bar{t}' = * = \bar{s}'))$ *is the same as* $f(X,\tilde{t}) \stackrel{?}{=} f(s_1',\tilde{s}) \rightsquigarrow (\{X \leftarrow s_1\}, \tilde{t} \stackrel{?}{=} \tilde{s})$. *Thus, rules 4 and the first case of rule 5 of our algorithm correspond to the first case of rule 15 of Kutsia's algorithm.*
- $X = * = seq(s_1, seq(X_1,\varepsilon)) \wedge normalize(seq(X_1,\bar{t})) = * = normalize(\bar{s}) \Longrightarrow$ $X = seq(s_1,\ seq(X_1,\varepsilon)) \wedge \bar{t}' = * = \bar{s}'$ *where* $\bar{t}'$ *and* $\bar{s}'$ *result from* $seq(X_1,\bar{t})$ *and* $\bar{s}$ *respectively by applying the substitutions* $X = seq(s_1, seq(X_1,\varepsilon))$ *and normalize. Then,* $\mathcal{K}(f(seq(X,\bar{t}))) \stackrel{?}{=} \mathcal{K}(f(seq(s_1,\bar{s}))) \rightsquigarrow (\mathcal{K}(X = seq(s_1, seq(X_1,\varepsilon))), \mathcal{K}(seq(X_1,\bar{t}') = * = \bar{s}'))$ *is the same as* $f(X,\tilde{t}) \stackrel{?}{=} f(s_1',\tilde{s}) \rightsquigarrow (\{X \leftarrow \ulcorner s_1, X_1 \urcorner\}, g(X_1, \tilde{t}) \stackrel{?}{=} g(\tilde{s}))$. *Thus, rules 4 and the second case of rule 5 of our algorithm correspond to the second case of rule 15 of Kutsia's algorithm.*

Rule 4 and 7 *This case is similar to the one above and thus showing that rules 4 and 7 of our algorithm correspond to rule 16 of Kutsia's algorithm.*

Rule 4 and 8 $f(seq(X,\bar{t})) = * = f(seq(Y,\bar{s})) \Longrightarrow$ *by 4 and then* $seq(X,\bar{t}) = * = seq(Y,\bar{s})$ $\Longrightarrow$ *by 8:*

- $X = * = Y \wedge \bar{t} = * = \bar{s} \Longrightarrow X = Y \wedge \bar{t} = * = \bar{s}$ *since* $X = * = Y$ *obviously succeeds. Then,* $\mathcal{K}(f(seq(X,\bar{t}))) \stackrel{?}{=} \mathcal{K}(f(seq(Y,\bar{s}))) \rightsquigarrow (\mathcal{K}(X = Y), \mathcal{K}(\bar{t}(X = Y) = * = \bar{s}(X = Y)))$ *is the same as* $f(X,\tilde{t}) \stackrel{?}{=} f(Y,\tilde{s}) \rightsquigarrow (\{X \leftarrow Y\}, \tilde{t} \stackrel{?}{=} \tilde{s})$. *Thus, rules 4 and first case of 8 of our algorithm correspond to first case of rule 17 of Kutsia's algorithm.*
- $X = * = seq(Y, seq(X_1,\varepsilon)) \wedge normalize(seq(X_1,\bar{t})) = * = normalize(\bar{s}) \Longrightarrow$ $X = seq(Y,\ seq(X_1,\varepsilon)) \wedge \bar{t}' = * = \bar{s}'$ *where* $\bar{t}'$ *and* $\bar{s}'$ *result from* $seq(X_1,\bar{t})$ *and* $\bar{s}$ *respectively by applying the substitutions* $X = seq(Y, seq(X_1,\varepsilon))$ *and normalize it. Then,* $\mathcal{K}(f(seq(X,\bar{t}))) \stackrel{?}{=} \mathcal{K}(f(seq(Y,\bar{s}))) \rightsquigarrow (\mathcal{K}(X = seq(Y, seq(X_1,\varepsilon))), \mathcal{K}(\bar{t}') = * = \bar{s}'))$ *is the same as* $f(X,\tilde{t}) \stackrel{?}{=} f(Y,\tilde{s}) \rightsquigarrow (\{X \leftarrow \ulcorner Y, X_1 \urcorner\}, g(X_1,\tilde{t}) \stackrel{?}{=} g(\tilde{s}))$. *Thus, rules 4 and the second case of rule 8 of our algorithm correspond to the second case of rule 17 of Kutsia's algorithm.*
- *This case is similar to the one above and thus, rules 4 and third case of rule 8 of our algorithm correspond to the third case of rule 17 of Kutsia's algorithm.*

Rule 4 and 9 *It is trivial by direct observation to see that rules 4 and 9 of our algorithm correspond to rule 18 of Kutsia's algorithm.*

Theorem 4.5.2 (Completeness) *Let $\tilde{t}$ and $\tilde{s}$ be sequences. Then, for each transformation rule in Kutsia algorithm of the form:*

$$\tilde{t} \stackrel{?}{=} \tilde{s} \rightsquigarrow (SU, FA)$$

where SU is a set of elements of the form $t_1 \leftarrow t_2$, FA is of the form $\tilde{t'} \stackrel{?}{=} \tilde{s'}$ and $i \in \{1,2\}$ there is a transformation rule in algorithm of figure 4.1 of the form:

$$\mathcal{K}^{-1}(\tilde{t}) = * = \mathcal{K}^{-1}(\tilde{s}) \Rightarrow \mathcal{K}^{-1}(SU) \wedge \mathcal{K}(FA^{-1}))$$

Proof 4.5.3 *The proof follows case by case:*

Rule 1 $\tilde{t} \stackrel{?}{=} \tilde{t} \rightsquigarrow (\varepsilon, \top)$, *then* $\mathcal{K}^{-1}(\tilde{t}) = * = \mathcal{K}^{-1}(\tilde{t}) \Rightarrow \mathcal{K}^{-1}(\varepsilon) \wedge \mathcal{K}^{-1}(\top)$ *and* $\bar{t} = * = \bar{s} \Rightarrow True$. *Thus, rule 1 of Kutsia's algorithm corresponds to the first rule of our algorithm.*

Rule 2 $X \stackrel{?}{=} \tilde{t} \rightsquigarrow (\{X \leftarrow t, \top\})$ *then* $\mathcal{K}^{-1}(X) = * = \mathcal{K}^{-1}(\tilde{t}) \Rightarrow \mathcal{K}^{-1}(X \stackrel{?}{=} t) \wedge \mathcal{K}^{-1}(\top)$ *and* $X = * = \bar{t} \Rightarrow X = \bar{t}$ *and thus rule 2 of Kutsia's algorithm corresponds to rule 2 of our algorithm.*

Rule 3 *This rule is similar to the one above and thus the result holds.*

Rule 13 $f(t_1, \tilde{t}) \stackrel{?}{=} f(s_1, \tilde{s}) \rightsquigarrow (\delta, g(\tilde{t}\delta) \stackrel{?}{=} g(\tilde{s}\delta))$ *where* $\delta = \{X_1 \leftarrow e_1, \ldots, X_n \leftarrow e_n\}$ *is the set of substitutions resulting from the successful unification of* $t_1 \stackrel{?}{=} s_1$. *Then,* $\mathcal{K}^{-1}(f(t_1, \tilde{t})) = * = \mathcal{K}^{-1}(f(s_1, \tilde{s})) \Rightarrow \mathcal{K}^{-1}(\delta) \wedge \mathcal{K}^{-1}(g(\tilde{t}\delta) \stackrel{?}{=} g(\tilde{s}\delta))$ *resulting in* $f(t'_1, \bar{t}) = * = f(s'_1, \bar{s}) \Rightarrow X_1 = e'_1 \wedge \ldots \wedge X_n = e'_n \bar{t} = * = \bar{s}$ *by applying rule 4 followed by rule 5 of our algorithm. Thus, rule 13 of Kutsia's algorithm corresponds to applying rule 4 followed by rule 5 of our algorithm.*

Rule 14 $f(X, \tilde{t}) \stackrel{?}{=} f(X, \tilde{s}) \rightsquigarrow (\{X \leftarrow X\}, g(\tilde{t}) \stackrel{?}{=} g(\tilde{s}))$. *Then,* $\mathcal{K}^{-1}\ (f(X, \tilde{t})) = * = \mathcal{K}^{-1}(f(X, \tilde{s})) \Rightarrow \mathcal{K}^{-1}(\{X \leftarrow X\}) \wedge \mathcal{K}^{-1}(g(\tilde{t}) \stackrel{?}{=} g(\tilde{s}))$ *resulting in* $f(X, \bar{t}) = * = f(X, \bar{s}) \Rightarrow X = X \wedge \bar{t} = * = \bar{s}$ *by applying rule 4 followed by rule 5 of our algorithm. Thus, rule 14 of Kutsia's algorithm corresponds to applying rule 4 followed by rule 5 of our algorithm.*

Rule 15

- $f(X, \tilde{t}) \stackrel{?}{=} f(t_1, \tilde{s}) \rightsquigarrow (\{X \leftarrow t_1\}, g(\tilde{t}\{X \leftarrow t_1\}) \stackrel{?}{=} g(\tilde{s}\{X \leftarrow t_1\}))$. *Then,* $\mathcal{K}^{-1}(f(X, \tilde{t})) = * = \mathcal{K}^{-1}(f(t_1, \tilde{s})) \Rightarrow \mathcal{K}^{-1}(\{X \leftarrow t_1\}) \wedge \mathcal{K}^{-1}(g(\tilde{t}\{X \leftarrow t_1\}) \stackrel{?}{=} g(\tilde{s}\{X \leftarrow t_1\}))$ *resulting in* $f(X, \bar{t}) = * = f(t'_1, \bar{s}) \Rightarrow X = t'_1 \wedge \bar{t} = * = \bar{s}$ *by applying rule 4 followed by rule 6 of our algorithm. Thus, the first case of rule 15 of Kutsia's algorithm corresponds to applying rule 4 followed by rule 6 of our algorithm.*

- $f(X,\tilde{t}) \stackrel{?}{=} f(t_1,\tilde{s}) \rightsquigarrow (\{X \leftarrow \ulcorner t_1, X_1 \urcorner\}, g(seq(X_1,\tilde{t})\{X \leftarrow \ulcorner t_1, X_1 \urcorner\}) \stackrel{?}{=} g(\tilde{s}\{X \leftarrow \ulcorner t_1, X_1 \urcorner\}))$. *Then,* $\mathcal{K}^{-1}(f(X,\tilde{t})) = * = \mathcal{K}^{-1}(f(t_1,\tilde{s})) \Rightarrow \mathcal{K}^{-1}(X \leftarrow \ulcorner t_1, X_1 \urcorner) \wedge \mathcal{K}^{-1}(g(X_1,\tilde{t}\{X \leftarrow \ulcorner t_1, X_1 \urcorner\}) \stackrel{?}{=} g(\tilde{s}\{X \leftarrow \ulcorner t_1, X_1 \urcorner\}))$ *resulting in* $f(X,\bar{t}) = * = f(t_1',\bar{s}) \Rightarrow X = seq(t_1', seq(X_1,\varepsilon)) \wedge \bar{t} \wedge \bar{s}$ *by applying rule 4 followed by rule 6 of our algorithm. Thus, the second case of rule 15 of Kutsia's algorithm corresponds to applying rule 4 followed by rule 6 of our algorithm.*

Rule 16 *This case is similar to the one above and thus rule 16 of Kutsia's algorithm corresponds to rule 4 followed by rule 7 of our algorithm.*

Rule 17 *This case is similar to cases of rules 14 and 15 and thus rule 17 of Kutsia's algorithm corresponds to rule 4 followed by rule 8 of our algorithm.*

Rule 18 *It is trivial by direct observation to see that rule 18 of Kustia's algorithm correspond to rule 4 followed by rule 9 of our algorithm.*

4.5.2 Examples

Here we present some examples of application of CLP(Flex) to the processing of XML Documents.

Example 4.5.7 ***Address Book translation.*** *In this example we use the address book document presented in section 4.4.1. In this address book we have sometimes records with a phone tag. We want to build a new XML document without this tag. Thus, we need to get all the records and ignore their phone tag (if they have one). This can be done by the following program (this example is similar to one presented in XDuce [HP00]):*

```
translate:-
   xml2pro('addressbook.xml','addressbook.dtd',Xml),
   process(Xml,NewXml),
   pro2xml(NewXml,'addressbook2.xml').

process(A,NewA):-
   findall(Record,records_without_phone(A,Record),LRecords),
   newdoc(addressbook,LRecords,NewA).

records_without_phone(A1,A2):-
   A1 =*= addressbook(_,record(name(N),address(A),_,email(E)),_),
   A2 = record(name(N),address(A),email(E)).
```

Predicate translate/0 first translates the file "addressbook.xml" into a CLP(Flex) term, which is processed by process/2, giving rise to a new CLP(Flex) term and then to the new document "addressbook2.xml". This last file contains the address records without the phone tag.

Example 4.5.8 ***Book Stores.*** *In this example we have two XML documents with a catalogue of books in each ("bookstore1.xml" and "bookstore2.xml"). These catalogues refer to*

two different book stores. Both "bookstore1.xml" and "bookstore2.xml" have the same DTD and may have similar books. One of this XML documents can be:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <book number="1">
    <name>Art of Computer Programming</name>
    <author>Donald Knuth</author>
    <price>140</price>
    <year>1998</year>
  </book>
  ...
  <book number="500">
    <name>Haskell: The Craft of Functional
                Programming (2nd Edition)</name>
    <author>Simon Thompson</author>
    <price>41</price>
    <year>1999</year>
  </book>
</catalog>
```

1. *To check which books are cheaper at bookstore 1 we have the following program:*

```
best_prices(B):-
     xml2pro('bookstore1.xml','bookstore1.dtd',T1),
     xml2pro('bookstore2.xml','bookstore2.dtd',T2),
     process(T1,T2,B).

process(Books1,Books2,[N,A]):-
     Books1 =*= catalog(_,book(name(N),author(A),price(P1),year(Y)),_),
     Books2 =*= catalog(_,book(name(N),author(A),price(P2),year(Y)),_),
     atom2number(P1,P1f),
     atom2number(P2,P2f),
     P1f < P2f.
```

   The predicate best_prices/1 *returns the cheaper books at "bookstore1.xml", one by one, by backtracking.*

2. *To get all the books from one author, the author of a book or all the pairs author/book, we have the following code:*

```
books_from(Author,Book):-
   xml2pro('bookstore1.xml','bookstore1.dtd',Xml),
   process2(Xml,Author,Book).

process2(Xml,Author,Book):-
   Xml =*= catalog(_,book((name(Book),author(Author),_)),_).
```

Here books_from/2 retrieves, by backtracking, every Author/Book names from file "bookstore1.xml".

The previous programs are rather simple. This stresses the highly declarative nature of CLP(Flex) when used for XML processing.

Example 4.5.9 *We defined syntactic sugar for sequence term in normal form:* $<>$ *stands for the empty sequence term and* $< t_1, \ldots, t_n >$ *stands for the sequence term in normal form,* $seq(t_1,\ seq(t_2, \ldots,\ seq(t_n,\ \varepsilon) \ldots))$.
In this example we have two XML files, a simple text.xml *file conforming to the following DTD:*

```
<!ELEMENT ref (#PCDATA)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT text (#PCDATA | ref | b)*>
```

and a bib.xml *file with simple references conforming to the DTD:*

```
<!ELEMENT bib (author, name)>
<!ELEMENT bibliography (bib+)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT name (#PCDATA)>
```

Our text.xml *< ref > elements contain author names which appear in the < author > element of* bib.xml. *Our goal is to process* text.xml *and* bib.xml *and generate* text2.xml *and* bib2.xml *files. The* text2.xml *file is almost the same than* text.xml *but < ref > elements are replaced by new < i > elements where the content is replaced by a number. This number represents the index of that reference ordered by author within the document. The* bib2.xml *contains only the references appearing in* text.xml, *ordered by author, and with a* label *element with the corresponding number that occurs in the* text2.xml. *The program follows:*

```
run:-
        xml2pro('text.xml','text.dtd',T),
        process(T).

process(text(T)):-
        process2(T,T2,[],Refs),
        xml2pro('bib.xml','bib.dtd',B),
        add_bib(Refs,B,BibXML,1),
        newdoc(bibliography,BibXML,NewBIB),
        pro2xml(text(T2),'text2.xml'),
        pro2xml(NewBIB,'bib2.xml').

process2(<>,<>,L,L).

process2(A,B,L,RRefs):-
        A =*= <X,ref(R),Y>,
        B =*= <X,i(NewVar),Y2>,!,
        insert_sorted(R,NewVar,L,Refs),
        process2(Y,Y2,Refs,RRefs).
```

```
process2(S,S,L,L).

add_bib([],_,[],_).

add_bib([(A,AI)|Refs],B,[bib(label(AI),author(A),name(N))|XML],I):-
        B =*= bibliography(_,bib(author(A),name(N)),_),
        I1 is I + 1,
        atom_chars(I,AI),
        add_bib(Refs,B,XML,I1).
```

The run *predicate starts the translation process. In a first step, the* process2 *predicate translates the original text into a new one where the* < *ref* > *elements are replaced by new* < *i* > *elements and the content is replaced by a new free variable. At the same time pairs* $(Ref_content, FreeVariable)$ *(corresponding to the author in* < *ref* > *and the new variable in* < *i* > *respectively) are inserted in a list ordered by* $Ref_content$*. At the end we have a list of pairs ordered by author. In a second step,* add_bib *queries the bibliography file, retrieving the references found in the text and replacing the free variables with their definite content, thus, avoiding a second processing of the text file. For example, if* text.xml *corresponds to:*

```
<text>
Mainstream languages for <b>XML</b> processing such as XSLT
<ref>W3C</ref>, XDuce <ref>Hosoya</ref>, CDuce <ref>Frish
Casagna and Benzaken</ref> and Xtatic <ref>Pierce</ref> rely
on the notion of trees with an arbitrary number of leaf nodes
 to abstract <b>XML</b> documents.
</text>
```

and the bib.xml *to:*

```
<bibliography>
<bib>
   <author>Coelho and Florido</author>
   <name>Type-based XML Processing in Logic Programming</name>
</bib>
<bib>
   <author>W3C</author>
   <name>XSL Transformations (XSLT). http://www.w3.org/TR/xslt/</name>
</bib>
<bib>
   <author>Hosoya</author>
   <name>XDuce: A Typed XML processing language</name>
</bib>
<bib>
   <author>Frish Casagna and Benzaken</author>
   <name>CDuce an XML-centric general-purpose language</name>
</bib>
<bib>
```

```
  <author>Pierce</author>
  <name>Xtatic. http://www.cis.upenn.edu/~bcpierce/xtatic/</name>
</bib>
</bibliography>
```

the resulting text2.xml *will be:*

```
<text>
Mainstream languages for <b>XML</b> processing such as XSLT
<i>4</i>, XDuce <i>2</i>, CDuce <i>1</i> and Xtatic <i>3</i>
 rely on the notion of trees with an arbitrary number of
leaf nodes to abstract <b>XML</b> documents.
</text>
```

and the bib2.xml*:*

```
<bibliography>
<bib>
  <label>1</label>
  <author>
     Frish Casagna and Benzaken
  </author>
  <name> CDuce an XML-centric general-purpose language
  </name>
</bib>
<bib>
  <label>2</label>
  <author>Hosoya</author>
  <name> XDuce: A Typed XML processing language</name>
</bib>
<bib>
  <label>3</label>
  <author>Pierce</author>
  <name> Xtatic. http://www.cis.upenn.edu/~bcpierce/xtatic/</name>
</bib>
<bib>
  <label>4</label>
  <author>W3C</author>
  <name> XSL Transformations (XSLT). http://www.w3.org/TR/xslt/ </name>
</bib>
</bibliography>
```

The previous example emphasizes the advantages of CLP(Flex) for XML processing, it shows an highly declarative and compact code and the advantages of using logic variables that permit to solve the problem by processing the text.xml document only once.

Example 4.5.10 *In [KH06], it was defined a specific language to denote relations between XML documents. This language used its own new syntax, based on mainstream functional languages for XML processing, and its definition stresses the usefulness of an approach*

based on the truly relational programming paradigm: logic programming. Being a relational language, XCentric can easily describe the structures of two documents and relate their subparts. For example, we can write the following simple predicate in XCentric for converting between fragments of two kinds of address books.

