HAL
open science

Static Analysis for Data-Centric Web Programming

Pierre Genevès

To cite this version:

Pierre Genevès. Static Analysis for Data-Centric Web Programming. Computer Science [cs]. Université Grenoble Alpes, 2014. tel-01102401

HAL Id: tel-01102401

https://inria.hal.science/tel-01102401v1

Submitted on 12 Jan 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Static Analysis for Data-Centric Web Programming

Pierre Genevès

November 21st, 2014

Document presented in partial fulfillment of the requirements for the degree of "Habilitation à Diriger les Recherches (HDR)" from the University of Grenoble Alpes, 2014.

Referees:

Boualem Benatallah
Giuseppe Castagna
Amedeo Napoli

Contents

1 Introduction **1**
1.1 Outline . . . 1
1.2 Societal Stakes and Scientific Challenges . . . 2
1.3 Original Research Approach Developed . . . 3
1.4 Contributions . . . 5
1.5 Evolution of Schemas . . . 6
1.6 Functions and Polymorphism . . . 6
1.7 Automated Analysis of Layouts . . . 7
1.8 Containment for Graph Queries . . . 8
1.9 Pointers to Other Related Results . . . 10

2 Evolution of Types **11**
2.1 Introduction . . . 12
2.2 Analysis Framework . . . 13
2.3 Logical Setting . . . 16
2.4 Analysis Predicates . . . 24
2.5 Impact of Standards' Evolution on Valid Documents . . . 28
2.6 Impact on Queries . . . 36
2.7 System Implementation . . . 38
2.8 Related Work . . . 38
2.9 Conclusion . . . 40

3 Functions and Polymorphism **41**
3.1 Introduction . . . 41
3.2 Semantic Subtyping Framework . . . 46
3.3 Tree Logic Framework . . . 50
3.4 Logical Encoding . . . 52
3.5 Polymorphism: Supporting Type Variables . . . 55
3.6 Implementation and Practical Experiments . . . 62
3.7 Related Work . . . 70
3.8 Conclusion . . . 71

4 Analysis of Cascading Style Sheets **73**
4.1 Introduction . . . 74
4.2 Current Practice . . . 75
4.3 CSS: An Overview . . . 76
4.4 Theoretical Foundations . . . 79
4.5 A Logical Modeling of CSS . . . 83
4.6 Prototype Implementation . . . 86
4.7 Reasoning with Style . . . 86

4.8 Automated CSS Size Reduction 92
4.9 Conclusions 97

5 Containment for a SPARQL Fragment 101
5.1 Introduction 102
5.2 Preliminaries 103
5.3 SPARQL Query Containment 115
5.4 μ-calculus 115
5.5 RDF Graphs as Transition Systems 118
5.6 Encoding SPARQL Query Containment 123
5.7 Experimental Investigations 132
5.8 Query Containment Solvers 133
5.9 Benchmark Design 134
5.10 Experimental Results 139
5.11 Related Work 142
5.12 Conclusions 146

6 Conclusion and Perspectives 149
6.1 Summary of Contributions 149
6.2 Perspectives 149

Bibliography 155

Chapter 1

Introduction

Contents

1.1 Outline 1
1.2 Societal Stakes and Scientific Challenges 2
1.3 Original Research Approach Developed 3
1.4 Contributions 5
1.5 Evolution of Schemas 6
1.6 Functions and Polymorphism 6
1.7 Automated Analysis of Layouts 7
1.8 Containment for Graph Queries 8
1.9 Pointers to Other Related Results 10

1.1 Outline

This document presents an excerpt from the research results that I have obtained since I received a PhD in December 2006. My research addresses one fundamental challenge of our time: building the necessary theoretical foundations and practical tools for ensuring guarantees on web applications such as robustness, security, privacy, and efficiency, towards a web of trust. For this purpose, I build static analysis methods that analyse the source code of web applications for the purpose of automatically detecting defects, or certifying guarantees such as the absence of certain kinds of errors, otherwise. The methods that I develop can also be used for performing semantically-verified optimisations, in compiler design for instance. One particularity of my approach resides in the introduction of novel reasoning techniques, based on new rich modal logics and innovative algorithmic techniques. Another particularity of my research is that it considers theoretical aspects, algorithmic aspects as well as applied aspects such as implementation techniques, experimental validation, and practical relevance. The overall goal of my research is to enable people to construct more reliable, secure, and efficient information systems.

The present Chapter introduces more context about my research, highlights specific aspects that constitute the originality and uniqueness of my approach, and gives a summary of four of my main contributions. These four contributions are detailed in the next Chapters: evolution of types in Chapter 2, functions and polymorphism in Chapter3, analysis of layouts in Chapter 4 and containment for a SPARQL fragment in Chapter 5. Chapter 6 discusses perspectives for further research.

1.2 Societal Stakes and Scientific Challenges

The need for data-centric guarantees on web applications

Today, our private life, our work, and more generally data correctness are put at risk by web applications. This is already witnessed by a number of failures with dramatic consequences. For instance, in June 2011, a breach in Citibank's web portal exposed confidential data concerning 200 000 customer accounts. In March 2011, a failure in the GlobalPayment system exposed 1.5 million of credit card numbers. The frequency of such failures is increasing rapidly. In February 2013, 250 000 Tweeter accounts were compromised. In August 2013, global web traffic brutally fell 40% when Google was down for 11 minutes. In January 2014, the credit card details of 20 million South Koreans were stolen.

My vision is that critical software is a notion no longer limited to systems traditionally considered as critical, such as found in airplanes and nuclear power plants for example. By manipulating unprecedented volumes of data from unprecedentedly large numbers of users, web applications are reaching a criticality level that was never attained before. They are on the verge of becoming the critical applications of the twenty-first century.

The aforementioned failures illustrate that considering web data as merely strings is insufficient as it exposes to vulnerabilities (e.g. code injection) and, more generally, it prevents accurate tracking and verification of data manipulations. Existing approaches for program verification fail to extend to web applications, for a simple but fundamental reason: web applications manipulate **web data that are very richly structured and heterogeneous** (document trees, knowledge graphs, nested key-value records). This makes a major difference with software traditionally considered as critical, which, in comparison, manipulate much simpler data. This is also a reason why we cannot reuse or extend existing methods but must develop novel, radically different, methods instead.

My research seeks to address this challenge by building **novel theoretical foundations** and **innovative practical software tools** for the static analysis of the source codes of web applications. Their purpose is to automatically detect defects, or otherwise prove guarantees such as the absence of errors and undesired behaviours. The overall objective is to open up new horizons for the development of science towards a web of trust, and to obtain concrete tools capable of verifying real-world web applications.

Challenges in the automated verification of web applications

The static analysis of a web application consists in automatically inferring properties about all its possible executions, encompassing all possible inputs and outputs, for all users. It is distinguished from testing (sometimes called dynamic analysis) which considers a limited number of executions, and thus, except for very small applications, cannot guarantee the absence of errors.

For a given class of defects, a *sound* static analysis method detects defects in a

exhaustive manner. It never wrongly considers that a program is safe when actually it is not. Soundness provides the payoff I am looking for: by soundly checking the web application, one can be certain that certain defects will certainly not happen. Short of this, we just have a bug-finder; while it may be useful, it does not constitute a sufficient basis for establishing security, privacy and robustness guarantees. Therefore, in my research, I look for sound static analysis methods, even though some results may also apply to bug-finding methods. *Complete* static analysis methods never raise false alarms in the sense that all detected defects are actual. Static analysis attempts to solve a problem that is generally unsolvable, in the sense that sound and complete (and terminating) static analysis methods are impossible for web applications in general[1]. The major consequence is that sound and terminating methods are bound to generate false alarms. Reducing the number of false alarms by increasing the precision of the analysis while keeping time and memory costs low constitutes a scientific challenge. Furthermore, this challenge comes with an inherent antagonism: on one hand the modelling with precision of real-world web applications motivates the search for more expressivity of the algebras under consideration, while, on the other hand, practical feasibility encourages the search for efficient algorithms with reasonable worst-case behaviour. This stimulates the search for relevant trade-offs between the expressive power of an algebra, algorithmic complexity, and appropriate algorithmic techniques that avoid the worst-case behaviour as much as possible.

One crucial particularity of web applications consists in processing rich data structured in various forms: ordered trees (such as most webpages and XML documents), unordered nested records (such as JSON records), and graphs (such as social networks and sets of RDF triples). These structures emerged as flexible – and de facto – ways of representing data on the web. However, we still miss foundations for their manipulation. The richness, heterogeneity and coexistence of different structures constitute an important scientific challenge. I do not envision progress in the static analysis of web applications unless breakthroughs are made. This is the central issue considered in my research.

1.3 Original Research Approach Developed

Over the years, I have been developing a research approach for advancing the state-of-the-art, which is original in several respects. Three essential aspects constitute the scientific originality of my approach:

1. I have been using and developing modal logics for the purpose of modelling web data structures;
2. I have been using and developing novel exact decision procedures and algorithmic techniques for the static analysis of applications that process web data;

[1] In computability theory, this is known as Rice's theorem: any property which is non-trivial (in the sense that it is neither always true nor always false) concerning the semantics of a turing-complete programming language is undecidable. In other terms, there is no algorithm that decides a non-trivial property on the program source code (as this would amount to solving Turing's halting problem).

3. I have focused on data-centric properties and important facets that constitute the particularity of web applications, including tree and graph shaped data, rapid evolution of schemas, and sophisticated layouts.

The scientific rationale for (1) and (2) above is that I want a solid rigorous and generic ground for developing proper foundations[2]. To this end, I have chosen to focus on extending a very specific kind of logics: modal logics. There are three essential reasons for that, of theoretical and algorithmic nature:

a. Modal logics are more robust by extension than classical logics, while being of similar or equal expressive power[3]. My research requires expressive logics for reasoning with web data constraints. For example, transitive closure is needed for capturing data constraints (as in e.g. RDFS and regular tree grammars) and navigation patterns (with e.g. regular expressions) found in queries[4].

b. Modal logics provide an interesting basis in the search for a common notation where the underlying logical data model can be transparently customized in various ways. It is possible to define interpretations of a logical formula on particular data models, without changing the logical language syntax. It is possible to restrict the data model at the purely syntactic level[5]. For a logic that admits the *finite tree model property*, it is even possible to use a decision procedure over finite trees in order to solve a problem over graphs. Modal logics thus provide an interesting foundation for a unifying formalism for reasoning about data structures which are ordered/unordered and graph or tree-shaped, which is exactly what we need to harness the heterogeneity of web data structures.

c. Unlike their tree automata counterparts, modal logics are agnostic to the notion of non-determinism. Determinization is a complex process for most tree automata. Instead, a formula (of a logic which is closed under negation[6]) can simply be put in negation normal form by propagating negation using De Morgan style laws. Furthermore, convenient representations (such as Hintikka sets

[2] As opposed to ad-hoc and hardly reusable methods sometimes found in, e.g., type theory.

[3] For instance, the alternation-free fragment of the μ-calculus over finite trees is equally as expressive than the weak monadic second order logic of two successors. The former is decidable in EXPTIME whereas the latter is decidable in hyper-exponential (non-elementary) time. This situation arises from a difference in representation: modal logics are, in general, less succinct than classical logics. This is why, for instance, monadic second-order logic (MSO) formulas, in their generality, cannot be solved for satisfiability in simple exponential-time by a decision procedure for μ-calculus: the translation from MSO to μ-calculus would involve hyper-exponential blow-ups in the syntactic representation.

[4] This is the reason why, e.g., first-order logic and its variants cannot be used to reason in general with regular constraints of web data. A fundamental limitation of first-order logic is that it cannot express transitive closure. I thus usually consider more expressive logics such as monadic-second order logic or its modal logic variant: the μ-calculus.

[5] For example, graph models can be restricted to be finite ordered tree models by a syntactic transformation of the formula and a conjunction that further restricts branches to be finite, as in [Genevès & Layaïda 2006].

[6] For a given logic, closure under negation ensures that the negation of any formula is also expressible in the logic.

and Fisher-Ladner closures) can be used for implementing effective decision procedures with semi-implicit techniques.

One motivation for (3) above originates from the fact that I am concerned with the applicability of my research and its practical relevance with respect to the aforementioned challenges. In this spirit, one particularity of my research is that it combines theoretical and applied aspects: I have been trying to focus on advancing relevant theories with experimental validation of implementations of these theories.

1.4 Contributions

In this context, this document presents an excerpt from the results that I have obtained since I received my PhD in December 2006. A few criteria were used for picking a subset of contributions. Basically, I chose to focus on a balanced set of self-contained results that are quite representative of my research approach, while illustrating different facets and particularities of web applications. In this spirit, I retained four major contributions:

Evolution of Schemas: these results allow to automatically and precisely monitor the impact of schema evolutions over sets of documents and queries that were formulated with respect to the initial schema subject to changes.

Functions and Polymorphism: these results allow to effectively decide subtyping for a rich type algebra equipped with function types and polymorphic types.

Automated Analysis of Layouts: these results allow to automatically detect bugs and prove properties such as the absence of certain kinds of rendering bugs in layouts designed using Cascading Style Sheets (CSS). In addition, novel semantical analyses for refactoring CSS style sheets can significantly reduce the size of style sheet used in the most popular web sites. The impact is significant as this directly amounts to reducing the resources required for global web traffic.

Containment for Graph Queries these results allow to decide the most essential static analysis problems such as the containment problem (and hence, the equivalence problem) for queries formulated in a fragment of the SPARQL language, in the presence of schema constraints.

I briefly review these contributions in the next Sections. Their detailed presentation constitute the topic of the following Chapters of this document.

The research results presented in this document have been obtained in collaboration with colleagues and students. For this reason, the "we" pronoun is used throughout the remaining part of this document. Specifically, the development of static analysis techniques for monitoring the impact of schema evolutions is a topic that I started in collaboration with Nabil Layaïda and Vincent Quint. The topic of function types and polymorphism was brought to my attention by Giuseppe Castagna, and addressed

in collaboration with Nils Gesbert that I supervised as a post-doc. The development of the first static analysis method for cascading style sheets is a topic that I started, in collaboration with Nabil Layaïda and Vincent Quint. Finally, query containment for SPARQL was the PhD topic of Melisachew Wudage Chekol, that I supervised in collaboration with Jérôme Euzenat and Nabil Layaïda.

1.5 Evolution of Schemas

In the ever-evolving context of the web, XML schemas continuously change in order to cope with the natural evolution of entities they describe. Schema changes have important consequences. First, existing documents valid with respect to the original schema are no longer guaranteed to fulfill the constraints described by the evolved schema. Second, the evolution also impacts programs manipulating documents whose structure is described by the original schema.

We have proposed a unifying framework for determining the effects of XML Schema evolution both on the validity of documents and on queries. The system is precious in analyzing various scenarios in which notions of forward and backward compatibilities for schemas are broken, and in which the result of a query may not be anymore what was expected. For example, when a content model is changing, is the change forward compatible, i.e. do the former documents remain valid? Does the change preserve a notion of backward compatibility, i.e. may new documents containing only the elements defined in the original schema be assembled in such a way that compatibility with the first schema is lost? What is the impact of the evolution of a schema on a given query or transformation?

The system provides a predicate language which allows one to formulate properties related to schema evolution. The system then relies on exact reasoning techniques to perform a fine-grained analysis. This yields either a formal proof of the property or a counter-example that can be used for debugging purposes. The system has been fully implemented and tested with real-world use cases, in particular with the main standard document formats used on the web, as defined by W3C. The system identifies precisely compatibility relations between document formats. In case these relations do not hold, the system can identify queries that must be reformulated in order to produce the expected results across successive schema versions.

These works, presented in Chapter 2, were published in ICFP'09 [Genevès *et al.* 2009] and TOIT'11 [Genevès *et al.* 2011].

1.6 Functions and Polymorphism

An important aspect when analysing web applications is the ability to develop modular analyses in order to support large applications. The simplest form of modularity is already present in most programs through the use of functions. A slightly more sophisticated form of modularity is the use of higher-order functions, i.e. functions that can be passed as parameters or returned as results like any other value such as

integers, lists, etc. The support for higher-order functions, although absent from the current XQuery standard language, appears as a requirement in the forthcoming third version of the standard [Engovatov & Robie 2010]. Another form of modularity found in web applications is the use of web services.

The development of rich type algebra that capture higher-order functions and parametric polymorphism is a response to the need of modular type systems. Higher-order functions and parametric polymorphism are two of the most powerful constructs in functional programming languages such as ML, Caml or Haskell. In the setting of XML, it is attractive to reach such powerful type systems where types can denote data types such as schemas and also computations. To that end, function types first need to be supported in the manner of [Benzaken *et al.* 2003, Castagna & Xu 2011]. As a second step, if functions can be made parametric using variables (parametric types), then they become more generic since they can operate on a large number of specific types. Such functions are well suited to promote code reuse.

In this setting, this work studies parametric polymorphism for type systems aiming at maintaining full static type-safety of functional programs (such as XQuery programs) that manipulate linked structures such as trees, potentially with higher-order functions. We consider a type algebra equipped with recursive, product, function (arrow), intersection, union, and complement types. We have showed how the subtyping relation between such type expressions can be decided through a logical approach. Our main result solves an open problem: we prove the decidability of the subtyping relation when this type algebra is extended with type variables. This provides a powerful polymorphic type system (using ML-style prenex polymorphism, where variables are implicitly universally quantified at top level), for which defining the subtyping relation is not obvious, as pointed out in [Castagna & Xu 2011], and for which no candidate definition of subtyping had been proved decidable before. The novelty, originality and strength of our solution reside in introducing a logical modeling for the semantic subtyping framework. Specifically, we model semantic subtyping in the finite tree logic presented in [Genevès *et al.* 2007] and rely on a slightly modified satisfiability solver in order to decide subtyping in practice. We obtain an EXPTIME ($2^{O(n)}$) complexity bound as well as an efficient implementation in practice.

These works, presented in Chapter 3, were published in ICFP'11 [Gesbert *et al.* 2011]. A journal article is under review.

1.7 Automated Analysis of Layouts

Cascading style sheets (CSS) play an increasingly important role on the web. Initially developed for separating content from presentation, they have been enriched to achieve more and more sophisticated layouts and renderings over the last few years. With the latest CSS3 standard, cascading style sheets constitute a crucial component in the design of web application user interfaces. Developing and maintaining cascading style sheets is an important issue to web developers as they suffer from the lack of rigorous methods. Existing techniques for debugging style sheets are empirical. We

have proposed a novel approach to fill this lack by introducing static analysis for style sheets. We have presented an original tool based on recent advances in tree logics.

From a theoretical perspective, CSS selectors could be related to XPath queries, for which an extensive static analysis has been conducted in [Genevès *et al.* 2007]. In this work, we dealt with the particular combinators and pseudo-classes found in CSS selectors. In particular, we have extended the logical solver from [Genevès *et al.* 2007], initially developed for XPath, to be able to reason about attribute values, by introducing an equality test that compares an attribute value to a constant. This is a worthy extension since it is sufficient for supporting CSS while preserving decidability for the extended logic (it is known that extending the logic with equality tests with variables results in undecidable logics, but this feature is not needed for CSS). In addition, we dealt with style properties and the propagation of values defined by the inheritance mechanism of CSS, which do not have any XPath counterpart.

From a practical perspective, there exists a whole class of dynamic analyses. Most of this technology relies on runtime debuggers that check the behavior of a CSS style sheet on a particular document instance. However, the aim of CSS is to be applied to an entire set of documents, usually defined by some schema. The existing runtime debugging tools help reducing the number of bugs. However, compared to our approach, they do not allow to prove properties over the whole set of documents to which the style sheet is intended to be applied. Therefore, our new approach and tool can be used in addition to debuggers to ensure a higher level of quality of CSS style sheets.

The tool is capable of statically detecting certain kinds of errors (such as inappropriate renderings), as well as proving properties related to sets of documents (such as coverage of styling information), in the presence or absence of schema information. This was the first attempt at statically analyzing CSS style sheets. This new static analysis tool can be used in addition to existing runtime debuggers to ensure a higher level of quality of CSS style sheets.

In addition, we presented a first prototype of static CSS semantical optimizer that is capable of automatically detecting and removing redundant property declarations and rules. Existing purely syntactic CSS optimizers might be used in conjunction with our tool, for performing complementary (and orthogonal) size reduction, toward the common goal of providing cleaner, lighter, and easily debuggable CSS files.

These works, presented in Chapter 4, were published in WWW'12 [Genevès *et al.* 2012] and [Bosch *et al.* 2014]. [Genevès *et al.* 2012] is the first scientific publication about the analysis of cascading style sheets, yet one of the most widely used standard on the web.

1.8 Containment for Graph Queries

We investigate the problem of query containment for the SPARQL language. This problem is defined as determining whether, for any graph, the result of one SPARQL query is included in the result of another query. Query containment is important in many areas, including program analysis, information integration, and query opti-

mization. For instance, if we can prove that two queries are equivalent for any graph (which reduces to two query containment checks), then we can safely substitute one query by another more efficient query version, while preserving the initial semantics of the program.

We address query containment for a fragment of the SPARQL language, under expressive description logic constraints. SPARQL is interpreted over graphs, hence we encode it in a graph logic, specifically the alternation-free fragment of the μ-calculus [Kozen 1983] with converse and nominals [Tanabe *et al.* 2008] interpreted over labeled transition systems.

We show that this logic is powerful enough to deal with query containment for the fragment of SPARQL made of basic and union graph patterns, in the presence of $\mathcal{ALCH}$ schema axioms.

In this logical encoding, RDF graphs become transition systems and queries and schema axioms become μ-calculus formulae. Therefore, SPARQL query containment can be reduced to unsatisfiability in the μ-calculus. This approach has several advantages: it can be used to precisely characterize the expressive power and complexity for fragments of the SPARQL language. In particular, due to the high expressive power of the μ-calculus, such an encoding is useful to characterize not only restrictions but also extensions of the SPARQL language. For example, a benefit of using a μ-calculus encoding is to take advantage of fixpoints and modalities for encoding recursion. These operators allow to deal with natural extensions of SPARQL such as path queries [Alkhateeb *et al.* 2009] or queries modulo RDF Schema. Finally, another advantage of such an approach is that the considered logic admits exponential time decision procedures that can be implemented in practice [Tanabe *et al.* 2005, Genevès *et al.* 2007, Tanabe *et al.* 2008]. This study allows to exploit these advantages.

In order to experimentally assess implementation strengths and limitations in this setting, we provide a first SPARQL containment test benchmark. It has been designed with respect to both the capabilities of existing solvers and the study of typical queries. Some solvers support optional constructs and cycles, while other solvers support projection, union of conjunctive queries and RDF Schemas. No solver currently supports all these features. The study of query demographics on DBPedia logs shows that the vast majority of queries are acyclic and a significant part of them uses union or projection. We thus test available solvers on their domain of applicability on three different benchmark suites. We report on the experimental results, and discuss to which extent, in spite of its complexity, SPARQL query containment is practicable for acyclic queries.

These works, presented in Chapter 5, were published in AAAI'12 [Chekol *et al.* 2012b] and ISWC'13 [Chekol *et al.* 2013]. Closely related results were published in DBPL'11 [Chekol *et al.* 2011] and IJCAR'12 [Chekol *et al.* 2012a]. A journal article is in preparation.

1.9 Pointers to Other Related Results

This document only presents an excerpt from the the results that I have obtained since December 2006. I have obtained other results, including closely related ones, that are not detailed in this document, but whose publications can be found online. In particular, among the most closely related results, I have obtained results on the extension of tree logics with counting constraints. These results, published at IJCAI'11 [Barcenas *et al.* 2011], were obtained in collaboration with Everardo Bárcenas Patiño, that I supervised as a PhD student, in collaboration with Nabil Layaïda and Alan Schmitt. I also built a prototype of an integrated development environment equipped with semantic static analyses. These results were obtained in collaboration with Lorena Gonçalves de Alcântara e Freitas that I supervised as a master intern, with Manh-Toan Nguyen that I supervised as a research engineer, and with Nabil Layaïda. These results were published in ICSE'10 [Genevès & Layaïda 2010], ICSE'11 [Genevès & Layaïda 2011] and TOIT'14 [Genevès & Layaïda 2014a].

Chapter 2

Evolution of Types

Contents

2.1 Introduction 12
2.2 Analysis Framework 13
2.3 Logical Setting 16
2.4 Analysis Predicates 24
2.5 Impact of Standards' Evolution on Valid Documents 28
2.6 Impact on Queries 36
2.7 System Implementation 38
2.8 Related Work 38
2.9 Conclusion 40

Abstract

In this chapter, the problem of XML Schema evolution is considered. In the ever-changing context of the web, XML schemas continuously change in order to cope with the natural evolution of entities they describe. Schema changes have important consequences. First, existing documents valid with respect to the original schema are no longer guaranteed to fulfill the constraints described by the evolved schema. Second, the evolution also impacts programs manipulating documents whose structure is described by the original schema.

We propose a unifying framework for determining the effects of XML Schema evolution both on the validity of documents and on queries. The system is very powerful in analyzing various scenarios in which forward/backward compatibility of schemas is broken, and in which the result of a query may not be anymore what was expected. Specifically, the system offers a predicate language which allows one to formulate properties related to schema evolution. The system then relies on exact reasoning techniques to perform a fine-grained analysis. This yields either a formal proof of the property or a counter-example that can be used for debugging purposes. The system has been fully implemented and tested with real-world use cases, in particular with the main standard document formats used on the web, as defined by W3C. The system identifies precisely compatibility relations between document formats. In case these relations do not hold, the system can identify queries that must be reformulated in order to produce the expected results across successive schema versions.

2.1 Introduction

XML is now commonplace on the web and in many information systems where it is used for representing all kinds of information resources, ranging from simple text documents such as RSS or Atom feeds to highly structured databases. In these dynamic environments, not only data are changing steadily but their schemas also get modified to cope with the evolution of the real world entities they describe.

Schema changes raise the issue of data consistency. Existing documents and data that were valid with a certain version of a schema may become invalid on a new version of the schema (forward incompatibility). Conversely, new documents created with the latest version of a schema may be invalid on some previous versions (backward incompatibility). In particular, there are two ways commonly used in the design of schemas. One consists in under constraining the schema in the earlier versions when the design is not completely stable and then constraining it in future versions progressively. The other way is more conservative and consists in constraining the schema first and then relaxing the constraints progressively. If we leave aside new elements and attributes introduced between two successive versions of a schema, this is particularly true for new combinations of elements (content models) added or restricted through regular expressions in W3C Document formats recommendations (see Section 2.5).

In addition, schemas may be written in different languages, such as DTD, XML Schema, or Relax-NG, to name only the most popular ones. And it is common practice to describe the same structure, or new versions of a structure, in different schema languages. Document formats developed by W3C provide a variety of examples: XHTML 1.0 has both DTDs and XML Schemas, while XHTML 2.0 has a Relax-NG definition; the schema for SVG Tiny 1.1 is a DTD, while version 1.2 is written in Relax-NG; MathML 1.01 has a DTD, MathML 2.0 has both a DTD and an XML Schema, and MathML 3.0 is developed with a Relax-NG schema and also published with a DTD and an XML Schema. An issue then is to make sure that schemas written in different languages are equivalent, *i.e.* they describe the same structure, possibly with some differences due to the expressivity of the language [Murata *et al.* 2005]. Another issue is to clearly identify the differences between two versions of the same schema expressed in different languages. Moreover, the issues of forward and backward compatibility of instances obviously remain when schema languages change from a version to another.

Validation, and then compatibility, is not the only purpose of a schema. Validation is usually the first step for safe processing of documents and data. It makes sure that documents and data are structured as expected and can then be processed safely. The next step is to actually access and select the various parts to be handled in each phase of an application. For this, query languages play a key role. As an example, when transforming a document with XSL, XPath queries are paramount to locate in the original document the data to be produced in the transformed document.

Queries are affected by schema evolutions. The structures they return may change depending on the version of the schema used by a document. When changing schema, a query may return nothing, or something different from what was expected, and obviously further processing based on this query is at risk.

These observations highlight the need for evaluating precisely and safely the impact of schema evolutions on existing and future instances of documents and data. They also show that it is important for software engineers to precisely know what parts of a processing chain have to be updated when schemas change. In this chapter we focus on the XPath query language which is used in many situations while processing XML documents and data. The XSL transformation language was already mentioned, but XPath is also present in XLink and XQuery for instance.

A part of this work concerning the impact of schema changes on XPath queries was presented at the ACM International Conference on Functional Programming (ICFP), 2009, [Genevès *et al.* 2009]. The present chapter aims at covering the more general issue of schema evolution by taking into account the impact on the validity of documents as well. In particular, we identify criteria for the evolution of standard XML Schemas. We present a framework for checking these criteria with the schemas specifying the main standard documents formats used on the web, as defined by W3C (see Section 2.5).

Outline

We first introduce the framework from a high-level perspective in Section 2.2: we describe how the whole system is assembled, and which XML schemas and queries are supported. In Section 2.3, we provide a more in-depth understanding of the underlying logic on which the system is built; in particular we explain how XML constructs are mapped to this logical representation. Based on this logical encodings, Section 2.4 introduces a predicate language specifically designed for assessing the impact of schema evolutions. The following sections respectively focus on applying the framework for studying the impact of schema evolutions on the validity of documents (Section 2.5) and on queries (Section 2.6). The full implementation of the system is presented in Section 2.7. Finally, we discuss related work in Section 2.8 before concluding in Section 2.9.

2.2 Analysis Framework

The main contribution of this chapter is a unifying framework that allows the automatic verification of properties related to XML schema evolution and its impact on the validity of documents and on queries. In particular, it offers the possibility of checking fine-grained properties of the behavior of queries with respect to successive versions of a given schema. The system can be used for checking relations between schemas and whether schema evolutions require a particular query to be updated. Whenever schema evolutions may induce query malfunctions, the system is able to generate annotated XML documents that exemplify bugs, with the goal of helping the programmer to understand and properly overcome undesired effects of schema evolutions.

The system relies on a predicate language (presented in Section 2.4) specifically designed for studying schema and query compatibility issues when schemas evolve. Specifically, predicates allow characterizing in a precise manner nodes subject to evolution. For instance, predicates allow to distinguish new nodes selected by the query

after a schema change from new nodes that appear in the modified schema. Predicates also allow to describe nodes that appear in new regions of a schema compared to its original version, or even in a new context described by a particular XPath expression. Predicates, together with the composition language provided in the system allow to express and analyze complex settings.

The system has been fully implemented [Genevès & Layaïda 2014b] and is outlined in Figure 2.1. It is composed of a parser for reading the text file description of the problem (which in turn uses specific parsers for schemas, queries, logical formulas, and predicates), compilers for translating schemas and queries into their logical representations, a solver for checking satisfiability of logical formulas, and a counter example XML tree generator (described in [Genevès *et al.* 2014]).

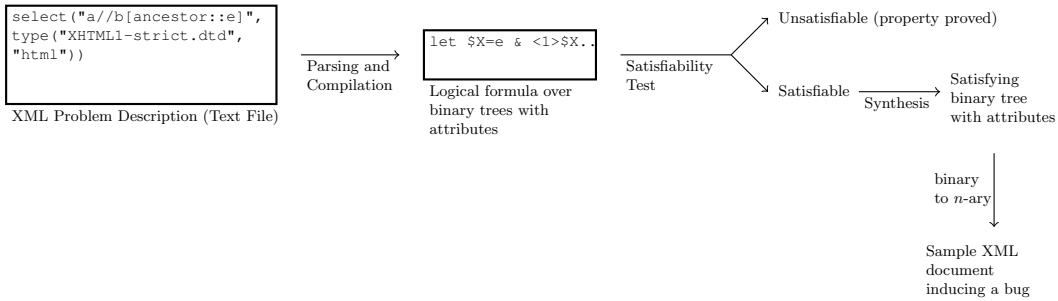


Figure 2.1: Framework Overview.

We first introduce the data model we consider for XML documents, schemas and queries.

2.2.1 XML Trees with Attributes

An XML document is considered as a finite tree of unbounded depth and arity, with two kinds of nodes respectively named elements and attributes. In such a tree, an element may have any number of children elements, and may carry zero, one or more attributes. Attributes are leaves. Elements are ordered whereas attributes are not, as illustrated on Figure 2.4. In this chapter, we focus on the nested structure of elements and attributes, and ignore XML data values.

2.2.2 Type Constraints

Our tree type expressions capture most of the schemas in use today either written using DTD, XML Schema, Relax NG, etc. Users may thus define constraints over XML documents with the language of their choice, and, more importantly, they may refer to most existing schemas for use with the system. Instead of having one parser/compiler per schema language, we rely on a common intermediate language in which all these languages are compiled. For the intermediate language we consider the standard class of regular tree grammars, commonly found in the literature [Hosoya *et al.* 2005], to which we have added the support of constraints over XML attributes (whose efficiency

is further discussed in section 2.3.3). In terms of expressive power, regular tree grammars support constraints over trees which are more expressive than local tree grammars (DTDs) and single-type tree grammars (XML schemas), capturing exactly the class of Relax NG schemas, and, more fundamentally finite tree automata (see [Murata *et al.* 2005] for a formal characterization of the respective expressive power of these languages). In practice, we have implemented parsers that produce this intermediate representation from a given DTD, XML Schema, or Relax NG schema. We have implemented one compiler from this representation into the logic. An advantage of this approach is that it is extensible: it is easy to know the supported features since (1) the intermediate language is well-characterized and made explicit, and (2) extending the system with new schema languages is easy since one does not need to implement new compilers into the logic (and prove soundness, completeness and polynomial-time translation), but rather simply express the new considered constraints in the intermediate language.

Specifically, our unifying internal representation for tree grammars is made of regular tree type expressions, extended with constraints over attributes. Assuming a set of variables ranged over by x, we define a tree type expression as follows:

τ ::=		tree type expression
	$\emptyset$	empty set
	$\texttt{()}$	empty sequence
	$\tau \mid \tau$	disjunction
	τ, τ	concatenation
	$l\,\texttt{(}a\texttt{)}\,\texttt{[}\tau\texttt{]}$	element definition
	x	variable
	$\texttt{let}\ \overline{x = \tau}\ \texttt{in}\ \tau$	binder

The `let` construct allows binding one or more variables to associated formulas. Since several variables can be bound at a time, the notation $\overline{x = \tau}$ is used for denoting a vector of variable bindings (possibly with mutual recursion).

We impose a usual restriction on the recursive use of variables: we allow unguarded (*i.e.* not enclosed by a label) recursive uses of variables, but restrict them to tail positions[1]. With that restriction, tree types expressions define regular tree languages. In addition, an element definition may involve simple attribute expressions that describe

[1]For instance, "$\texttt{let}\ x = l\,\texttt{(}a\texttt{)}\,\texttt{[}\tau\texttt{]}, x \mid \texttt{()}\ \texttt{in}\ x$" is allowed.

which attributes the defined element may (or may not) carry:

a	$::=$		attribute expression
		$()$	empty list
		$list \mid a$	disjunction
$list$	$::=$		attribute list
		$list, list$	commutative concatenation
		$l?$	optional attribute
		l	required attribute
		$\neg l$	prohibited attribute

We use the usual semantics of regular tree types found in [Hosoya *et al.* 2005] and [Genevès *et al.* 2007].

2.2.3 Queries

The set of XPath expressions we consider is given by the syntax shown on Figure 2.2. The semantics of XPath expressions is described in [Clark & DeRose 1999], and more formally in [Genevès *et al.* 2007]. We observed that, in practice, many XPath expressions contain syntactic sugars that can also fit into this fragment. Figure 2.3 presents how our XPath parser rewrites some commonly found XPath patterns into the fragment of Figure 2.2, where the notation $(axis{::}nt)^k$ stands for the composition of k successive path steps of the same form: $\underbrace{axis{::}nt/.../axis{::}nt}_{k \text{ steps}}$.

The next Section presents the logic underlying the predicate language.

2.3 Logical Setting

It is well-known that there exist bijective encodings between unranked trees (trees of unbounded arity) and binary trees [Thomas 1990]. Owing to these encodings binary trees may be used instead of unranked trees without loss of generality. In the sequel, we rely on a simple "first-child & next-sibling" encoding of unranked trees. In this encoding, the first child of an element node is preserved in the binary tree representation, whereas siblings of this node are appended as right successors in the binary representation. Attributes are left unchanged by this encoding. For instance, Figure 2.5 presents how the sample tree of Figure 2.4 is mapped.

The logic we introduce below, used as the core of our framework, operates on such binary trees with attributes.

2.3.1 Logical Formulas

The concrete syntax of logical formulas is shown on Figure 2.6, where the meta-syntax $\langle X \rangle^{\oplus}$ means one or more occurences of X separated by commas. The user can directly encode formulas with this syntax in text files to be used with the system [Genevès &

query	::=		
		$/path$	absolute path
		$path$	relative path
		$query \mid query$	union
		$query \cap query$	intersection
path	::=		
		$path/path$	path composition
		$path[qualifier]$	qualified path
		$axis{::}nt$	step
qualifier	::=		
		$qualifier$ and $qualifier$	conjunction
		$qualifier$ or $qualifier$	disjunction
		not($qualifier$)	negation
		$path$	path
		$path/@nt$	attribute path
		$@nt$	attribute step
nt	::=		node test
		σ	node label
		$*$	any node label
axis	::=		tree navigation axis
		self | child | parent	
		descendant | ancestor	
		descendant-or-self	
		ancestor-or-self	
		following-sibling	
		preceding-sibling	
		following | preceding	

Figure 2.2: XPath Expressions.