```
translate(<person(name(N),C1)>,<card(person-name(N),C2)>):-
  address_content(C1,C2).

address_content(<>,<>).

address_content(<A1,address(A),A2>,<L1,location(A),L2>):-
  address_content(<A1,A2>,<L1,L2>).
```

This relates a person *element and a* card *element, where the pattern of the first argument of translate requires the* person *to contain a* name *element followed by a sequence of* address *elements, and the second argument describes the structure of the* card *containing a* person-name *element followed by a sequence of* location *elements. Variable N, which appears in both arguments, specifies that its corresponding subparts, the contents of* name *and* person-name*, are the same. Predicate address_content relates* address *in a person element with* location *in the corresponding* card *element. This is trivially expressed in an unification-based relational language such as XCentric, but impossible to express in a functional (thus unidirectional) language based on pattern matching. Note that variables occurring in sequences, are interpreted in the domain of sequences, thus, in this program, unification is not Prolog unification, but the non-standard unification of XCentric. Also note the gain in modularity, this predicate can be used in three different ways:*

1. *To transform an XML document with the format specified by the first argument of translate into the format specified by its second argument which corresponds to call translate with the first argument ground.*

2. *To do the opposite transformation which corresponds to call translate with the second argument ground.*

3. *To guarantee that two different documents in the two different formats are related in the way specified by the predicate which corresponds to call translate with both arguments ground.*

In a functional-based language these three different behaviors have to be implemented by three different functions.

4.6 Discussion

As we can see by the examples presented, flexible arity unification provides an highly declarative model for XML processing, which can be seen by comparing XML processing in

Prolog with our approach. The algorithms are straightforward and based in an established theory providing a sound theoretical core.

XML provides schemas for defining types for documents. Our next step is the integration of types in our programming language and seeing which benefits they bring.

Chapter 5

XCentric: Typed Unification based XML Processing

5.1 Introduction

In this chapter we present XCentric, a typed logic programming language for XML processing with flexible arity unification. We start with motivating examples of XML processing in XCentric which show the usefulness of the use of types both for type validation and as a shorter representation of sequences of elements improving the declarativeness of querying and processing. We describe how we extend regular types [Zob90] to *regular expression types* which in turn describe schemas for XML. We define a notation for describing *type sequences* and use it as the internal notation for regular expression types. We present two different approaches: *dynamic typing* where types are translated to programs and used at run time to validate data and *static typing* where we define algorithms for intersection of type sequences and extend the unification algorithm presented in chapter 4 to use types. Finally we present some benchmark results comparing unification and pattern matching with and without type checking. The webpage of XCentric is:

`http://www.ncc.up.pt/xcentric/`

This work was partially presented in [CF06a], [CF07b] and [CF07c].

5.2 XML processing in XCentric

In this section we motivate the reader with examples of application of the XCentric language to XML processing. XCentric extends CLP(Flex) with regular expression types which are a model for XML grammars such as DTDs. Types are now used in unification corresponding to the concept of *regular expression unification.*

For example, the following declaration introduces regular expression types describing terms in a simple bibliographic database (a type rule in XCentric is represented by the declaration :-type α - - -> $\tau_1; \ldots ; \tau_n$ where α is the type name and τ_i is a description of that type):

```
:-type bib ---> bib(book+).
:-type book ---> book(author+, name).
:-type author ---> author(string).
:-type name ---> name(string).
```

Type expressions of the form $f(...)$ classify tree nodes with the label f (XML structures of the form $< f > ... < /f >$). Type expressions of the form $t*$ denote a sequence of arbitrary many ts, and $t+$ denotes a sequence of arbitrary many ts with at least one t. Thus terms with type *bib* have *bib* as functor and their arguments are a sequence of one or more books. Terms with type *book* have *book* as functor and their arguments are a sequence consisting of one or more authors followed by the name of the book.

The next type describes arbitrary sequences of authors with at least two authors:

```
:-type type_a ---> (author(string),author(string)+).
```

To get the names of all the books with two or more authors we just need the following query:

```
?-bib(_,book(X::type_a,name(N)),_)=*=BibDoc::bib.
```

This unifies two terms typed by *bib*. The type declaration in the first argument is not needed because it can be easily reconstructed from the term. In this case we bind the variable N to the name's content of the first book with at least two authors. Note how the type *type_a* in the first argument of the unification is used to jump over an arbitrary number of arguments and extract the name of the first book with at least two authors. All the results can then be obtained, one by one, by backtracking.

Next we describe Regular Expression Types and give some examples of the application of the typed unification to XML processing.

5.2.1 Regular Expression Types

Regular expression types, describe sequences of values: a* (sequence of zero or more a's), a+ (sequence of one or more a's), a? (zero or one a), a|b (a or b) and a,b (a followed by b). We translate regular expression types to our internal sequence notation (here applied to types):

$$\begin{array}{lll} a* & \Rightarrow & \alpha_* \rightarrow \{\varepsilon, seq(a, \alpha_*)\} \\ a+ & \Rightarrow & \alpha_+ \rightarrow \{a, seq(a, \alpha_+)\} \\ a? & \Rightarrow & \alpha_? \rightarrow \{\varepsilon,\ seq(a, \varepsilon)\} \\ a|b & \Rightarrow & \alpha_| \rightarrow \{a, b\} \\ a, b & \Rightarrow & \alpha_{seq} \rightarrow \{seq(a, seq(b, \varepsilon))\} \end{array}$$

Note that DTDs (Document Type Definition) [W3C04c] can be trivially translated to regular expression types.

Example 5.2.1 *The DTD,*

```
<!ELEMENT a (#PCDATA)>
<!ELEMENT b (c+)>
<!ELEMENT c (#PCDATA)>
<!ELEMENT d (#PCDATA)>
<!ELEMENT e (#PCDATA)>
<!ELEMENT l (a*, b, e?, d)>
```

corresponds to the regular expression type:

$$\alpha \rightarrow l(a(string)*, b(c(string)+), e(string)?, d(string))$$

5.2.2 XML Schema support

XCentric also has some support for *XML Schema* [W3C04f].

5.2.2.1 Basic Types

We can explicit declare the following base types:

- *string*;
- *integer*;
- *float*;
- *boolean*.

5.2.2.2 Occurrences of sequences

We can declare the maximum and minimum number of occurrences of an element: $\tau_{\{min,max\}}$ means a sequence of elements of type τ where the length of the sequence is between min and max. We can use *unbounded* if the maximum is any number greater than min The example presented in section 5.2, for the type of a sequence of two or more authors, can be written as:

```
type type_a ---> author(string){2,unbounded}.
```

Type *type_a* represents every sequence of two or more authors.

5.2.2.3 Orderless sequences

Another feature from XML Schema which is available in XCentric is orderless sequences. $\{\tau_1 \& \tau_2 \& \ldots \& \tau_n\}$ means a sequence of elements of types $\tau_1, \tau_2, \ldots, \tau_n$ that can occur in any order. Consider the following type:

```
type mix ---> rec({name(string) & address(string) & email(string)}).
```

Type *mix* represents a record with three elements where their order doesn't matter.

5.2.3 Examples

Here we present some examples of use of typed unification. From now on, whenever a variable is presented without any type information it is implicitly associated with the universal type *any*.

Example 5.2.2 *Consider the simple XML file:*

```
<addressbook>
        <record>
                <name>John</name>
                <address>New York</address>
                <email>john.ny@mailserver.com</email>
        </record>
        <record>
                <name>Sofia</name>
                <address>Rio de Janeiro</address>
                <phone>123456789</phone>
                <email>sofia.brasil@mail.br</email>
        </record>
</addressbook>
```

This XML file is valid accordingly to the following DTD:

```
<!ELEMENT addressbook (record*)>
<!ELEMENT record (name, address, phone?, email)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

In XCentric this DTD is translated to type_addr*:*

```
:-type type_addr ---> addressbook(record(name(string), address(string),
                                          phone(string)?,
                                          email(string))*)
```

Note that, in this address book we have sometimes records with a phone tag. We want to build a new XML document without this tag. Thus, we need to get all the records and ignore their phone tag (if they have one). This can be done by the following program (this example is similar to one presented in XDuce [HP00]):

```
:-type type_a ---> addressbook(record(name(string), address(string),
                                       phone(string)?,
                                       email(string))*).
```

```
:-type type_r ---> record(name(string),address(string),email(string)).

translate:-
  xml2pro('addressbook.xml',Xml),
  process(Xml,NewXml),
  pro2xml(NewXml,'addressbook2.xml').

process(A,NewA):-
  findall(Record,records_without_phone(A,Record),LRecords),
  newdoc(addressbook,LRecords,NewA).

records_without_phone(A1,A2):-
  A1 =*= addressbook(_,record(name(N),address(A),_,email(E)),_)::type_a,
  A2 = record(name(N),address(A),email(E))::type_r.
```

Predicate translate/0 first translates the file "addressbook.xml" into a term, which is processed by process/2, giving rise to a new term and then to the new document "addressbook2.xml". This last file contains the address records without the phone tag. Note that findall *is a predicate that returns a list of items satisfying a given condition. In this specific case it returns a list* LRecord *of all of records satisfying the predicate* records_without_phone*.*

Example 5.2.3 *In this example we have two XML documents with a catalogue of books in each ("bookstore1.xml" and "bookstore2.xml"). These catalogues refer to two different book stores. Both "bookstore1.xml" and "bookstore2.xml" have the same DTD and may have similar books. One of this XML documents can be:*

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  ...
  <book number="500">
    <name>Haskell: The Craft of Functional
                  Programming (2nd Edition)</name>
    <author>Simon Thompson</author>
    <price>41</price>
    <year>1999</year>
  </book>
  ...
</catalog>
```

To check which books are cheaper at bookstore 1 we have the following program:

```
:-type type_c ---> catalog(book(name(string),author(string),
                              price(string),year(string))*).

best_prices(B):-
    xml2pro('bookstore1.xml',T1),
    xml2pro('bookstore2.xml',T2),
    process(T1,T2,B).
```

```
process(Books1,Books2,[N,A]):-
    Books1 =*= catalog(_,book(name(N),author(A),price(P1),year(Y)),_)::type_c ,
    Books2 =*= catalog(_,book(name(N),author(A),price(P2),year(Y)),_)::type_c ,
    atom2number(P1,P1f),
    atom2number(P2,P2f),
    P1f < P2f.
```

The predicate best_prices/1 *returns the cheaper books at "bookstore1.xml", one by one, by backtracking.*

Example 5.2.4 *Consider an addressbook where the email may occur zero or more times:*

```
<!ELEMENT addressbook (record*)>
<!ELEMENT record (name,address ,phone?,email*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

To create a new document with all the records *containing at least one* email*:*

```
:-type type_a ---> record(name(string),address(string),phone(string)?,
                                                        email(string)+).
translate:-
  xml2pro('addressbook.xml',Xml),
  process(Xml,NewXml),
  pro2xml(NewXml,'addressbook2.xml').

process(A,NewA):-
  findall(Record,records_with_email(A,Record),LRecords),
  newdoc(addressbook ,LRecords,NewA).

records_with_email(A1,X):-
  A1 =*= addressbook(_,X::type_a ,_).
```

Note that besides using types to check the content of the document, we use them as a short representation for sequences that can be seen in:

```
A1 =*= addressbook(_,X::type_a ,_).
```

Example 5.2.5 *Given the following XML file containing a list of people and jobs they had:*

```
<doc>
  <person>
     <name>David Pereira </name>
     <previous_job>
        <description>Development Manager</description>
        <address>NUS Software</address>
     </previous_job>

     <previous_job>
```

```
      <description>Senior industry analyst</description>
      <address>TELIN Corp.</address>
     </previous_job>

     <previous_job>
      <description>Software Architect</description>
      <address>MicroToft</address>
     </previous_job>
  </person>

  <person>
    <name>Tania Magalhaes </name>
    <previous_job>
     <description> Graphics specialist </description>
     <address>Silicone Graphics</address>
    </previous_job>

    <previous_job>
     <description>PHP web developer</description>
     <address>Rincohn</address>
    </previous_job>
   </person>

  <person>
    <name>Nuno Pereira</name>
  </person>
</doc>
```

which is valid accordingly to the following XML Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:schema>
  <xs:element name="previous_job">
   <xs:complexType>
    <xs:sequence>
      <xs:element ref="description"/>
      <xs:element ref="address"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>

  <xs:element name="person">
   <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
       <xs:element ref="previous_job" minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
```

```
  <xs:element name="doc">
   <xs:complexType>
    <xs:sequence>
      <xs:element ref="person" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>

 <xs:element name="name" type="xs:string"/>
 <xs:element name="description" type="xs:string"/>
 <xs:element name="address" type="xs:string"/>

</xs:schema>
```

Get all the names of persons that had between 2 and 4 jobs:

```
:- type jobs ---> previous_job(description(string),address(string)){2,4}.

twofour(N):-
          xml2pro('.jobs.xml',Oc),
          doc(_,person(name(N),X::jobs),_) =*= Oc.
```

Example 5.2.6 *Consider the following simple addressbook:*

```
<abook>
  <person>
    <name>Liliana</name>
    <address>Lisboa</address>
  </person>

  <person>
    <address>V.N. Gaia</address>
    <name>Judite</name>
    <email>jud@mail.pt</email>
  </person>

  <person>
    <email>tania@mail.pt</email>
    <name>Tania</name>
    <address>Regua</address>
  </person>

  <person>
   <name>Rosa</name>
   <email>rosa@mail.pt</email>
   <address>Ermesinde</address>
  </person>
</abook>
```

which is valid accordingly to the following XML Schema:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:schema>
  <xs:element name="person">
   <xs:complexType>
    <xs:all>
      <xs:element ref="email" minOccurs="0" maxOccurs="1"/>
      <xs:element ref="name"/>
      <xs:element ref="address"/>
    </xs:all>
   </xs:complexType>
  </xs:element>

  <xs:element name="name" type="xs:string"/>
  <xs:element name="email" type="xs:string"/>
  <xs:element name="address" type="xs:string"/>

  <xs:element name="abook">
   <xs:complexType>
    <xs:sequence>
     <xs:element ref="person" maxOccurs="unbounded"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
</xs:schema>
```

get the names of all the persons:

```
:- type mix ---> person({name(string) & address(string) & email(string)?}).

getnames(N):-
        xml2pro('mixaddress.xml',Mix),
        abook(_,X::mix,_) =*= Mix,
        person(_,name(N),_) =*= X.
```

We now proceed with the explanation of the concepts behind XCentric.

5.3 Regular Types

The next definitions and examples introduce the notion of Regular Types along the lines presented in [DZ92].

Definition 5.3.1 *Assuming an infinite set of* type symbols*, a* type term *is defined as follows:*

1. *A constant symbol is a type term* $(a, b, c, ...)$.

2. *A variable is a type term* $(x, y, ...)$.

3. *A type symbol is a type term* $(\alpha, \beta, ...)$

4. *If* f *is a flexible arity function symbol and each* τ_i *is a type term,* $f(\tau_1, ..., \tau_n)$ *is a type term.*

Example 5.3.1 *Let* a *be a constant symbol,* α *a type symbol and* x *a variable. The expressions* a, α, x *and* $f(a, \alpha, x)$ *are* type terms. *If the expression is variable free, we call it a* pure type term. *The expressions* a, α *and* $f(\alpha, g(\beta), c)$ *are* pure type terms.

Definition 5.3.2 *A* type rule *is an expression of the form* $\alpha \rightarrow \Upsilon$ *where* α *is a type symbol (represents types) and* Υ *is a set of pure type terms. We will use* T *to represent a set of type rules.*

Sets of type rules correspond to *regular term grammars* [Tha73].

Example 5.3.2 *Let* α *and* β *be type symbols. Then* $\alpha \rightarrow \{a, b\}$ *and* $\beta \rightarrow \{nil, tree(\beta,\ \alpha, \beta)\}$ *are type rules.*

Definition 5.3.3 *A type symbol* α *is defined by a set of type rules* T *if there exists a type rule* $\alpha \rightarrow \Upsilon \in T$.

We make some assumptions:

1. The constant symbols are partitioned in non-empty subsets, called *base types*. Some examples are, *string*, *int*, and *number*.

2. The existence of μ, the universal type, and ϕ representing the empty type.

3. Each type symbol occurring in a set of type rules T is either μ, ϕ, a base type symbol, or a type symbol defined in T, and each type symbol defined in T has exactly one defining rule in T.

Regular types are the class of types that can be specified by sets of type rules.

Example 5.3.3 *Let* α_i *be the type of the* i^{th} *argument of append. The predicate append is defined as follows:*

append(nil,L,L) $\leftarrow$
append(.(X,RX),Y,.(X,RZ)) $\leftarrow$ *append(RX,Y,RZ).*

Regular types for α_1,α_2,α_3 *are:*

$$\begin{array}{lcl} \alpha_1 & \rightarrow & \{nil, .(\mu, \alpha_1)\} \\ \alpha_2 & \rightarrow & \{\mu\} \\ \alpha_3 & \rightarrow & \{\alpha_2, .(\mu, \alpha_3)\} \end{array}$$

5.4 Sequence Types

Regular types can be used to type sequences, where every sequence has a type α_{seq} with type rule, $\alpha_{seq} \rightarrow R$ and R may contain constants, types and/or sequences of types.

Example 5.4.1 *Given the type symbol α_{seq}, functor f, constant c and type rule $\alpha_{seq} \rightarrow \{\varepsilon, seq(f(c), \alpha_{seq})\}$, α_{seq} describes sequences of elements of the form $f(c)$. For example, ε, $seq(f(c), \varepsilon)$, $seq(f(c), seq(f(c), \varepsilon))$, ...*

Definition 5.4.1 *We define* sequence type *as a sequence containing regular types.*

Example 5.4.2 *Given type α with type rule, $\alpha \rightarrow \{a, b\}$, $seq(\alpha, \varepsilon)$ is a sequence type.*

5.5 Dynamic Typing

In this section we present dynamic typing in XCentric. This is one way of having typed-unification where types are translated to programs which accept terms and validate them in run-time. These programs can be created directly from the type definition and used at runtime to validate the data being processed. We start by describing types as programs and then we explain their role in the unification process.

5.5.1 Types as programs

It is well known that regular types can be associated with unary logic programs (see [Zob87, YS90, FD92, FSVY91]). For every type symbol α, there is a predicate definition α, such that $\alpha(t)$ is true if and only if t is a term with type α (note that we are using the type symbol as the predicate symbol). The universal type μ is defined by the predicate $\mu(X) \leftarrow$. For every other type symbol, α, where $\alpha \rightarrow \Upsilon$ and $\tau \in \Upsilon$, the program has a clause,

$$\alpha(t) \leftarrow \beta(X_1) \wedge \ldots \wedge \beta(X_n)$$

where t is τ with each type symbol β_i replaced by a new variable x_i.

Example 5.5.1 *Let α_{list} be a type symbol with type rule, $\alpha_{list} \rightarrow \{nil, .(a, \alpha_{list})\}$. α_{list} can be associated with the program:*

$$\begin{aligned} \alpha_{list}(nil) &\leftarrow \\ \alpha_{list}(.(a, X)) &\leftarrow \alpha_{list}(X) \end{aligned}$$

Example 5.5.2 *Let α_{seq_a} be a type symbol with type rule, $\alpha_{seq_a} \rightarrow \{\varepsilon, seq(a, \alpha_{seq_a})\}$. α_{seq_a} can be associated with the program:*

$$\begin{aligned} \alpha_{seq_a}(\varepsilon) &\leftarrow \\ \alpha_{seq_a}(seq(t_1, \bar{t})) &\leftarrow \alpha_{seq_a}(\bar{t}) \end{aligned}$$

Let $\alpha_1, \ldots, \alpha_n$ be type symbols defined by the rules in T. Φ_T denotes the program that defines μ (universal type) and $\alpha_1, \ldots, \alpha_n$.

Example 5.5.3 *Let* $T = \{\alpha \to \{\varepsilon, a\}, \beta \to \{seq(\alpha, \gamma)\}, \gamma \to \{\varepsilon, seq(d, \gamma)\}\}$ *then,* Φ_T *is:*

$$\begin{array}{lll} \alpha(\varepsilon) & \leftarrow & \\ \alpha(a) & \leftarrow & \\ \beta(seq(X_1, X_2)) & \leftarrow & \alpha(X_1), \gamma(X_2) \\ \gamma(\varepsilon) & \leftarrow & \\ \gamma(seq(d, X_2)) & \leftarrow & \gamma(X_2) \end{array}$$

The *type* associated with a type symbol α in a set of type rules T, is the set of terms occurring as arguments to the unary predicate α in the minimal model M_{Φ_T} (details about standard semantics of logic programs can be found in [Llo87]).

Definition 5.5.1 *Let T be a set of type rules. The* type *associated with the pure type term τ with respect to T is given by the following recursive definition (the first line applies when τ is constant symbol c, the second one applies when τ is type symbol α and the third one when τ is the pure type term $f(\tau_1, \ldots, \tau_n)$):*

$$[\tau]_T = \begin{cases} \{c\} \\ \{t | \alpha(t) \in M_{\Phi_T}\} \\ \{f(t_1, \ldots, t_n) | t_i \in [\tau_i]_T, 1 \leq i \leq n\} \end{cases}$$

Types that can be described by $[\tau]_T$, where T is a set of type rules are called regular types*. Informally, $[\tau]_T$ is the set of terms that can be derived from τ by repeated application of rules in T.*

Example 5.5.4 *Let T be the set of type rules $\{\alpha \to \{\varepsilon,\ seq(a,\ \alpha)\}\}$, then $[\alpha]_T = \{\varepsilon,$ $seq(a, \varepsilon),\ seq(a, seq(a,\ \varepsilon)), \ldots\}$*

5.5.2 Types in the unification process

In this section we explain the role of types in the unification process.

Definition 5.5.2 *A* type declaration *for a term t with respect to a set of type rules T is a pair $t :: \alpha$ where α is a type symbol defined in T.*

Note that in any term t, it is only necessary to declare types for variables since this is enough to reconstruct the type for t. When a term t is typed by μ (the universal type) we will omit the type declaration. We now define the notion of typed unification.

Definition 5.5.3 *If t_1 and t_2 are terms and α_1 and α_2 are types, then $t_1 :: \alpha_1 = * = t_2 :: \alpha_2$ denotes unification of typed terms with flexible arity symbols.*

An equation $t_1 :: \alpha_1 = * = t_2 :: \alpha_2$ is *solvable* if and only if there is an assignment of sequences or ground terms, respectively, to variables therein such that the equation evaluates to *true*, i.e. such that after that assignment the terms become equal and belong to $[\alpha_1]_T \cap [\alpha_2]_T$.

Example 5.5.5 *Consider the equation* $a(X, b, Y) :: \alpha_a = * = a(a, b, b, b) :: \mu$*, where* α_a *is defined by the type rules:*

$$\begin{array}{lcl} \alpha_a & \longrightarrow & \{a(\alpha_x, b, \alpha_y)\} \\ \alpha_x & \longrightarrow & \{\mu\} \\ \alpha_y & \longrightarrow & \{b, (b, \alpha_y)\} \end{array}$$

then this unification gives only two results:

1. $X = a$ *and* $Y = \ulcorner b, b \urcorner$

2. $X = \ulcorner a, b \urcorner$ *and* $Y = b$

Note that without the types the solution $X = \ulcorner a, b, b \urcorner$ *and* $Y = \varepsilon$ *would also be valid.*

In the implementation an equation of the form $t_1 :: \alpha_1 = * = t_2 :: \alpha_2$ is translated to the following query:

$$?\text{ - } t_1 = * = t_2, \alpha_1(t_1), \alpha_2(t_2).$$

and the respective predicate definitions for α_1 and α_2 (as described in section 5.3.2). Correctness of $t_1 :: \alpha_1(t_1) = * = t_2 :: \alpha_2(t_2)$ comes for free from the correctness of the untyped version of $= * =$ (presented in the previous chapter) and noticing that if $?{-}t_1 = * = t_2, \alpha_1(t_1), \alpha_2(t_2)$. succeeds then $t_1\theta \in [\alpha_1]_T \cap [\alpha_2]_T$, where θ is the substitution resulting from $t_1 = * = t_2$.

Example 5.5.6 *Consider again example 5.5.5. The generated predicates for the types are:*

$$\begin{array}{lcl} \alpha_a(A) & \leftarrow & A = * = a(_, b, Y), \alpha_y(Y). \\ \alpha_y(Y) & \leftarrow & Y = * = b. \\ \alpha_y(Y) & \leftarrow & Y = * =< b, Y_1 >, \alpha_y(Y_1). \end{array}$$

5.6 Static Typing

Here we describe how can we do static typing. We define sequence type intersection and use it at compile time to check if the unification can be successful accordingly with the declared types for each of the terms involved.

5.6.1 Type intersection

The intersection of types plays an important role in the unification algorithm described in this paper. In this section we present an intersection algorithm along with some auxiliary functions and examples.

We now add some new definitions to the ones presented in chapter 4:

Definition 5.6.1 *We define equality between sequences as:*

1. $\varepsilon \stackrel{?}{=} \varepsilon$
2. $\tilde{t_1}, \tilde{t_2} \stackrel{?}{=} \tilde{t_3}, \tilde{t_4}$ *iff* $\tilde{t_1} \stackrel{?}{=} \tilde{t_3}$ *and* $\tilde{t_2} \stackrel{?}{=} \tilde{t_4}$
3. $\varepsilon, \tilde{t} \stackrel{?}{=} \tilde{t}$
4. $\tilde{t}, \varepsilon \stackrel{?}{=} \tilde{t}$

Definition 5.6.2 *Given the sequence term $\bar{t}$, function* ts *translates $\bar{t}$ to the equivalent sequence $\tilde{t}$.* ts *is defined as follows:*

$$\begin{array}{ccc} ts(\varepsilon) & = & \varepsilon \\ ts(f(\bar{t})) & = & f(ts(\bar{t})) \\ ts(seq(t_1, \bar{t})) & = & ts(t_1), ts(\bar{t}) \\ ts(t) & = & t, \textit{ otherwise} \end{array}$$

Example 5.6.1 *Let's $seq(seq(\varepsilon, a), seq(b, \varepsilon)$ be a sequence term, then:*

$$\begin{array}{rcl} ts(seq(seq(\varepsilon, a), seq(b, \varepsilon)) & = & ts(seq(\varepsilon, a)), ts(seq(b, \varepsilon)) \\ & = & ts(\varepsilon), ts(a), ts(b), ts(\varepsilon) \\ & = & \varepsilon, a, b, \varepsilon \end{array}$$

By definition 5.6.1, $\varepsilon, a, b, \varepsilon = a, b$.

Lemma 5.6.1 *Let t_1 and t_2 be two sequence terms then:*

$$ts(t_1 + +t_2) \stackrel{?}{=} ts(t1), ts(t2)$$

Proof 5.6.1 *By induction on the length of the sequence. The base case corresponds to concatenate a sequence with ε, in this case, $ts(\bar{t}) \stackrel{?}{=} ts(\bar{t})$ is obviously true. Our hypothesis states that, given two sequences $\bar{t}$ and $\bar{s}$, then $ts(\bar{t} + +\bar{s}) \stackrel{?}{=} ts(\bar{t}), ts(\bar{s})$. Let's prove that $ts(seq(t_1, \bar{t_2}))$ ++ $\bar{t_3}) \stackrel{?}{=} ts(seq(t_1, \bar{t_2}))$,$ts(\bar{t_3})$.*

$$\begin{array}{rcl} ts(seq(t_1, \bar{t_2}) + +\bar{t_3}) & = & ts(seq(t_1, \bar{t_2} + +\bar{t_3}) \text{ , by definition of ++} \\ & = & ts(t_1), ts(\bar{t_2} + +\bar{t_3}), \text{ by the definition of ts} \\ & = & ts(t_1), ts(\bar{t_2}), ts(\bar{t_3}), \text{ by the I.H.} \end{array}$$

Now,

$$ts(seq(t_1, \bar{t_2})), ts(\bar{t_3}) = ts(t_1), ts(\bar{t_2}), ts(\bar{t_3}), \text{ by the definition of ts.}$$

Obviously,

$$ts(t_1), ts(\bar{t_2}), ts(\bar{t_3}) \stackrel{?}{=} ts(t_1), ts(\bar{t_2}), ts(\bar{t_3}). \square$$

Sometimes we need to *normalize* the sequences in order to avoid returning types of sequences that are not in normal form.

Definition 5.6.3 *Given a sequence type, we define sequence type* normalization *as:*

$$\begin{array}{lcl} norm_{\cap}(\varepsilon) & = & \varepsilon \\ norm_{\cap}(t) & = & seq(t,\varepsilon), \text{ if } t \text{ is a constant or a type symbol} \\ norm_{\cap}(f(t)) & = & seq(f(norm_{\cap}(t)),\varepsilon) \\ norm_{\cap}(seq(t_1,\bar{t})) & = & norm_{\cap}(t_1) \mathbin{++} norm_{\cap}(\bar{t}) \end{array}$$

The *ts* function can be trivially lifted to sets of terms by just applying *ts* to every element of the set.

Lemma 5.6.2 (Correctness of normalization) *Let $\bar{t}$ be a sequence term. Then,*

$$ts([\bar{t}]_T) \stackrel{?}{=} ts([norm_{\cap}(\bar{t})]_T)$$

Proof 5.6.2 *We use induction on the length of the sequence term. The base cases are:*

- $ts([\varepsilon]_T) \stackrel{?}{=} ts([norm_{\cap}(\varepsilon)]_T)$. *We have that* $ts([\varepsilon]_T) = ts(\varepsilon) = \varepsilon$ *and* $ts([norm_{\cap}(\varepsilon)]_T) = ts([\varepsilon]_T)$, *by the definition of* $norm_{\cap}$ *and* $ts([\varepsilon]_T) = \varepsilon$.

- $ts([t]_T) \stackrel{?}{=} ts([norm_{\cap}(t)]_T)$, *where t is a constant, then* $ts([t]_T) = t$ *and* $ts([norm_{\cap}(t)]_T) = t$.

The induction case follows:

- $ts([t]_T) \stackrel{?}{=} ts([norm_{\cap}(t)]_T)$, *where t is a type symbol. Here we have that,* $[t]_T$ *is going to lead to a set of* t_i *where each* t_i *is the result of the evaluation of t based on rules in T. For* $[norm_{\cap}(t)]_T$, *we will get a set of* $norm_{\cap}(t_i)$. *Since* $t_i = [t_i]_{\{\}}$ *and* $norm_{\cap}(t_i) = [norm_{\cap}(t_i)]_{\{\}}$, $ts(t) \stackrel{?}{=} ts(norm_{\cap}(t))$ *follows by the induction hypothesis.*

- $ts([f(\bar{t})]_T) \stackrel{?}{=} ts([norm_{\cap}\ (f(\bar{t}))]_T)$. *We have that* $[f(\bar{t})]_T = \{f(\bar{t}') \mid \bar{t}' \in [\bar{t}]_T\}$ *and* $ts([f(\bar{t})]_T) = \{ts(f(\bar{t}')) \mid \bar{t}' \in [t_i]_T\}$. *Now,* $[norm_{\cap}\ (f(\bar{t}))]_T = [f(norm_{\cap}\ (\bar{t}))]_T$, *by the definition of* $norm_{\cap}$ *and* $[f(\ norm_{\cap}\ (\bar{t}))]_T = \{f(norm_{\cap}\ (\bar{t}')) \mid \bar{t}' \in [\bar{t}]_T\}$. *We have that,* $ts(\ [norm_{\cap}\ (f(\bar{t}))]_T) = \{ts(f(norm_{\cap}\ (\bar{t}')))| \ \bar{t}' \in [\bar{t}]_T\}$. *By the I.H.* $ts([\bar{t}]_T) \stackrel{?}{=} ts([norm_{\cap}\ (\bar{t})]_T)$, *thus* $ts([f(\bar{t})]_T) \stackrel{?}{=} ts(f([norm_{\cap}\ (\bar{t})]_T)) \stackrel{?}{=} ts([norm_{\cap}\ (f(\bar{t}))]_T)$

- $ts([seq(t_1,\bar{t})]_T) \stackrel{?}{=} ts([norm_{\cap}\ (seq(t_1,\bar{t}))]_T)$. *We have that,* $[seq(t_1,\ \bar{t})]_T = \{seq(t'_1,\bar{t}') \mid t'_1 \in [t_1]_T \wedge \bar{t}' \in [\bar{t}]_T\}$ *and* $ts([seq(t_1,\bar{t})]_T) = \{ts(seq(t'_1,\ \bar{t}')) \mid t'_1 \in [t_1]_T \wedge \bar{t}' \in [\bar{t}]_T\} = \{ts(t'_1),\ ts(\bar{t}') \mid t'_1 \in [t_1]_T \wedge \bar{t}' \in [\bar{t}]_T\}$, *by the definition of ts. Now,* $[norm_{\cap}\ (seq(t_1,\bar{t}))]_T = \{norm_{\cap}\ (seq(t'_1,\bar{t}')) \mid t'_1 \in [t_1]_T \wedge \bar{t}' \in [\bar{t}]_T\}$ *and* $ts([norm_{\cap}\ (seq(t_1,\bar{t}))]_T) = \{ts(norm_{\cap}\ (seq(t'_1,\bar{t}'))) \mid t'_1 \in [t_1]_T \wedge \bar{t}' \in [\bar{t}]_T\} = \{ts(norm_{\cap}(t'_1) \mathbin{++} norm_{\cap}(\bar{t}')) \mid t'_1 \in [t_1]_T \wedge \bar{t}' \in [\bar{t}]_T\}$, *by the definition of* $norm_{\cap}$. *This results in* $\{ts(norm_{\cap}(t'_1)),\ ts(norm_{\cap}(\bar{t}')) \mid t'_1 \in [t_1]_T \wedge \bar{t}' \in [\bar{t}]_T\}$, *by proposition 5.6.1. Like in the previous cases we can apply the induction hypothesis and the result follows.*

Empty types are types that only describe the empty set. Another useful function is *empty*, described bellow: it returns $true$ if a type is empty or $false$ otherwise.

Definition 5.6.4 *Given a type of a sequence α and a set of rules T, the following function tests if α is empty:*

$$\begin{array}{lcl} empty(\alpha, T) & = & true,\ if\ \alpha \in T\ or\ \alpha = \phi \\ empty(\alpha, T) & = & false,\ if\ \alpha\ is\ a\ constant\ or\ \mu \\ empty(\alpha, T) & = & empty(\alpha_1, T \cup \{\alpha\}) \wedge, \ldots, \wedge, empty(\alpha_n, T \cup \{\alpha\}), \\ & & if\ \alpha \rightarrow \{\alpha_1, \ldots, \alpha_n\} \\ empty(f(s), T) & = & empty(s, T) \end{array}$$

Note that the function empty is initially called as $empty(\alpha, \emptyset)$

The correctness of *empty* follows by the correction of a similar function applied to general regular types in [Zob90].

Example 5.6.2 *Let α be a type symbol defined by the type rule, $\alpha \rightarrow \{seq(a, \alpha)\}$, α is empty.*

We now present the sequence type intersection algorithm. We use two global variables, $TRules$ for the rules describing the types and T which stores the intersections already made in order to avoid cycles. Whenever more than one case is applicable the first one is used. The algorithm is described in figure 5.1

Example 5.6.3 *Let's calculate the intersection $seq(\alpha, \varepsilon) \cap seq(a, \alpha)$ with $TRules = \{\alpha \rightarrow \{\varepsilon,\ seq(a, \alpha\}\ \}$ and $T = \{\}$. This intersection fits the case number 9, $seq(\alpha, \varepsilon) \cap seq(a, \alpha) = \alpha_f$, where $T = \{\ (seq(\alpha, \varepsilon),\ seq(a, \alpha)\ , \alpha_f)\ \}$. Two cases follow:*

1. *$norm_\cap(seq(\varepsilon, \varepsilon)) \cap norm_\cap(seq(a, \alpha)) = \alpha_{f1}$, this results in $\varepsilon \cap seq(a, \alpha) = \phi$, thus $\alpha_{f1} = \phi$.*

2. *$norm_\cap(seq(seq(a, \alpha), \varepsilon)) \cap norm_\cap(seq(a, \alpha)) = \alpha_{f2}$. Then by case 1, $seq(a,\ seq(\alpha,\ \varepsilon)) \cap seq(a,\ seq(\alpha, \varepsilon)) = seq(a,\ seq(\alpha,\ \varepsilon))$, thus $\alpha_{f2} = seq(a,\ seq(\alpha,\ \varepsilon))$*

Thus we have $\alpha_f \rightarrow \{seq(a, seq(\alpha, \varepsilon))\}$.

Example 5.6.4 *Given the two sequence types, $seq(\alpha, \varepsilon)$ and $seq(\beta, \varepsilon)$ and type rules $\alpha \rightarrow \{\varepsilon, seq(a, \alpha)\}$ and $\beta \rightarrow \{a, seq(a, \alpha)\}$. Let's calculate $seq(\alpha, \varepsilon) \cap seq(\beta, \varepsilon)$. By case 9, $seq(\alpha, \varepsilon) \cap seq(\beta, \varepsilon) = \gamma$, $T = \{\ (seq(\alpha, \varepsilon), seq(\beta, \varepsilon), \gamma)\}$, and:*

- *$norm_\cap(seq(\varepsilon, \varepsilon)) \cap norm_\cap(seq(a, \varepsilon)) = \phi$*
- *$norm_\cap(seq(\varepsilon, \varepsilon)) \cap norm_\cap(seq(seq(a, \alpha), \varepsilon)) = \phi$*

(1) $X \cap Y = X$, if $X == Y$ [1]

(2) $X \cap Y = Z$, if $(X, Y, Z) \in T$ or $(Y, X, Z) \in T$

(3) $X \cap seq(\mu, \varepsilon) = X$

(4) $seq(\mu, \varepsilon) \cap Y = Y$

(5) $f(\tau_1) \cap f(\tau_2) = f(\tau_1 \cap \tau_2)$

(6) $seq(\tau_1, \bar{\tau_1}) \cap seq(\tau_2, \bar{\tau_2}) = norm_{\cap}(seq(\tau_1 \cap \tau_2, \bar{\tau_1} \cap \bar{\tau_2}))$

(7) $seq(\alpha, \varepsilon) \cap \varepsilon = \varepsilon$,
if
$(seq(\alpha, \varepsilon), \varepsilon, \alpha') \notin T$,
$T = T \cup \{(seq(\alpha, \varepsilon), \varepsilon, \varepsilon)\}$,
$\alpha \to \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in TRules$
and $\exists_i . norm_{\cap}(\alpha_i) \cap \varepsilon = \varepsilon$

(8) $\varepsilon \cap seq(\alpha, \varepsilon) = \varepsilon$,
if
$(seq(\alpha, \varepsilon), \varepsilon, \alpha') \notin T$,
$T = T \cup \{(seq(\alpha, \varepsilon), \varepsilon, \varepsilon)\}$,
$\alpha \to \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in TRules$
and $\exists_i . norm_{\cap}(\alpha_i) \cap \varepsilon = \varepsilon$

(9) $seq(\alpha, \bar{\tau_1}) \cap seq(\tau_2, \bar{\tau_2}) = \alpha_f$, where α_f is a new type symbol and given
that $\alpha \to \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in TRules$,
$T = T \cup \{(seq(\alpha, \bar{\tau_1}), seq(\tau_2, \bar{\tau_2}), \alpha_f)\}$, $NR = \{\}$
for each α_i,
  $norm_{\cap}(seq(\alpha_i, \bar{\tau_1})) \cap seq(\tau_2, \bar{\tau_2}) = \alpha_i$,
  if $empty(\alpha_i, \{\}) = false$
  then $NR = NR \cup \{\alpha_i\}$, $1 \leq i \leq n$
end_for_each
if NR is $\{\}$ then $\alpha_f = \phi$
else $TRules = TRules \cup \{\alpha_f \to NR\}$

(10) $seq(\tau_1, \bar{\tau_1}) \cap seq(\alpha, \bar{\tau_2}) = \alpha_f$, where α_f is a new type symbol and given
that $\alpha \to \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in TRules$,
$T = T \cup \{(seq(\tau_1, \bar{\tau_1}), seq(\alpha, \bar{\tau_2}), \alpha_f)\}$, $NR = \{\}$
for each α_i,
  $norm_{\cap}(seq(\alpha_i, \bar{\tau_1})) \cap seq(\tau_1, \bar{\tau_2}) = \alpha_{fi}$,
  if $empty(\alpha_i, \{\}) = false$
  then $NR = NR \cup \{\alpha_i\}$, $1 \leq i \leq n$
end_for_each
if NR is $\{\}$ then $\alpha_f = \phi$
else $TRules = TRules \cup \{\alpha_f \to NR\}$

(11) $seq(\alpha, \bar{\tau_1}) \cap seq(\beta, \bar{\tau_2}) = \gamma$, where γ is a new type symbol and given that,
$\alpha \to \{\alpha_1, \alpha_2, \ldots, \alpha_n\} \in TRules$,
$\beta \to \{\beta_1, \beta_2, \ldots, \beta_m\} \in TRules$,
$T = T \cup \{(seq(\alpha, \bar{\tau_1}), seq(\beta, \bar{\tau_2}), \gamma)\}$, $NR = \{\}$
for each α_i and β_j
  $norm_{\cap}(seq(\alpha_i, \bar{\tau_1})) \cap norm_{\cap}(seq(\beta_j, \bar{\tau_2})) = \gamma_k$,
  if $empty(\gamma_k,) = false$,
  $then\, NR = NR \cup \{\gamma_k\}$, $1 \leq k \leq w$
  where $w = n * m$,
end_for_each
if NR is $\{\}$ then $\gamma = \phi$
else $TRules = TRules \cup \{\gamma \to NR\}$

(12) $\tau_1 \cap \tau_2 = \phi$, if none of the previous rules is applicable.

Figure 5.1: Intersection of sequences of types

- *$norm_\cap(seq(a,\alpha),\varepsilon)) \cap norm_\cap(seq(a,\varepsilon)) = seq(a, seq(\alpha,\varepsilon)) \cap seq(a,\varepsilon)$ thus by case 6:*
  - *$a \cap a = a$.*
  - *$seq(\alpha,\varepsilon) \cap \varepsilon = \varepsilon$, by rule 7.*

  This results in $seq(a,\varepsilon)$.
- *$norm_\cap(seq(seq(a,\ \alpha),\ \varepsilon)) \cap norm_\cap\ (seq(seq(a,\ \alpha),\ \varepsilon)) = seq(a,\ seq(\alpha,\ \varepsilon)) \cap seq(a,\ seq(\alpha,\ \varepsilon)) = seq(a,\ seq(\alpha,\ \varepsilon))$, by rule 1.*

Now, the intersection γ is defined by $\gamma \rightarrow \{\ seq(a,\varepsilon),\ seq(a,\ seq(\alpha,\ \varepsilon))\}$, $\alpha \rightarrow \{\varepsilon, seq(a,\alpha)\}$.

Example 5.6.5 *Let's calculate the intersection of the following sequences:*

$$seq(l(seq(\alpha_1, seq(\alpha_2,\varepsilon))),\varepsilon) \cap seq(l(seq(a, seq(a, seq(\alpha_5,\varepsilon)))),\varepsilon)$$

with,

$$\begin{array}{lcl}
\alpha_1 & \rightarrow & \{\varepsilon, seq(a,\alpha_1)\} \\
\alpha_2 & \rightarrow & \{seq(b(seq(\alpha_3, seq(\alpha_4,\varepsilon))),\varepsilon)\} \\
\alpha_3 & \rightarrow & \{seq(c,\varepsilon), seq(c,\alpha_3)\} \\
\alpha_4 & \rightarrow & \{\varepsilon, seq(d,\alpha_4)\} \\
\alpha_5 & \rightarrow & \{seq(b(seq(\alpha_6,\varepsilon)),\varepsilon)\} \\
\alpha_6 & \rightarrow & \{\varepsilon, seq(c,\alpha_6)\}
\end{array}$$

For the sake of simplicity we will omit some steps. From rule 6 we have that

$$\begin{array}{c}
seq(l(seq(\alpha_1, seq(\alpha_2,\varepsilon))),\varepsilon) \cap seq(l(seq(a, seq(a, seq(\alpha_5,\varepsilon)))),\varepsilon) = \\
seq(l(seq(\alpha_1, seq(\alpha_2,\varepsilon))) \cap l(seq(a, seq(a, seq(\alpha_5,\varepsilon)))),\varepsilon \cap \varepsilon)
\end{array}$$

by rule number 5, $l(seq(\alpha_1, seq(\alpha_2,\varepsilon))) \cap l(seq\ (a, seq\ (a, seq(\alpha_5,\varepsilon))))$ yields $l(seq\ (\alpha_1, seq\ (\alpha_2,\varepsilon)) \cap seq(a,\ seq(a,\ seq(\alpha_5,\varepsilon))))$.