Layaïda 2014b]. This concrete syntax is used as a single unifying notation throughout all the chapter.

The semantics of logical formulas corresponds to the classical semantics of a μ-calculus interpreted over finite tree structures. A formula is satisfiable iff there exists a finite binary tree with attributes for which the formula holds at some node. This is formally defined in [Genevès *et al.* 2007], and we review it informally below through a series of examples.

There is a difference between an element name and an atomic proposition[2]: an element has one and only one element name, whereas it can satisfy multiple atomic propositions. We use atomic propositions to attach specific information to tree nodes, not related to their XML labeling. For example, the start context (a reserved atomic proposition) is used to mark the starting context nodes for evaluating XPath expressions.

[2]In practice, an atomic proposition must start with a "_".

$$
\begin{aligned}
nt[\text{position}() = 1] &\rightsquigarrow nt[\text{not}(\text{preceding-sibling}::nt)] \\
nt[\text{position}() = \text{last}()] &\rightsquigarrow nt[\text{not}(\text{following-sibling}::nt)] \\
nt[\text{position}() = \underbrace{k}_{k>1}] &\rightsquigarrow nt[(\text{preceding-sibling}::nt)^{k-1}] \\
\text{count}(path) = 0 &\rightsquigarrow \text{not}(path) \\
\text{count}(path) > 0 &\rightsquigarrow path \\
\text{count}(nt) > \underbrace{k}_{k>0} &\rightsquigarrow nt/(\text{following-sibling}::nt)^{k}
\end{aligned}
$$

$$
\begin{aligned}
&\text{preceding-sibling}::*[\text{position}() = \text{last}() \text{ and } qualifier] \\
&\qquad \rightsquigarrow \text{preceding-sibling}::*[\text{not}(\text{preceding-sibling}::*) \text{ and } qualifier]
\end{aligned}
$$

Figure 2.3: Syntactic Sugars and their Rewritings.

```
<r c="␣" a="␣" b="␣">
 <s d="␣">
  <v/><w/><x e="␣"/>
 </s>
 <t/>
 <u/>
</r>
```

XML Notation

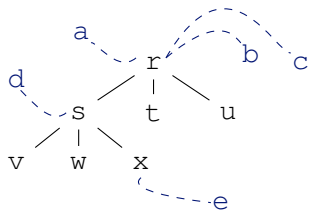


Figure 2.4: Sample XML Tree with Attributes.

The logic uses modalities for navigating in binary trees. A modality $<p>\varphi$ can be read as follows: "there exists a successor node by program p such that φ holds at this successor". As shown on Figure 2.6, a program p is simply one of the four basic programs $\{1, 2, -1, -2\}$. Program 1 allows navigating from a node down to its first successor, and program 2 allows navigating from a node down to its second successor. The logic also features converse programs −1 and −2 for navigating upward in binary trees, respectively from the first successor to its parent and from the second successor to its previous sibling. Table on Figure 2.7 gives some simple formulas using modalities for navigating in binary trees, together with sample satisfying trees, in binary and unranked tree representations.

The logic allows expressing recursion in trees through the recursive binder. For example the recursive formula:

```
let $X = b | <2>$X in $X
```

means that either the current node is named b or there is a sibling of the current node which is named b. For this purpose, the variable $X is bound to the subformula b

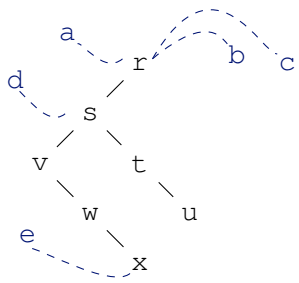


Figure 2.5: Binary Encoding of Tree of Figure 2.4.

`| <2>$X` which contains an occurence of `$X` (therefore defining the recursion). The scope of this binding is the subformula that follows the "`in`" symbol of the formula, that is `$X`. The entire formula can thus be seen as a compact recursive notation for a infinitely nested formula of the form:

```
b | <2>(b | <2>(b | <2>(...)))
```

Recursion allows expressing global properties. For instance, the recursive formula:

```
~ let $X = a | <1>$X | <2>$X in $X
```

expresses the absence of nodes named `a` in the whole subtree of the current node (including the current node). Furthermore, the fixpoint operator makes possible to bind several variables at a time, which is specifically useful for expressing mutual recursion. For example, the mutually recursive formula:

```
let
  $X = (a & <2>$Y) | <1>$X | <2>$X,
  $Y = b | <2>$Y
in $X
```

asserts that there is a node somewhere in the subtree such that this node is named `a` and it has at least one sibling which is named `b`. Binding several variables at a time provides a very expressive yet succinct notation for expressing mutually recursive structural patterns (that are common in XML Schemas, for instance).

From a theoretical perspective, the recursive binder `let $X` $= \varphi$ `in` φ corresponds to the fixpoint operators of the μ-calculus. It is shown in [Genevès *et al.* 2007] that the least fixpoint and the greatest fixpoint operators of the μ-calculus coincide over finite tree structures, for a restricted class of formulas called *cycle-free* formulas.

2.3.2 Queries

The logic is expressive enough to capture the set of XPath expressions presented in Section 2.2.3. For example, Figure 2.8 illustrates how the sample XPath expression:

φ	::=		formula
		`T`	true
		`F`	false
		l	element name
		p	atomic proposition
		`#`	start context
		φ `|` φ	disjunction
		φ `&` φ	conjunction
		φ `=>` φ	implication
		φ `<=>` φ	equivalence
		`(`φ`)`	parenthesized formula
		`~`φ	negation
		`<`p`>`φ	existential modality
		`<`l`>T`	attribute named l
		`$X`	variable
		`let` $\langle$`$X` $= \varphi\rangle^{\oplus}$ `in` φ	binder for recursion
		predicate	predicate (See Section 2.4)
p	::=		program inside modalities
		`1`	first child
		`2`	next sibling
		`-1`	parent
		`-2`	previous sibling

Figure 2.6: Concrete Syntax of Formulas.

```
child::r[child::w/@att]
```

is expressed in the logic. From a given context in an XML document, this expression selects all `r` child nodes which have at least one `w` child with an attribute `att`. Figure 2.8 shows how it is expressed in the logic, on the binary tree representation. The formula holds for `r` nodes which are selected by the expression. The first part of the formula, φ, corresponds to the step `child::r` which selects candidates `r` nodes. The second part, ψ, navigates downward in the subtrees of these candidate nodes to verify that they have at least one immediate `w` child with an attribute `att`.

This example illustrates the need for converse programs inside modalities. The translated XPath expression only uses forward axes (child and attribute), nevertheless both forward and backward modalities are required for its logical translation. Without converse programs we would have been unable to differentiate selected nodes from nodes whose existence is simply tested. More generally, properties must often be stated on both the ancestors and the descendants of the selected node. Equipping the logic with both forward and converse programs is therefore crucial. Logics without converse programs may only be used for solving XPath emptiness but cannot be used for solving other decision problems such as containment efficiently.

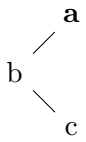
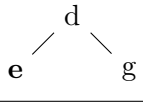
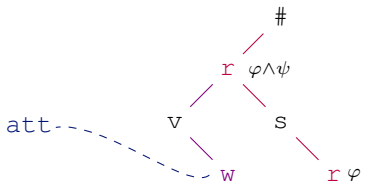
Sample Formula	Tree	XML
`a & <1>b`		`<a><b/></a>`
`a & <1>(b & <2>c)`		`<a><b/><c/></a>`
`e & <-1>(d & <2>g)`		`<d><e/></d><g/>`
`f & <-2>(g & ~<2>T)`	none	none

Figure 2.7: Sample Formulas and Satisfying Trees.



Translated Query: `child::r[child::w/@att]`

Translation:

$$\underbrace{\texttt{r \& (let \$X=<-1>\# | <-2>\$X)}}_{\varphi}\ \underbrace{\texttt{\& <1>let \$Y=w \& <att>T | <2>\$Y}}_{\psi}$$

Figure 2.8: XPath Translation Example.

A systematic translation of XPath expressions into the logic is given in [Genevès *et al.* 2007]. In this chapter, we extended it to deal with attributes. We implemented a compiler that takes any expression of the fragment of Figure 2.2 and computes its logical translation. With the help of this compiler, we extend the syntax of logical formulas with a logical predicate `select("`*query*`",`φ`)`. This predicate compiles the XPath expression *query* given as parameter into the logic, starting from a context that satisfies φ. The XPath expression to be given as parameter must match the syntax of the XPath fragment shown on Figure 2.2 (or Figure 2.3). In a similar manner, we introduce the predicate `exists("`*query*`",`φ`)` which tests the existence of *query* from a context satisfying φ, in a qualifier-like manner (without moving to its result). Additionally, the predicate `select("`*query*`")` is introduced as a shortcut for `select("`*query*`",#)`, where # simply marks the initial context node of the XPath expression[3]. The predicate `exists("`*query*`")` is a shortcut for `exists("`*query*`",T)`. These syntactic extensions

[3]This mark is especially useful for comparing two or more XPath expressions from the same context.

of the logic allow the user to easily embed XPath expressions and formulate decision problems out of them (like *e.g.* containment or any other boolean combination). In the next sections we explain how the framework allows combining queries with schema information for formulating problems.

2.3.3 Tree Types

Tree type expressions are compiled into the logic in two steps: the first stage translates them into binary tree type expressions, and the second step actually compiles this intermediate representation into the logic. The translation procedure from tree type expressions to binary tree type expressions is well-known and detailed in [Genevès 2006]. The syntax of output expressions follows:

τ ::=		binary tree type expression
	$\emptyset$	empty set
	$\texttt{()}$	empty tree
	$\tau \mid \tau$	disjunction
	$l\,\texttt{(}a\texttt{)}\,\texttt{[}x, x\texttt{]}$	element definition
	$\texttt{let}\ \overline{x = \tau}\ \texttt{in}\ \tau$	binder

Attribute expressions are not concerned by this transformation to binary form: they are simply attached, unchanged, to new (binary) element definitions. Finally, binary tree type expressions are compiled into the logic. This translation step was introduced and proven correct in [Genevès *et al.* 2007]. Originally, the translation takes a tree type expression τ and returns the corresponding logical formula. Here, we extend it slightly but crucially: the logical translation of an expression τ is given by the function $\mathrm{tr}(\tau)^{\psi}_{\varphi}$ defined below, that takes additional arguments φ and ψ:

$$
\begin{aligned}
\mathrm{tr}(\tau)^{\psi}_{\varphi} &\stackrel{\text{def}}{=} \mathtt{F} \quad \text{for} \quad \tau = \emptyset, \texttt{()} \\
\mathrm{tr}(\tau_1 \mid \tau_2)^{\psi}_{\varphi} &\stackrel{\text{def}}{=} \mathrm{tr}(\tau_1)^{\psi}_{\varphi} \ \texttt{|}\ \mathrm{tr}(\tau_2)^{\psi}_{\varphi} \\
\mathrm{tr}(l\,\texttt{(}a\texttt{)}\,\texttt{[}x_1, x_2\texttt{]})^{\psi}_{\varphi} &\stackrel{\text{def}}{=} (l\ \texttt{\&}\ \varphi\ \texttt{\&}\ \mathrm{tra}(a)\ \texttt{\&}\ s_1(x_1)\ \texttt{\&}\ s_2(x_2))\ \texttt{|}\ \psi \\
\mathrm{tr}(\texttt{let}\ \overline{x_i = \tau_i}\ \texttt{in}\ \tau)^{\psi}_{\varphi} &\stackrel{\text{def}}{=} \texttt{let}\ \overline{\texttt{\$X}_i = \mathrm{tr}(\tau_i)^{\psi}_{\varphi}}\ \texttt{in}\ \mathrm{tr}(\tau)^{\psi}_{\varphi}
\end{aligned}
$$

The addition of φ and ψ (respectively in a new conjunction and a new disjunction) is a key element for the definition of predicates in Section 2.4. More precisely, this allows marking type sub-expressions so that they can be distinguished in predicates, as explained in Section 2.3.4. In addition, φ and ψ are either true, false, or simple atomic propositions. Thus, it is worth noticing that their addition does not affect the linear complexity of tree type translation. The function $s_{\cdot}(\cdot)$ describes the type for each successor:

$$
s_p(x) \quad = \quad \begin{cases} \texttt{\~{}<}p\texttt{>T} & \textit{if } x \textit{ is bound to } \texttt{()} \\ \texttt{\~{}<}p\texttt{>T} \ \texttt{|}\ \texttt{<}p\texttt{>\$X} & \textit{if nullable}(x) \\ \texttt{<}p\texttt{>\$X} & \textit{if not nullable}(x) \end{cases}
$$

according to the predicate $nullable(x)$ which indicates whether the type $T \neq$ `()` bound to x contains the empty tree.

The function $\mathrm{tra}(a)$ compiles attribute expressions associated with element definitions as follows:

$$
\begin{aligned}
\mathrm{tra}(\texttt{()}) &\stackrel{\text{def}}{=} \mathrm{notothers}(\texttt{()}) \\
\mathrm{tra}(list \mid a) &\stackrel{\text{def}}{=} \mathrm{tra}(list)\ \texttt{\&}\ \mathrm{notothers}(list) \\
\mathrm{tra}(list, list') &\stackrel{\text{def}}{=} \mathrm{tra}(list)\ \texttt{\&}\ \mathrm{tra}(list') \\
\mathrm{tra}(l\texttt{?}) &\stackrel{\text{def}}{=} l\ \texttt{|}\ \texttt{\~{}}\,l \\
\mathrm{tra}(l) &\stackrel{\text{def}}{=} l \\
\mathrm{tra}(\neg l) &\stackrel{\text{def}}{=} \texttt{\~{}}\,l
\end{aligned}
$$

In usual schemas (*e.g.* DTDs, XML Schemas) when no attribute is specified for a given element, it simply means no attribute is allowed for the defined element. This convention must be explicitly stated in the logic. This is the role of the function "notothers($list$)" which returns the negated disjunction of all attributes not present in $list$. As a result, taking attributes into account comes at an extra-cost. The above translation appends a (potentially very large) formula in which all attributes occur, for each element definition. In practice, a placeholder atomic proposition is inserted until the full set of attributes involved in the problem formulation is known. When the whole formula has been parsed, placeholders are replaced by the conjunction of negated attributes they denote. This extra-cost can be observed in practice, and the system allows two modes of operations: with or without attributes[4]. Nevertheless the system is still capable of handling real world DTDs (such as the DTD of XHTML 1.0 Strict) with attributes. This is due to (1) the limited expressive power of languages such as DTD that do not allow for disjunction over attribute expressions (like "$list \mid a$"); and, more importantly, (2) the satisfiability-testing algorithm which is implemented using symbolic techniques [Genevès *et al.* 2014].

Tree type expressions form the common internal representation for a variety of XML schema definition languages. In practice, the logical translation of a tree type expression τ are obtained directly from a variety of formalisms for defining schemas, including DTD, XML Schema, and Relax NG. For this purpose, the syntax of logical formulas is extended with a predicate `type("·",·)`. The logical translation of an existing schema is returned by `type("`f`",`l`)` where f is a file path to the schema file and l is the element name to be considered as the entry point (root) of the given schema. Any occurence of this predicate will parse the given schema, extract its internal tree type representation τ, compile it into the logic and return the logical formula $\mathrm{tr}(\tau)^{\mathrm{F}}_{\mathrm{T}}$.

[4]The optional argument "-attributes" must be supplied for attributes to be considered.

2.3.4 Type Tagging

A tag (or "color") is introduced in the compilation of schemas with the purpose of marking all node types of a specific schema. A tag is simply a fresh atomic proposition passed as a parameter to the translation of a tree type expression. For example: $\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathrm{xhtml}}$ is the logical translation of τ where each element definition is annotated with the atomic proposition "xhtml". With the help of tags, it becomes possible to refer to the element types in any context. For instance, one may formulate $\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathrm{xhtml}}$ | $\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathrm{smil}}$ for denoting the union of all τ and τ' documents, while keeping a way to distinguish element types; even if some element names are shared by the two type expressions.

Tagging becomes even more useful for characterizing evolutions between successive versions of a single schema. In this setting, we need a way to distinguish nodes allowed by a newer schema version from nodes allowed by an older version. This distinction must not be based only on element names, but also on content models. Assume for instance that τ' is a newer version of schema τ. If we are interested in the set of trees allowed by τ' but not allowed by τ then we may formulate:

$$\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathtt{T}} \ \&\ \tilde{}\ \mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$$

If we now want to check more fine-grained properties, we may rather be interested in the following (tagged) formulation:

$$\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathrm{all}} \ \&\ \tilde{}\ \mathrm{tr}(\tau)^{\tilde{}\,\mathrm{old\_complement}}_{\mathtt{T}}$$

In this manner, we can distinguish elements that were added in τ' and whose names did not occur in τ, from elements whose names already occured in τ but whose content model changed in τ', for instance.

In practice, a type is tagged using the predicate $\mathtt{type}(\mathtt{"}f\mathtt{"}, l, \varphi, \varphi')$ which parses the specified schema, converts it into its logical representation τ and returns the formula $\mathrm{tr}(\tau)^{\varphi'}_{\varphi}$. This kind of type tagging is useful for studying the consequences of schema updates over queries, as presented in the next sections.

2.4 Analysis Predicates

This section introduces the basic analysis tasks offered to XML application designers for assessing the impact of schema evolutions. In particular, we propose a means for identifying the precise reasons for type mismatches or changes in query results under type constraints.

For this purpose, we build on our query and type expression compilers, and define additional predicates that facilitate the formulation of decision problems at a higher level of abstraction. Specifically, these predicates are introduced as logical macros with the goal of allowing system usage while focusing (only) on the XML-side properties, and keeping underlying logical issues transparent for the user. Ultimately, we regard the set of basic logical formulas (such as modalities and recursive binders) as an assembly

language, into which predicates are translated.

We illustrate this principle with two simple predicates designed for checking backward-compatibility of schemas, and query satisfiability in the presence of a schema.

- The predicate `backward_incompatible`(τ, τ') takes two type expressions as parameters, and assumes τ' is an altered version of τ. This predicate is unsatisfiable iff all instances of τ' are also valid against τ. Any occurrence of this predicate in the input formula will automatically be compiled as $\mathrm{tr}(\tau')^{\mathtt{F}}_{\mathtt{T}}$ `&` $\tilde{}\,\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$.

- The predicate `non_empty("`*query*`"`$,\tau$) takes an XPath expression (with the syntax defined on Figure 2.2) and a type expression as parameters, and is unsatisfiable iff the query always returns an empty set of nodes when evaluated on an XML document valid against τ. This predicate compiles into `select("`*query*`"`$,\mathrm{tr}(\tau)^{\mathtt{F}}_{\mathtt{T}}$ `& #`) where the top-level predicate `select("`*query*`"`$,\varphi$) compiles the XPath expression *query* into the logic, starting from a context that satisfies φ, as explained in Section 2.3.2. This can be used to check whether the modification of the schema does not contradict any part of the query.

Notice that the predicate `non_empty("`*query*`"`$,\tau$) can be used for checking whether a query that is valid[5] against a schema remains valid with an updated version of a schema. In other terms, this predicate allows determining whether a query that must always return a non-empty result (whatever the tree on which it is evaluated) keeps verifying the same property with a new version of a schema.

A second, more-elaborate, class of predicates allows formulating problems that combine both a query *query* and two type expressions τ, τ' (where τ' is assumed to be a evolved version of τ):

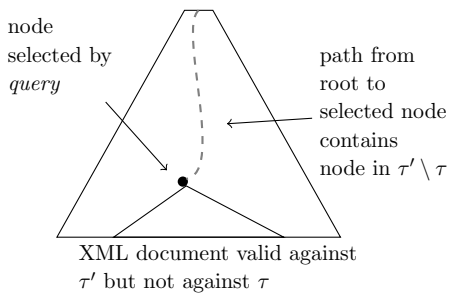
- `new_element_name("`*query*`"`$,\tau,\tau'$) is satisfied iff the query *query* selects elements whose names did not occur at all in τ. This is especially useful for queries whose last navigation step contains a "$\star$" node test and may thus select unexpected elements. This predicate is compiled into:

  $$\tilde{}\,\mathtt{element}(\tau)\ \mathtt{\&}\ \mathtt{select}(\mathtt{"}\mathit{query}\mathtt{"}, \mathrm{tr}(\tau')^{\mathtt{F}}_{\mathtt{T}})$$

  where `element`(τ) is another predicate that builds the disjunction of all element names occuring in τ. In a similar manner, the predicate `attribute`(φ) builds the logical disjunction of all attribute names used in φ.

- `new_region("`*query*`"`$,\tau,\tau'$) is satisfied iff the query *query* selects elements whose names already occurred in τ, but such that these nodes now occur in a new context in τ'. In this setting, the path from the root of the document to a node selected by the XPath expression *query* contains a node whose type is defined in τ' but not in τ as illustrated below:

[5] We say that a query is *valid* iff its negation is unsatisfiable.

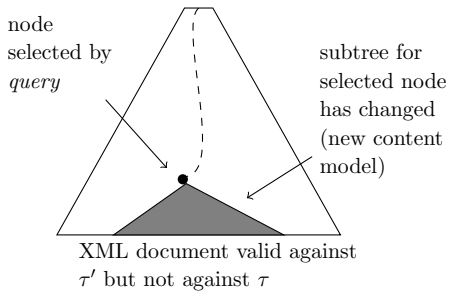


The predicate `new_region("query"`$,\tau,\tau')$ is logically defined as follows:

$$
\begin{aligned}
&\texttt{new\_region("}\mathit{query}\texttt{"},\tau,\tau') \stackrel{\text{def}}{=} \\
&\qquad \texttt{select("}\mathit{query}\texttt{"}, \mathrm{tr}(\tau')^{\texttt{F}}_{\texttt{\_all}} \ \texttt{\&}\,\tilde{}\,\mathrm{tr}(\tau)^{\tilde{}\texttt{\_old\_complement}}_{\texttt{T}}) \\
&\qquad\qquad \texttt{\&}\,\tilde{}\,\texttt{added\_element}(\tau,\tau') \\
&\qquad\qquad \texttt{\&}\ \texttt{ancestor(\_old\_complement)} \\
&\qquad\qquad \texttt{\&}\,\tilde{}\,\texttt{descendant(\_old\_complement)} \\
&\qquad\qquad \texttt{\&}\,\tilde{}\,\texttt{following(\_old\_complement)} \\
&\qquad\qquad \texttt{\&}\,\tilde{}\,\texttt{preceding(\_old\_complement)}
\end{aligned}
$$

The previous definition heavily relies on the partition of tree nodes defined by XPath axes, as illustrated by Figure 2.9. The definition of `new_region("query"`$,\tau,\tau')$ uses an auxiliary predicate `added_element`(τ,τ') that builds the disjunction of all element names defined in τ' but not in τ (or in other terms, elements that were added in τ'). In a similar manner, the predicate `added_attribute`(φ,φ') builds the disjunction of all attribute names defined in τ' but not in τ. The predicate `new_region("query"`$,\tau,\tau')$ is useful for checking whether a query selects a different set of nodes with τ' than with τ because selected elements may occur in new regions of the document due to changes brought by τ'.

- `new_content("query"`$,\tau,\tau')$ is satisfied iff the query *query* selects elements whose names were already defined in τ, but whose content model has changed due to evolutions brought by τ', as illustrated below:



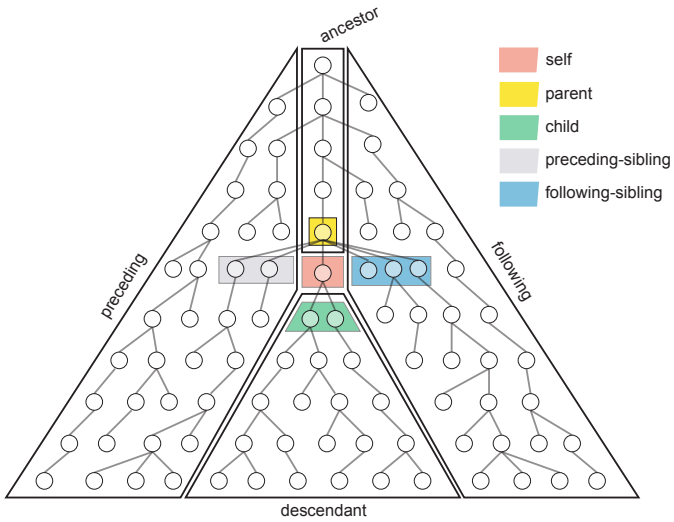


Figure 2.9: XPath axes: partition of tree nodes.

The definition of $\texttt{new\_content}(\texttt{"}\mathit{query}\texttt{"}, \tau, \tau')$ follows:

$$
\begin{aligned}
&\texttt{new\_content}(\texttt{"}\mathit{query}\texttt{"}, \tau, \tau') \stackrel{\text{def}}{=} \\
&\qquad \texttt{select}(\texttt{"}\mathit{query}\texttt{"}, \mathrm{tr}(\tau')^{\texttt{F}}_{\texttt{\_all}} \ \texttt{\&}\ \texttt{\~{}}\,\mathrm{tr}(\tau)^{\texttt{\~{}}\texttt{\_old\_complement}}_{\texttt{T}}) \\
&\qquad\qquad \texttt{\&}\ \texttt{\~{}}\,\texttt{added\_element}(\tau, \tau') \\
&\qquad\qquad \texttt{\&}\ \texttt{\~{}}\,\texttt{ancestor}(\texttt{added\_element}(\tau, \tau')) \\
&\qquad\qquad \texttt{\&}\ \texttt{descendant}(\texttt{\_old\_complement}) \\
&\qquad\qquad \texttt{\&}\ \texttt{\~{}}\,\texttt{following}(\texttt{\_old\_complement}) \\
&\qquad\qquad \texttt{\&}\ \texttt{\~{}}\,\texttt{preceding}(\texttt{\_old\_complement})
\end{aligned}
$$

The predicate $\texttt{new\_content}(\texttt{"}\mathit{query}\texttt{"}, \tau, \tau')$ can be used for ensuring that XPath expressions will not return nodes with a possibly new content model that may cause problems. For instance, this allows checking whether an XPath expression whose resulting node set is converted to a string value (as in, *e.g.* XPath expressions used in XSLT "value-of" instructions) is affected by the changes from τ to τ'.

- $\texttt{new\_sibling}(\texttt{"}\mathit{query}\texttt{"}, \tau, \tau')$ is satisfied iff the query *query* selects elements whose names already occurred in τ, but such that they now occur with new potential siblings due to τ'. The notion of context, here, is extended to be not only the chain of ancestors from the selected node to the root but also the set of previous and following siblings of the selected node.

The previously defined predicates can be used to help the programmer identify precisely how type constraint evolutions affect queries. They can even be combined with usual

logical connectives to formulate even more sophisticated problems. For example, let us define the predicate `exclude`(φ) which is satisfiable iff there is no node that satisfies φ in the whole tree. This predicate can be used for excluding specific element names or even nodes selected by a given XPath expression. It is defined as follows:

$$\texttt{exclude}(\varphi) \stackrel{\text{def}}{=} \texttt{\~{}ancestor-or-self}(\texttt{descendant-or-self}(\varphi))$$

This predicate can also be used for checking properties in an iterative manner, refining the property to be tested at each step. It can also be used for verifying fine-grained properties. For instance, one may check whether τ' defines the same set of trees as τ modulo new element names that were added in τ' with the following formulation:

$$\texttt{\~{}}(\tau \texttt{ <=> } \tau') \texttt{ \& exclude}(\texttt{added\_element}(\tau, \tau'))$$

This allows identifying that, during the type evolution from τ to τ', the query results change has not been caused by the type extension but by new compositions of nodes from the older type.

In practice, instead of taking internal tree type representations (as defined in Section 2.2.2) as parameters, most predicates do actually take any logical formula as parameter, or even schema paths as parameters. We believe this facilitates predicates usage and, most notably, how they can be composed together. Figure 2.10 gives the syntax of built-in predicates as they are implemented in the system, where f is a file path to a DTD (.dtd), XML Schema (.xsd), or Relax NG (.rng). In addition of aforementioned predicates, the predicate `descendant`(φ) forces the existence of a node satisfying φ in the subtree, and *predicate-name*($\langle\varphi\rangle^{\oplus}$) is a call to a custom predicate, as explained in the next section.

Custom Predicates

Following the same spirit, users may also define their own custom predicates. The full syntax of XML logical specifications to be used with the system is defined on Figure 2.11, where the meta-syntax $\langle X\rangle^{\oplus}$ means one or more occurrence of X separated by commas. A global problem specification can be any formula (as defined on Figure 2.6), or a list of custom predicate definitions separated by semicolons and followed by a formula. A custom predicate may have parameters that are instanciated with actual formulas when the custom predicate is called (as shown on Figure 2.10). A formula bound to a custom predicate may include calls to other predicates, but not to the currently defined predicate (recursive definitions must be made through the let binder shown on Figure 2.6).

2.5 Impact of Standards' Evolution on Valid Documents

As depicted on Fig. 2.1, the whole system relies on a satisfiability solver for the underlying logic. The main principe of the satisfiability-solver is an exhaustive search for a tree that satisfies the formula. The search relies on a least fixpoint computation

$$
\begin{array}{ll}
\textit{predicate} & ::= \\
& \texttt{select}(\texttt{"}\textit{query}\texttt{"}) \\
& \texttt{select}(\texttt{"}\textit{query}\texttt{"}, \varphi) \\
& \texttt{exists}(\texttt{"}\textit{query}\texttt{"}) \\
& \texttt{exists}(\texttt{"}\textit{query}\texttt{"}, \varphi) \\
& \\
& \texttt{type}(\texttt{"}f\texttt{"}, l) \\
& \texttt{type}(\texttt{"}f\texttt{"}, l, \varphi, \varphi') \\
& \texttt{forward\_incompatible}(\varphi, \varphi') \\
& \texttt{backward\_incompatible}(\varphi, \varphi') \\
& \\
& \texttt{element}(\varphi) \\
& \texttt{attribute}(\varphi) \\
& \texttt{descendant}(\varphi) \\
& \texttt{exclude}(\varphi) \\
& \texttt{added\_element}(\varphi, \varphi') \\
& \texttt{added\_attribute}(\varphi, \varphi') \\
& \\
& \texttt{non\_empty}(\texttt{"}\textit{query}\texttt{"}, \varphi) \\
& \texttt{new\_element\_name}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l) \\
& \texttt{new\_region}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l) \\
& \texttt{new\_sibling}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l) \\
& \texttt{new\_content}(\texttt{"}\textit{query}\texttt{"}, \texttt{"}f\texttt{"}, \texttt{"}f'\texttt{"}, l) \\
& \textit{predicate-name}(\langle\varphi\rangle^{\oplus})
\end{array}
$$

Figure 2.10: Syntax of Predicates for XML Reasoning.

that starts from all possible leaves and attempt to plug every possible parent node at each further step. The algorithm terminates once the initial formula has been found to hold in a given node of the tree. Otherwise, the algorithm terminates when no more parent nodes can be added. The algorithm, as well as proofs of its soundness and completeness, optimal complexity, and implementation techniques are detailed in [Genevès *et al.* 2007].

We have carried out extensive experiments of the system in real world settings, e.g. with popular web schemas such as XHTML, MathML, SVG, SMIL (Table on Figure 2.12 gives details related to their respective sizes). In this section, we show how the tool can be used to analyze different situations where schemas changes have important consequences on the validity of existing documents.

One major role of organizations such as W3C is to contribute to the standardization effort leading to a unique widely accepted set of constraints for a given class of documents. Designing a normative specification is a complex process, which is made even harder by a few important considerations. For example, when a language is designed, one need to take into account how future versions of that language can evolve.

$$
\begin{array}{lll}
\mathit{spec} & ::= & \\
 & \varphi & \text{formula (see Fig. 2.6)} \\
 & \mathit{def}\,;\varphi & \\
\mathit{def} & ::= & \\
 & \mathit{predicate\text{-}name}(\langle l\rangle^{\oplus}) = \varphi' & \text{custom definition} \\
 & \mathit{def}\,;\mathit{def} & \text{list of definitions}
\end{array}
$$

Figure 2.11: Global Syntax for Specifying Problems.

Schema	Variables	Elements	Attributes
XHTML 1.0 basic DTD	71	52	57
XHTML 1.1 basic DTD	89	67	83
MathML 1.01 DTD	137	127	72
MathML 2.0 DTD	194	181	97

Figure 2.12: Sizes of Some Considered Schemas.

For a particular version of a language, not only the schema constraints allowed by that version need to be considered but also how they can be modified in future versions. This allows to address how an implementation of this version should process document variants added by future schema versions.

Specifically, we identify three different properties for a specification:

- *Forward compatibility*: All instances of an older specification should be valid with respect to newer specifications. This ensures that a document can still be processed properly with applications implementing newer specifications.

- *Backward compatibility without added elements/attributes*: New combinations of old elements are not supposed to be introduced in later specifications. Otherwise, an application implementing an older specification will not able to process a document that conforms to some future specification, even if this document does not contain any element or attribute introduced as extensions.

- *Equivalence between schema versions*: A given specification can be expressed in a variety of schema definition languages like DTD, XML Schema, Relax NG. We expect the different schema versions of the same specification to define the same set of documents modulo the expressivity of the schema language [Murata *et al.* 2005].

An XML schema definition (whether normative or not) often evolves over time, as new needs often result in new features usually introduced as new elements and

attributes. However we believe that this normal evolution should not break the three previous properties.

We report below on using the framework for characterizing the evolution of the main standard document formats used on the web, including W3C XHTML, SMIL, SVG and MathML, based on the criteria identified above. This kind of analyses yield important observations on the validity of, potentially, billions of documents.

XHTML Basic

The first test consists in analyzing the relationship (forward and backward compatibility) between XHTML basic 1.0 and XHTML basic 1.1 schemas. In particular, backward compatibility can be checked by the following command:

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
```

Executing the test yields a counter example as the new schema contains new element names. The counter example (shown below) contains a `style` element occurring as a child of `head`, which is not permitted in XHTML basic 1.0:

```
<html>
  <head>
    <title/>
    <style type="_otherV"/>
  </head>
  <body/>
</html>
```

The next step consists in focusing on the relationship between both schemas excluding these new elements. This can be formulated by the following command:

```
backward_incompatible("xhtml-basic10.dtd",
                      "xhtml-basic11.dtd", "html")
& exclude(added_element(
      type("xhtml-basic10.dtd","html"),
      type("xhtml-basic11.dtd", "html")))
```

The result of the test shows a counter example document that proves that XHTML basic 1.1 is not backward compatible with XHTML basic 1.0 even if new elements are not considered. In particular, the content model of the `label` element cannot have an `a` element in XHTML basic 1.0 while it can in XHTML basic 1.1. The counter example produced by the solver is shown below:

```
<html>
  <head>
    <object>
```

```
      <label>
        <a href="...">
          <img/>
        </a>
        <img/>
      </label>
      <param/>
    </object>
    <meta/>
    <title/>
    <base/>
  </head>
  <body/>
</html>

XTML basic 1.0 validity error: element a is not
declared in label list of possible children
```

SMIL

The second test consists in analyzing the relationship (forward and backward compatibility) between several versions of the SMIL standard[6], namely versions 1.0, 2.0, and 3.0. In particular, forward compatibility between 1.0 and 2.0 can be checked by the following command:

```
forward_incompatible("SMIL10.dtd", "SMIL20.dtd", "smil")
```

The result of the test shows a counter example document that proves that there exist valid SMIL 1.0 documents that are not valid anymore with respect to SMIL 2.0. In fact that is because the content model of the `layout` element is defined as `any` in SMIL 1.0, whereas it is more restricted in SMIL 2.0. We observe that introducing `any` is a choice that has important consequences. Indeed, a document that was playable with 1.0 implementations may no longer be playable using 2.0 implementations. The counter example produced by the solver is shown below:

```
<smil>
  <head>
    <layout>
      <meta content="_otherV" name="_otherV"/>
    </layout>
  </head>
</smil>

SMIL 2.0 validity error:
Element layout content does not follow the DTD,
```

[6]The first author was a member of the W3C SMIL working group and a co-author of SMIL 2.0 and 2.1.