From rule number 9 a new type symbol α_{f1} is generated and it is defined by two intersections, $seq(\alpha_2,\ \varepsilon) \cap seq(a,\ seq(a,\ seq(\alpha_5,\ \varepsilon)))$, which results in ϕ and $seq(a,\ seq(\alpha_1,\ seq(\alpha_2,\ \varepsilon))) \cap seq(a,\ seq(a,\ seq(\alpha_5,\ \varepsilon)))$.

Then, by rule number 6 this is $seq(a,\ seq(\alpha_1,\ seq(\alpha_2,\ \varepsilon)) \cap seq(a,\ seq(\alpha_5,\ \varepsilon)))$.

Thus by rule number 9 we have a new type symbol, α_{f2} defined by:

1. *$seq(\alpha_2,\varepsilon) \cap seq(a, seq(\alpha_5,\varepsilon)) = \phi$*
2. *$seq(a, seq(\alpha_1,\ seq(\alpha_2,\varepsilon))) \cap seq(a, seq(\alpha_5,\ \varepsilon)) = seq(a\ , seq(\alpha_1,\ seq(\alpha_2,\varepsilon)) \cap seq(\alpha_5,\varepsilon))$, by rule 6. Now, by rule 11, $seq(\alpha_1,\ seq(\alpha_2,\ \varepsilon)) \cap seq(\alpha_5,\ \varepsilon)$ results in a new type symbol, α_{f3} defined by:*

(a) *seq*(α_2, ε) $\cap$ *seq*(α_5, ε), *by rule 11 results in a new type symbol* α_{f4} *with type rule defined by* *seq*(*b*(*seq*(α_3, *seq*(α_4 ,ε))), ε) $\cap$ *seq*(*b*(*seq*(α_6, ε)), ε) *which, by rule 5 is* *seq*(*b*(*seq*(α_3, *seq*(α_4, ε)) $\cap$ *seq*(α_6, ε)), ε). *Thus by rule 11,* *seq*(α_3, *seq*(α_4, ε)) $\cap$ *seq*(α_6, ε) *results in a new type symbol,* α_{f5} *where:*

i. $seq(c, seq(\alpha_4, \varepsilon)) \cap \varepsilon = \phi$

ii. $seq(c, seq(\alpha_3,\ seq(\alpha_4, \varepsilon))) \cap \varepsilon = \phi$

iii. $seq(c, seq(\alpha_4, \varepsilon)) \cap seq(c, \alpha_6)$ *by rule 6 is* $seq(c, seq(\alpha_4, \varepsilon) \cap \alpha_6)$ *and* $seq(\alpha_4, \varepsilon)$ $\cap$ $seq(\alpha_6, \varepsilon)$ *results in a new type symbol* α_{f6} *where:*

A. $\varepsilon \cap \varepsilon = \varepsilon$

B. $\varepsilon \cap seq(c, seq(\alpha_6, \varepsilon)) = \phi$

C. $seq(d, \varepsilon) \cap \varepsilon = \phi$

D. $seq(d, \varepsilon) \cap seq(c, seq(\alpha_6, \varepsilon)) = \phi$

So the result is $seq(c, \alpha_{f6})$, *where* $\alpha_{f6} \rightarrow \{\varepsilon\}$

iv. $seq(c, seq(\alpha_3, seq(\alpha_4, \varepsilon))) \cap seq(c, \alpha_6)$, *by rule 6 is* *seq*(*c*, *seq*(α_3, *seq*(α_4, ε)) $\cap$ α_6) *and* *seq*(α_3 ,*seq*(α_4, ε)) $\cap$ *seq*(α_6 ,ε) *results, by rule 11 in a new type symbol,* α_{f7} *where:*

A. $seq(c, seq(\alpha_4, \varepsilon)) \cap \varepsilon = \phi$

B. $seq(c, seq(\alpha_3, seq(\alpha_4, \varepsilon))) \cap \varepsilon = \phi$

C. *seq*(*c*, *seq*(α_4, ε)) $\cap$ *seq*(*c*, *seq*(α_6, ε)), *by rule 6 is* *seq*(*c*, *seq*(α_4, ε) $\cap$ *seq*(α_6, ε)), *but this was already done before, thus the result is* *seq*(*c*, α_{f6}).

D. *seq*(*c*, *seq*(α_3, *seq*(α_4, ε))) $\cap$ *seq*(*c*, *seq*(α_6, ε)) *which, by rule 6 is* *seq*(*c*, *seq*(α_3, *seq*(α_4, ε)) $\cap$ *seq*(α_6, ε)), *but* *seq*(α_3, *seq*(α_4, ε)) $\cap$ *seq*(α_6, ε) *was already tried before, this results in* *seq*(*c*, α_{f7}).

So we have that $\alpha_{f7} \rightarrow \{seq(c, \alpha_{f6}), seq(c, \alpha_{f7})\}$,

This results in $\alpha_{f5} \rightarrow \{seq(c,\ \alpha_{f6},\ seq(c,\ \alpha_{f7})\}$ *and* $\alpha_{f4} \rightarrow \{seq(b(\alpha_{f5}),\ \varepsilon)\}$

(b) $seq(a, seq(\alpha_1, seq(\alpha_2, \varepsilon))) \cap seq(\alpha_5, \varepsilon) = \phi$

Thus, $\alpha_{f3} \rightarrow \{\alpha_{f4}\}$

Now, $\alpha_{f2} \rightarrow \{\ seq(a,\ \alpha_{f3})\ \}$ *and* $\alpha_{f1} \rightarrow \{\ seq(a,\ \alpha_{f2})\ \}$.

The final result is *seq*(*l*(α_{f1}), ε). *This result can be simplified to* *seq*(*l*(*seq*(*a*, *seq*(*a*, α_f))), ε) *where* $\alpha_f \rightarrow \{seq(b(seq(c,\ \alpha'_f)),\ \varepsilon)\}$ *and* $\alpha'_f \rightarrow \{\varepsilon,\ seq(c, \alpha'_f)\}$.

Lemma 5.6.3 *Let* α_1 *and* α_2 *be type terms defined by a set of type rules* $TRules$, *where each type symbol in* α_1 *and* α_2 *and* $TRules$ *has exactly one defining rule in* $TRules$. *If* $\alpha_1 \cap \alpha_2$ *returns* α_f *and set of type rules* T' *then* α_f *is a type term defined by the set of type rules* T' *and each type symbol in* α_f *and* T' *has exactly one defining rule in* T'.

Proof 5.6.3 *It is straightforward to see that the result of intersection is always a type term. New type rules are introduced in cases 9, 10 and 11 where the body of a new type rule is computed recursively by the intersection. New type symbols are only introduced in cases 9, 10 and 11 where a new type rule for this new type is always introduced. The result follows by induction on the number of calls to the intersection.*

Theorem 5.6.1 (Correctness of intersection) *Let α_1 and α_2 be type terms defined by a set of type rules T. Then $\alpha_1 \cap \alpha_2$ terminates and returns α_f and T' and $[\alpha_f]_{T'} = [\alpha_1]_T \cap [\alpha_2]_T$.*

Proof 5.6.4 *Let's prove the theorem above in two steps:*

Termination *Consider the set T, from rule 2, there are never two triples $(\alpha_1, \beta_1, \gamma_1) \in T$ and $(\alpha_2, \beta_2, \gamma_2) \in T$ such that $\alpha_1 = \alpha_2$ and $\beta_1 = \beta_2$. Thus each (α_1, α_2) is distinct. From alternatives 9, 10 and 11, for each $(\alpha_1, \beta_1, \gamma_1) \in T$, α_1 and α_2 are either subterms of the initial terms or are subterms of terms occurring on the right-hand side of type rules in the set of type rules given as input to intersection. Thus the set of possible (α_1, α_2) pairs is finite. Let T_1 be the set of pairs $\{(\alpha_1, \beta_1) | (\alpha_1, \beta_1, \gamma_1) \in T\}$ and $\bar{T_1}$ be the finite set of possible (α_1, α_2) pairs not in T_1. Suppose that pairs of the form $(\bar{T_1}, \alpha_1)$ are ordered such that $(\bar{T_1}, \alpha_1) \leq (\bar{T_2}, \alpha_2)$ if $\bar{T_1}$ is a subset of $\bar{T_2}$ or if this is not the case, α_1 is strictly a subterm of α_2. It is easily shown that $\leq$ is well-founded [BN98]. The pair $(\bar{T_1}, \alpha_1)$ decreases by the previous order in each recursive call to intersection. Thus intersection terminates.*

Correctness *Let's now prove that the result is correct using induction on the cardinality of the set $[\tau]_T$. The base case corresponds to steps number 1 through 4 and 12, the result follows by direct observation of each step result. Let's now see the other steps:*

Case 5 *Here the case follows by a direct application of the induction hypothesis.*

Case 6 *$t_1 \cap s_1$ fits one of the base cases and by the induction hypothesis $\bar{t} \cap \bar{s}$ is correct. Thus, the result holds for seq($t_1 \cap s_1$, $\bar{t} \cap \bar{s}$). Since the norm$_\cap$ function has no effect in the type symbols in a sequence (it just rewrites the sequence in its normal form), the result also holds for norm$_\cap$(seq($t_1 \cap s_1$, $\bar{t} \cap \bar{s}$)).*

Case 7 *In this case we test if some α_{1i} is ε. We can apply the induction hypothesis to norm$_\cap$ (seq(α, ε)) $\cap\ \varepsilon$ since $[\alpha_{1i}]_T \subset [\alpha]_T$ and the result holds.*

Case 8 *This case is analogous to the previous.*

Case 9 *Since $[\alpha_{1i}]_T \subset [\alpha]_T$, by the induction hypothesis norm$_\cap$ (seq(α_{1i}, $\bar{t}$)) $\cap$ seq(s_1, $\bar{s}$) $= \alpha_{fi}$ is correct. Thus, α_f is also correct.*

Case 10 *This case is analogous to the previous.*

Case 11 *This case is similar to the two above. Since $[\alpha_{1i}]_T \subset [\alpha]_T$ and $[\beta_{1i}]_T \subset [\beta]_T$ the result follows by the induction hypothesis.* □

5.6.2 Typed unification for terms with flexible arity

Here we describe a typed unification procedure. Types are annotated for the variables in sequences of pairs, *Var::Type*, where *Var* is a free variable and *Type* is a type symbol describing the set of terms that can be instantiated with *Var*. Every time a variable is associated with a new type and a term (rules 6, 7 and 8), all the occurrences of this variable are updated with the new information.

Example 5.6.6 *Given the variable X and the non-empty type α, defined by the rule $\alpha \rightarrow \{\varepsilon,\ seq(a, \alpha)\}$, $X :: \alpha$ describes variable X constrained to sequences of zero or more a's.*

Definition 5.6.5 *A* typed term *is a term containing type annotations.*

We now define some auxiliary functions.

Definition 5.6.6 *Given a sequence that may have type annotations, the* type *function, returns its type.*

$$
\begin{array}{lcl}
type(\varepsilon) & = & \varepsilon \\
type(seq(X :: \alpha, \bar{s}) & = & seq(\alpha, type(\bar{s})),\ X \text{ is a variable and } \bar{s} \text{ is a sequence} \\
type(seq(f(\bar{t}), \bar{s})) & = & seq(f(type(\bar{t})), type(\bar{s})),\ \bar{t} \text{ and } \bar{s} \text{ are sequences} \\
type(seq(c, \bar{s})) & = & seq(c, type(\bar{s})),\ c \text{ is a constant and } \bar{s} \text{ is a sequence}
\end{array}
$$

Definition 5.6.7 *Given a sequence term, we define sequence term* normalization *as:*

$$
\begin{array}{lcl}
norm_*(\varepsilon) & = & \varepsilon \\
norm_*(t) & = & seq(t,\varepsilon),\ \text{if } t \text{ is a constant or variable with} \\
 & & \text{type annotation.} \\
norm_*(f(\bar{t})) & = & seq(f(norm_*(\bar{t})),\varepsilon) \\
norm_*(seq(t_1,\bar{t})) & = & norm_*(t_1)\ {+}{+}\ norm_*(\bar{t})
\end{array}
$$

Theorem 5.6.2 *Given a sequence term $\bar{t}$, describing sequence $ts(\bar{t}) = t$, the application of the normalization function to $\bar{t}$ does not change t:*

$$ts(norm_*(\bar{t})) \stackrel{?}{=} ts(\bar{t})$$

Proof 5.6.5 *Let's prove by induction on the length of the sequence term. Our base cases are:*

- $ts(norm_*(\varepsilon)) \stackrel{?}{=} ts(\varepsilon)$*, we have that* $ts(norm_*(\varepsilon)) = ts(\varepsilon)$*, by the definition of* $norm_*$*, so the result holds.*
- $ts(norm_*(t)) \stackrel{?}{=} ts(t)$*, where t is a constant or variable with type annotation. Here we have that* $ts(norm_*(t)) = ts(seq(t, \varepsilon))$*, by the definition of* $norm_*$ *and* $ts(seq(t, \varepsilon)) =$

$ts(t)$, $ts(\varepsilon)$, by the definition of ts *and $ts(t)$, $ts(\varepsilon) = t, \varepsilon$, again by the definition of* ts. *Thus, since $ts(t) = t$, by the definition of* ts *and $t, \varepsilon \stackrel{?}{=} t$, by definition 5.6.1, the result holds.*

Induction step:

- *$ts(norm_*(f(\bar{t}))) \stackrel{?}{=} ts(f(t))$, we have that $ts(norm_* \ (f(\bar{t}))) = ts(f(norm_* \ (\bar{t})))$, by the definition of $norm_*$ and $ts(f(norm_* \ (\bar{t}))) = f(ts(norm_* \ (\bar{t})))$, by the definition of* ts. *Now, $ts(f(t)) = f(ts(t))$, by the definition of* ts *and by the induction hypothesis the result holds.*
- *$ts(norm_* \ (seq(t_1, \bar{t}))) \stackrel{?}{=} ts(seq(t_1, \bar{t}))$, we have that $ts(norm_* \ (seq(t_1, \bar{t}))) = ts(norm_* \ (t_1) \ {+}{+} \ norm_* \ (\bar{t}))$ by the definition of $norm_*$ and $ts(norm_* \ (t_1) \ {+}{+} \ norm_* \ (\bar{t})) = ts(norm_* \ (t_1)), ts(norm_* \ (\bar{t}))$ by proposition 5.6.1. Now, $ts(seq(t_1, \bar{t})) = ts(t_1), ts(\bar{t})$, by definition of* ts. *By the induction hypothesis the result holds.*

Definition 5.6.8 *Given a sequence term $\bar{t}$ and a type α, such that all the types in $\bar{t}$ and α are defined by the set T and $[\alpha]_T \subseteq [type(\bar{t})]_T$, the function* type rewrite (tr) *is defined as follows:*

$$
\begin{array}{lcl}
tr(c, c) & = & c, \text{ if } c \text{ is a constant} \\
tr(c, seq(c, \varepsilon)) & = & c, \text{ if } c \text{ is a constant} \\
tr(X :: \alpha, \beta) & = & X :: \beta, \ \alpha \text{ and } \beta \text{ are type symbols} \\
tr(X :: \alpha, \bar{t}) & = & X :: \alpha_1, \text{ where } \alpha \text{ is a type symbol, } \bar{t} \text{ is a} \\
 & & \text{sequence term or constant and } \alpha_1 \text{ is a new} \\
 & & \text{type symbol defined by } \alpha_1 \to \{\bar{t}\} \\
tr(f(\bar{t}_1), f(\bar{t}_2)) & = & f(tr(\bar{t}_1, \bar{t}_2)) \\
tr(seq(t, \varepsilon), s) & = & seq(tr(t, s), \varepsilon), \text{ if } s \text{ is not of the form } seq(s_1, \bar{s}) \\
tr(\bar{t}, \alpha) & = & \bar{t}' \text{ where } \alpha \to \{\alpha_1, \ldots, \alpha_n\} \text{ and} \\
 & & tr(\bar{t}, \alpha_1) = \bar{t}'_1 \\
 & & \vdots \\
 & & tr(\bar{t}, \alpha_n) = \bar{t}'_n \\
 & & \text{where} \\
 & & vars(\bar{t}'_1) = \{X_1 :: \beta_{11}, \ldots, X_n :: \beta_{1n}\} \\
 & & \vdots \\
 & & vars(\bar{t}'_n) = \{X_1 :: \beta_{1n}, \ldots, X_n :: \beta_{nn}\} \\
 & & \text{for each } X_i \text{ create a new type symbol } \gamma_i \\
 & & \text{where } \gamma_i \to \{\beta_{i1}, \ldots, \beta_{in}\}, \text{ now } \bar{t}' \text{ is } \bar{t}, \\
 & & \text{where each type associated with each} \\
 & & \text{variable } X_i \text{ is replaced by } \gamma_i \\
tr(seq(t_1, \bar{t}), seq(s_1, \bar{s})) & = & seq(tr(t_1, s_1), tr(\bar{t}, \bar{s}))
\end{array}
$$

tr rewrites the original sequence term accordingly to α, where the result is a new sequence $\bar{t}'$, such that $type(\bar{t}') = \alpha$. Type rewrite will be used in the unification procedure presented ahead. The idea is to rewrite a sequence $\bar{t}$ using a type α, where α was calculated by the intersection of $type(\bar{t})$ with another type and α is not an empty type.

Example 5.6.7 *Given a sequence of* a's *or* b's *with an* a *at the head represented by* $seq(a, seq(X :: \alpha))$ *where* $\alpha \rightarrow \{\varepsilon, seq(a,\varepsilon), seq(b,\varepsilon)\}$, *its type can be restricted to a sequence of one or more* a's *given by* $seq(a, seq(\alpha_1, \varepsilon))$ *where,* $\alpha_1 \rightarrow \{\varepsilon, seq(a,\varepsilon)\}$. *This is done by applying tr:*

$$\begin{array}{ll} tr(seq(a, seq(X :: \alpha, \varepsilon)), seq(a, seq(\alpha_1, \varepsilon))) & = \\ seq(tr(a,a), tr(seq(X :: \alpha, \varepsilon), seq(\alpha_1, \varepsilon))) & = \\ seq(a, seq(tr(X :: \alpha, \alpha_1), tr(\varepsilon, \varepsilon))) & = \\ seq(a, seq(X :: \alpha_1, \varepsilon)) & \end{array}$$

as a result X *is now typed by* α_1.

Theorem 5.6.3 *Given a sequence* $\bar{t}$ *and a type* α, *which results from the intersection of* $type(\bar{t})$ *with another sequence type, then:*

$$tr(\bar{t}, \alpha) = \bar{s} \Rightarrow type(\bar{s}) = \alpha$$

Proof 5.6.6 *We prove this result by induction on the number of steps of the procedure. The base cases correspond to the first, second and third cases. In the first case, a constant is rewritten to the same constant, in the second and third cases, a variable is rewritten accordingly to the type of the second argument of* tr *and thus the result holds. The induction hypothesis cases follows:*

- $tr(f(\bar{t}_1), f(\bar{t}_2)) = f(tr(\bar{t}_1, \bar{t}_2))$, *by the induction hypothesis, the result holds for* $tr(\bar{t}_1, \bar{t}_2)$ *and trivially, also holds for* $f(tr(\bar{t}_1, \bar{t}_2))$.
- $tr(seq(t,\varepsilon), s) = seq(tr(t,s),\varepsilon)$, *since* s *is not in sequence form, it must be a constant or of the form* $f(\bar{s})$, *the intersection must be done with* t. $tr(t,s)$ *will then fit one of the first four cases, thus it outputs a correct result and* $seq(tr(t,s),\varepsilon)$ *is also correct.*
- $tr(\bar{t}, \alpha) = tr(\bar{t}, \bar{s})$, *where* $\alpha \rightarrow \{\alpha_1, \ldots, \alpha_n\}$ *and* $\bar{t}$ *is a sequence. In this case we can apply the induction hypothesis to each of the* $tr(\bar{t}, \alpha_i)$, *and* $type(\bar{t_i}') = \alpha_i$ *the final result is a new sequence where the old type of variables was replaced by a new ones generated by the union of all the types computed by the rewriting of the original sequence with each one of the members of* α, *thus, obtaining a sequence of type* α *and the result holds.*
- $tr(seq(t_1, \bar{t}), seq(s_1, \bar{s})) = seq(tr(t_1, s_1), tr(\bar{t}, \bar{s}))$, *in this case the result holds by applying the induction hypothesis to both arguments.* $\square$

The procedure for unification of sequences with type annotations is defined in figures 5.2 and 5.3. Every unification made during the procedure execution and every type reassignment made by function *tr* have an immediate impact in the terms involved in the related unification. When a new unification is started, $t_1 = * = t_2$, t_1 and t_2 are translated to sequence terms if necessary. Note that the rules are stored in a global variable $TRules$.

Success

(1)	t	$=*=$	s	$\Longrightarrow$	True, if $t == s$
(2)	$X :: \alpha$	$=*=$	t	$\Longrightarrow$	$X = tr(norm_*(t), \alpha_1)$ if X does not occur in t and seq$(\alpha, \varepsilon) \cap type(t) = \alpha_1$ and $empty(\alpha_1) = false$
(3)	t	$=*=$	$X :: \alpha$	$\Longrightarrow$	$X = tr(norm_*(t), \alpha_1)$ if X does not occur in t and $seq(\alpha, \varepsilon) \cap type(t) = \alpha_1$ and $empty(\alpha_1) = false$

Figure 5.2: Success rules

Example 5.6.8 *Given variables X, Y and types α, β with type rules $\alpha \rightarrow \{f(seq(a, \varepsilon))\}$ and $\beta \rightarrow \{seq(a, \varepsilon),\ seq(b, \varepsilon)\}$, let's calculate $X :: \alpha = * = f(Y :: \beta)$ which is the same than $seq(X :: \alpha, \varepsilon) = * = seq(f(seq(Y :: \beta, \varepsilon), \varepsilon))$ By rule number 6 this unification results in two cases:*

- *$seq(\alpha, \varepsilon) \cap type(seq(f(seq(Y :: \beta, \varepsilon), \varepsilon))) = seq(\alpha, \varepsilon) \cap seq(f(seq(\beta, \varepsilon), \varepsilon))$ which results in a new type symbol α_f where, $\alpha_f \rightarrow \{f(seq(\alpha'_f, \varepsilon))\}$ and $\alpha'_f \rightarrow \{seq(a, \varepsilon)\}$. Since $empty(\alpha_f) = false$ we have $X = tr(norm_*(seq(f(seq(Y :: \beta, \varepsilon), \varepsilon))), \alpha_f)$, where:*

$$\begin{array}{ll} tr(norm_*(seq(f(seq(Y :: \beta, \varepsilon), \varepsilon)), \alpha_f)) & = \\ tr(seq(f(seq(Y :: \beta, \varepsilon)), \varepsilon), \alpha_f) & = \\ seq(tr(f(seq(Y :: \beta, \varepsilon)), \alpha_f), \varepsilon) & = \\ seq(tr(f(seq(Y :: \beta, \varepsilon)), f(seq(\alpha'_f, \varepsilon))), \varepsilon) & = \\ seq(tr(f(seq(Y :: \beta, \varepsilon)), f(seq(\alpha'_f, \varepsilon))), \varepsilon) & = \\ seq(f(tr(seq(Y :: \beta, \varepsilon), seq(\alpha'_f, \varepsilon))), \varepsilon) & = \\ seq(f(seq(tr(Y :: \beta, \alpha'_f), tr(\varepsilon, \varepsilon))), \varepsilon) & = \\ seq(f(seq(Y :: \alpha'_f, \varepsilon)), \varepsilon) & \end{array}$$

  Now, $\varepsilon = * = \varepsilon$ is ε and the procedure succeeds with $X = seq(f(seq(Y :: \alpha'_f, \varepsilon)), \varepsilon)$

- *$seq(\alpha, \varepsilon) \cap type(seq(f(seq(Y :: \beta, \varepsilon)seq(X_1 :: \beta, \varepsilon))))$, where X_1 is a new variable and β_1 a new type symbol defined by $\beta_1 \rightarrow \{\mu\}$. This intersection is empty and the procedure fails.*

Example 5.6.9 *Given the variable X and the types α, with type rule $\alpha \rightarrow \{\varepsilon, seq(a, \alpha)\}$, let's calculate $seq(X :: \alpha, \varepsilon) = * = seq(a, seq(a, \varepsilon))$ By rule number 6 this unification results in two cases:*

Eliminate

(4) $f(\bar{t}) \quad =*= \quad f(\bar{s}) \quad \Longrightarrow \quad \bar{t} =*= \bar{s}$

(5) $seq(t_1,\bar{t}) \quad =*= \quad seq(s_1,\bar{s}) \quad \Longrightarrow \quad t_1 =*= s_1,$
$norm_*(\bar{t}) =*= norm_*(\bar{s})$

(6) $seq(X :: \alpha,\bar{t}) \quad =*= \quad seq(s_1,\bar{s}) \quad \Longrightarrow \quad X = tr(norm_*(s_1),\alpha_1),$
if X does not occur in s_1,
$seq(\alpha,\varepsilon) \cap type(seq(s_1,\varepsilon)) = \alpha_1$
and $empty(\alpha_1) = false$
$norm_*(\bar{t}) =*= norm_*(\bar{s})$.

$\Longrightarrow \quad X = tr(seq(s_1, seq(X_1 :: \beta,\varepsilon)),\alpha_1),$
where X_1 is a new variable, β
a new type symbol defined by $\beta \to \{\mu\}$
and β_1 the new type of X_1 in X,
$\alpha_1 = seq(\alpha,\varepsilon) \cap type(seq(s_1, seq(X_1 :: \beta,\varepsilon)))$
and $empty(\alpha_1) = false$,
$norm_*(seq(X_1 :: \beta_1,\bar{t})) =*= norm_*(\bar{s})$.

(7) $seq(t_1,\bar{t}) \quad =*= \quad seq(X :: \alpha,\bar{s}) \quad \Longrightarrow \quad X = tr(norm_*(t_1),\alpha_1),$
if X does not occur in t_1,
$seq(\alpha,\varepsilon) \cap type(seq(t_1,\varepsilon)) = \alpha_1$
and $empty(\alpha_1) = false$
$norm_*(\bar{t}) =*= norm_*(\bar{s})$.

$\Longrightarrow \quad X = tr(seq(t_1, seq(X_1 :: \beta,\varepsilon)),\alpha_1),$
where X_1 is a new variable, β
a new type symbol defined by $\beta \to \{\mu\}$
and β_1 the new type of X_1 in X,
$\alpha_1 = seq(\alpha,\varepsilon) \cap type(seq(t_1, seq(X_1 :: \beta,\varepsilon)))$
and $empty(\alpha_1) = false$,
$norm_*(\bar{t}) =*= norm_*(seq(X_1 :: \beta_1,\bar{s}))$,

(8) $seq(X :: \alpha,\bar{t}) \quad =*= \quad seq(Y :: \beta,\bar{s}) \quad \Longrightarrow \quad X :: \gamma = Y :: \gamma$, where
$\gamma = seq(\alpha,\varepsilon) \cap seq(\beta,\varepsilon), empty(\gamma) = false,$
$\bar{t} =*= \bar{s}$.

$\Longrightarrow \quad X = tr(seq(Y :: \beta, seq(X_1 :: \beta_1,\varepsilon)),\gamma)$, where
X_1 is a new variable and β_1 is a new type
symbol defined by $\beta_1 \to \{\mu\}$ and
$\gamma = seq(\alpha,\varepsilon) \cap type(seq(Y :: \beta, seq(X_1 :: \beta_1,\varepsilon)),$
$empty(\gamma) = false$ and β_2 is the new type
of X_1 in X
$norm_*(seq(X_1 :: \beta_2,\bar{t})) =*= norm_*(\bar{s}),$
X, Y and X_1 are distinct variables.

$\Longrightarrow \quad Y = tr(seq(X :: \alpha, seq(Y_1 :: \alpha_1,\varepsilon)),\gamma)$, where
Y_1 is a new variable and α_1 is a new type
symbol defined by $\alpha_1 \to \{\mu\}$ and
$\gamma = seq(\beta,\varepsilon) \cap type(seq(X :: \alpha, seq(Y_1 :: \alpha_1,\varepsilon))),$
$empty(\gamma) = false$ and α_2 is the new type
of Y_1 in Y,
$norm_*(\bar{t}) =*= norm_*(seq(Y_1 :: \alpha_2,\bar{s})),$
X, Y and Y_1 are distinct variables.

Split

(9) $seq(t_1,\bar{t}) \quad =*= \quad seq(s_1,\bar{s}) \quad \Longrightarrow \quad$ if $t_1 =*= s_1 \Longrightarrow r_1 =*= q_1$ then
$norm_*(seq(r_1,\bar{t})) =*= norm_*(seq(q_1,\bar{s}))$

$\vdots$

$\Longrightarrow \quad$ if $t_1 =*= s_1 \Longrightarrow r_w =*= q_w,$
$norm_*(seq(r_w,\bar{t}_n)) =*= norm_*(seq(q_w,\bar{s})),$
where t_1 and s_1 are compound terms.

Figure 5.3: Eliminate and split rules

- *$seq(\alpha, \varepsilon) \cap seq(a, \varepsilon)$ returns a new type α_{f1} where $\alpha_{f1} \rightarrow \{seq(a, \varepsilon)\}$. α_f is not empty, Thus the procedure proceeds evaluating $\varepsilon = * = seq(a, \varepsilon)$ which will fail.*
- *In the second case, a new variable, X_1 is created and a new type symbol β, is associated to it such that $\beta \rightarrow \{\mu\}$. By the intersection, a new type symbol α_{f2} is created with type rule $\alpha_{f2} \rightarrow \{seq(a, \alpha_{f3})\}$ and $\alpha_{f3} \rightarrow \{\varepsilon, seq(a, \alpha_{f3})\}$. Now $X = tr(seq(a, seq(X_1 :: \beta, \varepsilon)), \alpha_{f2})$, where:*

$$\begin{array}{ll} tr(seq(a, seq(X_1 :: \beta, \varepsilon)), \alpha_{f2}) & = \\ tr(seq(a, seq(X_1 :: \beta, \varepsilon)), seq(a, \alpha_{f3})) & = \\ seq(tr(a, a), tr(seq(X_1 :: \beta, \varepsilon), \alpha_{f3})) & = \\ seq(a, seq(tr(X_1 :: \beta, \alpha_{f3}), \varepsilon)) & = \\ seq(a, seq(X_1 :: \alpha_{f3}, \varepsilon)) & \end{array}$$

Now, the procedure proceeds with the unification $seq(X_1 :: \alpha_{f3}, \varepsilon) = * = seq(a, \varepsilon)$.Thus it intersects $seq(\alpha_{f3}, \varepsilon)$ with $seq(a, \varepsilon)$ resulting in $\alpha_{f4} \rightarrow \{seq(a, \varepsilon)\}$ and X_1 is instantiated with $seq(a, \varepsilon)$. The procedure succeeds with $X = seq(a, seq(a, \varepsilon)$.

Example 5.6.10 *Given the variables X, Y and the types α, β with type rules $\alpha \rightarrow \{seq(a, \varepsilon), seq(b, \varepsilon)\}$ and $\beta \rightarrow \{seq(a, \varepsilon)\}$, let's calculate $seq(X :: \alpha, seq(X :: \alpha, \varepsilon)) = * = seq(Y :: \beta, seq(a, \varepsilon))$. By rule number 8 this unification results in three cases. In the first one a new type symbol α_{f1} is generated from the intersection of $seq(\alpha, \varepsilon)$ and $seq(\beta, \varepsilon)$ where α_{f1} is defined by the type rule, $\alpha_{f1} \rightarrow \{seq(a, \varepsilon)\}$. Since α_{f1} is not empty the procedure proceeds instantiating $X :: \alpha_{f1}$ with $Y :: \alpha_{f1}$ and unifying the remaining sequence, $seq(X :: \alpha_{f1}, \varepsilon) = * = seq(a, \varepsilon)$. This last unification results in a new type symbol, α_{f2} such that $\alpha_{f2} \rightarrow \{seq(a, \varepsilon)\}$ and the procedure succeeds with $X = Y = seq(a, \varepsilon)$. The remaining two cases fail.*

Definition 5.6.9 *Let $\bar{t}$ and $\bar{s}$ be sequences, X a variable and α a type symbol. Then, $t_1\{X = t_2\}$ is the sequence resulting from replacing all occurrences of $X :: \alpha$ in t_1 by t_2.*

As seen in chapter 3, each unification problem generates a tree. Each branch of the tree corresponds to a finite computation leading to one of two possible results, success if the unification procedure ends returning a substitution, or failure if the unification fails. These trees may have an infinite number of branches. Each successful branch yields some answer composed by set $\Theta = \{ \theta_1, \ldots, \theta_n\}$ where each θ_i is a substitution associated with step i of the computation described by the branch.

An unification problem succeeds if at least one branch of the associated tree is successful.

In the following theorem we will use the notion of substitution presented in [Kut02c].

Theorem 5.6.4 (Soundness) *Let Θ be the answer substitution of the equation $t_1 = * = t_2$. Then:*

1. $\Theta t_1 = \Theta t_2$

2. $type(t_1) \cap type(t_2) = type(t)$, *where* $t = \Theta t_1 = \Theta t_2$

Proof 5.6.7 *Let's prove the theorem above by induction on the length of* Θ*. The base cases correspond to rules 1 to 3. By theorems 5.6.3 and 5.6.1 the result follows. Let's prove the result for the remaining cases:*

Rule 4 *Follows directly by the induction hypothesis.*

Rule 5 *We can apply the induction hypothesis to* $t_1 = * = s_1$ *and to* $norm_*(\bar{t}) = * = norm_*(\bar{s})$ *and then the result holds for* $seq(t_1, \bar{t}) = seq(s_1, \bar{s})$*.*

Rule 6 *This rule has two possible cases:*

- *In this case a new substitution* θ_1 *is produced. This substitution is a consequence of the unification of* $X :: \alpha$ *with* $tr(norm_*(s_1), \alpha_1)$*. Now, by theorems 5.6.3 and 5.6.1 and the intersection procedure,* $type(X) = \alpha_1$ *which is the type of the intersection of* $seq(\alpha, \varepsilon)$ *and* $type(seq(s_1, \varepsilon))$*, thus the result holds for* θ_1*.* $norm_*(\bar{t}) = * = norm_*(\bar{s})$ *generates an answer with the set of substitutions* $\{\theta_2, \ldots, \theta_n\}$ *and by the induction hypothesis it also holds for* $\{\theta_1, \theta_2, \ldots, \theta_n\}$*.*
- *In the second case a new substitution is generated. This substitution,* $X = tr(seq(s_1,\ seq(X_1 :: \beta,\ \varepsilon)), \alpha_1)$*, implies the successful intersection,* $seq(\alpha, \varepsilon) \cap seq(s_1, seq(\beta, \varepsilon)) = \alpha_1$*, thus by theorem 5.6.3, the type of the resulting term is* α_1 *and the result holds.* $norm_*(seq(X_1 :: \beta_1, \bar{t})) = * = norm_*(\bar{s})$ *generates an answer with a set of substitutions* $\{\theta_2,\ \ldots,\ \theta_n\}$ *and the result holds for* $\{\theta_1, \theta_2, \ldots, \theta_n\}$ *by the induction hypothesis.*

Rule 7 *This case is analogous to the previous one.*

Rule 8 *This rule has three possible cases:*

- *Let* θ_1 *be the substitution* $X :: \gamma = Y :: \gamma$*. Then* X *and* Y *are unified with a new type* γ *associated to them. This new type is the result of the intersection of the types of* X *and* Y*, thus the result holds.* $\bar{t} = * = \bar{s}$ *generate an answer with the set of substitutions* $\{\theta_2, \ldots, \theta_n\}$ *and the result holds for* $\{\theta_1, \theta_2, \ldots, \theta_n\}$ *by the induction hypothesis.*
- *This case is similar to the second case of rule 6.*
- *This case is analogous to the previous one.*

Rule 9 *The proof of this rule follows directly by the induction hypothesis.*

(1)	t	$=\tilde{}=$	s	$\Longrightarrow$	True, if $t == s$ [2]
(2)	X	$=\tilde{}=$	t	$\Longrightarrow$	$X = t$.
(2)	$f(\bar{t})$	$=\tilde{}=$	$f(\bar{s})$	$\Longrightarrow$	$\bar{t} =\tilde{}= \bar{s}$
(3)	$seq(t_1, \bar{t})$	$=\tilde{}=$	$seq(s_1, \bar{s})$	$\Longrightarrow$	$t_1 =\tilde{}= s_1$, $\bar{t} =\tilde{}= \bar{s}$
(4)	$seq(X, \bar{t})$	$=\tilde{}=$	$seq(s_1, \bar{s})$	$\Longrightarrow$	$X = \varepsilon$, $\bar{t} =\tilde{}= \bar{s}$.
				$\Longrightarrow$	$X = seq(s_1, seq(X_1, \varepsilon))$, $norm(seq(X_1, \bar{t})) =\tilde{}= norm(\bar{s})$, where X_1 is a new variable.

Figure 5.4: Algorithm for sequence matching

5.7 Pattern Matching

In most cases, XML processing programs have equations where one of the sides is ground (the XML file). Thus, pattern matching is a useful mechanism for XML processing also in logic programming which is an unification based paradigm. In addition to unification, XCentric also provides an operator for pattern matching here denoted by $=\tilde{}=$. The pattern matching algorithm is defined in figure 5.4. This algorithm is simply the unification algorithm presented in figure 4.1 considering only the cases where only one of the sides of the unification has variables. Thus, its correctness and termination are straightforward from the observation of the proofs for unification presented before.

The main reason for the implementation of this pattern matching algorithm is to improve efficiency. With variables restricted to one of the sides we can include projection in the transformation rules in a straightforward manner, thus avoiding a very inefficient step. On the other hand the user cannot use this to build an incomplete XML document like the one presented in example 4.5.9.

5.7.1 Incomplete terms in depth

We provide builtins that allow the programmer to find a sequence of elements at arbitrary depth, to search for the *nth* occurrence of a sequence of elements and to count the number of occurrences of a sequence. The predicates are *deep/2*, *deepp/3* and *deepc/3*, respectively. These predicates are implemented using pattern matching.

Definition 5.7.1 *Given the sequences s_1 and $\bar{s}$, where $\bar{s}$ has no variables, deep is defined as follows:*

$$deep(S_1, \bar{s}) \quad :- \quad deep_1(< S_1, X >, \bar{s}).$$

$$\begin{array}{lll} deep_1(S_1, \bar{s}) & :- & S_1 =\tilde{}= \bar{s}. \\ deep_1(S_1, < f(\bar{S_2}), \bar{s} >) & :- & deep_1(S_1, \bar{S_2}) \vee\ deep_1(S_1, \bar{s}). \\ deep_1(S_1, < S_2, \bar{s} >) & :- & deep_1(S_1, \bar{s}). \end{array}$$

The other built-ins that search a given occurrence of a sequence of terms(*deepc*) or count the number of occurrences of a sequence of terms (*deepp*) are simply a generalizations of the previous definition using counters.

For example, consider we have an XML document bound to the variable *XML*. Suppose we want to find a sequence of elements between two elements named *incision* and store it in variable *Critical*. We can do the query:

? - deep(<incision(_),Critical,incision(_)>,XML).

If we want to find the text of the third occurrence of element author in document Bib we can simply write:

? - deepp(author(T),Bib,3).

Here, variable T will be instantiated with the name of the author. If we want to count the number of occurrences of *author* elements in document *Book* we can simply do:

? - deepc(author(_),Book,C).

Note that any of these predicates use sequences of elements. We now present some examples that use the *deep* family of built-ins.

Example 5.7.1 *This example is based on a medical report using the HL7 Patient Record Architecture and inspired by the XQuery use cases available at [W3C05b]. Given* report1.xml*:*

```
<report>
  <section>
    <section_title>Procedure</section_title>
     <section_content>
      The patient was taken to the operating room where she was placed ...
      <anesthesia>induced under general anesthesia.</anesthesia>
      <prep>
        <action>A Foley catheter was placed to decompress the bladder</action>
        and the abdomen was then prepped and draped in sterile fashion.
      </prep>
      <incision>
        A curvilinear incision was made
        <geography>in the midline immediately infraumbilical</geography>
        and the subcutaneous tissue was divided
        <instrument>using electrocautery.</instrument>
      </incision>
      The fascia was identified and
      <action>#2 0 Maxon stay sutures were placed on each side of the ...
      </action>
      <incision>
        The fascia was divided using
        <instrument>electrocautery</instrument>
```

```
      and the peritoneum was entered.
    </incision>
    <observation>The small bowel was identified.</observation> and
    <action> the <instrument>Hasson trocar</instrument>
     was placed under direct visualization. </action>
    <action>
      The <instrument>trocar</instrument>
      was secured to the fascia using the stay sutures. </action>
   </section_content>
  </section>
</report>
```

1. *Find what happened between the first* incision *and the second* incision *and write the result in a file named critical.xml:*

```
translate:-
     xml2pro('report1.xml',Rep),
     deep(<incision(_),Critical,incision(_)>,Rep),
     newdoc(critical_sequence,[],Critical,FL),
     pro2xml(FL,'critical.xml').
```

   The result is:

```
<critical_sequence>
        The fascia was identified and<action> #2 0 Maxon stay sutures
        were placed on each side of the midline.</action>
</critical_sequence>
```

2. *In the Procedure section what Instruments were used in the second Incision?*

```
instruments:-
        xml2pro('report1.xml',Rep),
        deep(section(section_title('Procedure'),S),Rep),
        deepp(incision(_,instrument(I),_),S,2),
        write('<instrument>'),write(I),write('</instrument>').
```

   Result:

```
<instrument>electrocautery</instrument>
```

The simplicity and declarativeness of our approach is stressed by comparing this solution to the one provided by W3C for XQuery use cases available at [W3C05b].

Example 5.7.2 *Given an XML document for a book validated by the following DTD (taken from the XQuery use cases available at [W3C05b]):*

```
<!ELEMENT book (title, author+, section+)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
```

```
<!ELEMENT section (title, (p | figure | section)* )>
<!ATTLIST section id          ID    #IMPLIED
                  difficulty CDATA #IMPLIED>
<!ELEMENT p (#PCDATA)>
<!ELEMENT figure (title, image)>
<!ATTLIST figure  width  CDATA  #REQUIRED
                  height CDATA  #REQUIRED >
<!ELEMENT image EMPTY>
<!ATTLIST image   source CDATA  #REQUIRED >
```

The following program counts how many sections and figures are in the book:

```
count:-
    xml2pro('book.xml',Book),
    deepc(section(_),Book,SC),
    deepc(figure(_),Book,FC),
    write('<section_count>'),write(SC),write('</section_count>'),nl,
    write('<figure_count>'),write(FC),write('</figure_count>').
```

5.8 Unification vs. Pattern Matching

In this section we present some benchmark results comparing execution of the same program with pattern matching and unification. The computer were the tests were carried on is a Pentium 4 Mobile @ 1.6 Ghz with 512MB of RAM running Debian Linux. These tests are similar to the ones performed for the CDuce language [BCF03].