```
expecting (region|topLayout|root-layout|regPoint)*,
got (meta)
```

The lesson here is that introducing very permissive content models (like `any`) has to be considered very seriously. Indeed, that means that all future versions of the standard should be at least as permissive. Otherwise, all content produced with earlier (more permissive) versions becomes at risk. Therefore, the initial content model has to be carefully designed in order to avoid such situations.

The following example is even worse. We check forward compatibility between SMIL 2.0 and 3.0:

```
forward_incompatible("SMIL20.dtd",
                          "SMIL30Language.dtd", "smil")
```

We obtain the following counter-example:

```
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
  <body>
    <switch>
      <animateMotion/>
    </switch>
    <a href="..."/>
  </body>
</smil>
```

This document is valid with respect to SMIL 2.0. However it does not validate with respect to SMIL 3.0. That is because the content model for the `switch` element was set to a more restrictive pattern in version 3.0 compared to 2.0, as the following validation error message suggests:

```
SMIL 3.0 validity error :
Element switch content does not follow the DTD,
expecting ((metadata | switch)* , ((((animate | set |
animateMotion | animateColor) , (metadata | switch)*)* ,
(((par | seq | excl | audio | video | animation | text |
 ... switch)*)+)) | (layout , (metadata | switch)*)*)),
got (animateMotion)
```

Now we would like to know if the bug is limited to the occurrence of the `animateMotion` element or whether it is more general. To this end, we progressively exclude elements named `animateMotion`, `set`, `animateColor`, and `animate`, as follows:

```
forward_incompatible("SMIL20.dtd",
                          "SMIL30Language.dtd", "smil")
  & exclude(animateMotion) & exclude(set)
  & exclude(animateColor) & exclude(animate)
```

We still obtain the following counter-example (valid w.r.t SMIL 2.0 but not w.r.t SMIL 3.0), which shows that the forward incompatibility is not limited to the occurence of the previous elements, but rather, to severe limitations of the `switch` content model introduced in 3.0. In other words, `switch` is an element which undermines SMIL forward compatibility.

```
<smil xmlns="http://www.w3.org/2001/SMIL20/Language">
  <body>
    <switch>
      <seq/>
      <area/>
    </switch>
    <switch/>
    <a href="..."/>
  </body>
</smil>
```

SVG

The SVG test consists in analyzing the relationship (forward and backward compatibility) between SVG 1.0 et 1.1. In particular, we examine the different profiles (tiny, basic and full) from 1.0 and compare them to 1.1 schemas. Backward compatibility can be checked by the following command:

```
forward_incompatible("svg10.dtd",
              "svg11-flat-20030114.dtd", "svg")
```

The test is unsatisfiable meaning that SVG 1.1 is formally proven to be forward compatible with SVG 1.0. This is good news as it means that all 1.0 documents will be supported with 1.1 conforming implementations, without any exception. In the case where a 1.0 document does not play with a 1.1 implementation, this indicates a bug in the implementation and not in the SVG specification.

We observe here that the common practice of including a single doctype declaration within a document is questionable, since a document is not only valid w.r.t a given schema but also w.r.t to all future forward-compatible versions. Keeping track of this mapping between a document and several schemas allows the document to be supported by a larger set of implementations.

Similar tests on the SVG 1.1 tiny, basic and full also exhibit good results. This corresponds to the definition of these three profiles as strict subsets of each other. Furthermore, we believe that the use of a modularized version of a schema (as opposed to a complete redefinition) has helped in avoiding compatibility problems.

We now focus on testing the backward compatibility between the SVG basic 1.1 profile and SVG 1.0 profile. The test fails even if new features are left aside:

```
backward_incompatible("svg10.dtd",
            "svg11-basic.dtd", "svg")
```

```
& exclude( added_element(type("svg10.dtd", "svg"),
                         type("svg11-basic.dtd","svg")))
& exclude(switch)
```

This test yields the following counter-example which confirms that there is actually a flaw in the 1.1 specification:

```
<svg>
  <image href="..." width="..." height="...">
    <title/>
    <title/>
  </image>
</svg>
```

as it allows two `title` elements to occur inside an `image` element, which was not allowed in the 1.0.

MathML

We apply a similar investigation approach to MathML 1.0 and its newer version 2.0. We formulate a backward compatibility test without elements that were added in version 2.0. Furthermore, we want to exclude immediate trivial counter-examples involving the use of the `declare` element as well as of the `math` element occuring within the `annotation-xml` element. For this purpose, we use the following formulation:

```
backward_incompatible("mathml.dtd","mathml2.dtd","math")
& exclude( added_element( type("mathml.dtd","math"),
                          type("mathml2.dtd","math")))
& exclude(declare)
& (~descendant(math))
```

that bans the `declare` element from occuring in the whole tree (achieved with the use of the `exclude(declare)` predicate), and prevents the `math` element from ocurring in the root's subtree (owing to the use of the `(~descendant(math))` predicate) The following counter-example is produced:

```
<math>
  <apply>
    <annotation-xml>
      <mprescripts/>
    </annotation-xml>
  </apply>
</math>
```

Such backward incompatibilities suggest that applications cannot simply ignore new elements from newer schemas, as the combination of older elements may evolve significantly from one version to another.

2.6 Impact on Queries

In this section, we report on using the framework in order to evaluate the consequences of schema changes on XPath queries such as the ones found in transformations like the MathML content to presentation conversion [Pietriga 2005].

MathML Content to Presentation Conversion

MathML is an XML format for describing mathematical notations and capturing both its mathematical structure and graphical rendering, also known as Content MathML and Presentation MathML respectively. The structure of a given equation is kept separate from the presentation and the rendering part can be generated from the structure description. This operation is usually carried out using an XSLT transformation that achieves the conversion. In this test series, we focus on the analysis of the queries contained in such a transformation sheet and evaluate the impact of the schema change from MathML 1.0 to MathML 2.0 on these queries.

Most of the queries contained in the transformation represent only a few patterns very similar up to element names. The following three patterns are the most frequently used:

```
Q1:  //apply[*[1][self::eq]]
Q2:  //apply[*[1][self::apply]/inverse]
Q3:  //sin[preceding-sibling::*[position()=last()
             and (self::compose or self::inverse)]]
```

The first test is formulated by the following command:

```
new_region("Q1","mathml.dtd","mathml2.dtd","math")
```

The result of the test shows a counter example document that proves that the query may select nodes in new contexts in MathML 2.0 compared to MathML 1.0. In particular, the query `Q1` selects `apply` elements whose ancestors can be `declare` elements, as indicated on the document produced by the solver[7]:

```
<math xmlns:solver="http://wam.inrialpes.fr/xml"
            solver:context="true">
  <declare>
    <apply solver:target="true">
      <eq/>
    </apply>
    <condition/>
  </declare>
</math>
```

[7]Notice that the solver automatically annotates a pair of nodes related by the query: when the query is evaluated from a node marked with the attribute `solver:context`, the node marked with `solver:target` is selected.

To evaluate the effect of this change, the counter example is filled with content and passed as an input parameter to the transformation. This shows immediately a bug in the transformation as the resulting document is not a MathML 2.0 presentation document. Based on this analysis, we know that the XSLT template associated with the match pattern `Q1` must be updated to cope with MathML evolution from version 1.0 to version 2.0.

The next test consists in evaluating the impact of the MathML type evolution for the query `Q2` while excluding all new elements added in MathML 2.0 from the test. This identifies whether old elements of MathML 1.0 can be composed in MathML 2.0 in a different manner. This can be performed with the following command:

```
new_content("Q2","mathml.dtd","mathml2.dtd","math")
& exclude(added_element(type("mathml.dtd","math"),
                        type("mathml2.dtd", "math")))
```

The test result shows an example document that effectively combines MathML 1.0 elements in a way that was not allowed in MathML 1.0 but permitted in MathML 2.0.

```
<math xmlns:solver="http://wam.inrialpes.fr/xml"
            solver:context="true">
  <apply solver:target="true">
    <apply>
      <inverse/>
    </apply>
    <annotation-xml>
      <math/>
    </annotation-xml>
    <condition/>
  </apply>
</math>
```

Similarly, the last test consists in evaluating the impact of the MathML type evolution for the query `Q3`, excluding all new elements added in MathML 2.0 and counter example documents containing `declare` elements (to avoid trivial counter examples):

```
new_region("Q3","mathml.dtd","mathml2.dtd","math")
& exclude(added_element(type("mathml.dtd","math"),
                        type("mathml2.dtd","math")))
                        & exclude(declare)
```

The counter example document shown below illustrates a case where the `sin` element occurs in a new context.

```
<math xmlns:solver="http://wam.inrialpes.fr/xml"
            solver:context="true">
  <apply>
```