For the first test, the file used is a simple addressbook like the following one:

```
<addrbook>
   <person>
        <name>Jorge Coelho</name>
        <tel>123456789</tel>
        <email>jcoelho@ncc.up.pt</email>
        <email>jcoelho@dcc.fc.up.pt</email>
   </person>
   <person>
        <name>M\'ario Florido</name>
        <tel>987654321</tel>
        <email>amf@ncc.up.pt</email>
   </person>
   ...
</addrbook>
```

We test a simple operation, translating an XML document for an addressbook which has *persons* with *name*, *tel* and one or more *email* in a new document were the emails are ignored. The code for the unification is the following one:

```
translate:-
        xml2pro('addrbook.xml',Addr),
```

Runs	1	2	3	4	5	6	7	8	9	10	**Result**
Pat	0,044	0,036	0,040	0,043	0,044	0,040	0,043	0,043	0,048	0,044	**0,043**
Unif	0,171	0,173	0,171	0,174	0,173	0,177	0,173	0,170	0,171	0,173	**0,173**

Table 5.1: Result of tests with 0,5 KB file for addressbook transformation

```
          addrbook([],PS) =*= Addr,
          mktelbook(PS,PS2),!,
          newdoc(addrbook,[],PS2,NewA),
          pro2xml(NewA,'addrbook2.xml').

mktelbook(<>,<>):-!.

mktelbook(<person([],name([],N),tel([],T),_),PS2>,
            <person([],name([],N),tel([],T),_),F2>):-
          mktelbook(PS2,F2),!.

mktelbook(<person(_),P2>,Final):-
          mktelbook(P2,Final).
```

the code for the pattern matching is the following one:

```
translate:-
          xml2pro('addrbook.xml',Addr),
          addrbook([],PS) =~= Addr,
          mktelbook(PS,PS2),!,
          newdoc(addrbook,[],PS2,NewA),
          pro2xml(NewA,'addrbook2.xml').

mktelbook(A,B):-
          A  =~= <>,
          B  =~= <>,!.

mktelbook(PS,Final):-
          <person([],name([],N),tel([],T),R),PS2> =~= PS,
          mktelbook(PS2,F2),!,
          Final =~= <person([],name([],N),tel([],T),R),F2>.

mktelbook(PS,Final):-
          <person(_),P2> =~= PS,
          mktelbook(P2,Final).
```

The file *addrbook.xml* is successively replaced by a file with 0,5 KB, 5 KB and 15 KB of randomly generated XML for an addressbook. Time was measured with the UNIX utility *time* and results are presented in tables 5.1, 5.8 and 5.8. The column **Result** contains the average results for the ten executions of the program.

The second case is more complex. Given an XML file describing a list of persons and their

Runs	1	2	3	4	5	6	7	8	9	10	**Result**
Pat	0,852	0,839	0,845	0,854	0,842	0,836	0,844	0,842	0,850	0,849	**0,845**
Unif	14,230	14,208	14,209	14,197	14,221	14,202	14,201	14,213	14,213	14,204	**14,2098**

Table 5.2: Result of tests with 5 KB file for addressbook transformation

Runs	1	2	3	4	5	6	7	8	9	10	**Result**
Pat	5,112	5,118	5,131	5,084	5,102	5,097	5,116	5,101	5,092	5,097	**5,105**
Unif	2,29.9	2,29.5	2,29.9	2,29.8	2,29.9	2,29.5	2,29.4	2,29.9	2,29.5	2,29.9	**2m29,72**

Table 5.3: Result of tests with 15 KB file for addressbook transformation

children, as the one presented next:

```
<doc>
  <person gender="m" >
   <name>John</name>
   <children>
     <person gender="m" >
      <name>Robert</name>
      <children>
       <person gender="f" >
        <name>Mary</name>
        <children/>
       </person>

       <person gender="m" >
        <name>Jack</name>
        <children>
         <person gender="f" >
          <name>Jill</name>
          <children>
           <person gender="f" >
            <name>Gloria</name>
            <children/>
           </person>
          </children>
         </person>
        </children>
       </person>
      </children>
    </person>
   </children>
 </person>
</doc>
```

The goal is to transform each of the person's record into *man* or *woman*. The attribute *gender* disappears and a new attribute with the name is introduced. The children are divided in a list

of men and a list of women. For the XML presented before the result of the transformation is:

```
<doc>
  <man name="John">
   <man name="Robert">
     <man name="Jack">
       <woman name="Jill">
         <woman name="Gloria"/>
       </woman>
     </man>
     <woman name="Mary"/>
   </man>
  </man>
</doc>
```

We tested these transformations using an unification approach and a pattern matching approach. In both of them we enabled and disabled type checking in order to evaluate the impact of this step. We used the following declared types:

```
:-type pm ---> person([attribute(gender,m)],name([],string),children).
:-type pf ---> person([attribute(gender,f)],name([],string),children).
:-type person ---> pm ; pf.
:-type children ---> children([],persons).
:-type persons ---> person*.

:-type male --->man([attribute(name,string)],sd).
:-type female --->woman([attribute(name,string)],sd).
:-type males --->male*.
:-type females --->female*.
:-type sd ---> (males,females).
```

The code for the unification approach follows:

```
translate:-
        xml2pro('split.xml',Person),
        doc([],PS) =~= Person,
        sperson(PS,PS2),
        newdoc(doc,[],PS2,NewA),
        pro2xml(NewA,'split2.xml').



sperson(PersonSeq,SD):-
        split_person(PersonSeq,Sons,Daug),
        SD =*= <Sons,Daug>.

split_person(<>,<>,<>):-!.

split_person(P1,P2,Women):-
    P1::persons =*= <person([attribute(gender,m)],name([],N),
                                        children([],Children)),PS>,!,
```

```
    split_person(PS,Men,Women),
    split_person(Children,S,D),
    P2::males =*= <man([attribute(name,N)],S,D),Men>.


split_person(P1,Men,P2):-
   P1::persons =*= <person([attribute(gender,f)],name([],N),
                                        children([],Children)),PS>,!,
   split_person(PS,Men,Women),
   split_person(Children,S,D),
   P2::females =*= <woman([attribute(name,N)],S,D),Women>.
```

The code for the pattern matching approach follows:

```
translate:-
        xml2pro('split.xml',Person),
        doc([],PS) =~= Person,
        sperson(PS,PS2),
        newdoc(doc,[],PS2,NewA),
        pro2xml(NewA,'split2.xml').


sperson(PersonSeq,SD):-
        split_person(PersonSeq,Sons,Daug),
        SD =~=<Sons,Daug>.

split_person(P1,P2,P3):-
        P1 =~= <>,
        P2 =~= <>,
        P3 =~= <>,!.

split_person(P1,P2,Women):-
  <person([attribute(gender,m)],name([],N),children([],Children)),PS>
                                                    =~=P1::persons,!,
  split_person(PS,Men,Women),
  split_person(Children,S,D),
  P2::males =~= <man([attribute(name,N)],S,D),Men>.


split_person(P1,Men,P2):-
 <person([attribute(gender,f)],name([],N),children([],Children)),PS>
                                                    =~= P1::persons,!,
   split_person(PS,Men,Women),
   split_person(Children,S,D),
   P2::females =~= <woman([attribute(name,N)],S,D),Women>.
```

The file *split.xml* is successively replaced by a file with 0,5 KB, 5 KB and 15 KB of randomly generated XML for a list of persons. Time was measured with the UNIX utility *time* and results are presented in tables 5.4, 5.8 and 5.8. The column **Result** contains the average

Runs	1	2	3	4	5	6	7	8	9	10	**Result**
Pat	0.070	0.066	0.066	0.078	0.072	0.071	0.067	0.071	0.070	0.071	**0.070**
Pat + Types	0.218	0.214	0.217	0.221	0.216	0.210	0.215	0.217	0.218	0.220	**0.216**
Unif	0.297	0.300	0.302	0.299	0.295	0.298	0.303	0.302	0.293	0.303	**0.299**
Unif + Types	0.436	0.444	0.445	0.438	0.439	0.441	0.439	0.438	0.438	0.444	**0.440**

Table 5.4: Result of tests with 0,5 KB file for person transformation

Runs	1	2	3	4	5	6	7	8	9	10	**Result**
Pat	0.883	0.883	0.876	0.870	0.874	0.891	0.905	0.884	0.875	0.884	**0.883**
Pat + Types	8.634	8.670	8.616	8.620	8.677	8.562	8.704	8.617	8.637	8.766	**8.650**
Unif	5.543	5.561	5.564	5.529	5.542	5.585	5.571	5.490	5.573	5.553	**5.550**
Unif + Types	13.26	13.32	13.31	13.33	13.35	13.32	13.25	13.32	13.33	13.31	**13.340**

Table 5.5: Result of tests with 5 KB file for person transformation

results for the ten executions of the program.

As expected pattern matching is more efficient than unification. The main reason for this results is the absence of projection in the pattern matching algorithm. Eliminating variables can be easily done in the transformation rules, thus avoiding that expensive step. The impact of type checking is also considerable. It can be explained by the implementation of types which relies on flexible arity unification. It is important to mention that we strongly rely in SWI-Prolog and thus, in one hand we inherit a sound API and, in the other hand, poor performance of SWI-Prolog itself. We strongly believe that performance can be much improved by a lower level implementation of unification and pattern matching.

5.9 Discussion

Here we described an extension to unification with regular expression types and its application to XML processing where types, besides they usual use as the guarantee of correctness accordingly to some pre-defined type information, give a quite compact specification of sequences of arguments with a similar structure. We presented two approaches, dynamic and static typing, describing how they are implemented in the XCentric language. We also present

Runs	1	2	3	4	5	6	7	8	9	10	**Result**
Pat	3.128	3.118	3.204	3.137	3.134	3.138	3.134	3.143	3.189	3.108	**3.158**
Pat + Types	31.98	31.98	31.92	32.12	32.00	32.06	32.12	31.96	31.95	31.96	**32.005**
Unif	20.70	20.70	20.70	20.73	21.39	20.63	20.73	20.99	20.50	20.80	**20.786**
Unif + Types	50.72	50.76	50.72	50.98	50.87	50.82	50.75	50.70	50.75	50.801	**50.785**

Table 5.6: Result of tests with 15 KB file for person transformation

benchmark results of Unification and Pattern Matching which naturally show the advantage of Pattern Matching. One possible way to improve the language is to automatically detect which cases don't need unification and apply pattern matching automatically. As a final note, we believe that the mixed environment of XCentric, which allows programming with and without types, increases flexibility in programming.

Chapter 6

Case-Study: Website Content Verification

6.1 Introduction

Managing the semantic content of a website is not easy and it is even harder when it is built by many different persons. Consider for example a website of a university. There, different persons such as teachers and administrative staff have access to some part of the website, for example their homepage. The website manager or the university administration may impose some constraints in the website construction, for example, every person must have his/her curriculum in his/her homepage or every teacher must have information about his/her teaching activities. Verifying these constraints may be a difficult task when we have a high number of pages to look up. Another question arises from usefulness of the information available in the website: can it be used to infer more information? For example, in our research laboratory, technical reports and other publications are added to a central web page. It is desirable to infer some statistical data about scientific production activities. In this chapter we present VeriFLog, an extension to XCentric for verification of content in websites. We present VeriFLog in two parts, first we show the adequacy of XCentric to accomplish the tasks here described and second we extend XCentric with the ability to automatically repair the webpages that don't obey to a given criteria along with implementation of compile-time and run-time specific techniques for rule consistency checking. This work was partially presented at [CF06b] and [CF07a]. The webpage of VeriFlog is: `http://www.ncc.up.pt/~jcoelho/veriflog/`.

6.2 XCentric for Website Content Verification

The main idea of this work is to provide an interface to semistructured data, syntactic validation and use XCentric as the rule language for semantic verification. By semantic

verification we mean verifying if the content of a website is correct with relation to a given criteria. For example, we have a web page with all the staff of the department grouped by category (Senior researcher, Phd researcher, Researcher and Assistant researcher). One verification we can do, is searching for people catalogued as one of the above mentioned categories but that don't belong to that category.

With our framework the programmer can also use the high declarative model of XCentric to infer new knowledge from web pages. For example, if every researcher of our department has its own publications in his/her homepage we can easily retrieve this data and get new statistical data from it, for example, how many publications in journals and international conferences by year.

As motivation, consider the following simple example: suppose we want to verify if the teachers of our university have in their XML-based home pages a correct email address. We have an XML file generated from our database with their data and can use the following code:

```
check(N,URL):-
    http2pro(URL,T),
    T =*= hpage(_,email(E),_),
    xml2pro('dbase.xml',DB),
    DB =*= db(_,record(name(N),_,email(E),_),_).
```

This program, written in XCentric, starts by using *http2pro* to retrieve from the web address in *URL*, the XML code, translating it to the internal notation and storing it in variable *T*. Then the non-standard unification operator =*= is used to find the teacher's email and store it in variable *E*. Using *xml2pro* the database file is translated to the internal representation and stored in variable *DB*. Then the record of the teacher named *N* is searched and the program succeeds only if both the email found in the web page and the one found in the database are the same. We can easily generalize the program to accept a list of teachers, verify the entire website and output error messages.

6.2.1 Examples

The next examples show the adequacy of XCentric to impose and verify constraints over websites. We encourage the reader to test the full versions of these examples and many others available at the VeriFLog site. There are four ways to retrieve HTML/XML data: directly from the web using *http2pro(URL,Term)*; directly from the web with validation using *http2pro(URL, DTDFile, Term)*, where the file in *URL* is validated with respect to the DTD given in *DTDFile*; from an xml file using *xml2pro(XMLFile,Term)* and from an xml file with validation using *xml2pro(XMLFile,DTDFile,Term)*.

Example 6.2.1 *In our department website we have an HTML page with a list of technical reports indexed by year. The html is presented next:*

```
<html>
 <head>
  <title>Technical Reports</title>
 </head>

<body>
  <h3>Technical Reports </h3>
  <h4>2005</h4>
  <ul>
   <li>Jorge Coelho and Mario Florido. Avoiding Infinite Loops in the Solving
    of Equations Involving Sequence Variables and Terms with Flexible Arity
    Function Symbols, Technical Report DCC-2005-01, DCC - FC &amp; LIACC,
    Universidade do Porto, March, 2005. </li>
   <li>Ana Paula Tomas. Emparelhamentos, Casamentos Estaveis e Algoritmos de
    Colocacao de Professores, Technical Report DCC-2005-02, DCC - FC &amp;
   LIACC, Universidade do Porto, March, 2005. (in Portuguese)</li>
  </ul>
   ...
  <h4>2004</h4>
  <ul>
   <li>Sabine Broda and Luis Damas. The Formula-Tree Proof Method, Technical
    Report DCC-2004-01, DCC - FC &amp; LIACC, Universidade do Porto, January,
    2004.
   </li>
    ...
   <li>Sandra Alves and Mario Florido. Type Inference for Programming
    Languages: A Constraint Logic Programming Approach, Technical Report DCC
   - FC &amp; LIACC, Universidade do Porto</li>
    ...
  </ul>

<h4>2003</h4>

 ...

 </body>
</html>
```

Suppose that we want to know if some publications are out of place with respect to the year of publication. We can simply do:

```
verify(URL):-
       http2pro(URL,TR),
       TR =*= html(_,body(_,h4(Y1),Seq,h4(Y2),_),_),
       atom_number(Y1,N1),atom_number(Y2,N2),
       N2 is N1 - 1,write('Year: '),write(Y1),nl,
       process(Seq,Y1).

process(Seq,Y):-
         Seq =*= ul(_,li(C),_),
```

```
    not(substring(Y,C)), write(C).
```

We start by using http2pro/2 to translate the web page given in variable URL into its internal representation, then we get the sequences of elements (variable Seq), between two adjacent years (variables Y1 and Y2).

```
TR =*= html(_,body(_,h4(Y1),Seq,h4(Y2),_),_),
```

We then check if some string (variable C) describing a technical report does not include the year that indexes that report:

```
Seq =*= ul(_,li(C),_),
not(substring(Y,C)),
```

Running this program over the previously mentioned web page will present the following output:

```
Year: 2005
Year: 2004
Sandra Alves and Mario Florido. Type Inference for Programming
Languages: A Constraint Logic Programming Approach, Technical
Report DCC - FC & LIACC, Universidade do Porto
```

Meaning that for the year 2004 it was found one publication that doesn't have a reference to the year it was published or the year is different from 2004.

Example 6.2.2 *Given the following XML file with a set of publications:*

```
<pubs>
...
  <pub>
    <author>Jorge Coelho</author>
    <author>Mrio Florido</author>
    <title>Type-based XML Processing in Logic Programming</title>
    <booktitle>Proceedings of the Fifth International Symposium on
              Practical Aspects of Declarative Languages (PADL'03)
    </booktitle>
    <pages>273-285</pages>
    <editor>V. Dahl, P. Wadler </editor>
    <volume>Lecture Notes in Computer Science 2562</volume>
    <address>New Orleans</address>
    <month>January</month>
    <year>2003</year>
    <publisher>Springer Verlag</publisher>
  </pub>

  <pub>
   <author>Mario Florido</author>
   <author>Luis Damas</author>
   <title>
```

```
      Linearization of the Lambda-Calculus and its Relation with
      Intersection Type Systems
    </title>
    <booktitle>Journal of Functional Programming</booktitle>
    <pages>519-546</pages>
    <volume>14</volume>
    <month>September</month>
    <year>2004</year>
   </pub>
...
</pubs>
```

The following code detects which ones have a publisher tag and which don't, and for those having a publisher tag, prints their content. Note that the way we use variables (unifying with zero or more elements) is not the standard way in Prolog but turns out to be very useful for XML queries:

```
verpub(URL):-
      http2pro(URL,Pubs),
      Pubs =*= pubs(_,pub(_,title(NP),Rem),_),
      write(NP),write(' '),decide(Rem),nl.

decide(<_,publisher(Pub),_>):-
      write('has publisher: '),write(Pub),!.

decide(_):-
     write('does not have publisher.').
```

We start by storing in variable Rem *the sequence of elements possible including the publisher tag. Then we use the special syntax for sequences:*

$$<_, \text{publisher(Pub)}, _>$$

in order to verify if the sequence includes the publisher tag. The result follows:

```
...
Type-based XML Processing in Logic Programming has publisher: Springer Verlag
Publication: Linearization of the Lambda-Calculus and its Relation with
             Intersection Type Systems does not have publisher.
...
```

Given a list of publishers we can easily check which publications we have that were published by these:

```
sci(URL,SCIPub):-
     http2pro(URL,Pubs),
     Pubs =*= pubs(_,pub(_,title(NP),_,publisher(P),_),_),
     member(P,SCIPub),
     write('Publication: '),write(NP),write(' by '),write(P),nl.
```

Here we are retrieving the title of publications to NP *and their publisher to* P *and then verifying if the publisher is one of the elements given in the list* SCIPub*. To the query:*

```
?- sci('http://www.ncc.up.pt/~jcoelho/veriflog/examples/pubs.xml',
       ['Springer Verlag', 'IEEE', 'ACM','Elsevier','Kluwer']).
```

the answer is:

```
...
Publication: Type-based XML Processing in Logic Programming by Springer Verlag
...
```

Example 6.2.3 *Given a teacher homepage, for example this one:*

```
<teacher>
   <name>Mario</name>
   <phone>+351 123456789</phone>
   <email>amf@ncc.up.pt</email>
   <courses>
     <name>Compilers</name>
   </courses>
</teacher>
```

We want to know if it includes information about the classes he teaches. We get that information from our database (in an XML file) and compare both:

```
DBase =*= staff(_,teacher(_,name(N),_,courses(_,class(C),_),_),_),
write('Database: '),write(C),
XML =*= teacher(_,name(N),_,courses(_,name(C),_),_),
write('>Found in Homepage<'),nl.
```

Here we query the database file stored in variable DBase *and for each teacher we store his name in variable* N *and the classes he teaches one by one in variable* C*. Then, we query the teacher homepage, stored in variable* XML *trying to find the same classes which were found in the database. The output follows:*

```
Database: Compilers >Found in Homepage<
Database: Logic
Database: Programming Languages
```

Meaning that, from the three courses found in the database only Compilers *was found in the teacher homepage. This can be easily generalized to a set of teachers (using http2pro/2 to retrieve each ones data).*

Example 6.2.4 *Given an XML file with all the staff of our department:*

```
<staff>
 ...
 <admin>
   <name>Susana Amorim</name>
 </admin>
```

```
...
 <researcher>
  <name>Mario Florido</name>
  <category>Senior</category>
 </researcher>
...
 <admin>
  <name>Amelia Rocha</name>
 </admin>
...
 <researcher>
  <name>Luis Antunes</name>
  <category>PhD</category>
 </researcher>
...
 <researcher>
  <name>Jorge Coelho</name>
 </researcher>
...
</staff>
```

and the publications file presented in example 6.2.2, we want to validate the staff file with respect to our DTD and query the files in order to know which people appears as co-authors of publications from our department and don't belong to our staff of researchers:

```
verify(PubsFile, StaffFile ,DTDStaff):-
        http2pro(PubsFile,ATerm),
        xml2pro(StaffFile ,DTDStaff, StaffTerm),
        pubaut(ATerm, StaffTerm).

pubaut(AT,ST):-
        AT =*= pubs(_,pub(_,author(A),_),_),
        nl, write('Searching: '),write(A),write(' '),
        ST =*= staff(_,researcher(name(A),_),_),
        write(' Found').
```

Here we query the publications file retrieved from the url given in PubsFile, *searching for authors of publications. Then we search our local XML file stored in* StaffFile *for the corresponding record. We output* Found *whenever an author of a publication appears in our staff list. Please note that inside the staff file, there is information about researchers and administrative people and our model allows the programmer to ignore the part of the file that is not of interest and focus on the problem. To the query:*

```
?- verify('http://www.ncc.up.pt/~jcoelho/veriflog/examples/pubs.xml',
          'researchers.xml','staff.dtd').
```

the result is:

```
...
Searching: Jorge Coelho            Found
```

```
Searching: Mario Florido            Found
Searching: Nuno Fonseca             Found
Searching: Herve Paulino
Searching: Luis Damas               Found
...
```

The previous examples illustrate the easy application of the XCentric language to the domain of website content verification. We now proceed with the extension of XCentric with specific features for website verification.

6.3 Extension with type and consistency checking

In this section we present an extension to the previous framework with the ability to automatically repair the web pages that don't obey to a given constraint that is being checked and to verify if the changes introduced do not violate any of the constraints provided for the website. We accomplish these tasks by defining *Sets of Web Constraints*, introducing type declarations and compile time type checking along with consistency verification and type-checking during run-time.

Let's now present a simple motivating example of use of the extended framework. Suppose we have a XML file describing a list of publications which is valid with respect to the following DTD:

```
<!ELEMENT bib(pub*)>
<!ELEMENT pub (author+, title, booktitle, volume?,year?, publisher?)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT booktitle (#PCDATA)>
<!ELEMENT volume (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
```

and we want to impose one simple constraint:

"If the year of publication is greater than 2006, the string "to appear" must occur in the "booktitle" content. If it doesn't then simply add it."

This can be done by the following program:

```
swc(URLPub,NewPub):-
    xml2pro(URLPub,XML),
    bind_type(XML,type_pub),
    r1(XML1,XML2),
    pro2xml(XML2,NewPub).

r1(X1,X2):-
  replace(pub(X, booktitle(BT1),Y),
```

```
 pub(X, booktitle(BT2),Y),X1,X2,
[Y =*= <_, year(Y1), _>,
 atom2number(Y1,Yn),
 Yn > 2006, not(substring("to appear",BT1)),
 concat(BT1,"(to appear)",BT2]).
```

In this program we retrieve an XML document from a url given in *URLPub*, then it is converted to the internal representation by builtin *xml2pro* and binded to type *type_pub* which represents the DTD presented before. The rule is described in predicate *r1*, where *pub* is used as a flexible arity functor (thus X and Y may have zero or more elements) and for all publications where the year is greater than 2006 and the string "to appear" does not occur in the booktitle, the string "to appear" is automatically added to the content of booktitle. This is done for all publications on $X1$ which violate the set of constraints resulting in a new document $X2$. During compile time a static analysis is carried on in order to see if the rule violates the declared type. During run-time the types are again used in order to verify that the values introduced don't change the document in wrong way. In case there is more than one rule, a consistency check is also made in order to insure that actions made by one rule does not violate other rule. We now describe the details of this extension.

6.3.1 Verification Framework

We start by defining a set of rules as constraints for a given website. We introduce special builtins for dealing with errors and finally show how the static-time verification and the run-time consistency checking work. In this section we focus on *syntactic validation*, *semantic verification* and *error correction*. The framework keeps the same tools for querying and inferring data as presented before. The framework can be used as a tool for the webmaster which feeds it with a set of pages to verify and correct them. Another approach can be the use of the framework as an auditing tool for everyone which publishes in the website. The webmaster describes the rules and the system is used to verify the content of any page prior to its publication online. The page can only be published if the system confirms it is correct accordingly to the imposed constraints.

6.3.1.1 Website Constraints

Definition 6.3.1 *We define a set of website constraints (SWC) as follows:*

- *A main rule whose input is a web page (WpageI) and output is a new web page (WpageO) resulting from the input page with the necessary changes in order to obey to the set of constraints imposed:*

$$
\begin{aligned}
swc(WpageI, WpageO) \quad :- \quad & \\
& r_1(WpageI, Wpage1), \\
& \vdots, \\
& r_n(Wpage_{n-1}, WpageO).
\end{aligned}
$$

- *Each rule r_i imposes some action to be taken in case some set of constraints is violated. We call these kind of rule an* action rule. *The rule may change the original document in order to make it obey to the constraint set.*

Note that $WpageI$ and $WpageO$ is our internal representation for XML data and thus $WpageI$ resulted from translating a web page from a URL or file by using one of the internal builtins available in the framework.

We now define three new builtins associated with action rules:

- $delete(S, WpageI, WpageO, L)$ - deletes sequence S which respect the constraints in L from $WpageI$ resulting in $WpageO$.
- $replace(S_1, S_2, WpageI, WpageO, L)$ - replace sequence S_1 which respect the constraints in L by sequence S_2 in $WpageI$ by $WpageO$.
- $failure(WebpageI, L, Mesg)$ - used when the error is to serious to be automatically solved. Message $Mesg$ is shown when the constraints in L are violated.

Elements in L are, for example, tests in the values in the sequences in order to verify they follow a certain criteria.

Example 6.3.1 *Given the following web page:*

```
<teacher>
  <name>Mario</name>
  <phone>+351 123456789</phone>
  <email>amf@ncc.up.pt</email>
  <teaching>
    <course>Compilers</course>
    <course>Theory of Computation</course>
  </teaching>
</teacher>
```

which is translated to the internal representation (stored in variable W_1):

```
teacher(
        name(``Mario''),
        phone(``+351 123456789''),
        email(``amf@ncc.up.pt''),
        teaching(course(``Compilers''))
                 course(``Theory of Computation''))).
```

If we apply:

$$delete(< phone(_), email(_) >, W_1, W_2, []).$$

W_1 will be translated to:

```
teacher(
        name(``Mario''),
        teaching(course(``Compilers''),
                 course(``Theory of Computation''))).
```

where the phone *and* email *tags have been deleted. If for example we want to delete all the names of courses which do not occur in our database (in variable DB), one can do:*

$$delete(course(N), W_1, W_2, [not(deep(course(N), DB))]).$$

If for example "Theory of Computation" does not occur in DB the result is:

```
teacher(
        name(``Mario''),
        teaching(course(``Compilers''))).
```

Example 6.3.2 *Given the following XML file for a catalog of books translated to a term in W_1:*

```
<catalog>
  ...
  <book number="500">
    <name>
     Haskell: The Craft of Functional Programming
     (2nd Edition)
    </name>
    <author>Simon Thompson</author>
    <price>41</price>
    <year>1999</year>
  </book>
  <book number="501">
    <name>
      Data on the Web
    </name>
     <author>Serge Abiteboul</author>
     <author>Peter Buneman</author>
     <author>Dan Suciu</author>
    <price>null</price>
    <year>2000</year>
  </book>
  ...
</catalog>
```

to replace all the prices with non-numeric values by 0 one can do:

$$replace(price(X), price(0), W_1, W_2, [not(number(X))]).$$

To reduce 10% to all the prices higher than 45 one can do:

$$replace(price(X), price(Y), W_1, W_2, [X > 45, Y = X - (X * 0.1)]).$$

Example 6.3.3 *Consider the XML file of example 6.3.1 in variable W, the following rule:*

$$failure(W, [not(deep(curriculum(_), W))], \text{"Error: Curriculum not found!"}).$$

results in stopping the verification and output of message "Error: Curriculum not found!" in case the curriculum *tag is not found.*

We now proceed with an example of running sets of website constraints.

Example 6.3.4 *Let's use again a teacher's webpage which is valid with respect to the following DTD:*

```
<!ELEMENT teacher (name, phone, email*, research, teaching, curriculum)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT research (pub*)>
<!ELEMENT pub (author+, title, booktitle, volume?, year?, publisher?)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT booktitle (#PCDATA)>
<!ELEMENT volume (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
<!ELEMENT teaching (course+)>
<!ELEMENT course (#PCDATA)>
<!ELEMENT curriculum (#PCDATA)>
```

This DTD can be trivially translated to the following regular expression type (and included in the program file):

```
:-type teacher ---> teacher(name(string), phone(string), email(string)*, research,
                                                         teaching, curriculum).
:-type name ---> name(string).
:-type phone ---> phone(string).
:-type email ---> email(string).
:-type research ---> research(pub*).
:-type pub ---> pub (author+, title, booktitle, volume?, year?, publisher?).
:-type author ---> author(string).
:-type title ---> title(string).
:-type booktitle ---> booktitle(string).
:-type volume ---> volume(string).
:-type year ---> year(string).
:-type publisher ---> publisher(string).
:-type teaching ---> teaching (course+).
```

```
:-type course ---> course(string).
:-type curriculum ---> curriculum(string).
```

We want to verify the following constraints:

- *The teacher must have a non empty curriculum tag in his homepage and this tag should include a small text were the teacher resumes his curriculum: he should include a reference to the degree and the year he obtained it. In case this rule is not obeyed, a small warning text should appear in the curriculum tag of the newly generated document.*
- *A well-formed email address should be available, otherwise the verification should be canceled and the error reported.*
- *All the publications the teacher has that don't appear in the central repository should be deleted.*

We can build the program presented in Fig. 6.1 to verify the above constraints over the teacher's webpage. It works as follows: the url is retrieved and stored in variable Tea using builtin xml2pro. *Then,* bind_type(Page,Type) *binds the webpage* Page *to the previous declared type* Type. *Without* bind_type *the compiler ignores the type information and thus would not be possible to find errors at compile time. The rules are described in the remaining lines of the program. Rule $r1$ verifies if the words "degree" and "year" occur inside the* curriculum *tag, in case they don't, the content of that tag is replaced with the message "Degree and year missing!". Rule $r2$ verifies if the email address is defined properly by checking if it includes "@ncc.up.pt", in case it doesn't the verification is cancelled and the error message "Valid email not found!" is shown. Rule $r3$ searches publication entries in the teacher webpage which are not present in the file "pubs.xml" and deletes them.*

6.3.1.2 Static verification of action rules

Static verification is made by static analysis of action rules. These rules possibly change the document by means of a delete or replace, thus having type information describing the document structure, it is possible to verify the validity of these transformations.

Definition 6.3.2 *We define the type of a term representing an XML document as:*

$$\begin{array}{lcl} type(X) & = & pcdata,\ if\ X\ is\ not\ of\ de\ form \\ & & f(s_1,\ldots,s_n)\ where\ n \geq 1 \\ type(f(s_1,\ldots,s_n)) & = & f(type(s_1),\ldots,type(s_n)) \end{array}$$

Example 6.3.5 *Given the following term:*

$$teacher(name("Jorge"), phone("123456789")),$$

```
start:-
        xml2pro('./examples/amf.xml',Tea),
        xml2pro('./examples/pubs.xml',Pub),
        assert(pubs(Pub)),
        bind_type(Tea,teacher),
        swc(Tea,Tea2),
        pro2xml(Tea2,'teacherchecked.xml'),
        retract(pubs(_)).


swc(In,Out):-
        r1(In,Out1),
        r2(Out1,Out2),
        r3(Out2,Out).

r1(A,B):-
     replace(curriculum(X),
             curriculum('Degree and year missing!'),A,B,
             [deep(curriculum(X),A),not(sub_string(X,'degree')),
                                    not(sub_string(X,'year'))]).

r2(A,A):-
     failure(A,[deep(email(X),A),
                not(sub_string(X,'@ncc.up.pt'))],
                'Valid email not found!').

r3(A,B):-
        delete(pub(X),A,B,[deep(pub(X),A),
                           X =*= <_,title(T),_>,
                           pubs(Pub),
                           not(deep(title(T),Pub))]).
```

Figure 6.1: Teacher's home page constraints

then,

$$type(teacher(name("Jorge"), phone("123456789"))) = \\ teacher(name(pcdata), email(pcdata)),$$

During compile time the type verification procedure verifies if transformations made by *replace* and *delete* rules are compatible with the declared type by following these steps:

- A type α is associated with document XML by a *bind_type* instruction.
- For any instruction of the form $delete(s_1, XML, XML2, L)$:
  - Verify if s_1* or $s_1?$ or $s_{1_{\{0,Max\}}}$ is found in the type, in case it isn't, report an error.
  - If $L \neq []$ and s_1+ or $s_{1_{N,Max}}$ where $N > 0$ was found, report a warning saying that if the constraints in L covers all the elements in the document, then $XML2$ will be an invalid document.
- For any instruction of the form $replace(s_1, s_2, XML, XML2, L)$, if $type(s_1) \neq type(s_2)$:
  - If s_1 and s_2 appear as a sequence in the type:
    - Report an error in cases were s_1 and s_2 appear as:
      - s_1, s_2 (because s_1 and s_2 must occur exactly once).
      - s_1, s_2*; s_1, s_2+; $s_1, s_2?$; $s_1, s_{2_{\{N',M'\}}}$, and $L = []$ (because s_1 must occur once).
      - s_1+, s_2; s_1*, s_2; $s_1?, s_2$; $s_{1_{\{N,M\}}}, s_2$; $s_{1_{\{N,M\}}}, s_2?$, if $N \neq 0$ and $M \neq 1$ (because only one s_2 is allowed).
      - s_1+, s_2+; s_1+, s_2*; $s_1+, s_{2_{\{N,M\}}}$; $s_1+, s_2?$, and $L = []$, (because at least one s_1 must occur).
      - $s_{1_{\{N,M\}}}, s_2+$; $s_{1_{\{N,M\}}}, s_{2_{\{N',M'\}}}$;$s_{1_{\{N,M\}}}, s_2*$: if $N \neq 0$ and $L = []$ (because s_1 must occur N times at least).
    - Report a warning in cases were s_1 and s_2 occur as:
      - $s_1, s_2?$; $s_{1_{\{0,1\}}}, s_2?$: s_2 *must not occur in the document.*
      - s_1+, s_2+; s_1+, s_2* and $L \neq []$ *the constraints in L cannot replace all s_1's.*
      - $s_1+, s_2?$; $s_1*, s_2?$ and $L \neq []$ *at most one s_1 can be translated and s_2 must not occur.*
      - $s_1*, s_{2_{\{N',M'\}}}$; $s_1?, s_{2_{\{N',M'\}}}$, *the number of s_1's plus the number of s_2's may not exceed M'.*
      - $s_1?, s_2?$: s_2 *must not occur.*
      - $s_{1_{\{N,M\}}}, s_2+$; $s_{1_{\{N,M\}}}, s_2*$, and $L \neq []$ *the number of s_1's which remain not replaced must be higher than N.*

- If s_1 and s_2 occur as a disjunction (|) in the type:
  - Report an error in cases were s_1 and s_2 occur as:
    - $s_1?|s_{2_{\{N',M'\}}}$ and $N' \geq 2$ (because only one s_1 will be translated).
    - $s_{1_{\{N,M\}}}|s_2$; $s_{1_{\{N,M\}}}|s_2?$, and $N' \geq 1$ (because only one s_2 is allowed).
    - $s_{1_{\{N,M\}}}, s_{2_{\{N',M'\}}}$, if $[N,M] \cap [N',M'] = \emptyset$ (because they will never have a compatible number of occurrences).
  - Report a warning in cases were s_1 and s_2 occur as:
    - $s_1+|s_2$; $s_1+|s_2?$; $s_1*|s_2$; $s_1*|s_2?$: *because only one* s_2 *is allowed.*
    - $s_1+|s_{2_{\{N,M\}}}$; $s_1*|s_{2_{\{N,M\}}}$, *because* $N \leq$ *number of* $s_1's \leq M$.
    - $s_{1_{\{0,M\}}}|s_2+$, *at least one* s_1 *must be translated.*
    - $s_{1_{\{N,M\}}}|s_{2_{\{N',M'\}}}$, $[N,M]\cap[N',M'] \neq \emptyset$ *because* $N' \leq$ *number of* $s_1's \leq M'$.

Example 6.3.6 *Using the same scenario as the one presented in example 6.3.4. Errors are reported if for example we try to apply some of the following actions:*

- *delete*(*name*(X), XML_1, XML_2, []) *since name is a tag that cannot be deleted from the document.*
- *replace*(*phone*(P), *email*(”*test@testmail.com*”), XML_1, XML_2, []), *since although we can have several emails, the phone element is mandatory.*

Warnings are reported if we try the following action:

- *delete*(*author*(A), XML_1, XML_2, [*internal_database* (DB), *not*($DB = * = db$(_, *name* (A), _))]), *here the goal is to delete all the authors whose name does not appear in another XML file in variable DB. A warning is produced warning that, if all the authors for a given record are covered, then the XML produced will be invalid.*

Note that the verifications were all made at compile time.

6.3.1.3 Run time consistency checking

Types declared for the documents are transformed in programs, as explained in section 5.5.1, and used for checking the content of the document given as input and the document generated as output.

Creating a new program from the original SWC which, after applying the verification, is executed in order to find any possible inconsistency in the set of constraints.

Given the following SWC:

```
swc(WpageI,WpageO):-
        r_1(WpageI,Wpage1),
        ...
```

```
            r_n(Wpage_{n-1},WpageO).

r_1(W1,W2):-...
...
r_n(W1,W2):-...
```

a new program is generated where the body of each rule is replaced by the constraints found inside each action and an error message. Then, after applying the original SWC, checking the output XML file using this new program is performed. This way it is possible to detect inconsistency errors between the rules.

Example 6.3.7 *Using the scenario presented in example 6.3.4, the generated program for consistency errors detection is:*

```
verify(WpageO):-
                r1(WpageO),
                r2(WpageO),
                r3(WpageO).

r1(W):-
    deep(curriculum(X),W),
    not(sub_string(X,'degree')),
    not(sub_string(X,'year')),
    write("Rule 1 fails!"),!.

r1(_).


r2(W):-
    deep(email(X),W),
    not(sub_string(X,'@ncc.up.pt')),
    write("Rule 2 fails!"),!.

r2(_).

r3(W):-
    deep(pub(X),W),
    X =*= <_,title(T),_>,
    pubs(Pub),
    not(deep(title(T),Pub)),
    write("Rule 3 fails!"),!.

r3(_).
```

Example 6.3.8 *Suppose we add to the teacher's SWC presented in example 6.3.4 the following constraint:*

```
r4(W1,W2):-
    delete(email(E),W1,W2,[]).
```

Although not detected at compile time, since not having an email is valid accordingly to the DTD, this error would be detected when checking the rules at run-time by:

```
r2(W):-
    deep(email(X),W),
    not(sub_string(X, '@ncc.up.pt')),
    write("Rule 2 fails!"),!,fail.
```

And thus rule number 4 made an inconsistent change in relation to rule number 2.

Note that the output XML document is again checked with respect to the declared types. This way we can find errors from actions which depend on values.

6.4 Discussion

VeriFLog is a framework for website verification using XCentric. We presented examples of application and a framework for website verification at compile-time and run-time using type verification of rules applicable to documents. Errors can be automatically solved by introducing actions that are executed whenever an error is found. Errors within the set of web constraints imposed by the system administrator are detected by type checking at compile time complemented with further type checking at run time and consistency analysis to detect constraints whose action may violate other constraints.

One may argue that, nowadays big websites tipically rely on Database Management Systems and constraints are imposed at the database level. Although a valid argument, there is a new type of websites arising, websites that allows visitors to add, remove, edit and change content. These websites are usually designated as wikis and the most famous is probably Wikipedia [Fou07]. Due to their nature, wikis are vulnerable to vandalism, thus, VeriFlog can be applied to filter content on these websites.

As a final word we should point that XML-based websites although not yet widespread, will probably become more and more popular in the next years thus increasing the utility of tools such as the one presented here.

Chapter 7

Conclusions

7.1 General Conclusions

With the work presented in this thesis we have shown that:

- It is possible to improve XML processing in logic programming by switching from standard unification to flexible arity unification.
- Schemas translated to regular types can be smoothly integrated with the flexible arity unification, providing static and runtime validation.
- There are applications where our approach can be used successfully.

7.2 Declarative XML Processing

With this work we improved XML processing by representing documents as terms and by implementing a new procedure for unification of flexible arity terms. This new kind of unification is quite useful for XML processing. When representing trees as terms, one can end with enormous terms representing big documents. With flexible arity term unification, we enable focusing on the parts of the document which matter (see example 4.2.1 for a comparison between the traditional XML processing approach and ours). Here, a sequence of elements can be binded with just one sequence variable.

We decided to use the procedure of Kutsia [Kut02c] as the basis for our work because it is the one that fits better in our initial goal, which was to define a highly declarative language for XML processing based on an extension of standard unification to denote the same objects denoted by XML: trees with an arbitrary number of leafs. Although our procedure is based on this previous one it has some differences motivated by its use as a constraint solving method in a CLP package:

- Kutsia procedure gave the whole set of solutions to an equality problem as output. We changed that point accordingly to the standard *backtracking* model of *Prolog*. We give as output one answer substitution and subsequent calls to the same query will result in different answer substitutions computed by backtracking. When every solution is computed the query fails indicating that there are no more solutions.

- a direct consequence of the previous point is that our implementation deals with infinite sets of solutions (see example 4.5.6). It simply gives all solutions by backtracking.

- Kutsia procedure was a new definition of unification for the case of terms with flexible arity symbols. Our implementation transforms the initial set of constraints into a different (larger) set of equalities solved by standard unification and uses standard *Prolog* unification for propagating substitutions.