```
    <annotation-xml>
      <math>
        <apply>
          <inverse/>
          <sin solver:target="true"/>
        </apply>
      </math>
    </annotation-xml>
  </apply>
</math>
```

Applying the transformation on previous examples yields documents which are neither MathML 1.0 nor MathML 2.0 valid. As a result, the stylesheet cannot be used safely over documents of the new type without modifications. In addition, the required changes to the stylesheet are not limited to the addition of new templates for MathML 2.0 elements. The templates that deal with the composition of MathML 1.0 elements should be revised as well.

2.7 System Implementation

We have implemented the whole software architecture described in Section 2.2 and illustrated on Figure 2.1. The tool [Genevès & Layaïda 2014b] is available online from:

`http://wam.inrialpes.fr/xml`

All the previous tests were processed in less than 30 seconds on an ordinary laptop computer running Mac OS X. The 30s correspond to the most complex use cases. Most complex means analyzing recursive forward/backward and qualified queries such as `Q3`, under evolution of large and heavily recursive schemas such as XHTML and MathML (large number of type variables, elements and attributes: see Table on Figure 2.12). These are the hardest cases measured in practice with the implementation. Most of other schemas and queries usually found in applications are much simpler than the ones presented in this chapter and will obviously be solved much faster. Given the variety of schemas occurring in practice, we focused on the most complex W3C standard schemas. The full online implementation [Genevès & Layaïda 2014b] allows to run all the tests described in the chapter as well as user-supplied ones. It shows intermediate compilation stages, generated formulae (in particular the translation of schemas into the logic), and reports on the performance of each step of the analysis.

2.8 Related Work

Schema evolution is an important topic and has been extensively explored in the context of relational, object-oriented, and XML databases. Most of the previous work for XML query reformulation is approached through reductions to relational problems [Beyer *et al.* 2005]. This is because schema evolution was considered as a storage

problem where the priority consists in ensuring data consistency across multiple relational schema versions. In such settings, two distinct schemas and an explicit description of the mapping between them are assumed as input. The problem then consists in reformulating a query expressed in terms of one schema into a semantically equivalent query in terms of the other schema: see [Yu & Popa 2005] and more recently [Moon *et al.* 2008] with references thereof.

In addition to the fundamental differences between XML and the relational data model, in the more general case of XML processing, schemas constantly evolve in a distributed, independent, and unpredictable environment. The relations between different schemas are not only unknown but hard to track. In this context, one priority is to help maintaining query consistency during these evolutions, which is still considered as a challenging problem [Sedlar 2005, Rose 2004]. The absence of evolution analysis tools for XML/XPath contrasts with the abundance of tools and methods routinely used in relational databases.

The work found in [Moro *et al.* 2007] discusses the impact of evolving XML schemas on query reformulation. Based on a taxonomy of XML schema changes during their evolution, the authors provide informal – not exact nor systematic – guidelines for writing queries which are less sensitive to schema evolution. In fact, studying query reformulation requires at least the ability to analyze the relationship between queries. For this reason, a closely related work is the problem of determining query containment and satisfiability under type constraints [Benedikt *et al.* 2005, Colazzo *et al.* 2006, Genevès *et al.* 2007]. These static analysis tasks are also notably useful for performing query optimization [Groppe *et al.* 2006].

The works found in [Benedikt *et al.* 2005, Groppe & Groppe 2008] study the complexity of XPath emptiness and containment for various fragments with or without type constraints (see [Benedikt & Koch 2009] and references thereof for a survey). In [Colazzo *et al.* 2004, Colazzo *et al.* 2006], a technique is presented for statically ensuring correctness of paths. The approach deals with emptiness of XPath expressions without reverse axes. The work presented in [Genevès *et al.* 2007] solves the more general problem of containment, including reverse axes.

The main distinctive idea pursued in this chapter is to develop a logical approach for guiding schema and query evolution. In contrast to the previous use of logics for proving properties such as query emptiness or equivalence, the goal here is different in that we seek to provide the necessary tools to produce relevant knowledge when such relations do not hold. From a complexity point-of-view, it is worth noticing that the addition of predicates does not increase complexity for the underlying logic shown in [Genevès *et al.* 2007].

We would also like to emphasize that, to the best of our knowledge, this work is the first to provide precise analyses of XML evolution, that was tested on real life use cases (such as XHTML and MathML types) and complex queries (involving recursive and backward navigation). As a consequence, in this context, analysis tools such as type-checkers [Hosoya & Pierce 2003, Benzaken *et al.* 2003, Møller & Schwartzbach 2005, Gapeyev *et al.* 2006, Castagna & Nguyen 2008] do no match the expressiveness, typing precision, and analysis capabilities of the work presented here.

2.9 Conclusion

In this chapter, we present an application of a unifying logical framework for verifying forward/backward compatibility issues caused by schemas evolution. We provide evidence that such a framework can be successfully used to overcome the obstacles of the analysis of XML schema evolution. This kind of analyses is widely considered as a challenging problem in XML programming. As mentioned earlier, the difficulty is twofold: first it requires dealing with large and complex language constructions such as XML types and XPath queries, and second, it requires modeling and reasoning about evolution of such constructions.

We presented the logical foundations of the framework. We then applied the framework for analyzing two major issues due to schema evolution: first, the consequence on the validity of documents and, second, the impact on queries. The presented system detected several compatibility problems in the main document formats used on the web. The same tool also allows XML designers to identify queries that need reformulation in order to produce the expected results across successive schema versions. With this tool designers can examine precisely the impact of schema changes over queries, therefore facilitating their reformulation.

We gave illustrations of how to use the tool for schema evolution on realistic examples. In particular, we considered typical situations in applications involving evolution of W3C schemas used on the web such as XHTML and MathML. We believe that the tool can be very useful for standard schema writers and maintainers in order to assist them enforce some level of quality assurance on compatibility between versions.

Chapter 3

Functions and Polymorphism

Contents

3.1 Introduction 41
3.2 Semantic Subtyping Framework 46
3.3 Tree Logic Framework 50
3.4 Logical Encoding 52
3.5 Polymorphism: Supporting Type Variables 55
3.6 Implementation and Practical Experiments 62
3.7 Related Work 70
3.8 Conclusion 71

Abstract

We consider a type algebra equipped with recursive, product, function, intersection, union, and complement types together with type variables and implicit universal quantification over them. We consider the subtyping relation recently defined by Castagna and Xu over such type expressions and show how this relation can be decided in EXPTIME, answering an open question. The novelty, originality and strength of our solution reside in introducing a logical modeling for the semantic subtyping framework. We model semantic subtyping in a tree logic and use a satisfiability-testing algorithm in order to decide subtyping. We report on practical experiments made with a full implementation of the system. This provides a powerful polymorphic type system aiming at maintaining full static type-safety of functional programs that manipulate trees, even with higher-order functions, which is particularly useful in the context of XML.

3.1 Introduction

In programming, subtyping represents a notion of safe substitutability: τ being a subtype of τ' means that wherever in the program something of type τ' is used, it is safe to use something of type τ instead. This property has a natural set-theoretic interpretation: the set of values which can safely replace something of type τ is included in the set of values which can safely replace something of type τ'.

The semantic subtyping approach consists of using this set-theoretic property to *define* the subtyping relation, rather than for example an axiomatic definition. Types are given an interpretation as sets and subtyping is defined as inclusion of interpretations.

The XML-centric functional language XDuce [Hosoya & Pierce 2003] uses this semantic approach to define the subtyping relation between datatypes. Datatypes in that language are intended to correspond to XML document types (as described for example by DTDs), i. e. regular tree grammars, and are built using pair construction, union, intersection, negation and recursion. The set-theoretic interpretation of a type is the regular language of trees it describes, so subtyping is inclusion of regular languages. XDuce however does not have higher-order functions, and the type system does not include functional types.

The XDuce type system was extended to include arrow types in the language CDuce [Benzaken *et al.* 2003]. In that language, boolean combinations of types can still be used, and intersections of arrow types are interpreted as the type of overloaded functions (which give a result of a different type depending on the type of their argument). Extending the set-theoretic interpretation of types accordingly, so that subtyping still corresponds to inclusion of interpretations, turns out to be non-trivial and the recipe for managing it is explained in [Frisch *et al.* 2008] — we summarise it, with a slightly different focus than the original paper, in Section 3.2.

More recently, extending the XDuce type algebra with type variables so as to support prenex parametric polymorphism, while keeping the semantic subtyping approach, has been studied in [Hosoya *et al.* 2009]. Again, doing the same in the presence of arrow types was more difficult, and a solution has only been proposed in 2011 by Castagna and Xu [Castagna & Xu 2011].

In both [Frisch *et al.* 2008] and [Castagna & Xu 2011], algorithms used to decide the subtyping relations rely on arrow elimination. It is well known that in a sensible subtyping relation, $\tau_1 \to \tau_2 \leqslant \tau_1' \to \tau_2'$ is equivalent to the conjunction of $\tau_1' \leqslant \tau_1$ and $\tau_2 \leqslant \tau_2'$, so that a subtyping decision problem involving arrows can be reduced to two problems not involving them. It gets more complicated than this example when things like intersections of arrow types are allowed but can still be done. In general, very schematically, the way the algorithms work is by splitting complex types into components and distinguishing cases repeatedly in order to reduce ultimately the problem to a series of elementary comparisons between base types. Because of that, adding new constructs to the type algebra mechanically complicates the algorithm: for example, the algorithm of [Castagna & Xu 2011] behaves like the one of [Frisch *et al.* 2008] for monomorphic types, but contains new rules for variable elimination in various cases depending of where they occur in the type. These additions were not easy to define and obscure the algorithm enough that proving that it terminated in all cases was difficult — it was in fact yet unproven when we first implemented the decision procedure we present here — and that its complexity is still unknown.

An interesting thing to note in these founding works about semantic subtyping is that, while the set-theoretic interpretation of types is used to give some insight and some theoretical backing to the subtyping relation, it does not play as fundamental a

role as we may think in the practical applications — one does not need the semantic subtyping theoretical development to use or even to understand the relation $\tau_1 \to \tau_2 \leqslant \tau_1' \to \tau_2' \Leftrightarrow \tau_1' \leqslant \tau_1 \wedge \tau_2 \leqslant \tau_2'$, after all, and the algorithm relies mostly on such transformations. The authors actually present the proof that a model of types can effectively be constructed as a way to justify that the subtyping relation makes sense as it is, which is nice to have but would not really be absolutely necessary; something almost cosmetic.

In the present chapter, we show in some sense how to push the semantic approach further, all the way into the decision algorithm — we could say we present a semantic approach to deciding semantic subtyping. We give the set-theoretic model of types a practical use: types are translated into logical formulas describing precisely the set of model elements corresponding to the type. A type being a subtype of another then corresponds to the logical implication of the corresponding formulas being valid. We show that domain elements can be represented by finite trees and that the formulas corresponding to types can be written in a μ-calculus of finite trees for which we have an efficient satisfiability checker. Deciding subtyping between two types can then be done by feeding to this checker the negation of the implication formula relating the two types — if this formula is unsatisfiable, the implication is valid and thus subtyping holds; otherwise, we can exhibit explicitly a domain element which disproves the implication, that is, which belongs to the first type but not the second.

A benefit of this fully logical approach is made clear in Section 3.5 where we show that extending the type algebra of [Frisch *et al.* 2008] with type variables and altering the subtyping relation accordingly, in the way described by [Castagna & Xu 2011], can be done in a very simple way and at effectively zero cost in our system. This in turn immediately proves that subtyping is still decidable in the extended framework of [Castagna & Xu 2011] (this was chronologically, by a few days, the first satisfiability proof for that relation), and furthermore gives a precise complexity bound for its decision, since the translation into logic is linear and the complexity of the solver is known — this complexity bound is one of our contributions, since no other proof of it currently exists.

3.1.1 The Need for Polymorphism and Subtyping: a Concrete Example

This work is motivated by a growing need for polymorphic type systems for programming languages that manipulate XML data. For instance, XQuery [Boag *et al.* 2007] is the standard query and functional language designed for querying collections of XML data. The support of higher-order functions, currently missing from XQuery, appears in the requirements for the forthcoming XQuery 3.0 language [Engovatov & Robie 2010]. This results in an increasing demand in algorithms for proving or disproving statements with polymorphic types, and with types of higher-order functions (like the traditional `map` and `fold` functions), or more generally, statements involving the subtyping relation over a type algebra with recursive, product, function, intersection, union, and complement types together with type variables and universal quantification

over them.

For example, let us consider a simple property relating polymorphic types of functions that manipulate lists. We consider a type α, and denote by $[\alpha]$ the type of α-lists (lists whose elements are of type α). The type τ of functions that process an α-list and return a boolean is written as follows:

$$\tau = \forall\alpha.\ [\alpha] \to \texttt{Bool}$$

where $\texttt{Bool} = \{\texttt{true}, \texttt{false}\}$ is the type containing only the two values `true` and `false` Now let us consider functions that distinguish α-lists of even length from α-lists of odd length: such a function returns `true` for lists with an even number of elements of type α, and returns `false` for lists with an odd number of elements of type α. One may represent the set of these functions by a type τ' written as follows:

$$\forall\alpha.\ \text{even}[\alpha] \to \{\texttt{true}\} \wedge \text{odd}[\alpha] \to \{\texttt{false}\}$$

where $\{\texttt{true}\}$ and $\{\texttt{false}\}$ are singleton types (containing just one value). If we make explicit the parametric types even$[\alpha]$ and odd$[\alpha]$, τ' becomes:

$$\tau' = \forall\alpha.\left(\begin{array}{lll} & \mu v.(\alpha \times (\alpha \times v)) \vee \text{nil} & \to \{\texttt{true}\} \\ \wedge & \mu v.(\alpha \times (\alpha \times v)) \vee (\alpha \times \text{nil}) & \to \{\texttt{false}\} \end{array}\right)$$

where $\times$ denotes the cartesian product, μ binds the variable v for denoting a recursive type, and nil is a singleton type. Obviously, a particular function of type τ' can also be seen as a less-specific function of type τ. In other terms, from a practical point of view, a function of type τ can be replaced by a more specific function of type τ' while preserving type-safety (however the converse is not true). This is exactly what captures the notion of subtyping; in that case we write:

$$\tau' \leqslant \tau \tag{3.1}$$

where $\leqslant$ denotes the subtyping relation which is under scrutiny in this chapter.

3.1.2 Semantic Subtyping with Logical Solvers

During the last few years, a growing interest has been seen in the use of logical solvers such as satisfiability-testing solvers and satisfiability-modulo solvers in the context of functional programming and static type checking [Bierman *et al.* 2010, Benedikt & Cheney 2010]. In particular, solvers for tree logics [Genevès *et al.* 2007, de Moura & Bjørner 2008] are used as basic building blocks for type systems for XQuery.

The main idea in this chapter is a type-checking algorithm for polymorphic types based on deciding subtyping through a logical solver. To decide whether τ is a subtype of type τ', we first construct equivalent logical formulas φ_τ and $\varphi_{\tau'}$ and then check the validity of the formula $\psi = \varphi_\tau \Rightarrow \varphi_{\tau'}$ by testing the unsatisfiability of $\neg\psi$ using the satisfiability-testing solver. This technique corresponds to semantic subtyping [Frisch *et al.* 2008] since the underlying logic is inherently tied to a set-theoretic interpretation.

Semantic subtyping has been applied to a wide variety of types including refinement types [Bierman *et al.* 2010] and types for XML such as regular tree types [Hosoya *et al.* 2005], function types [Benzaken *et al.* 2003], and XPath [Clark & DeRose 1999] expressions [Genevès *et al.* 2007].

This fruitful connection between logics, their decision procedures, and programming languages permitted to equip the latter with rich type systems for sophisticated programming constructs such as expressive pattern-matching and querying techniques. The potential benefits of this interconnection crucially depend on the expressivity of the underlying logics. Therefore, there is an increasing demand for more and more expressiveness. For example, in the context of XML:

- SMT solvers like [de Moura & Bjørner 2008] offer an expressive power that corresponds to a fragment of first-order logic in order to solve the intersection problem between two queries [Benedikt & Cheney 2010];

- Full first-order logic solvers over finite trees [Genevès *et al.* 2007] solve containment and equivalence of XPath expressions;

- Monadic second-order logic solvers over trees, and – equivalent yet much more effective – satisfiability-solvers for μ-calculus over trees [Genevès *et al.* 2007] are used to solve query containment problems in the presence of type constraints.

3.1.3 Contributions of the Chapter

The novelty of our work is threefold. It is the first work that:

- Proves the decidability of semantic subtyping for polymorphic types with function, product, intersection, union, and complement types, as defined by Castagna and Xu [Castagna & Xu 2011], and gives a precise complexity upper-bound: $2^{(n)}$, where n is the size of types being checked. Decidability was only conjectured by Castagna and Xu before our result, although they have now proved it independently; our result on complexity is still the only one. In addition, we provide an effective implementation of the decision procedure.

- Produces counterexamples whenever subtyping does not hold with polymorphic and arrow types. These counterexamples are valuable for programmers as they represent evidence that the relation does not hold.

- Pushes the integration between programming languages and logical solvers to a very high level. The logic in use is not only capable of ranging over higher order functions, but it is also capable of expressing values from semantic domains that correspond to monadic second-order logic such as XML tree types [Genevès *et al.* 2007]. This shows that such solvers can become the core of XML-centric functional languages type-checkers such as those used in CDuce [Benzaken *et al.* 2003] or XDuce [Hosoya & Pierce 2003].

3.1.4 Structure of the Chapter

We introduce the semantic subtyping framework in Section 3.2 where we start with the monomorphic type algebra (without type variables). We present the tree logic in which we model semantic subtyping in Section 3.3. We detail the logical encoding of types in Section 3.4. Then, in Section 3.5 we extend the type algebra with type variables, and state the main result of the chapter: we show how to decide the subtyping relation for the polymorphic case in exponential time. We report on practical experiments using the implementation in Section 3.6. Finally, we discuss related work in Section 3.7 before concluding in Section 3.8.

3.2 Semantic Subtyping Framework

In this section, we present the type algebra we consider: we introduce its syntax and define its semantics using a set-theoretic interpretation. This framework is the one described at length in [Frisch *et al.* 2008]; we summarise its main features and give the intuitions behind it, using a slightly different point of view than the original paper, but we refer the reader to that paper for technical details.

We will then extend this framework with type variables in Section 3.5.

3.2.1 Types

Type terms are defined using the following grammar:

$$
\begin{array}{rcll}
\tau & ::= & & \\
 & & b & \text{basic type} \\
 & | & \tau \times \tau & \text{product type} \\
 & | & \tau \to \tau & \text{function type} \\
 & | & \tau \vee \tau & \text{union type} \\
 & | & \neg\tau & \text{complement type} \\
 & | & \mathbf{0} & \text{empty type} \\
 & | & v & \text{recursion variable} \\
 & | & \mu v.\tau & \text{recursive type}
\end{array}
$$

We consider μ as a binder and define the notions of free and bound variables and closed terms as standard. A type is a closed type term which is *well-formed* in the sense that:

- The negation operator only occurs in front of *closed* types;
- Both operands of an arrow constructor must be closed types as well;
- Every occurrence of a recursion variable is separated from its binder by at least one occurrence of the product or arrow constructor (guarded recursion).

So, for example, $\mu v.\mathbf{0} \vee v$ is not well-formed, nor is $\mu v.\mathbf{0} \to \neg v$.

Additionally, the following abbreviations are defined:

$$\tau_1 \wedge \tau_2 \stackrel{\text{def}}{=} \neg(\neg\tau_1 \vee \neg\tau_2)$$

and

$$\mathbf{1} = \neg\mathbf{0}$$

3.2.2 Set-theoretic interpretation

3.2.2.1 Underlying ideas

Before defining formally how types shall be interpreted, let us summarise the ideas which lead to that interpretation.

Consider a programming language whose values are constants from a set $\mathcal{C}$, pairs of values, and functions. We consider the different kinds of values disjoint, i. e. for example no value can simultaneously be a pair and a function. Let $\mathcal{W}$ be the set of all values in the language. The basic idea of the semantic subtyping framework is to interpret the types of the above algebra as subsets of $\mathcal{W}$, giving $\vee$ and $\neg$ the meaning of set-theoretic union and complement, and to define subtyping as set inclusion of interpretations.

Suppose we have an interpretation of base types b as sets of constants. As long as we don't use arrows, it is straightforward to define a set-theoretic semantics for $\times$. The recursive type $\mu v.\tau$ can be interpreted as a least fixpoint.

The usual interpretation of a function type however is operational rather than set-theoretic. Indeed, we can consider in the general case that when applying a function to an argument, a computation is triggered which can, possibly nondeterministically, either yield a value, yield an error or yield nothing (i. e. not terminate). The intended meaning of the type $\tau_1 \to \tau_2$ is that whenever applied to an argument of type τ_1, the function returns either a value of type τ_2 or nothing. An important feature of this framework is that it allows overloaded functions: a function f can return something of type τ_2 when given an argument of type τ_1, and return something of the completely different type τ_3 when given an argument of type τ_4. In that case, f has *both* type $\tau_1 \to \tau_2$ and type $\tau_4 \to \tau_3$, and since the type algebra allows boolean combinations of types, it also has type $\tau_1 \to \tau_2 \wedge \tau_4 \to \tau_3$, which is more precise than each simple arrow type.

This operational definition of arrow types makes impractical to interpret them as sets of actual function values defined in the considered language. Rather, [Frisch *et al.* 2008] propose to use the associated abstract functions, i. e. sets of pairs of an antecedent and a result. Note that because the computational functions are allowed to be nondeterministic, the abstract ones are not necessarily functions in the mathematical sense but more general relations. Formally, an abstract function is a subset of $\mathcal{W} \times (\mathcal{W} \cup \{\Omega\})$. Each pair (d, d') in the set means that, when given d as an argument, the function *may* yield d' as a result. If d does not appear as the first element of any pair, the operational interpretation is that the function can still accept d as an argument but will not yield a result: this represents a computation which does not

terminate. A pair of the form (d, Ω) is used to represent a function rejecting d as an argument: when given d, it yields an error.

The set of abstract functions of type $\tau_1 \to \tau_2$ can then be defined simply as all sets of pairs (d, d') such that whenever d is of type τ_1, d' is of type τ_2. This is called the extensional interpretation of function types in [Frisch *et al.* 2008]. Formally:

$$\mathbb{E}[\![\tau_1 \to \tau_2]\!] \stackrel{\text{def}}{=} \{S \subset \mathcal{W} \times (\mathcal{W} \cup \{\Omega\}) \mid (d, d') \in S \wedge (d : \tau_1) \Rightarrow (d : \tau_2)\}$$

where $(d : \tau)$ means the value d has type τ. Boolean combinators can be interpreted as the corresponding set-theoretic operations on extensional interpretations[1], and subtyping corresponds to inclusion between sets of abstract functions.

This extensional interpretation has the problem that not all abstract functions can have concrete implementations in the language, for cardinality reasons: the set of concrete functions is included in $\mathcal{W}$ since they are values themselves, but the set of all possible abstract functions is $\mathcal{P}(\mathcal{W} \times (\mathcal{W} \cup \{\Omega\}))$. However, inclusion between the extensional interpretations of two types clearly implies inclusion between the sets of values of those types, and for the converse implication to hold, it suffices that every type whose extensional interpretation is non-empty has a witness in the language. Indeed, because we have boolean combinators in the type algebra, the question of inclusion reduces to a question of emptiness.

It is not immediately obvious that a language with that property (i. e. that whenever there exists an abstract function of some type, there is also a function of that type in the language) exists. However the following property makes it easy to define one: whenever there exists an abstract function of some type, there also exists a **finite** abstract function (i. e. the set of pairs is finite) of the same type. To get an intuition of why this is true, remark that for an abstract function to have type $\tau_1 \to \tau_2$, it suffices that it contains *no* pair (a, b) with $(a : \tau_1)$ and $(b : \neg\tau_2)$. For it to have type $\neg(\tau_1 \to \tau_2)$, it suffices that it contains *one* such pair. Since the type algebra only allows *finite* boolean combinations of types, it is quite clearly impossible to build a type constraint that would be satisfied only by infinite sets of pairs.

Therefore, if we consider an abstract language where function values are simply finite lists of pairs of values, with the semantics described above, the semantic subtyping relation it induces on types is the same as any sufficiently expressive concrete language with the same set of base constants. We now define formally our semantic domain.

3.2.2.2 Formal definitions

Consider an arbitrary set $\mathcal{C}$ of constants. From it, we define the semantic domain $\mathcal{D}$ as the set of ds generated by the following grammar, where c ranges over constants in

[1] The attentive reader may remark that the complement of an arrow type includes not just all functions which do not have that type but also all non-functional values. In the full formal development in [Frisch *et al.* 2008], the extensional interpretation of a type is actually a subset of the disjoint union of non-functional values and abstract functions, so that this kind of things is taken into account.

$\mathcal{C}$:

$$
\begin{array}{rcll}
d & ::= & & \text{domain element} \\
 & & c & \text{base constant} \\
 & | & (d, d) & \text{pair} \\
 & | & \{(d, d'), \ldots, (d, d')\} & \text{function} \\
d' & ::= & & \text{extended domain element} \\
 & & d & \\
 & | & \Omega & \text{error}
\end{array}
$$

We suppose we have an interpretation $\mathbb{B}[\![\cdot]\!]$ of basic types b as subsets of $\mathcal{C}$.

To properly define the typing relation between extended domain elements and types, we first define a structural ordering relation $\trianglelefteq$ on $(\mathcal{D} \cup \{\Omega\}) \times T$ where T is the set of types:

- On extended domain elements, we use the ordering $d'_1 \trianglelefteq d'_2$ if d'_1 is a subterm of d'_2;
- Let the *shallow depth* of a type term be the longest path, in its syntactic tree, starting from the root and consisting only of μ, $\vee$, and $\neg$ nodes. We order types by $\tau_1 \trianglelefteq \tau_2$ if the shallow depth of τ_1 is less than the shallow depth of τ_2;
- Pairs are ordered lexicographically, i. e. $(d'_1, \tau_1) \trianglelefteq (d'_2, \tau_2)$ if either $d'_1 \triangleleft d'_2$ or $d'_1 = d'_2$ and $\tau_1 \trianglelefteq \tau_2$.

Recall the well-formedness constraint on types : in the syntactic tree, a recursion variable is always separated from its binder by a $\times$ or $\rightarrow$ constructor. This implies that the unfolding of a recursive type always has a strictly smaller shallow depth than the original type: $\mu v.\tau \triangleright \tau\{^{\mu v.\tau}/_{v}\}$; indeed, the substitution may increase the depth of the syntactic tree, but only below a $\times$ of $\rightarrow$ node, so it does not affect its shallow depth.

The predicate $(d' : \tau)$ where d' is either an element of $\mathcal{D}$ or Ω and τ is a type can now be defined recursively in the following way:

$$
\begin{aligned}
(\Omega : \tau) &= \mathsf{false} \\
(c : b) &= c \in \mathbb{B}[\![b]\!] \\
((d_1, d_2) : \tau_1 \times \tau_2) &= (d_1 : \tau_1) \wedge (d_2 : \tau_2) \\
(\{(d_1, d'_1), \ldots, (d_n, d'_n)\} : \tau_1 \rightarrow \tau_2) &= \forall i, (d_i : \tau_1) \Rightarrow (d'_i : \tau_2) \\
(d : \tau_1 \vee \tau_2) &= (d : \tau_1) \vee (d : \tau_2) \\
(d : \neg\tau) &= \neg(d : \tau) \\
(d : \mu v.\tau) &= (d : \tau\{^{\mu v.\tau}/_{v}\}) \\
(d : \tau) &= \mathsf{false} \text{ in any other case}
\end{aligned}
$$

It is easy to check that all occurrences of the predicate on the right-hand side of the definition are for pairs strictly smaller, with respect to $\trianglelefteq$, than the one on the left. Because all terms and types are finite, this makes the definition well-founded.

The interpretation of types as parts of $\mathcal{D}$ is then defined as $[\![\tau]\!] = \{d \mid (d : \tau)\}$. Note that Ω is not part of any type, as expected.

In this framework, we consider XML types as regular tree languages. An XML tree type is interpreted as the set of documents that match the type.

Finally, the subtyping relation is defined as $\tau_1 \leqslant \tau_2 \Leftrightarrow [\![\tau_1]\!] \subset [\![\tau_2]\!]$, or, equivalently, $[\![\tau_1 \wedge \neg\tau_2]\!] = \emptyset$.

3.3 Tree Logic Framework

In this section we introduce the logic in which we model the semantic subtyping framework. This logic is a subset of the one proposed in [Genevès *et al.* 2007]: a variant of μ-calculus whose models are finite trees. We first introduce below the syntax and semantics of the logic, before tuning it for representing types.

3.3.1 Formulas

Formulas are defined thus:

φ, ψ	$::=$		formula
		$\top$	true
	$\mid$	$\sigma \quad \mid \quad \neg\sigma$	atomic proposition (negated)
	$\mid$	X	variable
	$\mid$	$\varphi \vee \psi$	disjunction
	$\mid$	$\varphi \wedge \psi$	conjunction
	$\mid$	$\langle a \rangle\, \varphi \quad \mid \quad \neg \langle a \rangle\, \top$	existential (negated)
	$\mid$	$\mu(X_i = \varphi_i)_{i \in I}$ in ψ	(least) n-ary fixpoint

where $a \in \{1, 2\}$ are *programs*, and I is a finite set. Atomic propositions σ correspond to labels from a countable set Σ. Additionnally, we use the abbreviation $\mu X.\varphi$ for $\mu(X = \varphi)$ in φ.

Intuitively, the logic allows one to formulate regular properties on unranked trees: the programs "1" and "2" are respectively used to access the first child node and the next sibling node in an unranked tree. For instance, the formula $a \wedge \langle 1 \rangle\, (b \wedge \langle 2 \rangle\, c)$ is satisfied at the root of the tree denoted by the term $a(b, c)$. The recursive formula $\langle 1 \rangle\, (\mu X.a \vee \langle 1 \rangle\, X \vee \langle 2 \rangle\, X)$ states the existence of some node labelled with "a" at an arbitrary depth in the subtree. We formalize those intuitions in the next sections.

3.3.2 Semantic domain

The semantic domain is the set $\mathcal{F}$ of focused trees defined by the following syntax, where we have an alphabet Σ of labels, ranged over by σ:

$$
\begin{array}{rcll}
t & ::= & \sigma[tl] & \text{tree} \\
tl & ::= & & \text{list of trees} \\
 & & \varepsilon & \quad \text{empty list} \\
 & | & t :: tl & \quad \text{cons cell} \\
c & ::= & & \text{context} \\
 & & (tl, \mathit{Top}, tl) & \quad \text{root of the tree} \\
 & | & (tl, c[\sigma], tl) & \quad \text{context node} \\
f & ::= & (t, c) & \text{focused tree}
\end{array}
$$

A focused tree (t, c) is a pair consisting of a tree t and its context c. The context $(tl, c[\sigma], tl)$ comprises three components: a list of trees at the left of the current tree in reverse order (the first element of the list is the tree immediately to the left of the current tree), the context above the tree, and a list of trees at the right of the current tree. The context above the tree may be *Top* if the current tree is at the root, otherwise it is of the form $c[\sigma]$ where σ is the label of the enclosing element and c is the context in which the enclosing element occurs.

The *name* of a focused tree is defined as $\mathtt{nm}(\sigma[tl], c) = \sigma$.

We now describe how to navigate focused trees, in binary style. There are four directions, or *modalities*, that can be followed: for a focused tree f, $f\,\langle 1\rangle$ changes the focus to the first child of the current tree, $f\,\langle 2\rangle$ changes the focus to the next sibling of the current tree, $f\,\langle \overline{1}\rangle$ changes the focus to the parent of the tree *if the current tree is a leftmost sibling*, and $f\,\langle \overline{2}\rangle$ changes the focus to the previous sibling.

Formally, we have:

$$
\begin{aligned}
(\sigma[t :: tl], c)\,\langle 1\rangle &\stackrel{\text{def}}{=} (t, (\varepsilon, c[\sigma], tl)) \\
(t, (tl_l, c[\sigma], t' :: tl_r))\,\langle 2\rangle &\stackrel{\text{def}}{=} (t', (t :: tl_l, c[\sigma], tl_r)) \\
(t, (\varepsilon, c[\sigma], tl))\,\langle \overline{1}\rangle &\stackrel{\text{def}}{=} (\sigma[t :: tl], c) \\
(t', (t :: tl_l, c[\sigma], tl_r))\,\langle \overline{2}\rangle &\stackrel{\text{def}}{=} (t, (tl_l, c[\sigma], t' :: tl_r))
\end{aligned}
$$

When the focused tree does not have the required shape, these operations are not defined.

3.3.3 Interpretation

Formulas are interpreted as subsets of $\mathcal{F}$ in the following way, where V is a mapping from variables to formulas:

$$[\![\top]\!]_V \stackrel{\text{def}}{=} \mathcal{F} \qquad [\![\sigma]\!]_V \stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) = \sigma\}$$
$$[\![X]\!]_V \stackrel{\text{def}}{=} V(X) \qquad [\![\neg\sigma]\!]_V \stackrel{\text{def}}{=} \{f \mid \mathtt{nm}(f) \neq \sigma\}$$
$$[\![\varphi \vee \psi]\!]_V \stackrel{\text{def}}{=} [\![\varphi]\!]_V \cup [\![\psi]\!]_V \qquad [\![\varphi \wedge \psi]\!]_V \stackrel{\text{def}}{=} [\![\varphi]\!]_V \cap [\![\psi]\!]_V$$

$$[\![\langle a\rangle\, \varphi]\!]_V \stackrel{\text{def}}{=} \{f\,\langle \overline{a}\rangle \mid f \in [\![\varphi]\!]_V \wedge f\,\langle \overline{a}\rangle \text{ defined}\}$$
$$[\![\neg\,\langle a\rangle\, \top]\!]_V \stackrel{\text{def}}{=} \{f \mid f\,\langle a\rangle \text{ undefined}\}$$

$$\begin{aligned}
&[\![\mu(X_i = \varphi_i)_{i\in I} \text{ in } \psi]\!]_V \stackrel{\text{def}}{=} \\
&\quad \text{let } S = \{(T_i) \in \mathcal{P}(\mathcal{F})^I \mid \forall j \in I, [\![\varphi_j]\!]_{V[\overline{T_i/X_i}]} \subset T_j\} \quad \text{in} \\
&\quad \text{let } (U_j) = \left(\textstyle\bigcap_{(T_i)\in S} T_j\right)_{j\in I} \quad \text{in} \quad [\![\psi]\!]_{V[\overline{U_i/X_i}]}
\end{aligned}$$

where $V[\overline{T_i/X_i}](X) = V(X)$ if $X \notin \{X_i\}$ and T_i if $X = X_i$.

The lemma 4.2 of [Genevès *et al.* 2007] says that the interpretation of a fixpoint formula is equal to the union of the interpretations of all its finite unfoldings (where unfolding is defined as usual). A consequence (detailed in [Genevès *et al.* 2007]) is that the logic is closed under negation, i. e. for any closed φ, $\neg\varphi$ can be expressed in the syntax using De Morgan's relations and this definition:

$$\neg\,\langle a\rangle\, \varphi \stackrel{\text{def}}{=} \neg\,\langle a\rangle\, \top \vee \langle a\rangle\, \neg\varphi$$
$$\neg\mu(X_i = \varphi_i) \text{ in } \psi \stackrel{\text{def}}{=} \mu(X_i = \neg\varphi_i\{\overline{X_i/\neg X_i}\}) \text{ in } \neg\psi\{\overline{X_i/\neg X_i}\}$$

In the following, we consider only closed formulas and write $[\![\varphi]\!]$ for $[\![\varphi]\!]_\emptyset$.

3.4 Logical Encoding

In the context of the present chapter, we want finite tree models of the logic to correspond to types introduced in section 3.2. Thus, we first extend the alphabet of node labels to be able to reason with type constructors. Then, we present the translation of a type into a logical formula.

3.4.1 Representation of domain elements

Let $\mathcal{T}$ be the set of (unfocused) trees. Set $\mathcal{C} = \{\mathbb{B}[tl] \mid tl \in \mathcal{T}^*\}$, where $\mathbb{B}$ is a label **not in** Σ : the set of trees with a distinguished root $\mathbb{B}$. Let $\mathcal{T}_{ext}$ be the set of trees obtained by extending Σ with the four extra labels $(\rightarrow), (\times), \mathbb{B}$ and Ω. Then $\mathcal{D}_\Omega$ can

straightforwardly be embedded into $\mathcal{T}_{ext}$ in the following way:

$$\begin{aligned}
\mathsf{tree}(c) &= c \\
\mathsf{tree}(\Omega) &= \Omega[\varepsilon] \\
\mathsf{tree}(d, d') &= (\times)[\mathsf{tree}(d) :: \mathsf{tree}(d') :: \varepsilon] \\
\mathsf{tree}(\{(d_1, d'_1), \ldots, (d_n, d'_n)\}) &= (\rightarrow)[\mathsf{tree}(d_1, d'_1) :: \ldots :: \mathsf{tree}(d_n, d'_n) :: \varepsilon]
\end{aligned}$$

The intuition of this tree representation is illustrated in Figure 3.1.

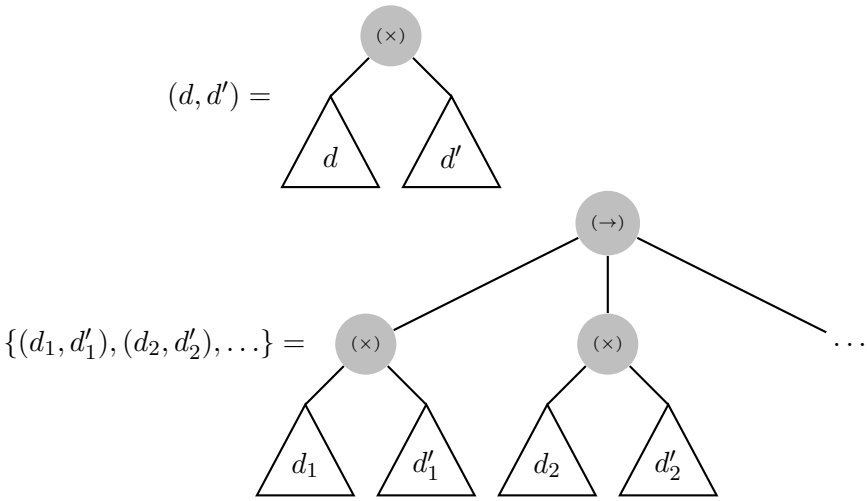


Figure 3.1: Pairs and functions are represented as trees with special labels.

In the following we consider this embedding implicitly done, so $\mathcal{D}_\Omega \subset \mathcal{T}_{ext}$.

3.4.2 Translation of types

First of all, we can define basic types b, which are to represent sets of trees with no special nodes but the distinguished root $\mathbb{B}$, as the (closed) base formulas of the logic. The full interpretation of formulas uses sets of focused trees, but note that a toplevel formula cannot contain any constraint on what is above or to the left of the node at focus, so it can be considered as describing just a list of trees. The interpretation of a base type will then be a $\mathbb{B}$ root whose list of children is described by the formula. Formally:

$$\mathbb{B}[\![\varphi]\!] \stackrel{\text{def}}{=} \{\mathbb{B}[t :: tl_2] \mid (t, (tl_1, c[\sigma], tl_2)) \in [\![\varphi]\!]\}$$

Note how the only part of the context taken into account in defining the semantics is the list of following siblings of the current node.

Then, we translate the types into *extended* formulas obtained (as for extended trees) by adding to Σ the labels $(\times), (\rightarrow), \Omega$ and $\mathbb{B}$. Straightforwardly these formulas denote lists of trees in $\mathcal{T}_{ext}$.

First, we define the following formulas:

$$
\begin{aligned}
\mathsf{isbase} &= \mu X.((\neg \langle 1\rangle \top \vee \langle 1\rangle X) \wedge (\neg \langle 2\rangle \top \vee \langle 2\rangle X) \\
&\qquad \wedge \neg\mathbb{B} \wedge \neg(\rightarrow) \wedge \neg(\times) \wedge \neg\Omega) \\
\mathsf{error} &= \Omega \wedge \neg \langle 1\rangle \top \\
\mathsf{isd} &= \mu X.(\\
&\qquad (\mathbb{B} \wedge \langle 1\rangle \mathsf{isbase})\vee \\
&\qquad ((\times) \wedge \langle 1\rangle (X \wedge \langle 2\rangle (X \wedge \neg \langle 2\rangle \top)))\vee \\
&\qquad ((\rightarrow) \wedge (\neg \langle 1\rangle \top\vee \\
&\qquad\quad \langle 1\rangle \mu Y.((\neg \langle 2\rangle \top \vee \langle 2\rangle Y)\wedge \\
&\qquad\qquad (\times) \wedge \langle 1\rangle (X \wedge \langle 2\rangle ((X \vee \mathsf{error}) \wedge \neg \langle 2\rangle \top)) \\
&\qquad))))
\end{aligned}
$$

isbase selects all tree lists which do not contain any of the special labels (the fixpoint is for selecting all the nodes). error is straightforward. isd selects all elements of $\mathcal{D}$ (actually, all tree lists whose first element is in $\mathcal{D}$): either they are a constant (a $\mathbb{B}$ node with a base list as children), or a pair (a $(\times)$ node with exactly two children each of which is itself in $\mathcal{D}$), or a function: a $(\rightarrow)$ node with either no children at all or a list of children (described by Y) all of which are pairs whose second element may be error.

We now associate to every type τ the formula $\mathsf{fullform}(\tau) = \mathsf{isd} \wedge \mathsf{form}(\tau)$, with $\mathsf{form}(\tau)$ defined as follows, where X_v is a different variable for every v and is also different from X:

$$
\begin{aligned}
\mathsf{form}(b) &= \mathbb{B} \wedge \langle 1\rangle b \\
\mathsf{form}(\tau_1 \times \tau_2) &= (\times) \wedge \langle 1\rangle (\mathsf{form}(\tau_1) \wedge \langle 2\rangle \mathsf{form}(\tau_2)) \\
\mathsf{form}(\tau_1 \rightarrow \tau_2) &= (\rightarrow) \wedge (\neg \langle 1\rangle \top\vee \\
&\qquad \langle 1\rangle \mu X.((\neg \langle 2\rangle \top \vee \langle 2\rangle X) \\
&\qquad\quad \wedge \langle 1\rangle (\neg\mathsf{form}(\tau_1) \vee \langle 2\rangle \mathsf{form}(\tau_2))) \\
&\qquad) \\
\mathsf{form}(\tau_1 \vee \tau_2) &= \mathsf{form}(\tau_1) \vee \mathsf{form}(\tau_2) \\
\mathsf{form}(\neg\tau) &= \neg\mathsf{form}(\tau) \\
\mathsf{form}(\mathbf{0}) &= \neg\top \\
\mathsf{form}(v) &= X_v \\
\mathsf{form}(\mu v.\tau) &= \mu X_v.\mathsf{form}(\tau)
\end{aligned}
$$

Recall that basic types b are themselves formulas, but that their interpretation as a type is different from their interpretation as a formula (see the first paragraph of Section 3.4.2 and the definition of $\mathbb{B}[\![\varphi]\!]$, the interpretation as a type, in terms of $[\![\varphi]\!]$,

the interpretation as a formula). This explains why the translation of b contains b itself. The translation of product types is simple: it describes a $(\times)$ node whose first child is described by $\mathsf{form}(\tau_1)$ and has a following sibling described by $\mathsf{form}(\tau_2)$. The translation of arrow types has a structure similar to what appeared in isd: it describes a $(\rightarrow)$ node with either no children or a list of children recursively described by X (each node has either no following sibling or a following sibling itself described by X). Each of these nodes must have a first child which either is not of type τ_1 or has a next sibling of type τ_2 — this means that these nodes represent pairs (d_i, d'_i) such that $(d_i : \tau_1) \Rightarrow (d'_i : \tau_2)$. The attentive reader may notice that the formula $\mathsf{form}(\tau_1 \rightarrow \tau_2)$ does not enforce in itself that all children of the $(\rightarrow)$ node are actually pairs; the reason for that is that isd already enforces it.

We can see that the formulas in the translation do not contain any $\langle 2 \rangle$ at toplevel (i. e. not under $\langle 1 \rangle$), nor does isd. This means they describe a single tree (they say nothing on its siblings), or in other words that in their interpretation as focused trees, the context is completely arbitrary, as it is not constrained in any way. Formally, we thus define the restricted interpretation of extended formulas as follows:

$$\mathbb{F}[\![\varphi]\!] \stackrel{\text{def}}{=} \{t \mid (t, c) \in [\![\varphi]\!]\}$$

That is, we drop the context completely.

Then we have $\mathbb{F}[\![\mathsf{fullform}(\tau)]\!] = [\![\tau]\!]$. This is a particular case of the property for polymorphic types which will be proved in the following section.

The main consequence of this property is that a type τ is empty if and only if the interpretation of the corresponding formula is empty — which is equivalent to the formula being unsatisfiable. Because there exists a satisfiability-checking algorithm for this tree logic [Genevès *et al.* 2007], this means that this translation gives an alternative way to decide the classical semantic subtyping relation as defined in [Frisch *et al.* 2008]. More interestingly, it yields a decision procedure for the subtyping relation *in the polymorphic case as well*, as we will explain in the next section.

3.5 Polymorphism: Supporting Type Variables

So far we have described a new, logic-based approach to a question — semantic subtyping in the presence of intersection, negation and arrow types — which had already been studied. We now show how this new approach allows us, in a very natural way, to encompass the latest work by adding polymorphism to the types along the lines of [Castagna & Xu 2011].

We add to the syntax of types *variables*, α, β, γ taken from a countable set $\mathcal{V}$. If τ is a polymorphic type, we write $\mathsf{var}(\tau)$ the set of variables it contains and call *ground type* a type with no variable. We sometimes write $\tau(\overline{\alpha})$ to indicate that $\mathsf{var}(\tau)$ is included in $\overline{\alpha}$.

Note that we only consider prenex (ML-style) parametric polymorphism, not higher-rank polymorphism, so there are no quantifiers in the syntax of types.

3.5.1 Subtyping in the polymorphic case: a problem of definition

Before defining formal interpretations for polymorphic types, we briefly review how extending the semantic subtyping framework to the polymorphic case has been addressed in previous work.

The intuition of subtyping in the presence of type variables is that $\tau_1(\overline{\alpha}) \leqslant \tau_2(\overline{\alpha})$ should hold true whenever, *independently of the variables* $\overline{\alpha}$, any value of type τ_1 has type τ_2 as well. However the correct definition of 'independently' is not obvious. It should look like this:

$$\forall \overline{\alpha}, [\![\tau_1(\overline{\alpha})]\!] \subset [\![\tau_2(\overline{\alpha})]\!]$$

but because variables are abstractions, it is not completely clear over what to quantify them. As mentioned in [Hosoya *et al.* 2009], a candidate — naive — definition would use *ground substitutions*, that is, if the inclusion of interpretations always holds when variables are replaced with ground types, then the subtyping relation holds:

$$\tau_1(\overline{\alpha}) \leqslant \tau_2(\overline{\alpha}) \Leftrightarrow \forall \overline{\tau} \text{ ground types}, [\![\tau_1(\overline{\tau}/\overline{\alpha})]\!] \subset [\![\tau_2(\overline{\tau}/\overline{\alpha})]\!] \tag{3.2}$$

Obviously the condition on the right must be *necessary* for subtyping to hold. But deciding that it is *sufficient* as well makes the relation unsatisfactory and somehow counterintuitive, as remarked in [Hosoya *et al.* 2009]. Indeed, suppose int is an *indivisible* type, that is, that it has no subtype beside $\mathbf{0}$ and itself. Then the following would hold:

$$\mathsf{int} \times \alpha \leqslant (\mathsf{int} \times \neg\mathsf{int}) \vee (\alpha \times \mathsf{int}) \tag{3.3}$$

This relation abuses the definition by taking advantage of the fact that for any ground type τ, either $[\![\mathsf{int}]\!] \subset [\![\tau]\!]$ or $[\![\tau]\!] \subset [\![\neg\mathsf{int}]\!]$. In the first case, because $[\![\tau]\!] \subset ([\![\neg\mathsf{int}]\!] \cup [\![\mathsf{int}]\!])$, we have $[\![\mathsf{int} \times \tau]\!] \subset [\![\mathsf{int} \times \neg\mathsf{int}]\!] \cup [\![\mathsf{int} \times \mathsf{int}]\!]$ and then the second member of the union is included in $[\![\tau \times \mathsf{int}]\!]$. In the second case, we directly have $[\![\mathsf{int} \times \tau]\!] \subset [\![\mathsf{int} \times \neg\mathsf{int}]\!]$.

This trick, which only works with indivisible ground types, not only shows that candidate definition (3.2) yields bizarre relations where a variable occurs in unrelated positions on both sides. It also means the candidate definition is very sensitive to the precise semantics of base types, since it distinguishes indivisible types from others. More precisely, it means that refining the collection of base types, for example by adding types even and odd, can break subtyping relations which held true without these new types — this is simply due to the fact that it increases the set over which $\overline{\tau}$ is quantified in (3.2), making the relation stricter. This could hardly be considered a nice feature of the subtyping relation.

The conclusion is thus that the types in (3.3) should be considered related *by chance* rather than by necessity, hence not in the subtyping relation, and that quantifying over all possible ground types is not enough; in other words, candidate definition (3.2) is too weak and does not properly reflect the intuition of 'independently of the variables'. Indeed, (3.3) is in fact dependent on the variable as we saw, the point being that there are only two cases and that the convoluted right-hand type is crafted so that the relation holds in both of them, though for different reasons.

In order to restrict the definition of subtyping, [Hosoya *et al.* 2009], which concentrates on XML types, uses a notion of *marking*: some parts of a value can be marked (using paths) as corresponding to a variable, and the relation 'a value has a type' is changed into 'a marked value matches a type', so the semantics of a type is not a set of values but of pairs of a value and a marking. This is designed so that it integrates well in the XDuce language, which has pattern-matching but no higher-order functions (hence no arrow types), so their system is tied to the operational semantics of matching and provides only a partial solution.

The question of finding the correct definition of semantic subtyping in the polymorphic case was finally settled very recently by Castagna and Xu [Castagna & Xu 2011]. Their definition does, in the same way as (3.2), follow the idea of a universal quantification over possible *meanings* of variables but solves the problem raised by (3.3) by using a much larger set of possible meanings — thus yielding a stricter relation. More precisely, variables are allowed to represent not just ground types but any arbitrary part of the semantic domain; furthermore, the semantic domain itself must be large enough, which is embodied by the notion of *convexity*. We refer the reader to [Castagna & Xu 2011] for a detailed discussion of this property and its relation to the notion of *parametricity* studied by Reynolds in [Reynolds 1983]; we will here limit ourselves to introducing the definitions strictly necessary for the discussion at hand.

In this work, we do not use this definition with its universal quantification directly. Rather, we retain from [Hosoya *et al.* 2009] the idea of tagging (pieces of) values which correspond to variables, but do so in a more abstract way, by extending the semantic domain, and define a *fixed* interpretation of polymorphic types in this extended domain as a straightforward extension of the monomorphic framework. We then show how to build a set-theoretic model of polymorphic types, in the sense of [Castagna & Xu 2011], based on this domain, and prove that the inclusion relation on fixed interpretations is equivalent to the full subtyping relation induced by this model. Finally, we explain briefly the notion of convexity and show that this model is convex, implying that this relation is, in fact, the semantic subtyping relation on polymorphic types, as defined in [Castagna & Xu 2011]. These steps are formally detailed in the following section.

3.5.2 Interpretation of polymorphic types

Let Λ be an infinite set of optional labels, and ι an injective function from the set of variables $\mathcal{V}$ to Λ. (It would be possible to set $\Lambda = \mathcal{V}$, but for clarity we prefer to distinguish *labels* which tag elements of the semantic domain from *variables* which occur in types.) We extend the grammar of (extended) trees by allowing any node to bear, in addition to its single σ label from $\Sigma \cup \{(\rightarrow), (\times), \mathbb{B}, \Omega\}$, any (finite) number of labels from Λ. We write it $\sigma_L[tl]$ where L is a finite part of Λ. We extend $\mathcal{C}$ and $\mathcal{D}$ accordingly. When using the non-tree form of types, for instance (d_1, d_2), we indicate the set of root labels on the bottom right like this: $(d_1, d_2)_L$ (here L is the set of labels borne by the $(\times)$ node constituting the root of the pair tree).

We then extend the predicate defining the interpretation of types given in Section

3.2.2.2 with the following additional case:

$$(\sigma_L[tl] : \alpha) = \iota(\alpha) \in L$$

In other words, the interpretation of a type variable is the set of all trees whose root bears the label corresponding to that variable. The other cases are unchanged, except that the semantic domain is now much larger. This means that the same definition leads to larger interpretations; in particular, the interpretation of a (nonempty) ground type is always an infinite set which contains all possible labellings for each of its trees.

Subtyping over polymorphic types is then defined, as before, as set inclusion between interpretations:

$$\tau_1(\overline{\alpha}) \leqslant \tau_2(\overline{\alpha}) \Leftrightarrow [\![\tau_1(\overline{\alpha})]\!] \subset [\![\tau_2(\overline{\alpha})]\!] \tag{3.4}$$

It may seem strange to give type *variables* a *fixed* interpretation, and on the other hand it may seem surprising that this definition of subtyping does not actually contain any quantification and is nevertheless stronger than (3.2) which contains one. The key point is that a form of universal quantification is implicit in the extension of the semantic domain: in some sense, the interpretation of a variable represents all possible values of the variable *at once*. Indeed, for any variable α and any tree d in the domain, there always exist both an infinity of copies of d which are in the interpretation of α and another infinity of copies which are not. From the point of view of logical satisfiability, this makes the domain big enough to contain all possible cases.

In order to show that, despite the appearances, Definition (3.4) accurately represents a relation that holds *independently of the variables*, we rely, as discussed above, on the formal framework developed by Castagna and Xu [Castagna & Xu 2011]. For this, we first introduce *assignments* η: functions from $\mathcal{V}$ to $\mathcal{P}(\mathcal{D})$ (where $\mathcal{D}$ is the extended semantic domain with labels). Thus an assignment attributes to each variable an arbitrary set of elements from the semantic domain.

We then define the interpretation of a type *relative to an assignment* in the following way: the predicate $(d' :_\eta \tau)$ is defined inductively in the same way as the $(d' : \tau)$ of Section 3.2.2.2 but with the additional clause:

$$(d :_\eta \alpha) = d \in \eta(\alpha).$$

The interpretation of the polymorphic type τ relative to the assignment η is then $[\![\tau]\!]\eta = \{d \mid (d :_\eta \tau)\}$. This defines an infinity of possible interpretations for a type, depending on the actual values assigned to the variables, and constitutes a set-theoretic model of types in the sense of [Castagna & Xu 2011]. The subtyping relation induced by this model is the following:

$$\tau_1(\overline{\alpha}) \leqslant \tau_2(\overline{\alpha}) \Leftrightarrow \forall \eta \in \mathcal{P}(\mathcal{D})^{\mathcal{V}}, [\![\tau_1(\overline{\alpha})]\!]\eta \subset [\![\tau_2(\overline{\alpha})]\!]\eta \tag{3.5}$$

which we can more easily compare to the candidate definition (3.2): it does in the same way quantify over possible meanings of the variables but uses a much larger set

of possible meanings, yielding a stricter relation. We will now prove that this relation is, for our particular model, actually equivalent to (3.4).

For this, let us first define the *canonical assignment* η_ι as follows:

$$\eta_\iota(\alpha) \stackrel{\text{def}}{=} \{\sigma_L[tl] \in \mathcal{D} \mid \iota(\alpha) \in L\}.$$

Then it is easily seen that the fixed interpretation $[\![\tau]\!]$ of a polymorphic type is the same as its interpretation relative to the canonical assignment, $[\![\tau]\!]\eta_\iota$. What we would like to prove is that the canonical assignment is somehow representative of all possible assignments, making the fixed interpretation sufficient for the purpose of defining subtyping. This is done by the following lemma and corollary.

Lemma 1. *Let V be a finite part of $\mathcal{V}$. Let η be an assignment. Let T be the set of all types τ such that $\mathsf{var}(\tau) \subset V$. Then there exists a function $F_V^\eta : \mathcal{D} \to \mathcal{D}$ such that: $\forall \tau \in T, \forall d \in \mathcal{D}, d \in [\![\tau]\!]\eta \Leftrightarrow F_V^\eta(d) \in [\![\tau]\!]\eta_\iota$.*

Proof. For d in $\mathcal{D}$, let $L(d) = \{\iota(\alpha) \mid \alpha \in V \wedge d \in \eta(\alpha)\}$. Since V is finite, $L(d)$ is finite as well. We define $F_V^\eta(d)$ inductively as follows:

- If $d = \mathbb{B}_L[tl]$ then $F_V^\eta(d) = \mathbb{B}_{L(d)}[tl]$
- If $d = (d_1, d_2)_L$ then $F_V^\eta(d) = (F_V^\eta(d_1), F_V^\eta(d_2))_{L(d)}$
- $F_V^\eta(\Omega) = \Omega$
- If $d = \{(d_1, d_1'), \ldots, (d_n, d_n')\}_L$ then
  $F_V^\eta(d) = \{(F_V^\eta(d_1), F_V^\eta(d_1')), \ldots, (F_V^\eta(d_n), F_V^\eta(d_n'))\}_{L(d)}$

So F_V^η preserves the structure but changes the labels so that the root node of $F_V^\eta(d)$ is labelled with $L(d)$ and so on inductively for its subterms.

Let $\mathcal{P}(d, \tau) = d \in [\![\tau]\!]\eta \Leftrightarrow F_V^\eta(d) \in [\![\tau]\!]\eta_\iota$. We prove that it holds for all pairs (d, τ) such that τ is in T by induction on those pairs, using the ordering relation $\trianglelefteq$ defined in Section 3.2.2.2, noticing that $\tau \in T$ implies that all subterms (and unfoldings) of τ are in T as well. The base cases are:

- If τ is a variable. Then it is in V by hypothesis and $\mathcal{P}(d, \tau)$ is true by definition of $L(d)$.
- If it is a base type. Then $\mathcal{P}(d, \tau)$ is true because the interpretation of τ is independent of assignments and labellings.

For the inductive cases, we suppose the property true for all strictly smaller pairs (d, τ) such that τ is in T.

- For the arrow and product cases, the inductive definition of F_V^η makes the result straightforward.
- For the negation and disjunction cases, the result is immediate from the induction hypothesis.

- For $\mu v.\tau$, recall that the well-formedness constraint on types implies that the type's unfolding has a strictly smaller shallow depth than the original type, hence we can use the induction hypothesis on the unfolding and conclude.

□

Corollary 1. *Let τ be a type.* $\bigcup_{\eta \in \mathcal{P}(\mathcal{D})^{\mathcal{V}}} [\![\tau]\!]\eta = \emptyset$ *if and only if* $[\![\tau]\!]\eta_\iota = \emptyset$.

Proof. If the union is not empty, there exists η and d such that $d \in [\![\tau]\!]\eta$. From the previous lemma we then have $F^{\eta}_{\mathsf{var}(\tau)}(d) \in [\![\tau]\!]\eta_\iota$. □

This corollary shows that the canonical assignment is representative of all possible assignments and implies that the subtyping relation defined by (3.4) is equivalent to the one defined by (3.5).

Convexity of the model. Definition (3.5) corresponds to semantic subtyping as defined in [Castagna & Xu 2011], but only on the condition that the underlying model of types be *convex*. Indeed, we can see that this definition is dependent on the set of possible assignments, which itself depends on the chosen (abstract) semantic domain, so it is reasonable to think that increasing the semantic domain could restrict the relation further. In other words, for the definition to be correct, the domain must be large enough to cover all cases. Castagna and Xu's *convexity* characterises this notion of 'large enough'. The property is the following: a set-theoretic model of types is *convex* if, whenever a finite collection of types τ_1 to τ_n each possess a nonempty interpretation relative to some assignment, then there exists a common assignment making all interpretations nonempty at once. This reflects the idea that there are enough elements in the domain to witness all the cases.

In our case, it comes as no surprise that the extended model of types is convex since any nonempty ground type has an infinite interpretation, which, as proved in [Castagna & Xu 2011], is a sufficient condition. But we need not even rely on this result since Corollary 1 proves a property even stronger than convexity: having a nonempty interpretation relative to *some* assignment is the same as having a nonempty interpretation relative to *the* common canonical assignment. This stronger property makes the apparently weaker relation defined by (3.4) equivalent, in our particular model, to the full semantic subtyping relation Castagna and Xu defined. This allows us to reduce the problem of deciding their relation to a question of inclusion between fixed interpretations, making the addition of polymorphism a mostly straightforward extension to the logical encoding we presented for the monomorphic case.

Interestingly, in [Castagna & Xu 2011] the authors suggest that convexity constrains the relation enough that it should allow reasoning on types, similarly to the way parametricity allowed Wadler [Wadler 1989] to deduce 'theorems for free' from typing information. The fact that our logical reasoning approach very naturally has this convexity property — indeed, it is difficult to think of a logical representation of variables which would not have it — seems to corroborate their intuition, although reasoning on types beyond deciding subtyping is currently left as future work.

We now show how the type system extended with type variables is encoded in our logic.

3.5.3 Logical encoding of variables

We extend the logic with atomic propositions α which behave similarly as σ except they are not mutually exclusive. The interpretation of these propositions is defined as:

$$[\![\alpha]\!] = \{(\sigma_L[tl], c) \mid \iota(\alpha) \in L\}$$

$$[\![\neg\alpha]\!] = \{(\sigma_L[tl], c) \mid \iota(\alpha) \notin L\}$$

The translation $\mathsf{form}(\tau)$ of types into formulas is extended in the obvious way by $\mathsf{form}(\alpha) = \alpha$.

Theorem 1. *With these extended definitions,* $\mathbb{F}[\![\mathsf{fullform}(\tau)]\!] = [\![\tau]\!]$.

Proof. Preliminary remark: whenever φ does not contain any $\langle 2\rangle$ at toplevel (which is the case of the formulas representing types), then $[\![\varphi]\!] = \mathbb{F}[\![\varphi]\!] \times \mathbf{C}$ where $\mathbf{C}$ is the set of all possible contexts. Hence, when considering such formulas, set-theoretic relations between full interpretations are equivalent to the same relations between first components.

First we check that $\mathbb{F}[\![\mathsf{isd}]\!] = \mathcal{D}$ and reformulate the statement as $\mathcal{D} \cap \mathbb{F}[\![\mathsf{form}(\tau)]\!] = [\![\tau]\!]$.

We make the embedding function tree explicit for greater clarity. What we have to show is that, for any d in $\mathcal{D}$, we have $(d : \tau)$ if and only if $(\mathsf{tree}(d), c)$ is in $[\![\mathsf{form}(\tau)]\!]$ for some (or, equivalently, for any) c.

The property is proved by induction on the pair (d, τ), following the definition of the predicate:

- For $(c : b)$ it holds by definition.
- For $((d_1, d_2)_L : \tau_1 \times \tau_2)$, let $f = (\mathsf{tree}((d_1, d_2)_L), c)$. f is in $[\![\mathsf{form}(\tau_1 \times \tau_2)]\!]$ if and only if $f\langle 1\rangle$ is in $[\![\mathsf{form}(\tau_1)]\!]$ and $f\langle 1\rangle\langle 2\rangle$ is in $[\![\mathsf{form}(\tau_2)]\!]$. (We already know that the node name is $(\times)$ by the structure of d.) Just see that the tree rooted at $f\langle 1\rangle$ is $\mathsf{tree}(d_1)$ and the one at $f\langle 1\rangle\langle 2\rangle$ is $\mathsf{tree}(d_2)$.
- For functions, use the finite unfolding property and the fact the set of pairs is finite, then see, similarly as above, that the correct properties are enforced when navigating the tree.
- For union, negation and empty types, use the preliminary remark.
- For $(d : \alpha)$, just see that $d \in \iota(\alpha)$ and $d \in \mathbb{F}[\![\alpha]\!]$ both mean that the root node of d, which is the node at focus in the formula, bears the label $\iota(\alpha)$.
- For $(d : \mu v.\tau)$, use the property that the interpretation of a fixpoint formula and its unfolding are the same (lemma 4.2 of [Genevès *et al.* 2007]).

□

Corollary 2. *$\tau_1 \leqslant \tau_2$ holds if and only if* $\mathsf{fullform}(\tau_1 \wedge \neg\tau_2)$*, or alternatively* $\mathsf{isd} \wedge \mathsf{form}(\tau_1) \wedge \neg\mathsf{form}(\tau_2)$*, is unsatisfiable.*

3.5.4 Complexity

Lemma 2. *Provided two types τ_1 and τ_2, the subtyping relation $\tau_1 \leqslant \tau_2$ can be decided in time $2^{\mathcal{O}(|\tau_1|+|\tau_2|)}$ where $|\tau_i|$ is the size of τ_i.*

Proof. The logical translation of types performed by the function $\mathsf{form}(\cdot)$ does not involve duplication of subformulas of variable size, therefore $\mathsf{form}(\tau)$ is of linear size with respect to $|\tau|$. Since isd has constant size, the whole translation $\mathsf{fullform}(\tau)$ is linear in terms of $|\tau|$. For testing satisfiability of the logical formula, we use the satisfiability-checking algorithm presented in [Genevès *et al.* 2007] whose time complexity is $2^{\mathcal{O}(n)}$ in terms of the formula size n. □

3.6 Implementation and Practical Experiments

In this section we report on some interesting lessons learned from practical experiments with the implementation of the system in order to prove relations in the type algebra. We first describe the main techniques used to implement the whole system, the minimal necessary background for using the implementation, and then we review and discuss several informative examples.

3.6.1 Implementation Principles

The algorithm for deciding the subtyping relation has been implemented on top of the satisfiability solver that was first introduced in [Genevès 2006, Genevès *et al.* 2007]. Since this algorithm constitutes the core of our implementation we briefly review its essential principles below and highlight its properties in the polymorphic setting.

Search universe and exponential complexity. The fundamental principle of the algorithm is to look for a finite tree that satisfies a given logical formula. For this purpose, it first constructs a compact representation of the relevant search universe in which to look for a tree model satisfying the given formula. This representation, called the Lean of the formula, is a set of subformulas of the initial formula. It is computed from the so-called Fisher-Ladner closure of the initial formula: it is composed of all the atomic propositions found in the formula, plus all distinct modal subformulas that can be obtained by unrolling fixpoints, and four basic "topological" formulas that indicate whether a given node admits some parent node, some child (or whether it is leaf and/or a root). This Lean set is important since its powerset precisely defines the search universe in which the algorithm looks for trees. For this reason, the time complexity of the algorithm is $2^{\mathcal{O}(n)}$ with respect to Lean size n. The acute reader may notice that the Lean size of a large logical formula is usually smaller than the

size of the formula measured as the number of all connectives and operands. This is because the Lean representation naturally eliminates duplicate subformulas and discards disjunctions and conjunctions at top level. Furthermore, we implemented an additional optimization: we rely on a binary encoding of indices of symbols in order to perform a logarithmic compression of the number of atomic propositions in the lean. The effect of this optimization can be observed for complex types with a large number of symbols, for which it is particularly effective[2].

Bottom-up search as a fixpoint computation. Once the Lean set is known, the algorithm starts traversing all relevant tree nodes in an attempt to build a satisfying tree. This search is performed in a bottom-up fashion, in the manner of a fixpoint computation. The algorithm considers a set of tree nodes whose subtrees have been proved consistent. The algorithm begins with the empty set of nodes, then at the first step, all possible leaves are added. Then, the algorithm repeatedly try to add new tree nodes to this set, until no more nodes can be added, i.e. a fixpoint has been reached. It is easy to observe that the algorithm terminates since, in the worst case (when the formula is unsatisfiable) it explores all the relevant nodes, that is, all subsets of the Lean, which is a finite set. At each step, whenever the algorithm is about to add a candidate node to the set of proved nodes, essential checks are performed to make sure that the higher tree rooted at the candidate node is logically consistent with subtrees already proved at earlier steps. In particular, modal formulas may impose constraints on successor nodes that must be checked for consistency when two nodes are connected. These checks are described formally in [Genevès *et al.* 2007]. At each step of the computation, the truth status of the initial formula given as input to the algorithm is tested at the freshly proved nodes. It the formula is found to hold at this node then the algorithm immediately terminates with a proof that the formula is satisfiable. This step by step approach offers several advantages. First, it opens the door to an implementation with semi-implicit techniques, and second, one can easily keep track of the current state of the set of proved nodes at each step in order to generate small satisfying trees.

Use of semi-implicit techniques. An important observation about the fixpoint computation is that for a given candidate node to be added to the set of proved nodes, the algorithm does not need to keep track of all possible subtrees that are consistent with the candidate node, but instead it is enough to find only one proved subtree for each successor of the candidate node. This observation has an important consequence: it makes it possible to avoid the explicit enumeration of all proved subtrees into memory. Instead checking the existence of at least one proved subtree per required successor of a candidate node is enough. This makes it possible to encode the algorithm with boolean functions operating on a bit-vector representation of the Lean set (as described in [Genevès 2006]), opening the door for an implementation based on Binary Decision

[2]This optimization can be turned on using the `-compressElts` argument on the command line offline version of the solver.

Diagrams (BDDs) [Bryant 1986]. BDDs provide a canonical representation of boolean functions. Experience has shown that this representation is very compact for very large Boolean functions. Furthermore, the effectiveness of operations over BDDs is notably well-known in the area of formal verification of systems [Edmund M. Clarke *et al.* 1999], in the context of simpler (less expressive) modal logics like $\mathcal{K}$ [Pan *et al.* 2006], and even in the context of much more complex problems that can be reduced to μ-calculus satisfiability testing, such as the problem of automatically detecting the impacts of a schema change on a regular query [Genevès *et al.* 2009]. Here again, the use of BDDs constitutes one of the major reasons why our approach performs well in practice.

Generation of Counter-Examples. The role of the satisfiability-solving algorithm is not limited to the partitioning of the set of logical formulas based on whether they are satisfiable or not: it can in addition generate a sample satisfying tree for satisfiable formulas. Technically, once the formula is found satisfiable at some node, the implementation reconstructs a sample satisfying tree in a top-down manner, starting from the root of the satisfying tree. It actually attempts to generate one of the smallest possible satisfying trees. For that purpose, a pointer to the current state of the set of proved nodes is kept at each step of the fixpoint computation. During the (re)construction of the satisfying tree, smaller proved subtrees are then preferred, resulting in a minimal satisfying tree.

In the context of our type algebra, the validity of a subtyping statement of the form $\tau_1 \leqslant \tau_2$ is checked by testing for the unsatisfiability of $\psi = \neg(\tau_1 \leqslant \tau_2)$. If ψ is unsatisfiable then τ_1 is a subtype of τ_2. If ψ is satisfiable, then the tree satisfying ψ generated by the algorithm represents a counter-example for the relation $\tau_1 \leqslant \tau_2$. Such a sample tree often happens to be of great practical value in order to ease the understanding of the reason(s) why the relation does not hold.

In the polymorphic setting, a counter-example is in principle, according to the extended semantics, a labelled tree. However, as mentioned in Section 3.5.2, whenever a formula is satisfiable there always exists an infinity of possible labellings which satisfy it. Therefore, rather than proposing just one labelled tree, the solver gives a minimal tree together with *labelling constraints* representing all labellings which make that particular tree a counter-example. Namely, for each variable α, every node will be labelled with α to indicate that it *must* be labelled with α for the formula to be satisfied, with $\neg\alpha$ to indicate that it *must not* be, or with nothing if label α is irrelevant for that particular node. This allows an easier interpretation of the counter-example in terms of assignments: the subtyping relation fails whenever the assignment for each variable α contains all the trees whose root is marked with α and none of those whose root is marked with $\neg\alpha$.

3.6.2 Using the Implementation

Our implementation is publicly available. Interaction with the system is offered through a user interface in a web browser. The whole system is available online at:

`http://wam.inrialpes.fr/websolver/`

A screenshot of the interface is given in Figure 3.2. The user can either enter a formula through area (1) of Figure 3.2 or select from pre-loaded analysis tasks offered in area (4) of Figure 3.2. The level of details displayed by the solver can be adjusted in area (2) of Figure 3.2 and makes it possible to inspect logical translations and statistics on problem size and the different operation costs. The results of the analysis are displayed in area (3) of Figure 3.2 together with counter-examples.

Concrete Syntax for Type Algebra. All the examples in the subsection that follows can be tested in our online prototype. For this purpose, the following table gives the correspondence between the syntax used in the chapter and the syntax that must be used in the implementation:

	Syntax	Implementation Syntax
Type variables	α, β, γ	`_a`, `_b`, `_g`
Type constructors	$\times, \rightarrow$	`*`, `->`
Recursive types	$\mu v.\tau$	`let $v = t in $v`
Basic types	$\mathbf{0}, \mathbf{1}$	`F`, `T`
Logical connectives	$\wedge, \vee, \neg, \Rightarrow$	`&`, `|`, `~`, `=>`
Subtyping	$\neg(\tau_1 \leqslant \tau_2)$	`nsubtype(t1,t2)`

Additionally, the embedding of a base formula of the logic into a base type is provided by curly braces: $\{\varphi\}$ is an abbreviation for $\mathsf{isbase} \wedge \langle 1 \rangle\, \varphi$.

3.6.3 Examples and Discussion

The goal of this subsection is to illustrate through some examples how our logical setting is natural and intuitive for proving subtyping relations. For example, one can prove simple properties such as the one below:

$$(\alpha \rightarrow \gamma) \wedge (\beta \rightarrow \gamma) \leqslant (\alpha \vee \beta) \rightarrow \gamma \tag{3.6}$$

This is formulated as follows:

```
nsubtype((_a -> _g) & (_b -> _g), (_a | _b) -> _g)
```

which is automatically compiled into the logical formula shown on Figure 3.3 and given to the satisfiability solver that returns:

```
Formula is unsatisfiable [16 ms].
```

which means that no satisfying tree was found for the formula, or, in other terms, that the negation of the formula is valid. The satisfiability solver is seen as a theorem prover since its run built a formal proof that property (3.6) holds.

webinterface

http://wam.inrialpes.fr/web-solver/webinterface.html

XML Reasoning Solver Project

Home | Demo | Publications | Team

Enter your formula below:

(1)

```
select("descendant::switch[ancestor::head]/descendant::seq
/descendant::audio[preceding-sibling::video]")
```

(4)

Select from these examples:

- XPath Satisfiability #1
- XPath Satisfiability #2
- XPath Containment #2
- XPath Equivalence
- Sample mu-formula with data-values
- XHTML Type Evolution
- MathML Query Evolution

Look at the user manual

(2)

Advanced Options | Check Satisfiability | Execution completed.

- XML Attributes
- Show Lean
- Show Formula
- Formula Statistics

(3)

```
Formula parsed and compiled [total time: 1 ms].

Computing Relevant Closure...
Computed Relevant Closure [0 ms].
Computed Lean [0 ms].
Lean size is 23. It contains 16 eventualities and 7 symbols.
Computing Fixpoint.....[2 ms].

Formula is satisfiable [total time: 4 ms].
A satisfying finite binary tree model is [2 ms]:
head(switch(seq(video(#, audio), #), #), #)
In XML syntax:
<head xmlns:solver="http://wam.inrialpes.fr/xml" solver:context="true">
 <switch>
  <seq>
   <video/>
   <audio solver:target="true"/>
  </seq>
 </switch>
</head>
```

This online demo is a 100% Java implementation of the solver that runs inside a Tomcat servlet. It is based on a thread-safe re-implemention of a BDD package (JavaBDD). However, the performance of this package is very slow compared to what can be achieved with an off-line solver implementation with native BDDs. Ask us if you are interested in the high-speed off-line version of the solver.

Figure 3.2: Screenshot of the Web-Based Interface.

```
(mu X8.(((
(let_mu
  X5=(((BASE & <1>(mu X4.(((~(<1>T) | <1>X4) & (~(<2>T) | <2>X4))
     & (~(ERROR) & ~(BASE) & ~(FUNCTION) & ~(PAIR)))))
       | (PAIR & <1>(X5 & <2>(X5 & ~(<2>T)))))
   | (FUNCTION & (~(<1>T) | <1>X6))),
  X6=(((~(<2>T) | <2>X6) & PAIR)
       & <1>(X5 & <2>((X5 | (ERROR & ~(<1>T))) & ~(<2>T))))
in
  X5) & ((FUNCTION & (~(<1>T) | <1>(mu X1.((~(<2>T)
         | <2>X1) & <1>(~(_a) | <2>_g)))))
      & (FUNCTION & (~(<1>T) | <1>(mu X2.((~(<2>T)
         | <2>X2) & <1>(~(_b) | <2>_g)))))))
       & (~(FUNCTION) | (<1>T & (~(<1>T)
         | <1>(mu X7.((<2>T & (~(<2>T) | <2>X7))
         | (~(<1>T) | <1>((_a | _b)
                  & (~(<2>T) | <2>~(_g))))))))))
| (<1>X8 | <2>X8)))
```

Figure 3.3: Logical translation tested for satisfiability.

Lists. Jerôme Vouillon [Vouillon 2006] uses simple examples with lists to illustrate polymorphism with recursive types. For instance, consider the type of lists of elements of type α:

$$\tau_{\text{list}} = \mu v.(\alpha \times v) \vee \text{nil}$$

where "nil" is a singleton type. The type of lists of an even number of such elements can be written as:

$$\tau_{\text{even}} = \mu v.(\alpha \times (\alpha \times v)) \vee \text{nil}$$

By giving the following formula to the solver:

```
nsubtype(let $v = (_a * _a * $v) | {nil} in $v,
               let $w= (_a * $w) | {nil} in $w )
```

which is found unsatisfiable, we prove that

$$\tau_{\text{even}} \leqslant \tau_{\text{list}}$$

If we now consider the type of lists of an odd number of elements of type α:

$$\tau_{\text{odd}} = \mu v.(\alpha \times (\alpha \times v)) \vee (\alpha \times \text{nil})$$

we can check additional properties in a similar manner, like:

$$(\tau_{\text{even}} \vee \tau_{\text{odd}} \leqslant \tau_{\text{list}}) \wedge (\tau_{\text{list}} \leqslant \tau_{\text{even}} \vee \tau_{\text{odd}})$$

The following formula corresponds to the example (3.1) of the introduction:

```
bool() = {true|false};
list() = let $l = (_a * $l) | {nil} in $l;
odd()  = let $o = (_a * _a * $o) | (_a * {nil}) in $o;
even() = let $e = (_a * _a * $e) | {nil} in $e;

nsubtype ( (odd() -> {true}) & (even() -> {false}),
           list() -> bool() )
```

This formula is found unsatisfiable by the solver, which proves the validity of the subtyping statement (3.1).

Hints about non-trivial relations. Giuseppe Castagna (see section 2.7 of [Castagna & Xu 2011]) gives some examples of non-trivial relations that hold in the type algebra. For instance, the reader can check that the types $\mathbf{1} \to \mathbf{0}$ and $\mathbf{0} \to \mathbf{1}$ can be seen as extrema among the function types:

$$\mathbf{1} \to \mathbf{0} \leqslant \alpha \to \beta \quad \text{and} \quad \alpha \to \beta \leqslant \mathbf{0} \to \mathbf{1}$$

Our system also permitted to detect an error in [Castagna & Xu 2011] and provided some helpful information to the authors of [Castagna & Xu 2011] in order to find the origin of the error and make corrections. Specifically, in a former version of [Castagna & Xu 2011], the following relation was considered:

$$(\neg\alpha \to \beta) \leqslant ((\mathbf{1} \to \mathbf{0}) \to \beta) \vee \alpha \tag{3.7}$$

Authors explained how this relation was proved by their algorithm. However, by encoding the relation in our system we found that this relation actually does not hold. Specifically, this is formulated as follows in our system:

```
nsubtype (~_a -> _b, ((T -> F) -> _b) | _a)
```

The satisfiability solver, when fed this formula, returns the following counter-example:

```
FUNCTION ~_a (PAIR(FUNCTION _a (#,  ~_b ERROR), #), #)
```

`FUNCTION` represents $(\to)$ and `PAIR` represents $(\times)$. This is a binary tree representation of the n-ary tree

$$(\to)_{\neg\alpha}[(\times)[(\to)_{\alpha}[\varepsilon] :: \Omega :: \varepsilon] :: \varepsilon]$$

which corresponds to the domain element

$$\{(\{\}_{\alpha}, \Omega)\}_{\neg\alpha}.$$

The inner $(\to)$ node has no children and thus represents the function which always diverges: $\{\}$. More precisely, it represents *a* copy f of this function that belongs to

the interpretation of α. The root ($\rightarrow$) node then represents a function which is *not* in $[\![\alpha]\!]$ and which to f associates an error, while diverging on any other input.

Now, why is it a counter-example to (3.7)? As the function diverges but on one input f and that input is in $[\![\alpha]\!]$, it is vacuously true that on all inputs in $[\![\neg\alpha]\!]$ for which it returns a result, this result is in $[\![\beta]\!]$. Thus it does have the type on the left-hand side. However, it does not have type α, nor does it have type $((\mathbf{1} \rightarrow \mathbf{0}) \rightarrow \beta)$. Indeed, f does have type $\mathbf{1} \rightarrow \mathbf{0}$ and our counter-example function associates to it an error, which is not in $[\![\beta]\!]$.

Big XML Types. One purpose of the next example is to illustrate how the system can handle large types. We consider the tree grammars that define the admissible structures of webpages conforming to the XHTML Basic 1.0 and XHTML Basic 1.1 specifications. We formulate a very simple example to check that the type of a function that modifies XHTML Basic 1.0 documents while preserving their validity is a subtype of the type of a more general transformation that produces XHTML Basic 1.1 output from XHTML Basic 1.0 input documents.

The formula given to the solver is the following[3]:

```
nsubtype({type("xhtml-basic10.dtd","html")}
           ->{type("xhtml-basic10.dtd","html")}
         ,
         {type("xhtml-basic10.dtd","html")}
           ->{type("xhtml-basic11.dtd", "html")})
```

The system first triggers the parsing of the two real world tree grammars whose sizes are significant: xhtml-basic10.dtd contains 52 tag names and 71 type variables, whereas xhtml-basic11.dtd contains 67 tag names and 89 type variables. Those grammars are then linearly compiled into logical formulas in order to compose the global logical formula. The entire expansion of this global formula is huge: it contains 1461 atomic propositions, 7150 modalities, 2410 variables, 8 (n-ary) fixpoint binders, 3715 negations, 2126 conjunctions and 3226 disjunctions. Those numbers include duplicate subformulas that are factorized by our lean set representation, as described in Section 3.6.1. The intermediate results are as follow:

```
Input parsed and compiled [total time: 1562 ms].
Computing Relevant Closure...
Computed Relevant Closure [533 ms].
Computed Lean [2 ms].
Lean size is 201. It contains 192 modal formulas and 9 symbols.
```

[3]The command line instruction to reproduce this test with the offline version of the solver is the following: `java -Xmx5g -Xms2g -Xmx2g -jar solver.jar fleche.txt -compressElts -stats` where the first three arguments increase the default heap size of the java virtual machine, "`fleche.txt`" is a text file containing the logical formula, "`-compressElts`" indicates that the binary encoding of symbols must be used in order to reduce the lean size, and "`-stats`" displays some formula statistics like e.g. the number of different connectives in the logical formula.

This means that the global search universe in which a tree is automatically looked for is composed of 2^{201} distinct tree nodes! Nevertheless, the fixpoint is computed in less than 18 seconds:

```
Fixpoint Computation Initialized [149 ms].
Computing Fixpoint...........[16542 ms].
Formula is unsatisfiable [17229 ms].
```

3.7 Related Work

We review below related works while recalling how the introduction of XML progressively renewed the interests in parametric polymorphism.

The seminal work by Hosoya, Vouillon and Pierce on a type system for XML [Hosoya *et al.* 2005] applied the theory of regular expression types and finite tree automata in the context of XML. The resulting language XDuce [Hosoya & Pierce 2003] is a strongly typed language featuring recursive, product, intersection, union, and complement types. The subtyping relation is decided through a reduction to containment of finite tree automata, which is known to be in EXPTIME. This work does not support function types nor polymorphism, but provided a ground for further research.

In particular, Frisch, Castagna and Benzaken provide a gentle introduction to semantic subtyping in [Frisch *et al.* 2008]. Semantic subtyping focuses on a set-theoretic interpretation, as opposed to traditional subtyping through direct syntactic rules. Our logical modeling presented in Section 3.4 naturally follows the semantic subtyping approach as the underlying logic has a set-theoretic semantics. Frisch, Castagna and Benzaken added function types to the semantic subtyping performed by XDuce's type system. This notably resulted in the CDuce language [Benzaken *et al.* 2003]. However, CDuce does not support type variables and thus lacks polymorphism.

Vouillon studied polymorphism in the context of regular types with arrow types in [Vouillon 2006]. Specifically, he introduced a pattern algebra and a subtyping relation defined by a set of syntactic inference rules. A semantic interpretation of subtyping is given by ground substitution of variables in patterns. The type algebra has the union connective but lacks negation and intersection. The resulting type system is thus less general than ours.

Polymorphism was also the focus of the later work found in [Hosoya *et al.* 2009]. In [Castagna & Xu 2011], it is explained that at that time a semantically defined polymorphic subtyping looked out of reach, even in the restrictive setting of [Hosoya & Pierce 2003], which did not account for higher-order functions. This is why [Hosoya *et al.* 2009] fell back on a somewhat syntactic approach linked to pattern-matching that seemed difficult to extend to higher-order functions. Our work shows however that such an extension was possible using similar basic ideas, only slightly more abstract.

The most closely related work is the one found in [Castagna & Xu 2011], which solves the problem of defining subtyping semantically in the polymorphic case for the first time, and addresses the problem of its decision through an ad-hoc and multi-step algorithm, which was only recently proved to terminate in all cases. Our approach

also addresses the problem of deciding their subtyping relation and solves it through a more direct, generic, natural and extensible approach since our solution relies on a modeling into a well-known modal logic (the μ-calculus) and on using a satisfiability solver such as the one proposed in [Genevès *et al.* 2007]. This logical connection also opens the way for extending polymorphic types with several features found in modal logics.

The work of [Bierman *et al.* 2010] follows the same spirit than ours: typechecking is subcontracted to an external logical solver. An SMT-solver is used to extend a type-checker for the language Dminor (a core dialect for M) with refinement type and type-tests. The type-checking relies on a semantic subtyping interpretation but neither function types nor polymorphism are considered. Therefore, their work is incomparable to ours.

The present work heavily relies on the work presented in [Genevès *et al.* 2007] since we repurpose the satisfiability-checking algorithm of [Genevès *et al.* 2007] for deciding the subtyping relation. The goal pursued in [Genevès *et al.* 2007] was very different in spirit: the goal was to decide containment of XPath queries in the presence of regular tree types. To this end, the decidability of a logic with converse for finite ordered trees is proved in a time complexity which is a simple exponential of the size of the formula. The present work builds on these results for solving semantic subtyping in the polymorphic case.

3.8 Conclusion

The main contribution of this chapter is to define a logical encoding of the subtyping relation defined in [Castagna & Xu 2011], yielding a decision algorithm for it. We prove that this relation is decidable with an upper-bound time complexity of $2^{(n)}$, where n is the size of types being checked. In addition, we provide an effective implementation of the decision procedure that works well in practice.

This work illustrates a tight integration between a functional language type-checker and a logical solver. The type-checker uses the logical solver for deciding subtyping, which in turn provides counter-examples (whenever subtyping does not hold) to the type-checker. These counterexamples are valuable for programmers as they represent evidence that the relation does not hold. As a result, our solver represents a very attractive back-end for functional programming languages type-checkers.

This result pushes the integration between programming languages and logical solvers to an advanced level. The proposed logical approach is not only capable of modeling higher order functions, but it is also capable of expressing values from semantic domains that correspond to monadic second-order logics such as XML tree types. This shows that such logical solvers can become the core of XML-centric functional languages type-checkers such as, e.g., those used in CDuce or XDuce.

Chapter 4

Analysis of Cascading Style Sheets

Contents

4.1 Introduction 74
4.2 Current Practice 75
4.3 CSS: An Overview 76
4.4 Theoretical Foundations 79
4.5 A Logical Modeling of CSS 83
4.6 Prototype Implementation 86
4.7 Reasoning with Style 86
4.8 Automated CSS Size Reduction 92
4.9 Conclusions 97

Abstract

Cascading Style Sheets (CSS) is a standard language for stylizing and formatting web documents. It plays an increasingly important role in web user experience. Developing and maintaining cascading style sheets (CSS) is an important issue to web developers as they suffer from the lack of rigorous methods. Most existing means rely on validators that check syntactic rules, and on runtime debuggers that check the behavior of a CSS style sheet on a particular document instance. However, the aim of most style sheets is to be applied to an entire set of documents, usually defined by some schema. To this end, a CSS style sheet is usually written w.r.t. a given schema. While usual debugging tools help reducing the number of bugs, they do not ultimately allow to prove properties over the whole set of documents to which the style sheet is intended to be applied.

We propose a novel approach to fill this lack. We present an original tool based on our recent advances in tree logics. The tool is capable of statically detecting a wide range of errors (such as empty CSS selectors and semantically equivalent selectors), as well as proving properties related to sets of documents (such as coverage of styling information), in the presence or absence of schema information. This new tool can be used in addition to existing runtime debuggers to ensure a higher level of quality of CSS style sheets.

In addition, we present a first prototype of static CSS semantical optimizer that is capable of automatically detecting and removing redundant property declarations and rules. Existing purely syntactic CSS optimizers might be used in conjunction with our tool, for performing complementary (and orthogonal) size reduction, toward the common goal of providing cleaner, lighter, and easily debuggable CSS files.

4.1 Introduction

"Style sheet languages are terribly under-researched" [Marden & Munson 1999]. This statement dates back from 1999, but it is still true. However, Cascading Style Sheets (CSS) [Lie 2005] was the first feature that was added to the initial foundations of the web (HTML, HTTP and URLs). While style has become a key component of web user experience, development tools for style sheets have involved very little basic research. As a result, empirical methods are the only means available to web developers for implementing and maintaining style sheets.

The research presented in this chapter addresses the issue of debugging CSS style sheets. At first glance, CSS appears to be a simple language, and from a syntactical perspective, it really is. Basically, a style sheet is simply a sequence of style rules. Each rule has a selector that specifies elements of interest in the document structure, and provides a value for a style property. The value is assigned to the corresponding property for all elements specified by the selector.

This apparent simplicity is contradicted by a number of combinatorial aspects, which bring a significant power to the CSS language, while making it a bit more complex. Style rules can be grouped to share the same selector, for specifying different properties that apply to the same elements. Style rules are also grouped by style sheets, and several style sheets may apply to a single document. A style sheet is usually external to the document it applies to, but it may also be embedded in the document, with the `style` element of HTML. Finally, several style rules may also be embedded within an element in a document with the `style` attribute. In addition, the same style property may appear several times in all these locations. The cascade sets the priority between several rules specifying the same property for the same elements.

As a consequence, when a style sheet does not work the way it was intended, it is very difficult to locate the origin of the problem. For this reason, the issue of debugging and maintaining style sheets is important to web developers. In this chapter, we propose a novel approach to this issue, based on recent advances in theoretical tools that handle XML structures and query languages for these structures.

The chapter is organized as follows. The next section reviews the methods and tools that web developers currently use to debug CSS style sheets. It is followed by an overview of the main features of CSS. The theoretical foundations on which the rest of the chapter is based are then summarized. This is mainly a tree logic that is used in section 4.5 for modeling CSS style sheets. Based on this model, section 4.6 presents a software tool for the static analysis of style sheets which is illustrated by typical applications and examples. The chapter closes with some perspectives.

4.2 Current Practice

Developers use basically two kinds of tools to find errors in CSS style sheets: validators and debuggers.

Validators address only syntactic issues. They check that a style sheet strictly follows the CSS grammar. These tools perform *static* checking: they analyze a style sheet for itself, independently of any web page to which it could be applied. A typical example of this family is the W3C CSS validator.[1] While they are useful, validators do not address the difficult issue of locating rules that do not behave as expected.

As opposed to validators, debuggers are *dynamic* tools. They are coupled with a formatting engine that executes style sheets by applying them to web pages and displaying the result. They allow the user to see how the formatter applies style rules to the tested documents. All modern web browsers now include debuggers, such as Firebug (Firefox), Developer Toolbar (Internet Explorer), Dragonfly (Opera), or Web Inspector (Safari).

These tools do not address only style sheets. They deal with the many facets of a web page (DOM tree, scripts, style) [Barton & Odvarko 2010], but they constitute the primary tool to debug style sheets. They help CSS debugging by providing a list of all style rules that apply to any element chosen by the user. All rules are displayed and any rule overridden by another through the cascade is struck through, thus helping developers to understand what style rules really apply to the chosen element. The origin of each rule (style sheet, style element, style attribute) is also presented. Rules can often be changed on the fly to quickly test alternative solutions.

Performances may be another issue. With complex style sheets, formatting may take some time. A tool such as the YSlow add-on for Mozilla may help to find performance issues, but it also addresses other aspects of performances in web pages, such as HTML and Javascript.

Other tools target CSS selectors specifically. Dust-Me Selectors, for instance, detects unused selectors dynamically, on a single HTML page or on a whole site.

Debugging style sheets after they have been written is not the only way to improve their quality. It could be done also at writing time. Two approaches are possible: generating style sheets automatically from some higher-level specification [Keller & Nussbaumer 2009] [Keller & Nussbaumer 2010] [Serrano 2010], and including debugging features in a CSS editor [Quint & Vatton 2007]. In the first case, the automatic tool is expected to generate bug-free style sheets, but the issue of debugging the higher-level specification remains. In the second case, the author gets assistance at the moment of creating the style rules, which helps her to create better style sheets.

To summarize, validators are the only tools available today that perform static analysis of a style sheet. The errors they report may potentially affect any web page the style sheet is applied to, and if they detect no errors, developers are sure that the style sheet will not have any syntactic issue whatever the page it is applied to.

Unfortunately, syntactic issues are only a small part of the debugging problem. To

[1] see http://jigsaw.w3.org/css-validator/

address the other issues, developers have only dynamic tools at their disposal. To get some confidence in their style sheets, they have to use these tools on a number of pages, but they can never get any complete assurance that these style sheets will not fail on some other page. The process is both painful and unsatisfactory. We believe that static analysis of the content of style sheets (not only their syntax) could considerably help developers in detecting errors and proving properties that are expected from style sheets, whatever the document they are applied to.

We have then developed a tool for the static analysis of CSS. After a brief review of the main features of CSS, we present a logical framework for modeling structured documents and selection of information in them, we show how CSS can be modeled in this logic, and we describe the tool based on this model.

4.3 CSS: An Overview

A style sheet C can be seen as a set R of rules, composed of simple rules R_i each composed of a single selector S_i and a set of pairs, each made of a property P_i and its value V_i. Selectors define which elements of a document the properties are applied to. Properties and their values define how those elements look like in the browser.

A selector is a chain of one or more sequences of simple selectors separated by combinators. Simple selectors considered here are of two types: the universal selector, noted `*`, and the type selector which is noted by the tag name of a given element, for example `h1`. For simplicity and without loss of generality, we consider that the rules are made of single selectors (the specification allows a comma separated list of selectors) which set a single property at a time (multiple properties are allowed for a given selector). It is easy to rewrite multiple selectors and property rules to a set of single selector and single property rules.

Selectors S_i, sometimes called patterns in the CSS specification [Çelik *et al.* 2011], define boolean functions of the form:

$$expression \times element \rightarrow boolean$$

that express whether or not a given element is selected by the selector expression.

In the following, we explore the main vehicle for setting CSS properties on document elements, namely combinators, structural pseudo-classes, and property inheritance.

4.3.1 Combinators

CSS combinators define relations between elements of a document. In CSS3, they come in three variants according to the specification:

- Descendant combinator: a descendant combinator describes a descendant relationship between two elements. A descendant combinator is made of the whitespace sign, for example “body p”.

- Child combinator: a child combinator describes a childhood relationship between two elements. This combinator is made of the > sign, for example "body ¿ p".
- Sibling combinator: there are two different sibling combinators, the adjacent sibling combinator and the general sibling combinator. They are noted with the + and ˜ signs respectively.

4.3.2 Structural pseudo-classes

Structural pseudo-classes permit to select elements based on positional information in the document tree. This positional information is based on calculating the position (via an index on sibling elements) of an element relatively to its parent. There are several pseudo-classes in the specification; we present just a few of them here. The others are similar with additional constraints on element types:

- :root pseudo-class: It represents an element that is the root of the document. In HTML 4, this is always the `html` element.
- :first-child and :last-child pseudo-classes: They represent an element that is the first child or the last-child of some other element respectively.
- :nth-child() pseudo-class : The :nth-child($an+b$) pseudo-class notation represents an element that has $an + b - 1$ siblings before it in the document tree, for any positive integer or zero value of n, and has a parent element.
- :nth-last-child(): The :nth-last-child($an + b$) pseudo-class notation represents an element that has $an+b-1$ siblings after it in the document tree, for any positive integer or zero value of n, and has a parent element.

Other pseudo-classes are defined in the specification based on both the element type and position. Examples are :first-of-type, :last-of-type and :only-of-type pseudo-classes.

The positional pseudo-classes are very useful to set properties (like foreground and background colors, or fonts) in HTML structures such as tables. The following example alternates four colors (zebra striping when two) in table rows:

```
tr:nth-of-type(4n+1) {color: navy;}
tr:nth-of-type(4n+2) {color: green;}
tr:nth-of-type(4n+3) {color: maroon;}
tr:nth-of-type(4n+4) {color: purple;}
```

4.3.3 CSS properties and inheritance

CSS inheritance works on a property by property basis. The mechanism for assigning a value to each property for each element is based on the following steps, in order of precedence. If the cascade results in a value, this value is used. Otherwise, if the property is defined by the specification as inherited and the element is not the

root of the document tree, the value of the property of the parent element is used (this situation also corresponds to a property with value `inherit`). Otherwise, the property's initial value is used.

The initial value is specific to each property and is indicated by the specification. The initial value for many properties is already `inherit`, and for most others (`border` for instance), inheriting the parent element's value is obviously not desirable. The allowed values for properties, their initial value, and whether they are inherited or not are summarized in the property table of the specification [Bos *et al.* 2011].

For example, with this style sheet and this HTML fragment:

```
div { background-color: white;
      color: blue;
      font-weight: normal; }
p {  background-color: inherit;
     color: inherit; }

<div>
  <p>
    Hello, world.
  </p>
</div>
```

the background color of the `div` element is set to white. The background color of the paragraph is also white, because its `background-color` property is set to `inherit` and the background color of the `div` parent element is white.

The `inherit` value does not require that the parent element have the same property set explicitly; it works from the computed value. In the above example, the `color` property of the paragraph has value `inherit`, but the computed value is `blue` because it inherits. The `font-weight` property of the `p` element is also set to `normal` since it is inherited by default.

When two selectors select the same element for a given property, the more "specific" one gets precedence. Specificity of selectors consists in counting a four integer vector corresponding to (1) whether the property is specified in a `style` attribute of not, (2) the number of `id` attributes in the selector, (3) the number of other attributes and pseudo-classes in the selector, (4) the number of element names in the selector. In our case, since we consider analyzing style properties on a possibly infinite set of HTML documents, we consider that selectors specificity is defined by the last integer corresponding to the number of element names. For example:

```
 *             {}  specificity = 0,0,0,0 */
 li            {}  specificity = 0,0,0,1 */
 ul li         {}  specificity = 0,0,0,2 */
 ul ol+li      {}  specificity = 0,0,0,3 */
```

Since specificity can be easily and statically computed before analysis, we consider that the corresponding number is provided for each selector by a function `Specificity`(S_i).

4.4 Theoretical Foundations

In this section, we present the static analysis technology on which our tool is based, which relies on automated verification of properties that are expressed as logical formulas over trees.

4.4.1 Approach overview

We use a tree logic capable of capturing the semantics of CSS selectors as well as schemas. Schemas we consider are regular tree grammars which capture most of the XML Schemas, Relax NG schemas, and DTDs. Our approach consists in modeling element selection performed by CSS selectors and structural constraints described by schema information into the tree logic. We then use an algorithm to check satisfiability of formulas of the logic. Such an algorithm defines a partition of the set of logical formulas: satisfiable formulas (for which there exist at least one tree, among those defined by the schema, that satisfies the constraints expressed by the formula) and remaining formulas which are unsatisfiable (no tree satisfies the formula). Alternatively (and equivalently), formulas can be divided into valid formulas (formulas which are satisfied by all trees) and invalid formulas (formulas that are not satisfied by at least one tree). The use of a satisfiability-testing algorithm allows proving validity of a given logical statement P by testing its negation ($\neg P$) for unsatisfiability.

In the sequel, we progressively introduce the tree logic and explain how it captures schemas and CSS selectors. We first present the data model of the logic and then we introduce the syntax of logical formulas through examples.

4.4.2 Data model

A document is considered as a finite tree of unbounded depth and arity, with two kinds of nodes respectively named elements and attributes. In such a tree, an element may have any number of children elements, and may carry zero, one or more attributes. Attributes are leaves with a value. Elements are ordered whereas attributes are not, as illustrated on Figure 4.1. The logic allows reasoning on such trees.