The generally small amount of code needed to solve the problems presented through this thesis shows that our approach is more declarative than the ones already available. Usually, XML processing languages use pattern matching for XML processing. With our approach we allow the construction of incomplete documents. This feature has two main advantages:

- When some information is not available at the time the document is processed, one can just leave free variables and bind them to sequences of code later when they become available (see example 4.5.9).

- It allows a bidirectional use of programs which is useful, for creating two different XML documents where the content of tags is the same, or for equivalent documents comparison (see example 4.5.10).

Since many practical problems are just queries we also implemented pattern matching as a built-in predicate. Pattern matching uses a different operator and thus can be explicitly used when the programmer just needs to query or process a ground document.

As the work was developed we felt the need of applying sequence matching in depth in a simple way. The *deep* family of predicates accomplish this task by matching sequences of elements with documents in depth. They provide a straightforward approach to querying in depth which can be seen in example 5.7.1, where a XQuery use case is translated to XCentric resulting in a simpler program.

7.3 Sequence Types

Regular types revealed to be a quite simple way to code DTDs. We defined a relation between regular types and regular expression types corresponding to DTDs, and used them for type verification at compile and run time. For the compile time verification we defined and implemented a sequence type intersection algorithm that verifies if unifications in the

domain of flexible arity terms are valid. Typed unification checks if the type of the resulting term (equal to the intersection of the types for the initial terms) is not empty. If the type is not empty then there are terms belonging to both sides of the unification which are common. Runtime checking is done by translating types to unary programs which use flexible arity term unification to check if a given sequence is valid against a given type, here represented by a program (see the illustrating example 5.5.6).

We improved run-time type checking by defining new types specifically representing some popular XML Schema types, such as:

- Basic types: string, integer float and boolean.
- Occurrences of sequences: one can define the maximum and minimum number of occurrences of a sequence of elements.
- Orderless sequences: one can declare sequences of elements that may appear in random order.

We implemented types as an optional feature: the programmer decides when to use them or not. With this approach we hope to please both the people which don't mind to use types in the context of logic programming and those who do mind.

7.4 Further Work

Here we address some lines of future work.

7.4.1 Implementation

The implementation was done using C for parsing of programs which are then translated to a Prolog-friendly syntax, and then all the remaining transformations (translation of types to programs, type verification, etc.) is made in Prolog. We choose SWI-Prolog to use its nice interface to semistructured data. One issue that must be solved is related with the performance. We identified two main problems:

- Overhead created by using the Prolog compiler.
- Exponential blow-up created by the Projection step in the flexible arity unification procedure.

These two subjects must be further studied in order to find better solutions. We think that implementing sequence unification directly in the WAM can improve performance.

7.4.2 Relation with other forms of unification

Flexible arity term unification has roots in word unification and equations over free semigroups. One interesting line of future work will be the study in more detail of the relation between Flexible Arity Term Unification and A-Unification [Plo72]. Another line of future work will be the study of the relation between flexible arity term unification and higher order unification [Koh94].

7.4.3 New Type Language

Regular types are useful for describing a language of types equivalent to DTDs. Nowadays, XML Schemas are becoming popular and the need for good languages to describe their types that can be used at compile time a relevant topic of research. We added some basic support for XML schema by introducing basic elements and enabling the run time check of occurrences of sequences of elements and orderless sequences of elements. A future line of research is to study new languages extending regular types that could be used to describe a subset of XML Schemas at compile time.

7.4.4 Applications

In this thesis we presented an application for verification and repairing web sites. The scenario we used was the website of a university. This kind of website has typically many people with access to modify different parts of it, for example teachers have access to their own page, and thus the generation of errors accordingly to some predefined criteria may be high. VeriFlog helps the website administrator to control and fix these errors.

A new type of websites is becoming popular, the wikis [LC01], where different people can freely update parts of it. The most famous of these is probably Wikipedia. Due to their nature this websites are vulnerable to vandalism and errors. One of the future topics of research is the study of the applicability of tools such as VeriFlog in the control of errors and vandalism in wikis.

Another future topic of research is the definition of sequence disunification algorithms and its application to Collaborative Schema Construction for XML. There is now ongoing work on this topic, presented in [CFK07].

Finally, we must mention that XCentric was made to XML processing but it may be applied to any domain where dealing with sequences is a key issue. We foresee that bio-informatics can be a relevant domain of application of XCentric in the future.

7.5 Summary

In this thesis we used a recently studied new kind of unification and new types to represent schemas, as the core features of a new language for XML processing. This new approach achieves a high declarative model to process XML data. We shown the applicability of this new language in the implementation of a tool for verifying and repair content of websites.

With this work we hope to show that logic programming (and CLP) is a suitable paradigm for the processing of XML data and also that XML is a strong motivation to take more seriously the development of new logic programming languages dealing with sequences (of arbitrary length) of terms.

Bibliography

[ABFR05a] María Alpuente, Demis Ballis, Moreno Falaschi, and Daniel Romero. A Rewriting-based Framework for Web Sites Verification. In *Electronic Notes in Theoretical Computer Science*, pages 41–61. Elsevier Science, 2005.

[ABFR05b] María Alpuente, Demis Ballis, Moreno Falaschi, and Daniel Romero. A semi-automatic methodology for repairing faulty web sites. In María Alpuente, Santiago Escobar, and Moreno Falaschi, editors, *WWV*. Departamento de Sistemas Informaticos y Computacion, Universidad Politecnica de Valencia, 2005.

[AW07] Krzysztof R. Apt and Mark Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[BBC+04] Nick Benton, Gavin Bierman, Luca Cardelli, Erik Meijer, Claudio Russo, and Wolfram Schulte. *Microsoft C-omega*. Microsoft Research, http://research.microsoft.com/Comega/, 2004.

[BCF03] Vronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN Int. Conference on Functional Programming*, Uppsala, Sweden, 2003.

[BDJ+00] B. Buchberger, C. Dupre, T. Jebelean, B. Konev, F. Kriftner, T. Kutsia, K. Nakagawa, F. Piroi, D. Vasaru, and W. Windsteiger. The Theorema System: Proving, Solving, and Computing for the Working Mathematician. Technical Report 00-38, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2000.

[BE00] François Bry and Norbert Eisinger. Data modeling with markup languages: A logic programming perspective. In *15th Workshop on Logic Programming and Constraint Systems, WLP 2000, Berlin*, 2000.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, New York, NY, USA, 1998.

[Bol00] Harold Boley. Relationships between logic programming and XML. In *Proc. 14th Workshop Logische Programmierung*, 2000.

[BS02a] François Bry and Sebastian Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *2nd Annual International Workshop Web and Databases*, volume 2593 of *LNCS*, 2002.

[BS02b] François Bry and Sebastian Schaffert. Towards a declarative query and transformation language for XML and semistructured data: simulation unification. In *Proc. of the 2002 International Conference on Logic Programming*, 2002.

[CF03] Jorge Coelho and Mário Florido. Type-based XML Processing in Logic Programming. In *Practical Aspects of Declarative Languages*, volume 2562 of *LNCS*, 2003.

[CF04] Jorge Coelho and Mário Florido. Clp(flex): Constraint logic programming applied to xml processing. In *Ontologies, Databases and Applications of SEmantics (ODBASE)*, volume 3291 of *LNCS*. Springer Verlag, 2004.

[CF05] Jorge Coelho and Mário Florido. Xml processing in logic programming. In João Correia Lopes José Carlos Ramalho, Alberto Simões, editor, *XATA2005, XML: Aplicações e Tecnologias Associadas (Vila Verde, Braga, 10 e 11 de Fevereiro de 2005)*. Universidade do Minho, Fevereiro 2005.

[CF06a] Jorge Coelho and Mário Florido. Unification with flexible arity symbols: a typed approach. In *20th International Workshop on Unification (UNIF'06)*, Seattle, USA, 2006.

[CF06b] Jorge Coelho and Mário Florido. VeriFLog: Constraint Logic Programming Applied to Verification of Website Content. In *Int. Workshop XML Research and Applications (XRA'06)*, volume 3842 of *LNCS*. Springer-Verlag, 2006.

[CF07a] Jorge Coelho and Mário Florido. Type-based static and dynamic website verification. In *The Second International Conference on Internet and Web Applications and Services*. IEEE Computer Society, 2007.

[CF07b] Jorge Coelho and Mário Florido. Xcentric: A logic-programming language for xml processing. In Torsten Grust, editor, *ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X)*, 2007.

[CF07c] Jorge Coelho and Mário Florido. Xcentric: Logic programming for xml processing. In *9th ACM International Workshop on Web Information and Data Management*. ACM Press, 2007.

[CFK07] Jorge Coelho, Mário Florido, and Temur Kutsia. Sequence disunification and its application in collaborative schema construction. In *International Workshop on Collaborative Knowledge Management for Web Information Systems*, Lecture Notes in Computer Science. Springer Verlag, 2007.

[Col90] Alain Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[Cor02] Microsoft Corp. *Microsoft Office 2003.* Microsoft, http://office.microsoft.com/, 2002.

[CS00] Sophie Cluet and Jerome Simon. Yatl: a functional and declarative language for xml. Draft manuscript, 2000.

[Des04] Thierry Despeyroux. Practical semantic analysis of web sites and documents. In Stuart I. Feldman, Mike Uretsky, Marc Najork, and Craig E. Wills, editors, *WWW*, pages 685–693. ACM, 2004.

[DL03] Silvano Dal Zilio and Denis Lugiez. XML Schema, Tree Logic and Sheaves Automata. In *RTA 2003 – 14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 246–263. Springer-Verlag, June 2003.

[DW06] Wlodzimierz Drabent and Artur Wilk. Combining XML querying with ontology reasoning: Xcerpt and DIG. In *Proceedings of RuleML-06 Workshop: Ontology and Rule Integration, Athens, Georgia, USA (11th November 2006)*, 2006.

[DZ92] Philip Dart and Justin Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming.* The MIT Press, 1992.

[ebX04] ebXML.org. *Electronic Business using eXtensible Markup Language (ebXML).* WWW, http://www.ebxml.org/, 2004.

[FD92] Mário Florido and Luís Damas. Types as theories. In *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming*, 1992.

[FFR97] Adam Farquhar, Richard Fikes, and James Rice. The Ontolingua Server: A tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46(6):707–727, 1997.

[Fou03] Phyton Software Foundation. *Phyton SAX 2.* WWW, http://www.python.org/doc/2.3/lib/markup.html, 2003.

[Fou07] Wikimedia Foundation. *Wikipedia: Multilingual, web-based, free content encyclopedia project.* World Wide Web, http://www.wikipedia.org/, 2007.

[FSC+03] Mary F. Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. Implementing xquery 1.0: The galax experience. In *VLDB*, pages 1077–1080, 2003.

[FSVY91] Thom W. Fruhwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *LICS*, pages 300–309. IEEE Press, 1991.

[GF92] Michael Genesereth and Richard Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual TR Logic-92-1. Technical report, Stanford University, Stanford, 1992.

[GH01] Daniel Cabeza Gras and Manuel V. Hermenegildo. Distributed www programming using (ciao-)prolog and the pillow library. *Theory and Practice of Logic Programming (TPLP)*, 1(3):251–282, 2001.

[GLPS05] Vladimir Gapeyev, Michael Y. Levin, Bwnjamin Pierce, and Alan Schmitt. The xtatic experience. In *ACM SIGPLAN Workshop on Programming Language Technologies for XML*, January 2005.

[HHK95] Monika Rauch Henzinger, Thomas A. Henzinger, and Peter W. Kopke. Computing simulations on finite and infinite graphs. In *FOCS*, pages 453–462, 1995.

[HLS04] Inc. Health Level Seven. *Health Level Seven.* WWW, http://www.hl7.org/, 2004.

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation, 3/E.* Adison-Wesley Publishing Company, Reading, Massachusets, USA, 2007.

[HP00] Haruo Hosoya and Benjamin Pierce. XDuce: A typed XML processing language. In *Third Int. Workshop on the Web and Databases*, volume 1997 of *LNCS*, 2000.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.

[IET03] IETF. *Unicode Transformation Format 8 bit.* World Wide Web, `http://www.ietf.org/rfc/rfc2279.txt`, 2003.

[Jaf90] Joxan Jaffar. Minimal and complete word unification. *Journal of the ACM*, 37(1):47–85, 1990.

[Jel04] Rick Jelliffe. *Schematron.* WWW, http://xml.ascc.net/ resource/schematron/schematron.html, 2004.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint Logic Programming. In *Proceedings of the Fourteenth Annual ACM Symp. on Principles of Programming Languages, POPL '87*, pages 111–119, Munich, Germany, 1987. ACM Press.

[JMSY92] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The clp(r) language and system. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, 1992.

[KCK+03] Howard Katz, Don Chamberlin, Michael Kay, Philip Wadler, and Denise Draper. *XQuery from the Experts: A Guide to the W3C XML Query Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[KH06] Shinya Kawanaka and Haruo Hosoya. biXid: a bidirectional transformation language for XML. In *International Conference on Functional Programming (ICFP)*, pages 201–214, 2006.

[KK03] Oleg Kiselyov and Shriram Krishnamurthi. SXSLT: Manipulation Language for XML. In *Practical Aspects of Declarative Languages*, volume 2562 of *LNCS*, 2003.

[KM04] Temur Kutsia and Mircea Marin. Unification procedure for terms with sequence variables and sequence functions (extended abstract). In *Proceedings of the 18th International Workshop on Unification (UNIF'04)*, Cork, Ireland, 5 June 2004.

[Koh94] Michael Kohlhase. *A Mechanization of Sorted Higher-Order Logic Based on the Resolution Principle*. PhD thesis, Universitt des Saarlandes, 1994.

[Kut02a] Temur Kutsia. Pattern unification with sequence variables and flexible arity symbols. *Electronic Notes on Theoretical Computer Science*, 66(5), 2002.

[Kut02b] Temur Kutsia. *Solving and Proving in Equational Theories with Sequence Variables and Flexible Arity Symbols*. PhD thesis, Johannes Kepler University, Linz, Austria, 2002.

[Kut02c] Temur Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In *Joint AISC'2002 - Calculemus'2002 conference*, LNAI, 2002.

[Kut04] Temur Kutsia. Solving equations involving sequence variables and sequence functions. In Bruno Buchberger and John A. Campbell, editors, *Proceedings of the 7th International Conference on Artificial Intelligence and Symbolic Computation, AISC'04*, volume 3249 of *Lecture Notes in Artificial Intelligence*, pages 157–170, Hagenberg, Austria, 22–24 September 2004. Springer Verlag.

[Kut05] Temur Kutsia. Context sequence matching for xml. In *Proceedings of the 1th Int. Workshop on Automated Specification and Verification of Web Sites*, 2005.

[LC01] Bo Leuf and Ward Cunningham. *The Wiki way: quick collaboration on the Web.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[Llo87] John Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[Mak77] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik USSR*, 103:147–236, 1977.

[May01] Wolfgang May. XPathLog: A Declarative, Native XML Data Manipulation Language. In *Int. Database Engineering & Applications Symposium (IDEAS '01)*, Grenoble, France, 2001. IEEE.

[Meg04] David Megginson. *Simple API for XML.* World Wide Web, `http://www.saxproject.org`, 2004.

[Mic03] Sun Microsystems. *Java Sun DOM API.* WWW, http://java.sun.com/j2se/1.4.2/docs/guide/plugin/dom/, 2003.

[Moz06] Mozilla. *XML User Interface Language (XUL).* Mozilla, http://www.mozilla.org/projects/xul/, 2006.

[MS99] Erik Meijer and Mark Shields. XMλ: A functional language for constructing and manipulating XML documents. (Draft), 1999.

[MT99] Enric Mayol and Ernest Teniente. A survey of current methods for integrity constraint maintenance and view updating. In *ER '99: Proceedings of the Workshops on Evolution and Change in Data Management, Reverse Engineering in Information Systems, and the World Wide Web and Conceptual Modeling*, pages 62–73, London, UK, 1999. Springer-Verlag.

[Ml04] Anders Mller. *Document Structure Description 2.* World Wide Web, http://www.brics.dk/DSD/dsd2.html, 2004.

[oC02] The Library of Congress. *Encoded Archival Description Application.* WWW, http://lcweb.loc.gov/ead/, 2002.

[oJ04] US Department of Justice. *Global Justice XML Data Model (GJXDM).* WWW, http://it.ojp.gov/jxdm, 2004.

[Plo72] G. Plotkin. Building in equational theories. *Machine Intelligence*, 7, 1972.

[Pro03] Perl XML Project. *Perl SAX 2.0.* WWW, http://perl-xml.sourceforge.net/perl-sax/, 2003.

[Pro04] Apache XML Project. *Xerces-C++ DOM API.* WWW, http://xml.apache.org/xerces-c/program-dom.html, 2004.

[Rob65] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[Rys01] Michael Rys. State-of-the-art xml support in rdbms: Microsoft sql server's xml features. *IEEE Data Eng. Bull.*, 24(2):3–11, 2001.

[SBm04] SBml.org. *Systems Biology Markup Language (SBML).* WWW, http://sbml.org/, 2004.

[Sch93] Klaus U. Schulz. Word unification and transformation of generalized equations. *Journal of Automated Reasoning*, 11(2):149–184, 1993.

[Sie78] Jörg H. Siekmann. Matching and unification problems. Technical report, University of Essex, 1978.

[Smo92] Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[Sof04] Enosys Software. *Enosys.* WWW, http://www.enosys.com/, 2004.

[Spe01] OASIS Committee Specification. *RELAX NG Specification.* WWW, `http://www.relaxng.org/`, 2001.

[SS94] Leon Sterling and Ehud Shapiro. *The Art of Prolog, 2nd edition.* The MIT Press, 1994.

[SW06] Hans Eric Svensson and Artur Wilk. XML Querying Using Ontological Information. In *Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning, Budva, Montenegro (10th–11th June 2006)*, volume 4187 of *LNCS*, pages 190–203. REWERSE, 2006.

[Tha73] James W. Thatcher. *Tree automata: An informal survey.* Prentice-Hall, 1973.

[vHvdM99] Frank van Harmelen and Jos van der Meer. Webmaster: Knowledge-based verification of web-pages. In Ibrahim F. Imam, Yves Kodratoff, Ayman El-Dessouki, and Moonis Ali, editors, *IEA/AIE*, volume 1611 of *Lecture Notes in Computer Science*, pages 256–265. Springer, 1999.

[W3C95] W3C. *Standard Generalized Markup Language.* World Wide Web, `http://www.w3.org/MarkUp/SGML/`, 1995.

[W3C98] W3C. *Vector Markup Language.* World Wide Web, `http://www.w3.org/TR/NOTE-VML/`, 1998.

[W3C99a] W3C. *The Extensible Stylesheet Language Family.* WWW, http://www.w3.org/Style/XSL/, 1999.

[W3C99b] W3C. *Namespaces in XML.* WWW, http://www.w3.org/TR/REC-xml-names/, 1999.

[W3C99c] W3C. *XPath.* World Wide Web, `http://www.w3.org/TR/xpath`, 1999.

[W3C01a] W3C. *MathML.* World Wide Web, `http://www.w3.org/Math/`, 2001.

[W3C01b] W3C. *XSL Transformations (XSLT).* World Wide Web, http://www.w3.org/TR/xslt/, 2001.

[W3C04a] W3C. *Document Object Model.* World Wide Web, `http://www.w3.org/DOM`, 2004.

[W3C04b] W3C. *Document Type Definition.* World Wide Web, `http://www.w3.org/TR/REC-xml`, 2004.

[W3C04c] W3C. *Extensible Markup Language (XML).* WWW, http://www.w3.org/XML/, 2004.

[W3C04d] W3C. *Extensible Markup Language (XML).* World Wide Web, http://www.w3.org/XML/, 2004.

[W3C04e] W3C. *XML Query Language.* WWW, http://www.w3.org/TR/xquery/, 2004.

[W3C04f] W3C. *XML Schema.* World Wide Web, `http://www.w3.org/XML/Schema/`, 2004.

[W3C04g] W3C. *XQuery 1.0 and XPath 2.0 Data Model.* WWW, http://www.w3.org/TR/xpath-datamodel/, 2004.

[W3C05a] W3C. *Synchronized Multimedia.* World Wide Web, `http://www.w3.org/AudioVideo/`, 2005.

[W3C05b] W3C. *XQuery Use Cases.* WWW, http://www.w3.org/TR/xquery-use-cases/, 2005.

[W3C06a] W3C. *HyperText Markup Language.* World Wide Web, `http://www.w3.org/MarkUp/`, 2006.

[W3C06b] W3C. *Web Services.* World Wide Web, `http://www.w3.org/2002/ws/`, 2006.

[W3C07] W3C. *World Wide Web Consortium.* World Wide Web, `http://www.w3.org/`, 2007.

[WD03] Artur Wilk and Wlodzimierz Drabent. On types for xml query language xcerpt. In *Principles and Practice of Semantic Web Reasoning*, volume 2901 of *LNCS*, 2003.

[WD06] Artur Wilk and Wlodzimierz Drabent. A Prototype of a Descriptive Type System for Xcerpt. In *Proceedings of 4th Workshop on Principles and Practice of Semantic Web Reasoning, Budva, Montenegro (10th–11th June 2006)*, volume 4187 of *LNCS*, pages 262–275. REWERSE, 2006.

[Wie06] Jan Wielemaker. *SWI Prolog.* WWW, http://www.swi-prolog.org/, 2006.

[YS90] Eyal Yardeni and Ehud Shapiro. A type system for logic programs. In *The Journal of Logic Programming*, 1990.

[Zob87] Justin Zobel. Derivation of polymorphic types for prolog programs. In *Proc. of the 1987 International Conference on Logic Programming.* MIT Press, 1987.

[Zob90] Justin Zobel. *Analysis of Logic Programs.* PhD thesis, Department of Computer Science, University of Melbourne, 1990.