```
<r c="␣" a="␣" b="␣">
 <s d="␣">
  <v/><w/><x e="␣"/>
 </s>
 <t/>
 <u/>
</r>
```

XML Notation

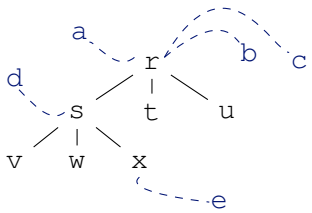


Figure 4.1: Sample XML Tree with Attributes.

Unranked and binary trees There exist bijective encodings between unranked trees (trees of unbounded arity) and binary trees. Owing to these encodings binary trees may be used instead of unranked trees without loss of generality. The logic operates on binary trees. The logic relies on the "first-child & next-sibling" encoding of unranked trees. In this encoding, the first child of a node is preserved in the binary tree representation, whereas siblings of this node are appended as right successors in the binary representation. The intuition of this encoding is illustrated on Figure 4.2 for a sample tree.

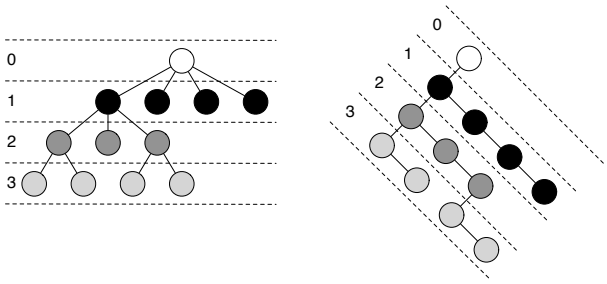


Figure 4.2: Binary encoding principle.

In the remaining of this chapter, the binary representation of a tree is implicitly considered, unless stated otherwise. From an XML point of view, notice that only the nested structure of XML elements (which are ordered) is encoded into a binary form like this. XML attributes (which are unordered) are left unchanged by this encoding. For instance, Figure 4.3 presents how the sample tree of Figure 4.1 is mapped.

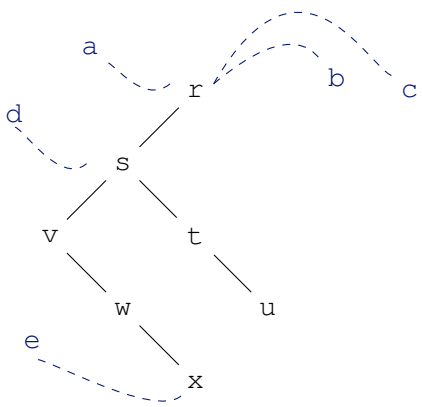


Figure 4.3: Binary encoding of tree of figure 4.1

4.4.3 A gentle introduction to tree logic

Navigating in trees with modalities The logic uses two *programs* for navigating in binary trees: the program `1` for navigating from a node down to its first successor and the program `2` for navigating from a node down to its second successor. The logic also features *converse programs* `-1` and `-2` for navigating upward in binary trees, respectively from the first and second successors to the parent node. Some basic logical formulas together with corresponding satisfying binary trees are shown in Figure 4.4.

Sample Formula	Satisfying Tree (focus in bold)	In XML

`a & <1>b`	**a**
	/
	b
		`<a>`
		`<b/>`
		`</a>`

`a & <2>b`	**a** \ b	`<a/><b/>`

`a & <1>(b & <2>c)`	**a**
	/
	b
	\
	c
		`<a>`
		`<b/><c/>`
		`</a>`

`e & <-1>(d&<2>g)`	d
	/ \
	e g
		`<d>`
		`<e/>`
		`</d><g/>`

`f & <-2>(g&~<2>T)`	none	none

Figure 4.4: Sample formulas using modalities.

The set of logical formulas is defined by the syntax given on Figure 4.5, where the meta-syntax $\langle X \rangle^{\oplus}$ means one or more occurrences of X separated by commas. Models of a formula are finite binary trees for which the formula is satisfied at some node. The semantics of logical formulas is formally defined in [Genevès 2006, Genevès *et al.* 2007]. Figure 4.4 gives basic formulas that use modalities for navigating in binary trees and node names, as well as sample satisfying trees in binary and XML notation.

Recursive formulas The logic allows expressing recursion in trees through the use of a fixpoint operator. For example the recursive formula:

```
let $X = b | <2>$X in $X
```

means that either the current node is named `b` or there is a sibling of the current node which is named `b`. For this purpose, the variable `$X` is bound to the subformula `b | <2>$X` which contains an occurrence of `$X` (therefore defining the recursion). The scope of this binding is the subformula that follows the `in` symbol of the formula, that is `$X`. The entire formula can thus be seen as a compact recursive notation for a

φ	::=		formula
		`T`	true
	|	`F`	false
	|	l	element name
	|	p	atomic proposition
	|	φ `|` φ	disjunction
	|	φ `&` φ	conjunction
	|	φ `=>` φ	implication
	|	φ `<=>` φ	equivalence
	|	`(`φ`)`	parenthesized formula
	|	`~`φ	negation
	|	`<`p`>`φ	existential modality
	|	`<`l`>T`	attribute named l
	|	`<`l`>'`v`'`	attribute l with value 'v'
	|	`$X`	variable
	|	`let` $\langle$`$X` $= \varphi\rangle^{\oplus}$ `in` φ	binder for recursion
p	::=		program inside modalities
		`1`	first child
	|	`2`	next sibling
	|	`-1`	parent
	|	`-2`	previous sibling

Figure 4.5: Syntax of logical formulas

infinitely nested formula of the form:

```
b | <2>(b | <2>(b | <2>(...)))
```

Recursion allows expressing global properties. For instance, the recursive formula:

```
~ let $X = a | <1>$X | <2>$X in $X
```

expresses the absence of nodes named `a` in the whole subtree of the current node (including the current node). Furthermore, the fixpoint operator makes possible to bind several variables at a time, which is specifically useful for expressing mutual recursion. For example, the mutually recursive formula:

```
let $X = (a & <2>$Y) | <1>$X | <2>$X, $Y = b | <2>$Y in $X
```

asserts that there is a node somewhere in the subtree such that this node is named `a` and it has at least one sibling which is named `b`. Binding several variables at a time provides a very expressive yet succinct notation for expressing mutually recursive structural patterns (that may occur in DTDs for instance).

The combination of modalities and recursion makes the logic one of the most expressive (yet decidable) logic known. For instance, most DTDs and schemas (specifically

regular tree grammars) can be expressed with the logic using recursion and (forward) modalities (see [Genevès 2006] or [Genevès *et al.* 2007] for details). The combination of converse programs and recursion allows expressing properties about previous siblings of a node for instance, which happens to be very useful for capturing the semantics of CSS selectors.

4.5 A Logical Modeling of CSS

4.5.1 Capturing selectors

CSS selectors are systematically translated into the logic: Figure 4.6 shows how the main combinators found in CSS selectors level 3 [Çelik *et al.* 2011] are mapped into their corresponding logical representation. The logical formula holds for elements that are selected by the CSS selector. Figure 4.7 presents how the structural and negation pseudo-classes of CSS level 3 are compiled into logical formulas. We have developed a general compiler that takes a CSS selector as input, systematically applies the translation rules, and outputs the corresponding logical formula. In the remaining part of this chapter, we denote this compiler by a compilation function $\texttt{F}(\cdot)$ so that we can refer to the logical translation of a selector S_i with $\texttt{F}(S_i)$.

For example, the selector S_1 = `ul li:nth-last-of-type(2)` selects any `li` element which is a second sibling of its type, counting from the last one, while being a descendant of some `ul` element. The corresponding logical formula is built in two steps. First, the translation of the descendant combinator (shown in Figure 4.6) is instantiated with the appropriate parameters `ul` and `li:nth-last-of-type(2)`, therefore the logical translation $\texttt{F}(S_1)$ is as follows:

$$\varphi \texttt{ \& let \$X= <-1>(}\psi \texttt{ | \$X) | <-2>\$X in \$X}$$

where $\varphi = \texttt{F(li:nth-last-of-type(2))}$ and $\psi = \texttt{F(ul)}$. As a second step, φ and ψ are computed:

- $\varphi =$ `li & let $X= <2>(li & ~let $Y=<2>li|<2>$Y in $Y) |<2>$X in $X` (see f_8 in Figure 4.7);
- $\psi =$ `ul` (see Figure 4.6).

Notice that the class attribute which very frequently used in style sheets is simply translated as an ordinary attribute (see class `foo` Figure 4.6).

4.5.2 Capturing Properties

In order to capture CSS properties, we consider that all elements in a schema, in HTML in particular, are augmented with the entire set of CSS properties encoded as attributes. For example, the following rule:

```
ul li:nth-last-of-type(2) {color:  green;}
```

Semantics	CSS	Tree Logic
Any element	`*`	`T`
Any 'p' element	`p`	`p`
Any child of some p element	`p > *`	`let $X= <-1>p \| <-2>$X in $X`
Any descendant 'b' of some 'a'	`a b`	`b&let $X=<-1>(a\|$X)|<-2>$X in $X`
Any element with class 'foo'	`.foo`	`<class>'foo'`
Any element with attribute 'title'	`*[title]`	`<title>T`
Any 'p' element with an 'a' child	Not possible	`p&<1>let $X= a \| <2>$X in $X`
Any adjacent next sibling of a 'p'	`p + *`	`<-2>p`
Any next sibling 'p' of a 'h1'	`h1 ~ p`	`p&let $X=<-2>h1\|<-2>$X in $X`
Any 'e' with 'foo' attr. value 'bar'	`e[foo="bar"]`	`e & <foo>bar`

Figure 4.6: Main CSS combinators and corresponding logical formulas.

is translated as `F`(S_1) `& <css:color>'green'` in the logic, with `F`(S_1) computed as explained above.

4.5.3 Capturing Inheritance

The CSS property value `inherit` is a very particular value which is not related to style, but instead it indicates how the property value must be computed. Specifically, a computed value $v \neq$ `inherit` is *obtained* for a property p at a given element iff:

- value of p is explicitly set to v at the given element (intuitively this has been set by some custom selector);
- value of p is not explicitly set to v at the given element, it is not set to `inherit` either at that element, but the initial value for this property is v;
- value of p is set to `inherit` at that element, the given element is the root, the initial value for this property happens to be v.
- value of p is set to `inherit` at that element, the given element is not the root, and value v is obtained for the parent element (by applying this case analysis recursively);

We model this inheritance mechanism for propagating values in logical terms. We introduce a predicate that logically describes each of those possible cases.

Semantics	CSS	Tree Logic
An 'e' element, root of the document	`e:root`	f_0
Any first child of a 'p' element	`p > *:first-child`	f_1
Any 'li' element, last child of a 'ol' element	`ol > li:last-child`	f_2
Any odd row of an HTML table	`tr:nth-child(odd)`	f_3
Any even row of an HTML table	`tr:nth-child(even)`	f_4
Any 'foo', third child of its parent element	`foo:nth-child(3)`	f_5
Any 'e', 2nd child of its parent, from the last one	`e:nth-last-child(2)`	f_6
Any 'e' element, second sibling among the 'e' 's	`e:nth-of-type(2)`	f_7
Any 'e', 2nd sibling of its type, from the last one	`e:nth-last-of-type(2)`	f_8
Any 'e' element, first sibling of its type	`e:first-of-type`	f_9
Any 'e' element, last sibling of its type	`e:last-of-type`	f_{10}
Any 'e' element, only child of its parent	`e:only-child`	f_{11}
Any 'e' element, only sibling of its type	`e:only-of-type`	f_{12}
Any 'e' element that has no children	`e:empty`	`e & ~<1>T`
Any 'e' element not matching selector s	`e:not(s)`	`e & ~s`

f_0 = `e & ~<-1>T & ~<-2>T`
f_1 = `~<-2>T & let $X=<-1>p|<-2>$X in $X`
f_2 = `~<2>T & li & let $X=<-1>ol|<-2>$X in $X`
f_3 = `tr & let $X=<-1>T|<-2><-2>$X in $X`
f_4 = `tr & let $X=<-2><-1>T|<-2><-2>$X in $X`
f_5 = `foo & <-2><-2>(~<-2>T) & let $X=<-1>T|<-2>$X in $X`
f_6 = `e & <2>(~<2>T) & let $X=<-1>T|<-2>$X in $X`
f_7 = `e & let $X= <-2>(e & ~let $Y=<-2>e|<-2>$Y in $Y) |<-2>$X in $X`
f_8 = `e & let $X= <2>(e & ~let $Y=<2>e|<2>$Y in $Y) |<2>$X in $X`
f_9 = `e & ~let $X=<-2>e|<-2>$X in $X`
f_{10} = `e & ~let $X=<2>e|<2>$X in $X`
f_{11} = `e & ~<2>T & ~<-2>T & let $X=<-1>T|<-2>$X in $X`
f_{12} = `e & ~let $X=<2>e|<2>$X in $X & ~let $Y=<-2>e|<-2>$Y in $Y`

Figure 4.7: Structural and negation pseudo-classes (CSS level 3) and corresponding logical formulas.

The predicate `inherit`(p, v) holds at a given element iff value v is obtained for property p at this element:

```
inherit(p,v) = let $X = <-1>( <p>'v'
                  | ~<p>'v' & ~<p>'inherit' & initialvalue(p,v)
                  | <p>'inherit' & ~<-1>T & ~<-2>T & initialvalue(p,v)
                  | <p>'inherit' & $X )
               | <-2>$X in $X
```

where `initialvalue`(p, v) is a predicate that holds iff property p has initial value v, as defined by the CSS recommendation (see [Bos *et al.* 2011]).

4.6 Prototype Implementation

We present here the tool we have developed based on the logical modeling presented in the previous section. Its architecture is outlined in Figure 4.8. It is composed of a set of parsers for reading the CSS and schema files (XML Schema, Relax NG, or DTD) together with a text file corresponding to problem description as a logical formula. Some compilers are used for translating schemas and CSS files into their logical representations. CSS files are first converted into the simplified form explained in Section 4.3. Then, the solver takes the overall problem formulation and checks it for satisfiability.

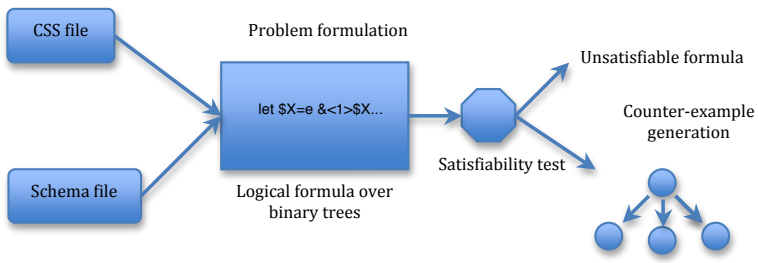


Figure 4.8: Overall architecture

The result of the analysis corresponds to two situations: either the formula is found unsatisfiable (meaning that the checked property holds for any tree), or it is satisfiable. In this case, the solver generates a counter-example document satisfying the formula (described in [Genevès *et al.* 2014]). The tool is available at: http://wam.inrialpes.fr/websolver

4.7 Reasoning with Style

In this section, we present some experiments highlighting how the analyzer works on some typical examples. These examples are simplified in order to make the analysis

easier to understand (but the same kind of analyses can be applied to more complex cases). Notice that users are not asked to type these formulas as generic tests are provided as a set of macros in the tool (see Section 4.7.1). We just detail some of them enough to explain how they work. For the same reasons, we use the simplified HTML DTD shown in Figure 4.9.

```
<!ELEMENT html (head,body)>
<!ELEMENT head (title)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT body ((div|table)*)>
<!ELEMENT table (tbody)>
<!ELEMENT tbody (tr+)>
<!ELEMENT tr (td+)>
<!ELEMENT td (div*)>
<!ELEMENT div (#PCDATA|div)*>
```

Figure 4.9: Simplified HTML DTD.

The first example is the verification of the behavior of a style sheet when it comes to displaying text in different font sizes. Indeed, setting the `font-size` property to the value `inherit` can be error-prone. Specifically, a computed `font-size` value repeatedly obtained by inheritance from a relative value like `80%` or `smaller` may result in tiny or unreadable text. The goal of the test here is to check whether the style sheet may yield such a bad rendering on some documents. This can be expressed logically by the following formula:

```
1. type("html.dtd","html") & ~<-1>T  & ~<-2>T
2. & let $CSS = (div => <font-size>'smaller')
3. & (~div => (~<font-size>T |<font-size>'normal'))
4. & (~<1>T | <1>$CSS) & (~<2>T | <2>$CSS) in $CSS
5. & let $Q = <font-size>'smaller'
   & ancestor(<font-size>'smaller'
   & ancestor(<font-size>'smaller')) | <1>$Q | <2>$Q in $Q
```

This formula is built from the sample style sheet of Figure 4.11. The first line allows translating the simplified schema of Figure 4.9 into a logical formula (omitted here). Notice that `~<-1>T & ~<-2>T` means that the element has no parent nor a previous sibling, i.e. it is the root element. Line 2 represents the logical counterpart of CSS rule 6 of the sample CSS of Figure 4.11. It corresponds to the logical implication "if an element is labeled `div` then the value of its `font-size` property must be `smaller`". Line 3 states that any other element than `div` (elements that are not concerned by the previous rules) may either carry no `font-size` property or if they do, then the value for `font-size` is set to `normal`. This models the default behavior of CSS for property `font-size` which is overridden by the rules for `div`. Line 4 in is charge of applying the style information to every element in the document.

Thus, lines 1 to 4 restrict the considered set of documents to those that are valid with respect to the DTD, and that have the style information defined by the style

sheet.

Now, line 5 formulates the question: "*may the application of my style sheet render some valid document with unreadable text due to font-size too small because of a triple application of the relative font-size value 'smaller'?*" In this example, the predicate `inherit`(p, v) is simply substituted by the simpler ancestor predicate. When fed with this formula, the logical solver explores all possible situations and produces the following counter-example (which is displayed in the browser as the picture on the left in Figure 4.10):

```
<html xmlns:solver="http://wam.inrialpes.fr/xml">
 <head>
  <title/>
 </head>
 <body>Body text
  <div font-size="smaller">text in first level of div
    <div font-size="smaller">text in second level of nested div
      <div font-size="smaller">text in third level of nested
       div </div>
    </div>
  </div>
 </body>
</html>
```

Notice that if we add the following rule

```
div div div {font-size:medium;}
```

in order to fix the style sheet, then the solver cannot find any counter-example anymore.

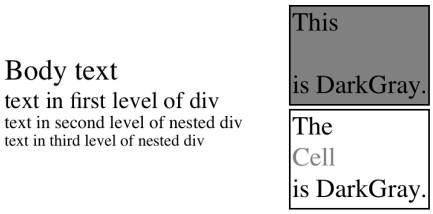


Figure 4.10: Generated counter-example layouts.

The second test consists in verifying that for a given style sheet, there is no document such that the style sheet generates text with the same color as the background color. For example, we consider the simple style sheet of Figure 4.11. The problem consists in testing wether rules that set the `color` and `background-color` properties together with the CSS inheritance mechanism may result in such a situation. This is expressed in logical terms is as follows:

```
1. type("html.dtd","html") & ~<-1>T  & ~<-2>T
2. & let $CSS =
```

```
   ((tr & let $V=<-1>T|<-2><-2>$V in $V)
   => <background-color>'LightGray')
3. & ((tr & let $X=<-2><-1>T|<-2><-2>$X in $X)
   => <background-color>'DarkGray')
4. & (~tr => (~<background-color>T |<background-color>'white'))
5. & (div => <color>'DarkGray')
6. & (~div => (~<color>T |<color>'black'))
7. & (~<1>T | <1>$CSS) & (~<2>T | <2>$CSS) in $CSS
8. & let $Q = (<color>'DarkGray'
       & ancestor(<background-color>'DarkGray'))
     | <1>$Q | <2>$Q in $Q
```

Line 2 and 3 are the logical counterparts of CSS rule 1 and 2 of the sample of Figure 4.11. They correspond to the logical implication "if an element is labeled `tr` and is at an even position (odd position respectively) among its siblings, then its background color must be 'LightGray' ('DarkGray' respectively)". Line 4 says that any other element than `tr` (elements that are not concerned by the previous rules) may either carry no `background-color` property or if they do, then its value is set to`white`. This models the default behavior of CSS for the `background-color` property which is overridden by the rules for `tr`. Similarly, line 5 is the logical implication that corresponds to the CSS rule on line 5 of Figure 4.11. Line 6 models the default behavior for the property color. Line 7 is in charge of applying the style information to every element in the document.

```
1.  tr:nth-child(even) {background-color:LightGray;}
2.  tr:nth-child(odd) {background-color:DarkGray;}
3.  table td {font-size: 16px;}
4.  td {font-size: 14px;}
5.  div {color:gray};
6.  div {font-size:smaller;}
```

Figure 4.11: Sample CSS style sheet.

Line 8 formulates the question: "*may the application of my style sheet render some valid document with unreadable text because it is displayed in the same color as the background?*" For the sake of simplicity, the ancestor predicate in this line models the default CSS inheritance behavior for the `background-color` property. A more general statement should use the predicate `inheritedValue`. When fed with this formula, the logical solver explores all possible situations and ends up with this counter-example (which is displayed in the browser as the picture on the right in Figure 4.10):

```
<html xmlns:solver="http://wam.inrialpes.fr/xml">
  <head>
    <title/>
  </head>
  <body>
    <table>
      <tbody>
```

```
        <tr background-color="LightGray">
          <td>
          This <div color="DarkGray"> Cell</div> is DarkGray.
          </td>
        </tr>
        <tr background-color="DarkGray">
          <td>
          The <div color="DarkGray"> Cell</div> is DarkGray.
          </td>
        </tr>
      </tbody>
    </table>
  </body>
</html>
```

The disclaimer text contained in the second cell of the table has both properties `color` and `background-color` set to `DarkGray`. This is caused by the default inheritance rules for these properties which are set to `inherit`. The disclaimer text has inherited its values from the enclosing `div` resulting in this color collision.

Now, we check the consistency of selectors in the style sheet of Figure 4.11. The test amounts to comparing the selectors for a given property. For example, if we focus on the `font-size` property for table element selectors `table td` and `td`, the test consists in checking the precise relation between selectors, equivalence or containment, against the HTML DTD for instance.

The problem formulation for the solver is as follows:

```
1.  type("html.dtd","html") & ~<-1>T  & ~<-2>T
2.  & let $Q =
      ~(td & ancestor(table) <=> td)
      | <1>$Q | <2>$Q in $Q
```

The formula is found unsatisfiable which means that the two selectors are equivalent in the presence of the DTD. Intuitively, here, both selectors are equivalent since under HTML schema constraints `td` always occurs under a `table` element. However, for `td` elements, CSS rule precedence gives higher priority to rule 5 which has specificity 2 compared to rule 6 with specificity of 1. Therefore, rule 6 will never be reachable by any HTML document. As a consequence, rule 6 can be safely removed from the style sheet.

When generalized to all selectors for a given property, this mechanism allows to clean up style sheets from such inapplicable rules, enhancing their readability, as seen in the next Section.

While in the case of HTML such situations can be detected by an expert designer, things become much harder when considering CSS for general XML documents. In particular, CSS rules (see selectors of [Werntges 2011]) for very structured schemas like Docbook [Walsh 1999] or DITA [Eberlein *et al.* 2010], tend to be much more involved as they use complex compositions of combinators with type elements.

4.7.1 Identifying and verifying generic issues

Our analyzer allow one to further analyze more generic issues that correspond to useful questions for CSS developers and that can arise in many style sheets. We describe how several such properties can be formulated and checked with our tool below. The tests described can be checked in the absence or in the presence of a schema. In the latter case, we use the logical translation of the schema that we insert as the initial context for the translation of selectors[2].

Emptiness of selectors This test is the generalization of the example presented above. The test consists in extracting every selector and testing its satisfiability against a given schema. We check $\mathtt{F}(S_i)$ for unsatisfiability. If $\mathtt{F}(S_i)$ is unsatisfiable then the selector is inconsistent and the corresponding style rule is always inactive.

Equivalence of selectors We check the validity of $\mathtt{F}(S_i) \Leftrightarrow \mathtt{F}(S_j)$ for $i \neq j$ pairwise by checking for the unsatisfiability of $\neg\,(\mathtt{F}(S_i) \Leftrightarrow \mathtt{F}(S_j))$. If two selectors are equivalent and if one has a lower specificity, i.e. $\mathtt{Specificity}(S_i) < \mathtt{Specificity}(S_j)$ then the rule for S_i is always inactive. If both have the same specificity then the first rule in the lexical order in the style sheet is always inactive since CSS favors the last one in this case.

The emptiness and equivalence tests for selectors can be used for tuning a CSS style sheet for a particular schema by pruning inactive CSS rules automatically.

Coverage without properties nor inheritance We check the validity of $\mathtt{T} \Rightarrow \bigcup_i S_i$, that is, we check the unsatisfiability of $\neg\,(\bigcup_i S_i)$. If this formula is unsatisfiable, then it means that some elements are not covered by any style sheet selectors. In other terms, the style properties set by the CSS developer do not cover all the possible elements of a document. If a rule with selector $\star$ exists then obviously all elements are covered. This test can be performed on the style sheet except $\star$ selectors, in order to capture the coverage of CSS properties other than those defined for all elements ($\star$).

Coverage with inheritance for a given property We want to determine, whether a given property p is set to some value v for all elements of a document, while taking into account the propagation of values defined by the inheritance mechanism of CSS. We define the predicate $\mathtt{customset}(p, v)$ as the disjunction of all selectors that set the value v for the property p. We check for the validity of the following formula ψ:

$$<p>\texttt{'}\,v\texttt{'} \Rightarrow (\mathtt{inherit}(p, v) \,|\, \mathtt{customset}(p, v))$$

or in other terms we check for the unsatisfiability of $\neg\psi$ where p is the property and v is the value for which we check the coverage. For example, we can think of a web designer building a style theme restricted to a limited set of colors. She may be willing

[2]The notion of context is explained in details in [Genevès *et al.* 2007].

to test whether all possible HTML documents and their respective elements do have only these colors without reverting to default ones.

$$\neg(\bigcup_{i=1,2} \texttt{<color>}c_i \Rightarrow (\bigcup_{i=1,2} \texttt{inherit}(\texttt{color}, c_i) \,|\, \texttt{customset}(\texttt{color}, c_i)))$$

If this formula is satisfiable, this means that there exist some document instance for which the style sheet renders some elements with another color than the c_i's, breaking the intended design.

4.8 Automated CSS Size Reduction

In this Section, we present another application of our analyses: the automated refactoring of CSS style sheets for reducing their size. We present how we can use semantical analyses for the purpose of automatically identifying redundant CSS declarations. We developed a prototype that effectively removes CSS declarations detected as redundant. We report on experimental results obtained with our prototype. To the best of our knowledge, these are the first experimental results concerning automated CSS size reduction based on a semantical analysis of CSS (as opposed to purely syntactic CSS optimizers).

Our tool is concerned with the detection of semantical relations between CSS selectors (e.g. mainly containment or equivalence). When some of these relations are detected, our tool might determine that a property declaration is unnecessary and it will thus be deleted, based on the *specificity* of selectors. In CSS, a selector's *specificity* is a vector of four integers (a, b, c, d), where $a = 1$ if the property is declared in a `style` attribute ($a = 0$ otherwise), b is the number of `id` attributes (of the form "#␣") in the selector, c is the number of other attributes and pseudo-classes in the selector, and d is the number of element names and pseudo-elements in the selector. Our tool exploits the facts that: (1) if selectors of two different rules have the same specificity, then the last rule in the style sheet gains precedence; (2) when several selectors point to the same set of elements, then the declarations under the one with higher *specificity* gain precedence.

4.8.1 Refactoring associated with containment of selectors

One fundamental relation between two CSS selectors is *containment*. For example we say that "`ul ¿ li`" is *contained* into "`li`" since any "`li`" element with an "`ul`" parent is indeed a "`li`" element. The existence of containment relations is determined by the analysis of the nested structures of elements and the sets of attributes carried by elements. A selector such as "`p.someclass`" is contained into "`p`", since any "`p`" element with "`class`" attribute "`someclass`" is indeed a "`p`" element.

Given two selectors S_b and S_p, S_b is contained in S_p iff any element pointed by S_b is also pointed by S_p. In this section we treat only *proper containment*, which means $S_b \subset S_p$ and not $S_b \subseteq S_p$. Under these circumstances, for each property declared

under both selectors, we propose two different refactoring procedures according to the selector's specificity:

Refactoring 1. ***Subset more specific****: delete the property declaration from S_b only if it has the same value set under both S_b and S_p.*

Refactoring 2. ***Subset less specific****: delete the property declaration from S_b, since the value set under S_p will always override the one under S_b.*

For example, consider the following code snippet:

Listing 4.1: containment_input.css

```
1 table.foo { color: #333;
2             font-size: 12px;
3             font-weight: bold }
4
5 table { color: #666;
6         font-size: 12px }
```

Note that "`table.foo`" $\subset$ "`table`" and the subset has a higher specificity $(0, 0, 1, 1)$ against $(0, 0, 0, 1)$ from the superset, so we are in the case of *optimization 1*. Consequently, we have to preserve the "`color: #333`" declaration as it will override the one from "`table`" when both rules apply. On the other hand, the "`font-size`" property statement can be removed from the subset as the same value is already pulled from the superset "`table`". The following code corresponds to the output of the analysis:

Listing 4.2: containment_output.css

```
1 table.foo { color: #333;
2             font-weight: bold }
3
4 table { color: #666;
5         font-size: 12px }
```

If "`table`" was more specific than "`table.foo`", *optimization 2* would apply and the "`color`" property declaration could be erased from the subset too as its value would always be overridden by the dominant "`#666`" set in the superset.

4.8.2 Refactoring associated with equivalence of selectors

Another relation between CSS selectors that can lead to some refactoring is *equivalence*. Two or more CSS selectors can be equivalent in several ways. Is not uncommon to find a rule R_i with a selector such as "`body`", and later in the same file another rule R_j with the same selector "`body`". Another case of equivalence could be a class selector "`.classname`" and the attribute selector "`[class='classname']`", or any similar case with the dual syntax for `id` attributes. These examples could be studied with basic string processing, but the logical and semantical analysis of selectors

allows us to detect more complex equivalences too such as the one between selectors "`p:nth-child(odd), p:nth-child(even)`" and "`p`", as every paragraph is either odd or even.

Given two selectors S_i and S_j, they are *equivalent* iff any element pointed by S_i is also pointed by S_j and vice-versa. In this context there is only one procedure over the bodies:

Refactoring 3. *For each property declared under both selectors, delete the statements under the less specific selector.*

To illustrate the analysis, consider the next listing:

Listing 4.3: equivalence_input.css

```
1 div#bar { font-style: italic;
2           border: none }
3
4 div[id='bar'] { color: #666;
5                 font-style: normal;
6                 border: none }
```

In this case we have two equivalent selectors, "`div#bar`" and "`div[id='bar']`" whose specificities are $(0, 1, 0, 1)$ and $(0, 0, 1, 1)$ respectively. For the properties declared under both rules, the values from "`div#bar`" will dominate, so the ones under "`div[id='bar']`" will never apply, as selectors point to the same set of elements. This means that for "`font-style`" and "`border`", the declarations can be safely erased from "`div[id='bar']`", resulting in the following output:

Listing 4.4: equivalence_output.css

```
1 div#bar { font-style: italic;
2           border: none }
3
4 div[id='bar'] { color: #666 }
```

4.8.3 Inheritance of properties

Whenever containment or equivalence relations are detected between selectors, several selectors point to the same elements, and specificity decides which declarations get precedence. In the context of inheritance, an element might be pointed by only one selector and yet be affected by declarations outside the concerning rule. This is because the declaration has been propagated from some ancestor through the inheritance mechanism. Consider a selector using a descendant combinator "$E_{anc} > E_{desc}$" with a inheritable property declaration $P_a : V_a$, and another selector "E_{anc}" with the same declaration. This statement might then be redundant as for E_{desc} the property might inherited from E_{desc}.

However we do not know if some document using this CSS file, presents a structure in which there is a certain element E_x placed in between E_{anc} and E_{desc}, and the property declarations for E_x alter the property inheritance among E_{anc} and E_{desc}. For example, a selector concerning only an attribute, such as "`[input]`", is free to be applied to any element on the document. Only certain CSS properties are inherited by default. Therefore, in our analyses, the amount of refactoring due to inheritance is a priori limited.

4.8.4 Reasoning over selectors

For detecting relations between selectors, we use the translation of CSS selectors into the tree logic described in the previous Sections and obtain logical formulas. We then formulate containment as logical implication, and test the formula for satisfiability using the logical solver, as mentioned in Section 4.6. For two selectors S_1 and S_2, Table 4.1 summarizes the tests performed, the four possible scenarios obtained according to the results, and the corresponding actions performed for optimizing a CSS.

$S_1 \subseteq S_2$	$S_1 \supseteq S_2$	Relation	Action
0	0	None	None
0	1	$S_1 \supset S_2$	Refactoring 1, 2
1	0	$S_1 \subset S_2$	Refactoring 1, 2
1	1	$S_1 \Leftrightarrow S_2$	Refactoring 3

Table 4.1: Actions associated with detected relations.

4.8.5 Processing template bodies

Once a relation between selectors S_1 and S_2 has been found, for each of the properties declared under both rules, we determine whether it is necessary or not, depending on three aspects: the relation between selectors, the selector's specificity, and whether the properties share the same value; as illustrated on examples in sections 4.8.1 and 4.8.2.

In some cases, the deletion of unnecessary property statements results in an empty rule. In this case, the whole rule is (safely) erased from the style sheet.

4.8.6 Optimization of search space and elapsed time

In a style sheet with n rules, each rule can be tested agains all rules but itself, adding up to a total of $n \times (n-1)$ possible tests. Given the diversity of elements in a HTML tree, tests concerning selector pairs such as "`body`" and "`p`" will not be uncommon. Only by adding a few basic pre checks, we will be able to determine the result of logical tests before actually processing it. We take advantage of two main observations for drastically reducing the number of pairwise tests:

1. *If two selectors point to elements with syntactically different names, they will never be contained into each other, in any of the two possible containment directions.*

2. *If a selector S_1 refers to one or more attributes that S_2 does not, S_1 will never contain S_2.*

4.8.7 Statistics tracking

The tool tracks some statistics about the analysis. First, while parsing it detects the total number of rules, the number of ignored ones, the number of possible tests, and the tests that were actually carried out. After all reasoning is done, the tool counts the number of relations between selectors, the modified rules, the number of deleted properties as well as the deleted bits. Finally, the time spent on each one of the analysis parts is also shown.

4.8.8 Room for improvements

Our prototype implements the aforementioned procedures for a significant CSS subset, sufficient for performing practical experiments with real-world style sheets. Our prototype can be improved in many respects, though. In particular, the library css-validator[3], that is used for parsing the CSS file and traversing the properties of the rules, could be improved. Some methods concerning comparisons of properties are not implemented, and thus some potential property deletions cannot be automatically carried out. It does not support browser specific properties yet.

Some CSS selectors' features are not supported yet, such as grouping, pseudo-classes, pseudo-elements, multiple class and id selectors and media queries. Consequently, the concerning selectors are ignored. With their implementation, additional refactoring might be performed.

4.8.9 Experimental Results

In order to get a representative collection of results, three different groups have been defined. The first group involves CSS code provided by frameworks and CMS, and is represented by *Bootstrap*, *Joomla* and *JQuery*. The second group consists in style sheets from complex web applications, and features *Instagram*, *Twitter* and *The Times*. Finally, we have extracted CSS files from some random web sites of medium complexity, which are *ACM DL*, *DocEng*, and *Inovallée*. Table 4.2 provides detailed information about the corresponding CSS file sizes and complexity.

After processing the aforementioned files, the tool has spotted on average 4.95%[4] of unnecessary properties, modifying 4.56% of the total rules. Of the relations found, 83.38% were containment ones, and the remaining 16.62% correspond to equivalence between selectors.

[3]see http://jigsaw.w3.org/css-validator/
[4]calculated over the properties that the tool supports.

Name	# bytes	# rules
Bootstrap (*Framework's CSS*)	127343	805
Joomla *(Template Beez20's CSS)*	30158	325
JQuery *(Framework's CSS)*	32891	349
Instagram *(www.instagram.com)*	123815	791
The Times *(www.thetimes.co.uk)*	89362	469
Twitter *(www.twitter.com)*	245473	2402
ACM DL *(dl.acm.org)*	11151	97
DocEng *(www.doceng2014.org)*	204970	1571
Inovallée *(www.inovallee.com)*	29930	189

Table 4.2: Dataset for the experiments.

It is clear that the style sheets from the first and second group present a low percentage of deleted properties (1.16% and 1.73% respectively). The same holds for the percentage of modified rules (1.59% for the first group and 2.19% for the second one). However, in the third group these numbers rise dramatically, reaching 11.05% of deleted properties and 16.95% of modified rules. Although these latter sites might not have involved as much testing as the first ones, they are not amateur web sites either.

4.8.10 Performance of the tool

The time taken to analyze each file is shown in Figure 4.13. A 34.30% of the rules have been ignored due to unsupported selectors commented in Section 4.8.8, and an average of 99.88%[5] of the them has been discarded by the optimization mechanism described in Section 4.8.6. Each test between selectors requires a median of 156.71 ms, so without the optimization mechanism, each file analysis would have taken a median of 16.60 hours, in contraposition to the 78.16 seconds that were actually required, still guaranteeing the same results.

4.9 Conclusions

In this chapter, we introduce the concept of static analysis for CSS style sheets. To the best of our knowledge, this is the first attempt at statically analyzing CSS style sheets. We propose an original tool based on recent advances in tree logics. The tool is capable of statically detecting a wide range of common errors, as well as proving properties related to sets of documents, such as coverage of styling information, in the presence or absence of schema information.

From a theoretical perspective, CSS selectors could be related to XPath queries, for which an extensive static analysis has been conducted in [Genevès *et al.* 2007].

[5]calculated over considered tests, discarding ignored ones.

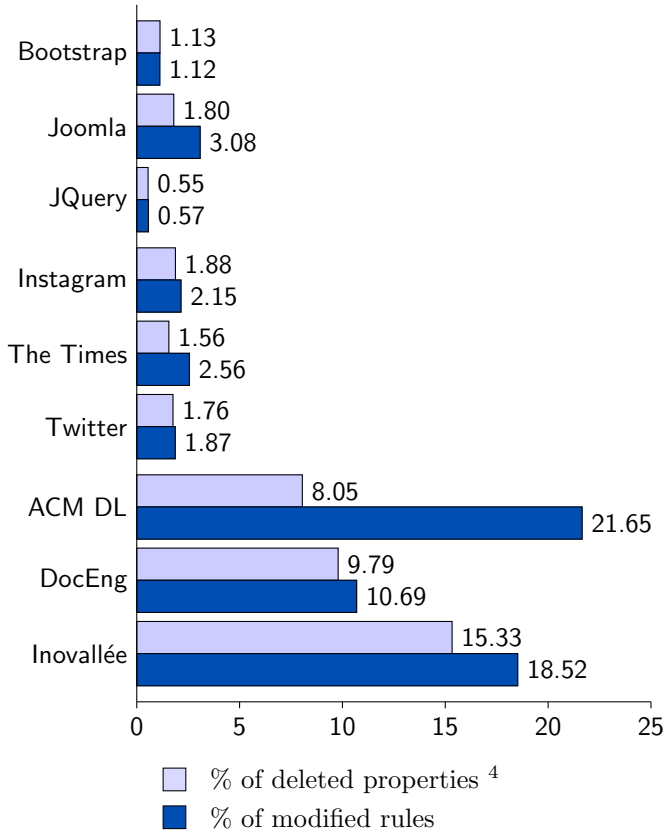


Figure 4.12: Refactoring performed.

In this chapter, we deal with the particuliar combinators and pseudo-classes found in CSS selectors. In particular, we have extended the logical solver, initially developed for XPath, to be able to reason about attribute values, by introducing an equality test that compares an attribute value to a constant. This is a worthy extension since it is sufficient for supporting CSS while preserving decidability for the extended logic (it is known that extending the logic with equality tests with variables results in undecidable logics, but this feature is not needed for CSS). In addition, we deal with style properties and the propagation of values defined by the inheritance mechanism of CSS, which do not have any XPath counterpart.

From a practical perspective, there exists a whole class of dynamic analyses. Most of this technology relies on runtime debuggers that check the behavior of a CSS style sheet on a particular document instance. However, the aim of CSS is to be applied to an entire set of documents, usually defined by some schema. The existing runtime debugging tools help reducing the number of bugs. However, compared to our approach, they do not allow to prove properties over the whole set of documents to which the style sheet is intended to be applied. Therefore, our new approach and tool can be used in addition to debuggers to ensure a higher level of quality of CSS style sheets.

Furthermore, we presented another tool that automatically detects and removes unnecessary property declarations in CSS files, based on the analysis of semantical

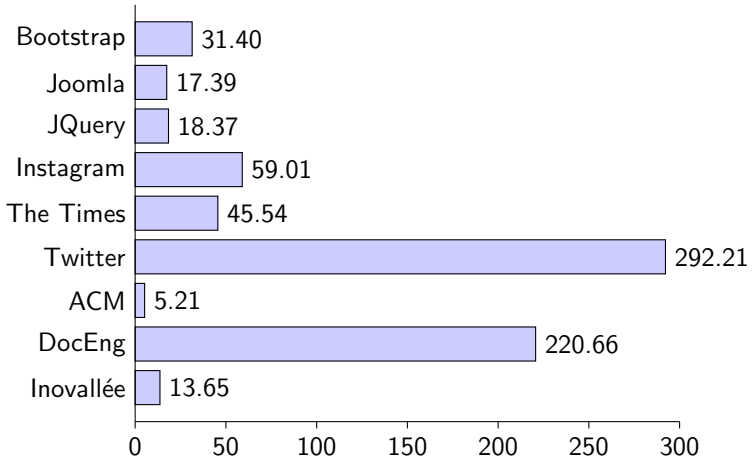


Figure 4.13: Elapsed time for the analyses (s).

relations between selectors. We provided a first prototype implementation, with many perspectives for further development. The benefit of this tool is to conduct precise semantical analyses, that go far beyond the capabilities of purely syntactic analyses done by current CSS optimizers. Generating equivalent but simpler CSS files not only improves the time spent in loading and formatting a web page, but also facilitates the debugging process of style sheets.

Despite the large number of unsupported features at this stage, the results obtained in section 4.8.9 already validate our approach: we have been able to detect large numbers of unnecessary property declarations in non-amateur web pages; and we have also found mistakes in the style sheets of some of the most popular web sites. The number of safe modifications can easily grow as more components of CSS are supported and more features are implemented, such as property inheritance, translation of pseudo-classes into query languages, analysis of media queries, merging of equivalent selectors or containment involving grouped selectors.

Chapter 5

Containment for a SPARQL Fragment

Contents

5.1 Introduction 102
5.2 Preliminaries 103
5.3 SPARQL Query Containment 115
5.4 μ-calculus 115
5.5 RDF Graphs as Transition Systems 118
5.6 Encoding SPARQL Query Containment 123
5.7 Experimental Investigations 132
5.8 Query Containment Solvers 133
5.9 Benchmark Design 134
5.10 Experimental Results 139
5.11 Related Work 142
5.12 Conclusions 146

Abstract

We investigate the problem of query containment for the SPARQL language. This problem is defined as determining whether, for any graph, the result of one SPARQL query is included in the result of another query. Query containment is important in many areas, including program analysis, information integration, and query optimization. For instance, if we can prove that two queries are equivalent for any graph (which reduces to two query containment checks), then we can safely substitute one query by another more efficient query version, while preserving the initial semantics of the program.

We address query containment for a fragment of the SPARQL language, under expressive description logic constraints. SPARQL is interpreted over graphs, hence we encode it in a graph logic, specifically the alternation-free fragment of the μ-calculus [Kozen 1983] with converse and nominals [Tanabe *et al.* 2008] interpreted over labeled transition systems.

We show that this logic is powerful enough to deal with query containment for the fragment of SPARQL made of basic and union graph patterns, in the presence of $\mathcal{ALCH}$ schema axioms.

In this logical encoding, RDF graphs become transition systems and queries and schema axioms become μ-calculus formulae. Therefore, SPARQL query containment can be reduced to unsatisfiability in the μ-calculus. This approach has several advantages: it can be used to precisely characterize the expressive power and complexity for fragments of the SPARQL language. In particular, due to the high expressive power of the μ-calculus, such an encoding is useful to characterize not only restrictions but also extensions of the SPARQL language. For example, a benefit of using a μ-calculus encoding is to take advantage of fixpoints and modalities for encoding recursion. These operators allow to deal with natural extensions of SPARQL such as path queries [Alkhateeb *et al.* 2009] or queries modulo RDF Schema. Finally, another advantage of such an approach is that the considered logic admits exponential time decision procedures that can be implemented in practice [Tanabe *et al.* 2005, Genevès *et al.* 2007, Tanabe *et al.* 2008]. This study allows to exploit these advantages.

In order to experimentally assess implementation strengths and limitations in this setting, we provide a first SPARQL containment test benchmark. It has been designed with respect to both the capabilities of existing solvers and the study of typical queries. Some solvers support optional constructs and cycles, while other solvers support projection, union of conjunctive queries and RDF Schemas. No solver currently supports all these features. The study of query demographics on DBPedia logs shows that the vast majority of queries are acyclic and a significant part of them uses union or projection. We thus test available solvers on their domain of applicability on three different benchmark suites. We report on the experimental results, and discuss to which extent, in spite of its complexity, SPARQL query containment is practicable for acyclic queries.

5.1 Introduction

Since its recommendation by the W3C, SPARQL has come to widespread use. Large datasets are being made available due to the rapid emergence of linked data, and queries are executed at remote endpoints. It becomes more and more important to be able to optimize queries, which requires analyzing them before evaluating them. We concentrate here on query containment analysis since query equivalence and query satisfiability can be reduced to query containment. The benefits of static analysis are well known for database query optimization and information integration. Indeed, the static analysis of queries may reveal, without trying to evaluate it, that a query will not return any answer. It may also be used for evaluating queries against precompiled views, which requires to know if the answers to the query against the view are the same as the answers to the database itself. The same benefits can apply for SPARQL queries. It is even more important in a distributed setting when federated queries are evaluated at different SPARQL endpoints. Before sending a query to an endpoint, it

is useful to know if this query has any chance to receive answers: this saves not only evaluation time but also communication time.

For all these reasons, the development of methods for the analysis of SPARQL queries and especially query containment is of uttermost importance. This is the problem investigated in this Chapter for the SPARQL query language interpreted over RDF graphs.

Outline and Contributions

This Chapter is organised in three main parts, as follows:

A common introductory part, dedicated to the presentation of required preliminary notions. We present RDF graphs and the SPARQL query language in Section 5.2. We then introduce the query containment problem for SPARQL in Section 5.3. We present the μ-calculus in which we encode this problem in Section 5.4.

A theoretical part, where we show how RDF graphs can be encoded as transition systems in Section 5.5. We then show how we encode SPARQL queries in terms of logical formulas and how we reduce SPARQL query containment to logical unsatisfiability in Section 5.6. We also characterize the computational complexity of our approach.

An experimental part, in which we characterize to which extent our approach is currently implementable and feasible in practice. In Section 5.7 we explain our experimental investigations and framework. We review query containment solvers and the state of the query landscape in Section 5.8. In order to experimentally assess implementation strengths and limitations, we provide a first SPARQL containment test benchmark in Section 5.9. We present and discuss the results of our experiments in Section 5.10 before discussing related works in Section 5.11 and presenting our conclusions in Section 5.12.

5.2 Preliminaries

5.2.1 RDF

The Resource Description Framework (RDF) is a language used to express structured information on the web as graphs. It was primarily intended for representing metadata about WWW resources, such as the title, author, and modification date of a web page, but it can be used for storing any other data. It is based on the concept of sets of triples of the form "(subject, predicate, object)" that constitute a graph.

We present a compact formalization of RDF [Hayes 2004]. Let U, B, and L be three disjoint infinite sets denoting the set of URIs (an URI identifies a resource), blank nodes (a blank node denotes an unidentified resource) and literals (a literal denotes a character string or some other type of data) respectively. We abbreviate any

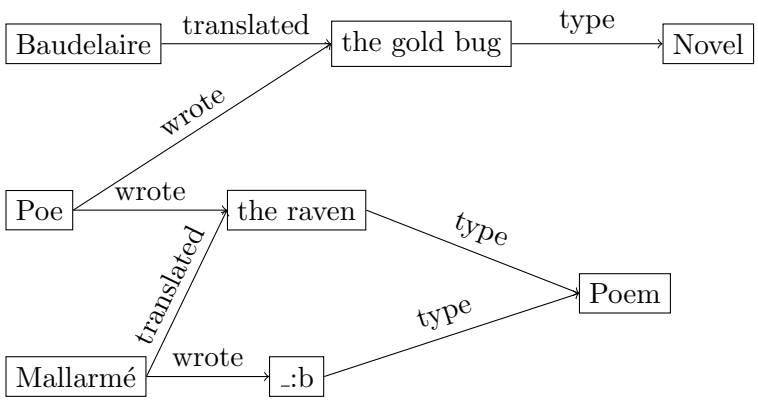


Figure 5.1: Example RDF graph G about writers and their works.

union of these sets as, for instance, UBL = U∪B∪L. We refer to U, B, and L as *terms* of the graph. A triple of the form $(s, p, o) \in \text{UB} \times U \times \text{UBL}$ is called an *RDF triple*. s is the *subject*, p is the *predicate*, and o is the *object* of the triple. Each triple can be thought of as an edge between the subject and the object labelled by the predicate, hence a set of RDF triples is often referred to as an *RDF graph*.

Example 1 (RDF Graph)**.** *Here are 8 triples of an RDF graph about writers and their works: (all identifiers correspond to URIs, _:b is a blank node):*

{ *(Poe,wrote,thegoldbug), (Baudelaire,translated,thegoldbug),*
(Poe,wrote,theraven), (Mallarmé,translated,theraven),
(theraven,type,Poem), (Mallarmé,wrote,_:b),
(_:b,type,Poem), (thegoldbug,type,Novel) }

These triples can also be represented graphically as shown in Figure 5.1.

RDF has a model theoretic semantics [Hayes 2004], that defines the notion of consequence between two RDF graphs, i.e., does an RDF graph G entails an RDF graph H (known as RDF entailment). The interpretation of an RDF graph is the set of all triples that are logically implied according to some set of axioms.

For example, let us consider an RDF graph G formed by an initial set S of triples. The interpretation of G, denoted by $[\![G]\!]$ is a superset of S where the elements in $[\![G]\!] \setminus S$ are deduced based on a chosen set of axioms. This set of axioms describes under which circumstances and conditions new triples are deduced from the initial ones. In practice, such a set of axioms is typically expressed using RDFS.

5.2.2 RDF Schema

RDF Schema (RDFS) [Hayes 2004] may be considered as a simple ontology language expressing subsumption relations between classes or properties. Technically, this is an RDF vocabulary used for expressing axioms constraining the interpretation of graphs. Hence, schemas are themselves RDF graphs. The RDFS vocabulary and its semantics are given in [Hayes 2004]. The W3C specifications introduce

two standard namespaces: the *RDF namespace* http://www.w3.org/1999/02/22-rdf-syntax-ns# (prefix *rdf*) and the *RDF Schema namespace* http://www.w3.org/2000/01/rdf-schema# (prefix *rdfs*). These namespaces comprise a set of URIs with predefined meaning. Below, we present some of the predefined vocabulary terms, the notation in the parenthesis is the vocabulary syntax used in this chapter:

- The predefined URI *rdf:type (type)* can be used for typing entities.
- *rdfs:subClassOf (sc)* and *rdfs:subPropertyOf (sp)* are used to describe subclass and subproperty relationships between classes and properties, respectively.
- The two classes *rdfs:Class (Class)* and *rdf:Property (Property)* can be used to assign a logical type to URIs.
- The URIs *rdfs:domain (dom)* and *rdfs:range (range)* can be used to specify the domain and range of properties.
- All things described by RDF are called *resources*, and are instances of the class *rdfs:Resource (Resource)*.

In [Hayes 2004], rules are given which allow to deduce or infer new triples using RDF Schema triples. In [Ter Horst 2005], it is shown that the standard set of entailment rules for RDFS, is incomplete and that this can be corrected by allowing blank nodes in predicate position. The rules shown below, taken from [Muñoz *et al.* 2007], fix this problem.

- Subclass (sc)

$$\frac{(a, sc, b)\ (b, sc, c)}{(a, sc, c)} \qquad \frac{(a, sc, b)\ (x, type, a)}{(x, type, b)} \tag{5.1}$$

- Subproperty (sp)

$$\frac{(a, sp, b)\ (b, sp, c)}{(a, sp, c)} \qquad \frac{(a, sp, b)\ (x, a, y)}{(x, b, y)} \tag{5.2}$$

- Typing (dom, range)

$$\frac{(a, dom, b)\ (x, a, y)}{(x, type, b)} \qquad \frac{(a, range, b)\ (x, a, y)}{(y, type, b)} \tag{5.3}$$

- Implicit Typing

$$\frac{(a, dom, b)\ (c, sp, a)\ (x, c, y)}{(x, type, b)} \qquad \frac{(a, range, b)\ (c, sp, a)\ (x, c, y)}{(y, type, b)} \tag{5.4}$$

- Subclass reflexivity

$$\frac{(a, type, Class)}{(a, sc, a)} \qquad \frac{(a, sc, b)}{(a, sc, a)\ (b, sc, b)} \tag{5.5}$$

- Subproperty reflexivity

$$\frac{(a, type, Property)}{(a, sp, a)} \qquad \frac{(x, a, y)}{(a, sp, a)} \tag{5.6}$$

- Resource

$$\frac{(a, b, c)}{(a, type, Resource)} \qquad \frac{(a, b, c)}{(c, type, Resource)} \qquad \frac{(a, type, Class)}{(a, sc, Resource)} \tag{5.7}$$

- Property

$$\frac{(a, b, c)}{(b, type, Property)} \tag{5.8}$$

- Class

$$\frac{(a, b, c)}{(a, type, Class)} \qquad \frac{(a, type, c)}{(c, type, Class)} \tag{5.9}$$

Example 2. *This example shows the usage of RDFS inference rules, consider the graph:*

$$\begin{aligned} G = \{&(john, child, mary), (child, sp, ancestor), \\ &(ancestor, dom, Person), (ancestor, range, Person)\} \end{aligned}$$

By applying either both typing (2.3) and subproperty (2.2) rules or implicit typing rule (2.5), it can be inferred that {(*john*,*type*,*Person*), (*mary*,*type*,*Person*), (*john*,*ancestor*,*mary*)}. *Hence, the deductive closure of the graph* G*, denoted as* $cl(G)$*, is:*

$$\begin{aligned} cl(G) = \{&(john, child, mary), (child, sp, ancestor), (ancestor, dom, Person) \\ &(john, type, Person), (mary, type, Person), \\ &(john, ancestor, mary)\} \end{aligned}$$

Note that additional triples that can be derived by reflexivity and other rules are not displayed in $cl(G)$*, it contains only a part of the closure graph.*

5.2.3 Entailment

Here we present simple entailment and RDFS entailment in RDF graphs, a more detailed discussion can be found in [Hayes 2004, Ter Horst 2005].

Simple RDF Entailment: simple entailment depends only on the basic logical form of RDF graphs and therefore holds for any vocabulary. Given two RDF graphs G_1 and G_2, a *map* from G_1 to G_2 is a function h from terms of G_1 to terms of G_2, preserving URIs and literals, such that for each triple $(a, b, c) \in G_1$ we have $(h(a), h(b), h(c)) \in G_2$.

An RDF graph G_1 *simply entails* G_2, denoted $G_1 \models_s G_2$, if and only if there exists a map from G_2 to G_1.

RDFS entailment: RDFS entailment captures the semantics added by the RDFS vocabulary. We write that $G_1 \models_{rule} G_2$ if G_2 can be derived from G_1 by iteratively applying rules in groups (*subclass*), (*subproperty*) and (*typing*). The *closure* of a graph G, denoted as $cl(G)$, is the graph obtained from it by iteratively applying the RDFS inference rules. We have that $G_1 \models_{rule} G_2$ if and only if $G_2 \in cl(G1)$. It turns out that G_1 RDFS-entails G_2, written $G_1 \models_{rdfs} G_2$, iff there is a graph G derived from G_1 by exhaustively applying the RDFS rules such that $G_1 \models_{rule} G$ and $G \models_s G_2$ [Ter Horst 2005]. Alternatively, $G_1 \models_{rdfs} G_2$ iff $cl(G_1, G_2) \models_s G_2$ where $cl(G_1, G_2)$ is the union of the closure of G_1 and G_2 obtained by exhaustively applying the RDFS inference rules.

5.2.4 Description Logics

Description Logics (DLs) are a family of knowledge representation (KR) formalisms that represent the knowledge of an application domain by first defining the relevant concepts of the domain (its terminology), and then using these concepts to specify properties of objects and individuals occurring in the domain (the world description) [Baader & Nutt 2003, Baader *et al.* 2007]. Alternatively, DLs are fragments of first-order logic that model a domain of interest in terms of concepts and roles denoting unary and binary predicates, respectivley [Baader *et al.* 2007]. DLs are equipped with a feature that allow for reasoning in a knowledge base. Reasoning enables one to infer implicitly represented knowledge from the knowledge that is explicitly contained in the knowledge base. A knowledge base (KB) comprises two components, the TBox and the ABox. The TBox introduces the terminology, i.e., the vocabulary for classes and properties in an application domain, while the ABox contains assertions about named individuals in terms of this vocabulary.

There are various types of description logics with differing expressivity, DL-Lite and its extensions, $\mathcal{SROIQ}(\mathbf{D})$ and its fragments, and $\mathcal{ALC}$ and its extensions [Baader *et al.* 2007,Horrocks *et al.* 2006]. In this chapter, we use the well-studied DL $\mathcal{SROIQ}(\mathbf{D})$ [Horrocks *et al.* 2006] that underlies the foundations of the web ontology language (OWL 2). We consider various fragments of this logic, mainly the $\mathcal{ALCH}$ fragment.

5.2.4.1 $\mathcal{SROIQ}(\mathbf{D})$

We consider here the following constructs occurring in expressive description logics: role hierarchy $\mathcal{H}$, role *transitivity* $\mathcal{S}$, role *composition* $\mathcal{R}$, *nominals* $\mathcal{O}$, role *inverse* $\mathcal{I}$, *qualified number restrictions* $\mathcal{Q}$, and *datatypes* $\mathbf{D}$. Recently, OWL 2 has become a W3C recommended ontology language. The logic underlying this ontology language is $\mathcal{SROIQ}(\mathbf{D})$. The $\mathcal{SROIQ}(\mathbf{D})$ KB satisfiability problem is 2NExpTime-complete [Horrocks *et al.* 2006,Kazakov 2008]. The syntax and semantics of this logic is presented

in Table 5.1 and 5.2. Notice that, even though we detail here its constructs, in this chapter we are interested in its fragments, mainly in $\mathcal{ALCH}$.

Syntax In $\mathcal{SROIQ}(\mathbf{D})$ concepts and roles are formed according to the syntax presented in Table 5.1, where R denotes an atomic role or its inverse, A represents an atomic concept, C denotes a complex concept, o refers to a nominal, and n is a non-negative integer. Additionally, the following abbreviations are used:

$$\begin{aligned} C_1 \sqcup C_2 &= \neg(\neg C_1 \sqcap \neg C_2) \\ \top &= \neg(\bot) \\ \forall R.C &= \neg(\exists R.\neg C) \end{aligned}$$

$\mathcal{SROIQ}(\mathbf{D})$ **Axioms:** The TBox is a finite set of axioms consisting of *concept inclusions*, *role inclusion*, *role transitivity*, and *role chain* axioms:

$$\begin{array}{cc} C_1 \sqsubseteq C_2 & R \sqsubseteq S \\ & R_1 \circ \cdots \circ R_n \sqsubseteq S \end{array}$$

Two concepts C_1 and C_2 are said to be equivalent, denoted as $C_1 \equiv C_2$, if and only if $C_1 \sqsubseteq C_2$ and $C_2 \sqsubseteq C_1$. Likewise, $R_1 \equiv R_2$ iff $R_1 \sqsubseteq R_2$ and $R_2 \sqsubseteq R_1$.

Example 3. $\mathcal{SROIQ}(\mathbf{D})$ *TBox axioms modeling a university domain.*

$$\begin{aligned} \mathit{PostgradStudent} &\sqsubseteq \mathit{Student} \\ \mathit{UndergradStudent} &\sqsubseteq \mathit{Student} \\ \mathit{Department} &\sqsubseteq \mathit{Faculty} \\ \mathit{Faculty} &\sqsubseteq \mathit{University} \\ \mathit{Staff} \sqcup \mathit{Student} &\sqsubseteq \bot \\ \mathit{Professor} &\sqsubseteq \mathit{Person} \\ \mathit{Student} &\sqsubseteq \mathit{Person} \\ \mathit{Chair} &\equiv \mathit{Person} \sqcap \exists \mathit{headOf}.\mathit{Department} \\ \mathit{Student} &\equiv \mathit{Person} \sqcap \exists \mathit{takesCourse}.\mathit{Course} \\ \mathit{Professor} &\equiv \mathit{Person} \sqcap \exists \mathit{givesCourse}.\mathit{Course} \\ \exists \mathit{headOf}.\top &\sqsubseteq \mathit{Professor} \\ \mathit{takesCourse} &\equiv \mathit{givesCourse}^{-} \end{aligned}$$

Here, we provide a small textual explanation for some of the TBox axioms shown above. The first axiom states that every postgraduate student is a student, the sixth axiom defines that every professor is a person, the ninth axiom asserts that every student is a person and takes a certain course and vice versa.

Semantics An interpretation, $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$, consists of a non-empty domain $\Delta^{\mathcal{I}}$ and an interpretation function $\cdot^{\mathcal{I}}$ that assigns to each object name o an element $o^{\mathcal{I}} \in \Delta^{\mathcal{I}}$, to each atomic concept A a subset $A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$ of the domain, and to each atomic role R a binary relation $R^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$ over the domain. The role and concept constructs can be interpreted in $\mathcal{I}$ as depicted in Tables 5.1.

Construct Name	Syntax	Semantics	
top concept	$\top$	$\top^{\mathcal{I}} = \Delta^{\mathcal{I}}$	$\mathcal{ALC}$
atomic concept	A	$A^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}$	
atomic role	R	$R^{\mathcal{I}} \sqsubseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}$	
conjunction	$C \sqcap D$	$C^{\mathcal{I}} \cap D^{\mathcal{I}}$	
disjunction	$C \sqcup D$	$C^{\mathcal{I}} \cup D^{\mathcal{I}}$	
negation	$\neg C$	$\Delta^{\mathcal{I}} \setminus C^{\mathcal{I}}$	
exists restriction	$\exists R.C$	$\{x \mid \exists y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\}$	
value restriction	$\forall R.C$	$\{x \mid \forall y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ implies } y \in C^{\mathcal{I}}\}$	
concept hierarchy	$C \sqsubseteq D$	$C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$	
role hierarchy	$R \sqsubseteq S$	$R^{\mathcal{I}} \subseteq S^{\mathcal{I}}$	$\mathcal{H}$
inverse role	R^{-}	$\{\langle x, y\rangle \mid \langle y, x\rangle \in R^{\mathcal{I}}\}$	$\mathcal{I}$
transitive role	$R \in R_{+}$	$R^{\mathcal{I}} = (R^{\mathcal{I}})^{+}$	$\mathcal{S}$
role chains	$R_1 \circ \cdots \circ R_n \sqsubseteq S$	$R_1^{\mathcal{I}} \circ \cdots \circ R_n^{\mathcal{I}} \subseteq S^{\mathcal{I}}$	$\mathcal{R}$
nominal	$\{o\}$	$\{o^{\mathcal{I}}\}$	$\mathcal{O}$
number restriction	$\geq n\ R$	$\{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}}\} \geq n\}$	$\mathcal{N}$
	$\leq n\ R$	$\{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}}\} \leq n\}$	
qualified number restriction	$\geq n\ R.C$	$\{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \geq n\}$	$\mathcal{Q}$
	$\leq n\ R.C$	$\{x \mid \sharp\{y.\langle x, y\rangle \in R^{\mathcal{I}} \text{ and } y \in C^{\mathcal{I}}\} \leq n\}$	

Table 5.1: Syntax and semantics of the $\mathcal{ALC}$ and $\mathcal{S}$ families of description Logics (courtesy of [Baader *et al.* 2007, Horrocks & Patel-Schneider 2010]).

An interpretation $\mathcal{I}$ *satisfies* an inclusion $C \sqsubseteq D$ iff $C^{\mathcal{I}} \subseteq D^{\mathcal{I}}$, and it satisfies an equality $C \equiv D$ iff $C^{\mathcal{I}} = D^{\mathcal{I}}$. If $\mathcal{T}$ is a set of axioms, then $\mathcal{I}$ satisfies $\mathcal{T}$ iff $\mathcal{I}$ satisfies each element of $\mathcal{T}$. If $\mathcal{I}$ satisfies an axiom (resp. a set of axioms), then we say that it is a *model* of this axiom (resp. set of axioms). Two axioms or two sets of axioms are *equivalent* if they have the same models.

Description logic datatype syntax and semantics Datatypes restrict the interactions between concrete and "abstract" parts of a knowledge base so as to avoid problems of undecidability and to simplify implementation, and are widely used in on-

Construct Name	Syntax	Semantics
datatype	D	$D^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}}_{\mathbf{D}}$
data value	v	$v^{\mathcal{I}} \in \Delta^{\mathcal{I}}_{\mathbf{D}}$
concrete role	T	$T^{\mathcal{I}} \subseteq \Delta^{\mathcal{I}} \times \Delta^{\mathcal{I}}_{\mathbf{D}}$
enumerated datatype	$\{v_1, \ldots, v_n\}$	$\{v_1^{\mathcal{I}}, \ldots, v_n^{\mathcal{I}}\}$
exists restriction	$\exists T.D$	$\{x \mid \exists y.\langle x, y\rangle \in T^{\mathcal{I}}$ and $y \in D^{\mathcal{I}}\}$
value restriction	$\forall T.D$	$\{x \mid \forall y.\langle x, y\rangle \in T^{\mathcal{I}}$ implies $y \in D^{\mathcal{I}}\}$
number restriction	$\geq n\ T$	$\{x \mid \sharp\{y.\langle x, y\rangle \in T^{\mathcal{I}}\} \geq n\}$
	$\leq n\ T$	$\{x \mid \#\{y.\langle x, y\rangle \in T^{\mathcal{I}}\} \leq n\}$
qualified number restriction	$\geq n\ T.D$	$\{x \mid \sharp\{y.\langle x, y\rangle \in T^{\mathcal{I}}$ and $y \in D^{\mathcal{I}}\} \geq n\}$
	$\leq n\ T.D$	$\{x \mid \sharp\{y.\langle x, y\rangle \in T^{\mathcal{I}}$ and $y \in D^{\mathcal{I}}\} \leq n\}$

Table 5.2: Syntax and Semantics of Description Logics Datatypes (taken from [Horrocks & Patel-Schneider 2010])

tology languages, including OWL and OWL 2. The syntax and semantics of datatypes is summarised in Table 5.2, where D is a datatype name, T is a concrete role, v is a data value and n is a non-negative integer. An interpretation, $\mathcal{I} = (\Delta^{\mathcal{I}}_{D}, \cdot^{\mathcal{I}})$, consists of a non-empty concrete domain $\Delta^{\mathcal{I}}_{D}$ and an interpretation function $\cdot^{\mathcal{I}}$.

Assertions about individuals In an ABox (Assertional Box), one describes a specific state of affairs of an application domain in terms of concepts and roles. Some of the concept and role atoms in the ABox may be defined names of the TBox. In the ABox, one introduces individuals, by giving them names, and one asserts properties of these individuals. We denote individual names by IRIs such as $o, o_1, o_2, \ldots, o_n$. Using concepts C and roles R, in an ABox, one can make two kinds of assertions:

$$o : C \qquad (o_1, o_2) : R$$

The first one, $o : C$, is called *concept assertion*: it states that o belongs to (the interpretation of) C, formally, $o^{\mathcal{I}} \subseteq C^{\mathcal{I}}$. The second one, $(o_1, o_2) : R$, is called *role assertion*: it states that o_1 is related by the role R with o_2, formally, $(o_1^{\mathcal{I}}, o_2^{\mathcal{I}}) \subseteq R^{\mathcal{I}}$. An ABox, denoted as $\mathcal{A}$, is a finite set of such assertions.

Example 4. *This example shows a set of ABox assertions that model a university domain.*

$$\begin{aligned}\mathcal{A} \ = \{&Jerome : Professor,\\ &SemanticWeb : Course,\\ &(Jerome, SemanticWeb) : givesCourse\}\end{aligned}$$

For expressive ontology languages, query entailment (and hence containment) in DLs ranging from $\mathcal{ALCI}$ to $\mathcal{SHIQ}$ is shown to be 2ExpTime in [Lutz 2008, Glimm

et al. 2008, Ortiz *et al.* 2008a, Eiter *et al.* 2009]. See Table 5.3 for a partial summary of the studies on query answering.

DL	Axioms	Entailment
$\mathcal{ALC}$	$C_1 \sqsubseteq C_2$	ExpTime [Lutz 2008]
$\mathcal{ALCH}$	$C_1 \sqsubseteq C_2, R_1 \sqsubseteq R_2$	ExpTime [Ortiz *et al.* 2008b]
$\mathcal{ALCI}$	$C_1 \sqsubseteq C_2, R_1 \sqsubseteq R_2$	2ExpTime-hard [Lutz 2008]
$\mathcal{ALCHI}$	$C_1 \sqsubseteq C_2, R_1 \sqsubseteq R_2$	2ExpTime [Calvanese *et al.* 1998]
$\mathcal{SH}$	$C_1 \sqsubseteq C_2, R_1 \sqsubseteq R_2$	2ExpTime-hard [Eiter *et al.* 2009]
$\mathcal{SHIQ}$	$C_1 \sqsubseteq C_2, R_1 \sqsubseteq R_2$	2ExpTime-complete [Glimm *et al.* 2008]
$\mathcal{SHOQ}$	$C_1 \sqsubseteq C_2, R_1 \sqsubseteq R_2$	2ExpTime-complete [Glimm *et al.* 2008]

Table 5.3: The complexity of query entailment for the fragments of $\mathcal{SHOIQ}$.

5.2.5 The SPARQL Query Language

SPARQL [Prud'hommeaux & Seaborne 2008] is a W3C recommended language for querying RDF graphs. We refer the reader to [Prud'hommeaux & Seaborne 2008] for the whole definition of the SPARQL standard, and to [Pérez *et al.* 2009b, Pérez *et al.* 2009a] for a more formal presentation of the syntax and semantics of SPARQL. In this Section we try to limit the introduced concepts to the minimum so that this Chapter is self-contained; while trying to give intuitions and examples for the most important notions.

5.2.5.1 Abstract Syntax

We consider an abstract syntax for SPARQL queries [Pérez *et al.* 2009a]. A triple pattern is a tuple $t \in \text{UBV} \times \text{UV} \times \text{UBLV}$. We recall that U is the set of URI references, B the set of blank nodes, L the set of literals and V a set of variables, all disjoint.

SPARQL is based on the notion of query patterns defined inductively from triple patterns. Triple patterns grouped together using SPARQL operators (AND, UNION, OPT[1]) form *query patterns* (or graph patterns). We do not consider FILTER query patterns here.

Definition 1 (Query Pattern)**.** *A* query pattern q *is inductively defined as follows:*

$$q ::= \; t \mid q_1 \textit{ AND } q_2 \mid q_1 \textit{ UNION } q_2 \mid q_1 \textit{ OPT } q_2$$

Typical SPARQL queries are built around the classical pattern SELECT U FROM G WHERE P such that FROM identifies an RDF graph G on which the query will be

[1] OPT is short for OPTIONAL.

evaluated, WHERE contains a graph pattern P that the query answers should satisfy and SELECT singles out the distinguished variables $U \subseteq V$ in the graph pattern (see Example 8).

Definition 2. *A* SPARQL SELECT query *is a query of the form* $q\{\overrightarrow{w}\}$ *where* $\overrightarrow{w}$ *is a tuple of variables that appear in* q *which are called* distinguished variables*, and* q *is a query pattern.*

Example 5 (SPARQL queries)**.** *Consider the following queries* $q_1\{?x\}$ *and* $q_2\{?x\}$ *on the graph of Example 1.* q_1 *selects all those who translated or wrote a poem whereas* q_2 *finds those who translated a poem or wrote anything else.*

q_1

```
SELECT ?x  WHERE  {
   { {?x ex:translated ?l} UNION {?x ex:wrote ?l} }
   ?l rdf:type ex:Poem .
}
```

q_2

```
SELECT ?x  WHERE {
  {?x ex:translated ?l . ?l rdf:type ex:Poem .} UNION
  {?x ex:wrote ?l }
}
```

q_1 *finds all those authors who either translated or wrote a poem whereas* q_2 *selects those authors who translated a poem or wrote something.*

Example 6. *This query selects, author names', where they live in, and the population of the city they live in, for those who wrote a poem and live in the same city they were born in.*

q_3

```
SELECT ?n ?loc ?p  WHERE {
   ?x ex:wrote ?l .
   ?x ex: hasName ?n
   ?l rdf:type ex:Poem .
   ?x ex:livesIn ?loc .
   ?x ex:bornIn ?loc .
   ?loc ex:population ?p .
}
```

5.2.5.2 Semantics

Intuitively, an answer for a SPARQL query is an assignment of values to the distinguished variables such that the query pattern is satisfied. One classical interpretation

of a SPARQL query is thus the set of all answers[2]. For a query q, we denote the set of answers with respect to a graph G as $ANS(q, G)$.

Example 7 (Answers to SPARQL queries)**.** *The answers to query* $q_1\{?x\}$ *and* $q_2\{?x\}$ *of Example 5 on graph* G *of Example 1 are respectively* $\{\langle \text{Poe}\rangle, \langle \text{Mallarme}\rangle\}$ *and* $\{\langle \text{Baudelaire}\rangle, \langle \text{Poe}\rangle, \langle \text{Mallarme}\rangle\}$*. Hence,* $[\![q_1\{?x\}]\!]_G \subseteq [\![q_2\{?x\}]\!]_G$*.*

Formally, the semantics of SPARQL queries is given by a partial mapping function ρ from V to UBL. The *domain* of ρ, $dom(\rho)$, is the subset of V on which ρ is defined. Two mappings ρ_1 and ρ_2 are said to be *compatible* if $\forall x \in dom(\rho_1) \cap dom(\rho_2)$, $\rho_1(x) = \rho_2(x)$. $\rho_1 \cup \rho_2$ is also a mapping (we use $\uplus$ when $\rho_1 \cap \rho_2 = \emptyset$). This allows for defining the join, union, and difference operations between two sets of mappings M_1, and M_2:

$$
\begin{aligned}
M_1 \bowtie M_2 &= \{\rho_1 \cup \rho_2 \mid \rho_1 \in M_1, \rho_2 \in M_2 \text{ are compatible mappings}\} \\
M_1 \cup M_2 &= \{\rho \mid \rho \in M_1 \text{ or } \rho \in M_2\} \\
M_1 \setminus M_2 &= \{\rho \in M_1 \mid \forall \rho_2 \in M_2, \rho \text{ and } \rho_2 \text{ are not compatible}\}
\end{aligned}
$$

The *evaluation* of query patterns over an RDF graph G is inductively defined:

$$
\begin{aligned}
[\![.]\!]_G &: q \rightarrow 2^{V \times UBL} \\
[\![t]\!]_G &= \{\rho \mid dom(\rho) = var(t) \text{ and } \rho(t) \in G\}
\end{aligned}
$$

where $var(t)$ is the set of variables occurring in t.

$$
\begin{aligned}
[\![q_1 \text{ AND } q_2]\!]_G &= [\![q_1]\!]_G \bowtie [\![q_2]\!]_G \\
[\![q_1 \text{ UNION } q_2]\!]_G &= [\![q_1]\!]_G \cup [\![q_2]\!]_G \\
[\![q_1 \text{ OPT } q_2]\!]_G &= ([\![q_1]\!]_G \bowtie [\![q_2]\!]_G) \cup ([\![q_1]\!]_G \setminus [\![q_2]\!]_G) \\
[\![q_1 \text{ MINUS } q_2]\!]_G &= [\![q_1]\!]_G \setminus [\![q_2]\!]_G \\
[\![q\{\overrightarrow{w}\}]\!]_G &= \pi_{\overrightarrow{w}}([\![q]\!]_G)
\end{aligned}
$$

The projection operator $\pi_{\overrightarrow{w}}$ selects only those part of the mappings relevant to variables in $\overrightarrow{w}$. The semantics of FILTER expressions is defined as: given a mapping ρ and a SPARQL constraint C, we say that ρ satisfies C, denoted by $\rho(C) = \top$, if:

- $C = BOUND(x)$ with $x \in dom(\rho)$,
- $C = (x = c)$ with $x \in dom(\rho)$ and $\rho(x) = c$,
- $C = (x = y)$ with $x, y \in dom(\rho)$ and $\rho(x) = \rho(y)$,
- $C = (x! = c)$ with $x \in dom(\rho)$ and $\rho(x) \neq c$,
- $C = (x! = y)$ with $x, y \in dom(\rho)$ and $\rho(x) \neq \rho(y)$,
- $C = (x < c)$ with $x \in dom(\rho)$ and $\rho(x) < c$,

[2]We consider SPARQL under set semantics which is practical in most cases and for which we show that query containment is decidable.

- $C = (x < y)$ with $x, y \in dom(\rho)$ and $\rho(x) < \rho(y)$,
- $C = (!C_1)$ with ρ does not satisfy C_1, in the following, we use $\rho(C) = \bot$ to denote this,
- $C = (C_1 \mid\mid C_2)$ with $\rho(C_1) = \top$ or $\rho(C_2) = \top$,
- $C = (C_1 \ \&\& \ C_2)$ with $\rho(C_1) = \top$ and $\rho(C_2) = \top$

It should be noted that there are semantic differences between the standard SPARQL semantics [Prud'hommeaux & Seaborne 2008] and the semantics that we use here (from [Pérez *et al.* 2009a]), e.g. for FILTERs within OPTIONALs, or with respect to the 3 valued semantics of FILTER expressions i.e., when evaluating FILTER expressions the answers can be either true, false, or error. For more details on the differences, we refer the reader to [Polleres 2012].

Definition 3 (Answers to a SPARQL query)**.** *Let $q\{\overrightarrow{w}\}$ be a SPARQL query, P its graph pattern, and G be an RDF graph, the set of* answers *to this query is defined as follows:*

$$[\![q\{\overrightarrow{w}\}]\!]_G = \{\rho \mid \rho \in \pi_{\overrightarrow{w}}([\![P]\!]_G)\}$$

5.2.5.3 Acyclic SPARQL queries.

Following the tradition from databases, we consider SPARQL queries as graphs. More specifically, a SPARQL query is represented as a bipartite graph, with two kinds of nodes: triple nodes and term nodes (representing URIs, blank nodes, and literals), as in Figure 5.2. If this graph contains a cycle going uniquely through variable and triple nodes, then the query is cyclic.

```
SELECT ?x WHERE {
  ?x :married ?y . ?y :knows ?z .
  ?z :knows ?r . ?r :knows ?y .
}
```

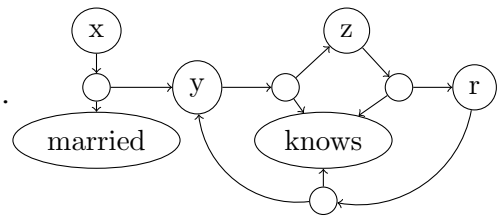


Figure 5.2: Cyclic query.

5.2.5.4 Query Arity

A *unary* SPARQL query is a query with one distinguished variable. Queries of Example 5 above are *unary*. In the general case, SPARQL queries are *n-ary*: they have n distinguished variables. The query of example 6 above is a 3-ary (ternary) query.

The arity of a query corresponds to the arity of the query answers. When there is an outer projection (i.e. a non-empty SELECT clause) it is defined by the number of distinguished variables, otherwise, it is defined by all free variables of the query. This is analogous to the arity of a predicate in logic.

5.3 SPARQL Query Containment

Intuitively, we say that a query is contained into another if, for any RDF graph, its answers are included in those of the other query.

Definition 4 (SPARQL query containment)**.** *Given queries q and q' with the same arity, q is contained in q', denoted $q \sqsubseteq q'$, if and only if $ANS(q, G) \subseteq ANS(q', G)$ for every RDF graph G.*

Example 8. *Consider the queries of Figure 5.3 that retrieve student information from a university dataset. We have $Qb \sqsubseteq Qa$ and $Qa \not\sqsubseteq Qb$.*

Select the URI and name of all students taking a course in either computer science or mathematics.

```
SELECT ?x ?y
WHERE {                                  Qa
    ?x a :Student . ?x :name ?y .
    ?x :takesCourse ?c .
    { ?c rdf:type :CsCourse . }
    UNION
    { ?c rdf:type :MathCourse . }
}
```

Select the URI and name of all master students taking a course in computer science.

```
SELECT ?x ?y
WHERE {                                  Qb
    ?x a :Student . ?x :name ?y .
    ?x :masterDegreeFrom ?master .
    ?x :takesCourse ?c .
    ?c rdf:type :CsCourse .
}
```

Figure 5.3: Two sample queries for which a containment relation holds.

The definitions of SPARQL query answers and of query containment exclude testing containment of query of differing arities. Indeed, it makes little sense in practice to replace a query with another query of a different arity. Projection can be used for making queries comparable.

When the queried graph is assumed to satisfy a particular ontology or schema, it is useful to take this schema into account because two queries may not be contained in each other without the schema, but they may under the schema constraints.

Definition 5 (SPARQL query containment with respect to a schema)**.** *Given a set of RDFS axioms $\mathcal{S}$ and two queries q and q' with the same arity, q is contained in q' with respect to $\mathcal{S}$, denoted $q \sqsubseteq_{\mathcal{S}} q'$, if and only if $ANS(q, G) \subseteq ANS(q', G)$ for every RDF graph G satisfying $\mathcal{S}$.*

5.4 μ-calculus

In this section, we present the μ-calculus that we use to encode the containment problem for a SPARQL fragment. The μ-calculus is a logic obtained by adding fixpoint operators to ordinary modal logic, or Hennessy-Milner logic [Kozen 1983]. The result is a very expressive logic, sufficient to subsume many other temporal logics such as CTL and CTL* [Blackburn *et al.* 2007]. The modal μ-calculus is easy to model-check,

and so makes a good 'back-end' logic for tools. In this chapter, we mainly use the μ-calculus with nominals and converse modalities.

The syntax of the μ-calculus is composed of countable sets of *atomic propositions* and *nominals* AP, a set of *variables* Var, a set of *programs and their respective converses* Prog for navigating in graphs. A μ-calculus formula, φ, can be defined inductively as follows:

$$\varphi \quad ::= \quad \top \mid \bot \mid q \mid X \mid \neg\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \langle a \rangle \varphi \mid [a]\varphi \mid \mu X \varphi \mid \nu X \varphi$$

where $q \in AP, X \in Var$ and $a \in Prog$ is a transition program or its converse $\bar{a}$. The greatest and least fixpoint operators (ν and μ) respectively introduce general and finite recursion in graphs [Kozen 1983]. A *sentence* is a formula with no free variable, i.e., each variable in the formula appears within the scope of μ or ν. Besides, we use the following syntactic sugars:

$$\begin{aligned}
\bot &= \neg\top \\
\varphi \vee \psi &= \neg(\neg\varphi \wedge \neg\psi) \\
[a]\varphi &= \neg\langle a \rangle \neg\varphi \\
\nu X.\varphi(X) &= \neg\mu X.\neg\varphi({}^{\neg X}/_{X}) \\
\varphi \Rightarrow \psi &= \neg\varphi \vee \psi \\
\varphi \Leftrightarrow \psi &= (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)
\end{aligned}$$

The semantics of the μ-calculus is given over a transition system, $K = (S, R, L)$ where S is a non-empty set of nodes, $R : Prog \to 2^{S \times S}$ is the transition function, and $L : AP \to 2^S$ assigns a set of nodes to each atomic proposition or nominal where it holds, such that $L(p)$ is a *singleton* for each nominal p. For converse programs, R can be extended as $R(\bar{a}) = \{(s', s) \mid (s, s') \in R(a)\}$. The valuation function $V : \text{Var} \to 2^S$ maps each variable into a set of nodes. For a valuation V, variable X, and a set of nodes $S' \subseteq S$, $V[X/S']$ is the valuation that is obtained from V by assigning S' to X. The semantics of a formula in terms of a transition system K (a.k.a. Kripke structure) and a valuation function is represented by $[\![\varphi]\!]^K_V \subseteq S$. The semantics of basic μ-calculus

formulae is defined as follows:

$$
\begin{aligned}
[\![\top]\!]_V^K &= S \\
[\![\bot]\!]_V^K &= \emptyset \\
[\![q]\!]_V^K &= L(q), q \in AP,\ L(q) \text{ is singleton if } q \text{ is a nominal} \\
[\![X]\!]_V^K &= V(X), X \in Var \\
[\![\neg\varphi]\!]_V^K &= S \backslash [\![\varphi]\!]_V^K \\
[\![\varphi \wedge \psi]\!]_V^K &= [\![\varphi]\!]_V^K \cap [\![\psi]\!]_V^K \\
[\![\varphi \vee \psi]\!]_V^K &= [\![\varphi]\!]_V^K \cup [\![\psi]\!]_V^K \\
[\![\langle a \rangle \varphi]\!]_V^K &= \{s \in S \mid \exists s' \in S.(s, s') \in R(a)\ \wedge s' \in [\![\varphi]\!]_V^K\} \\
[\![[a]\varphi]\!]_V^K &= \{s \in S \mid \forall s' \in S.(s, s') \in R(a) \Rightarrow s' \in [\![\varphi]\!]_V^K\} \\
[\![\mu X \varphi]\!]_V^K &= \bigcap \{S' \subseteq S \mid [\![\varphi]\!]_{V[X/S']}^K \subseteq S'\} \\
[\![\nu X \varphi]\!]_V^K &= \bigcup \{S' \subseteq S \mid S' \subseteq [\![\varphi]\!]_{V[X/S']}^K\}
\end{aligned}
$$

Note that the evaluation of sentences is independent of valuations and hence we define the following. For a sentence φ, a Kripke structure $K = (S, R, L)$, and $s \in S$, we denote $K, s \models \varphi$ if and only if $s \in [\![\varphi]\!]^K$, henceforth K is considered as a *model* of φ. In other words, K is considered as a model of ϕ if there exists an $s \in S$ such that $K, s \models \phi$. If a sentence has a model, then it is called *satisfiable*.

Another variety of the μ-calculus is the *μ-calculus with graded modalities*. Given a transition program or its converse a and a non-negative integer n, graded modalities generalize standard existential $\langle n, a \rangle$ and universal $[n, a]$ modalities [Kupferman *et al.* 2002]. For instance, $\langle n, a \rangle$ expresses that there exist at least n accessible states satisfying a certain formula and $[n, a] = \neg\langle n, a \rangle\neg$. The full μ-calculus, with graded modalities, converse modalities, and nominals, is undecidable [Bonatti *et al.* 2006] whereas its fragments are well-behaved as shown in Table 5.4, where $\mathcal{O}$ denotes nominals, $\mathcal{N}$ is for number restrictions (graded modalities), and $\mathcal{I}$ denotes converse modalities.

μ **fragments**	**Complexity**	**Source**
μ	ExpTime	[Kozen 1983]
$+\mathcal{N}$	ExpTime	[Kupferman *et al.* 2002]
$+\mathcal{O}+\mathcal{N}$	ExpTime	[Bonatti *et al.* 2006]
$+\mathcal{I}+\mathcal{N}$	ExpTime	[Bonatti *et al.* 2006]
$+\mathcal{O}+\mathcal{I}$	ExpTime	[Sattler & Vardi 2001]
$+\mathcal{O}+\mathcal{I}+\mathcal{N}$	Undecidable	[Bonatti & Peron 2004]

Table 5.4: Fragments of the full modal μ-calculus

To study SPARQL query containment, only a specific subset of the μ-calculus presented above, namely the *alternation-free* modal μ-calculus with nominals and converse [Tanabe *et al.* 2008], is of interest. A μ-calculus formula φ is alternation-free if $\mu X.\varphi_1$ (respectively $\nu X.\varphi_1$) is a subformula of φ and $\nu Y.\varphi_2$ (respectively $\mu Y.\varphi_2$) is a subformula of φ_1 then X does not occur freely in φ_2. For instance, $\nu X.\mu Y.\langle s\rangle Y \wedge a) \vee \langle p\rangle X$ is alternation-free but $\nu X.\mu Y.\langle s\rangle Y \wedge X) \vee a$ is not since X bound by ν appears freely in the scope of μY.

5.5 RDF Graphs as Transition Systems

In this section, we show how to translate RDF graphs into labeled transition systems. First of all, translating RDF graphs into transition systems is necessary in order to restrict the models of the μ-calculus formula obtained from the translation of queries. Additionally, if RDF graphs can be translated into transition systems, then model checking can be used to evaluate SPARQL queries. In fact, in this regard, there is already some progress as presented in the literature [Mateescu *et al.* 2009] where it is possible to extend and encode SPARQL queries in a logic and use model checking to evaluate the result of the query.

There are several ways of encoding RDF graphs as transition systems, for instance, consider the following:

- for each triple $(s, p, o) \in G$, s and o become nodes of the transition system and p is a transition program, there is an edge $\langle s, o\rangle$ where transition from node s to o and vice versa can be done using program p and its converse $\bar{p}$ respectively. While this approach is simple and intuitive, it does not work in the general case, i.e., in an RDF graph predicates or properties can also be nodes in an RDF graph as shown in Figure 5.4, thus, p cannot be a transition program.

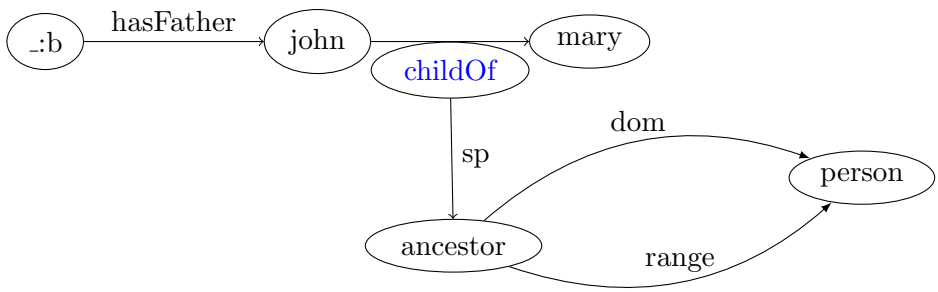


Figure 5.4: An RDF graph where a predicate appears as a node

- for each triple $(s, p, o) \in G$, s, p, and o become atomic propositions that are true in the states n_s, n_p, and n_o respectively of a transition system, there are are edges $\langle n_s, n_p\rangle$ and $\langle n_p, n_o\rangle$ that are accessible through transition programs 1 and 2 respectively. Similar to the above approach, this translation procedure does

not work in the general case when encoding RDF schema graphs. For example, consider an RDF graph that contains the triple (subPropertyOf, subPropertyOf, subPropertyOf).

- the last approach considers encoding RDF graphs as bipartite graphs, i.e., for each $t = (s, p, o) \in G$ introduce two sets of nodes in the transition system: one set for each triple n_t and another set for each element of the triple n_s, n_p, and n_o where atomic propositions s, p and o are set to be true respectively. Additionally, there are edges $\langle n_s, n_t \rangle$, $\langle n_t, n_p \rangle$, and $\langle n_t, n_o \rangle$ in the transitions system that are accessible through programs s, p, o and their converses respectively. The idea of representing RDF triples as other types of graphs (for instance, hypergraphs) was first introduced in [Baget 2005], in fact, this translation coincides with the notion of reification of n-ary relations [Calvanese *et al.* 2008] that is one edge from the triple node to subject, predicate, and object nodes of the triple in this case. This approach overcomes the limitations of the other two approaches. Thus, in the following, we discuss in detail how this technique works.

5.5.1 Encoding of RDF graphs

An RDF graph is encoded as a transition system in which nodes correspond to RDF entities and RDF triples. Edges relate entities to the triples they occur in. Different edges are used for distinguishing the functions (subject, object, predicate). Expressing predicates as nodes, instead of atomic programs, makes it possible to deal with full RDF expressiveness in which a predicate may also be the subject or object of a statement.

Definition 6 (Transition system associated to an RDF graph)**.** *Given an RDF graph, $G \subseteq UB \times U \times UBL$, the transition system associated to G, $\sigma(G) = (S, R, \mathsf{L})$ over $AP = \mathrm{U_G B_G L_G} \cup \{s', s''\}$, is such that:*

- *$S = S' \cup S''$ with S' and S'' the smallest sets such that $\forall u \in U_G, \exists n^u \in S'$, $\forall b \in B_G, \exists n^b \in S'$, $\forall l \in L_G, \exists n^l \in S'$, and $\forall t \in G, \exists n^t \in S''$,*
- *$\forall t = (s, p, o) \in G$, $\langle n^s, n^t \rangle \in R(s)$, $\langle n^t, n^p \rangle \in R(p)$, and $\langle n^t, n^o \rangle \in R(o)$,*
- *$\mathsf{L} : AP \to 2^S$; $\forall u \in U_G, \mathsf{L}(u) = \{n^u\}$, $\forall b \in B_G, \mathsf{L}(b) = S'$, $\mathsf{L}(s') = S'$, $\forall l \in L_G, \mathsf{L}(l) = \{n^l\}$ and $\mathsf{L}(s'') = S''$,*
- *$\forall n^t, n^{t'} \in S''$, $\langle n^t, n^{t'} \rangle \in R(d)$.*

The program d is introduced to render each triple accessible to the others and thus facilitate the encoding of queries. The function σ associates what we call a *restricted transition system* to any RDF graph. Formally, we say that a transition system K is a *restricted transition system* iff there exists an RDF graph G such that $K = \sigma(G)$.

A restricted transition system is thus a bipartite graph composed of two sets of nodes: S', those corresponding to RDF entities, and S'', those corresponding to RDF triples. For example, Figure 5.5 shows the restricted transition system associated with the graph of Example 1.

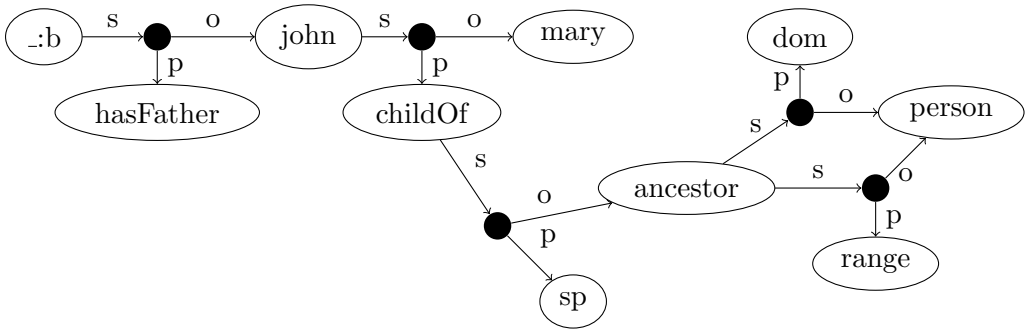


Figure 5.5: Transition system encoding the RDF graph of Example 1. Nodes in S'' are black anonymous nodes; nodes in S' are the other nodes (d-transitions are not displayed).

Given that the logic chosen to determine containment is μ-calculus with nominals (lacking functionality or number restrictions), one cannot impose that each triple node is connected to exactly one node for each of the three triple-components (subject, predicate, and object). However, we can impose a lighter restriction to achieve this by taking advantage of the technique introduced in [Genevès & Layaïda 2006]. Since it is not possible to ensure that there is only one successor, then we restrict all the successors to bear the same constraints. They thus become interchangeable (bisimulation). To do this, we introduce a rewriting function f such that all occurrences of $\langle a \rangle \varphi$ (existential formulas) are replaced by $\langle a \rangle \top \wedge [a]\varphi$. As such, f is inductively defined on the structure of a μ-calculus formula as follows:

$$
\begin{aligned}
f(\top) &= \top \\
f(q) &= q \quad q \in AP \cup Nom \\
f(X) &= X \quad X \in Var \\
f(\neg\varphi) &= \neg f(\varphi) \\
f(\varphi \wedge \psi) &= f(\varphi) \wedge f(\psi) \\
f(\varphi \vee \psi) &= f(\varphi) \vee f(\psi) \\
f(\langle a \rangle \varphi) &= \langle a \rangle \top \wedge [a] f(\varphi) \quad a \in \{\bar{s}, p, o\} \\
f(\langle a \rangle \varphi) &= \langle a \rangle f(\varphi) \quad a \in \{d, s, \bar{p}, \bar{o}\} \\
f([a]\varphi) &= [a] f(\varphi) \quad a \in \text{Prog} \\
f(\mu X.\varphi) &= \mu X.f(\varphi) \\
f(\nu X.\varphi) &= \nu X.f(\varphi)
\end{aligned}
$$

Thus, when checking for query containment, we assume that the formulas are rewritten using function f. Along with that, we also consider the following restrictions:

- The set of programs is fixed: $Prog = \{s, p, o, d, \bar{s}, \bar{p}, \bar{o}, \bar{d}\}$.
- A model must be a restricted transition system.

The last constraint can be expressed in the μ-calculus as follows:

Proposition 1 (RDF restriction on transition systems)**.** *Let φ be a formula that can be stated over a restricted transition system, φ is satisfied by some restricted transition system if and only if $f(\varphi) \wedge \varphi_r$ is satisfied by some transition system over Prog, i.e. $\exists K_r.[\![\varphi]\!]^{K_r} \neq \emptyset \iff \exists K.[\![f(\varphi) \wedge \varphi_r]\!]^{K} \neq \emptyset$, where:*

$$\varphi_r = \quad \nu X.\ \theta \wedge \kappa\ \wedge (\neg\langle d\rangle\top \vee \langle d\rangle X)$$

in which $\theta = \langle\bar{s}\rangle s' \wedge \langle p\rangle s' \wedge \langle o\rangle s' \wedge \neg\langle s\rangle\top \wedge \neg\langle\bar{p}\rangle\top \wedge \neg\langle\bar{o}\rangle\top$, and
$\kappa = [\bar{s}]\xi \wedge [p]\xi \wedge [o]\xi$ with

$$\xi = \neg\langle\bar{s}\rangle\top \wedge \neg\langle o\rangle\top \wedge \neg\langle p\rangle\top \wedge \neg\langle d\rangle\top \wedge \neg\langle\bar{d}\rangle\top \wedge \neg\langle s\rangle s' \wedge \neg\langle\bar{o}\rangle s' \wedge \neg\langle\bar{p}\rangle s'$$

The formula φ_r ensures that θ and κ hold in every node reachable by a d edge, i.e., in every s'' node. The formula θ forces each s'' node to have a subject, predicate and object. The formula $f(\varphi)$ enforces reification (makes sure that each s'' node is connected to one subject, one predicate, and one object node). The formula κ navigates from a s'' node to every reachable s' node, and forces the latter not to be directly connected to other subject, predicate or object nodes.

Proof. ($\Rightarrow$) Assume that $\exists K_r[\![f(\varphi)]\!]^{K_r} \neq \emptyset$, since φ_r is satisfied by any restricted transition system, one gets $[\![\varphi_r]\!]^{K_r} \neq \emptyset$. Hence it follows that, $\exists K_r[\![f(\varphi)]\!]^{K_r} \neq \emptyset$ and $[\![\varphi_r]\!]^{K_r} \neq \emptyset$ which imply $\exists K_r[\![f(\varphi)]\!]^{K_r} \wedge [\![\varphi_r]\!]^{K_r} \neq \emptyset$. From this, using the semantics of μ-calculus formula, one obtains $\exists K_r[\![f(\varphi) \wedge \varphi_r]\!]^{K_r} \neq \emptyset$. Since a restricted transition system is also a transition system, $K_r \subseteq K$, it follows that $\exists K.[\![f(\varphi) \wedge \varphi_r]\!]^{K} \neq \emptyset$.
($\Leftarrow$) Assume that $\exists K[\![f(\varphi) \wedge \varphi_r]\!]^{K} \neq \emptyset$. We construct a restricted transition system model $K_r = (S_r = S_r^{'} \cup S_r^{''}, R_r, L_r)$ and a function $g : K_r \to K$ from $K = (S, R, L)$. Add a node n_0' to S_r with $g(n_0') = n_0$ where $f(\varphi) \wedge \varphi_r$ is satisfied in K. Suppose we have constructed a node n_r of S_r. For $j \in \{s, p, o\}$, if there is $n \in S$ with $(g(n_r), n) \in R(j)$, then pick one such n and add a node n_r' to S_r with $g(n_r') = n$. In such construction, if there are concurrent $\bar{s}, p, o$ transitions from an $S_r^{''}$ node, we retain one transition for each modality. This is because, if such transitions are part of the model that satisfy $f(\varphi) \wedge \varphi_r$, then they will be under the influence of the constraints $f(.)$ and φ_r, and will bear these constraints. However, if they belong to K that does not satisfy the aforementioned formula, then cutting them will not affect the capacity of the model to be a model for the formula. Finally, for an atomic proposition p, $L_r(p) = \{n_r \in S_r \mid g(n_r) \in L(p)\}$.
The RDF triple structure is maintained in K_r i.e. $\langle(s, s''), (s'', p), (s'', o)\rangle$ is valid throughout the graph. If there were node pairs outside of this structure, then φ_r will not be satisfied. Throughout the graph, θ, $f(.)$ and κ ensure that for each triple node $s'' \in S_r$, there exists an incoming subject edge, an outgoing property edge, and an outgoing object edge. Hence, $[\![\varphi_r]\!]^{K_r} \neq \emptyset$.
To verify that $[\![f(\varphi)]\!]^{K_r} \neq \emptyset$, it is enough to show that $[\![f(\varphi)]\!]^{K} \neq \emptyset \Rightarrow [\![f(\varphi)]\!]^{K_r} \neq \emptyset$ by induction on the structure of $f(\varphi)$. $\square$

If a μ-calculus formula ψ appears under the scope of a least μ or greatest ν fixed point operator over all the programs $\{s, p, o, d, \bar{s}, \bar{p}, \bar{o}, \bar{d}\}$ as, $\mu X.\psi \vee \langle s\rangle X \vee \langle p\rangle X \vee \cdots$ or $\nu X.\psi \wedge \langle s\rangle X \wedge \langle p\rangle X \wedge \cdots$, then, for the sake of legibility, we denote the formulae by $\mathit{lfp}(X, \psi)$ and $\mathit{gfp}(X, \psi)$, respectively.

So far we have showed how RDF graphs can be translated into transition systems over which the μ-calculus formulas are translated. In the following, we propose methods that are used to encode schema axioms and queries as μ-calculus formulas. Thus, at a later point, we use these encodings to reduce containment test into the validity problem.

5.5.2 Encoding $\mathcal{ALCH}$ Schemas

In this section, we provide the encoding of $\mathcal{ALCH}$ axioms. These encodings are used together with query encodings to determine if any two queries are contained in each other.

Definition 7 (μ-calculus encoding of an $\mathcal{ALCH}$ schema)**.** *Given a set of axioms $c_1, c_2, ..., c_n$ of a schema $\mathcal{C}$, the μ-calculus encoding of $\mathcal{C}$ is:*

$$\eta(\mathcal{C}) = \eta(c_1) \wedge \eta(c_2) \wedge ... \wedge \eta(c_n).$$

Where η translates each axiom into an equivalent formula using ω which recursively encodes concepts and roles:

- *Concept Inclusion*

$$\begin{aligned}
\eta(C_1 \sqsubseteq C_2) &= \mathit{gfp}\big(X, \omega(C_1) \Rightarrow \omega(C_2)\big)\\
\omega(\bot) &= \bot\\
\omega(A) &= A\\
\omega(\neg C) &= \neg\omega(C)\\
\omega(C_1 \sqcap C_2) &= \omega(C_1) \wedge \omega(C_2)\\
\omega(\exists R.C) &= \langle s\rangle\big(\langle p\rangle R \wedge \langle o\rangle(\langle s\rangle\langle o\rangle\omega(C))\big)\\
\omega(\forall R.C) &= [s]\big([p]R \Rightarrow [o]([s][o]\omega(C))\big)\\
\omega(\exists R^{-}.C) &= \langle \bar{o}\rangle\big(\langle p\rangle R \wedge \langle \bar{s}\rangle(\langle s\rangle\langle o\rangle\omega(C))\big)\\
\omega(\forall R^{-}.C) &= [\bar{o}]\big([p]R \Rightarrow [\bar{s}]([s][o]\omega(C))\big)
\end{aligned}$$

- *Role Inclusion*

$$\eta(R_1 \sqsubseteq R_2) = \mathit{gfp}\big(X, R_1 \Rightarrow R_2\big)$$

Next, we provide procedures to translate unions of conjunctive SPARQL queries, i.e., SPARQL queries that only use UNION and AND, into μ-calculus formulas.

5.6 Encoding SPARQL Query Containment

In this section, we encode queries as μ-calculus formulas. Then, we reduce query containment under schemas to μ-calculus unsatisfiability test and prove the correctness of this reduction.

5.6.1 Encoding Queries as μ-calculus Formulae

In this section, we discuss the encoding of the containment problem $q_1\{\overrightarrow{w}\} \sqsubseteq q_2\{\overrightarrow{w}\}$. For any query $q\{\overrightarrow{w}\}$, we call the variables in $\overrightarrow{w}$ *distinguished* or answer variables. Furthermore, we denote the *non-distinguished* or existential variables in q by $ndvar(q)$, the URIs/constants by $uris(q)$, and the distinguished variables by $dvar(q)$. When encoding $q_1 \sqsubseteq q_2$, we call q_1 left-hand side query and q_2 right-hand side query. q_1 is a union of conjunctive SPARQL query whereas q_2 is a union of conjunctive SPARQL$_{cdfc}$ query (cf. Definition 10).

Queries are translated into μ-calculus formulas. The principle of the translation is that each triple pattern is associated with a sub-formula stating the existence of the triple somewhere in the graph. Hence, they are quantified by μ so as to put them out of the context of a state. In this translation, variables are replaced by nominals or some formula that are satisfied when they are at the corresponding position in such triple relations. A function called $\mathcal{A}$ is used to encode queries inductively on the structure of query patterns. AND and UNION are translated into boolean connectives $\wedge$ and $\vee$ respectively.

Encoding left-hand side query:

The encoding of the left hand side query is done by freezing all the terms in the query. In other words, q is frozen: every term in q becomes a nominal in the μ-calculus.

The function $\mathcal{A}$ computes recursively a μ-calculus formula corresponding to q. The encoding of the SPARQL query is $\mathcal{A}(q)$ such that:

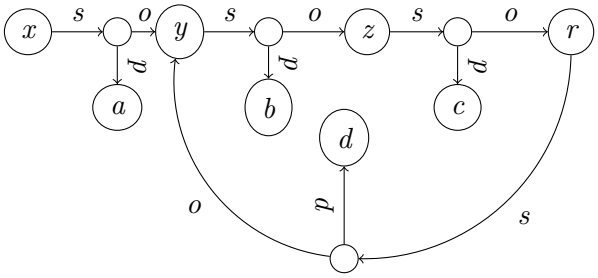
$$\begin{aligned}
\mathcal{A}((x,y,z)) &= \mathit{lfp}\big(X, \langle \bar{s}\rangle x \wedge \langle p\rangle y \wedge \langle o\rangle z\big)\\
\mathcal{A}(q_1 \text{ AND } q_2) &= \mathcal{A}(q_1) \wedge \mathcal{A}(q_2)\\
\mathcal{A}(q_1 \text{ UNION } q_2) &= \mathcal{A}(q_1) \vee \mathcal{A}(q_2)
\end{aligned}$$

In order to encode the right-hand side query, we need to define the notion of cyclic queries.

Definition 8 (Cyclic Query)**.** *A SPARQL query is referred to as cyclic if a transition graph induced from the query patterns is cyclic. The transition graph*[3] *is constructed in the same way as the transition system of Definition 6.*

[3]The transition graph is similar to the tuple-graph used in [Calvanese *et al.* 2008] to detect the dependency among variables.

Example 9. *Let q be the query* $q\{x\} = (x, a, y), (y, b, z), (z, c, r), (r, d, y)$ *where* $ndvar(q) = \{y, z, r\}$ *and* $dvar(q) = \{x\}$. *q is cyclic, as shown graphically,*

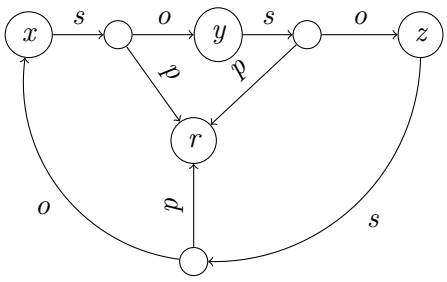


In this example, the cyclic component involves non-distinguished variables and constants $\{b, c, d\}$.

Example 10. *Consider the following cyclic query:*

$$q\{\} = (x, r, y)(y, r, z)(z, r, x)$$

a graph obtained from the graph patterns is shown below:



We can identify various features from this example:

- cyclicity*: the query contains a cycle,*
- distinguished variable-free*: the query does not refer to any distinguished variable,*
- constant-free*: the query does not refer to any constant.*

We refer to such cycles as constant and distinguished variable-free cycles (cdfc) *and denote such queries as* $SPARQL_{cdfc}$.

Definition 9. *A cdfc component of a query is a connected component of the query graph that:*

- *contains no constants,*
- *contains no distinguished variables,*
- *contains a cycle.*

Definition 10. $SPARQL_{cdfc}$ *is the set of SPARQL queries which do not contain any cdfc component.*

Encoding right-hand side query:

the encoding of the right-hand side query q' is different from that of the left due to the non-distinguished variables that appear in cycles in the query. The distinguished variables and constants are encoded as nominals whereas the non-distinguished variables $ndvar(q')$ are encoded as follows:

- If a non-distinguished variable x occurs only once in q', x is encoded as $\top$.

- If a non-distinguished variable appears multiple times in q', then we produce a set of mappings $m = \{m_1, \ldots, m_n\}$ such that each m_i contains formula assignments to the non-distinguished variables. m is produced as follows:

  - we denote the union of the set of distinguished variables and constants of q' by $\mathbf{X}$, i.e., $\mathbf{X} = uris(q') \cup dvar(q')$,

  - for any triple $t = (s, p, o)$, functions f_s, f_p, and f_o return the subject, predicate, and object of t respectively,

$$f_s((s,p,o)) = s$$
$$f_p((s,p,o)) = p$$
$$f_o((s,p,o)) = o$$

  - for each multiply occurring non-distinguished variable x_l, given that $\{x_1, \ldots, x_k\} \in ndvar(q')$, assign it one of the triple patterns $t_j \in q'$ where it appears in, i.e., x_l appears in the triple pattern t_j, from that we obtain m_i's as:

$$m_i = \bigcup_{l=1}^{k} \{x_l \mapsto \alpha(x_l, t_j) \mid x_l \in t_j\}$$

$$\alpha(x,t) = \begin{cases} \langle s \rangle \langle p \rangle f_p(t) & \text{if } x = f_s(t) \text{ and } f_p(t) \in \mathbf{X} \\ \langle s \rangle \langle o \rangle f_o(t) & \text{if } x = f_s(t) \text{ and } f_o(t) \in \mathbf{X} \\ \langle \bar{p} \rangle \langle \bar{s} \rangle f_s(t) & \text{if } x = f_p(t) \text{ and } f_s(t) \in \mathbf{X} \\ \langle \bar{o} \rangle \langle o \rangle f_o(t) & \text{if } x = f_p(t) \text{ and } f_o(t) \in \mathbf{X} \\ \langle \bar{o} \rangle \langle p \rangle f_p(t) & \text{if } x = f_o(t) \text{ and } f_p(t) \in \mathbf{X} \\ \langle \bar{o} \rangle \langle \bar{s} \rangle f_s(t) & \text{if } x = f_o(t) \text{ and } f_s(t) \in \mathbf{X} \end{cases}$$

  Note that there is an exponential number of m_i's in terms of the number of non-distinguished variables. More precisely, there are at most $O(n^k)$ mappings, where n is the number of triples where non-distinguished variables appear, and k is the number of non-distinguished variables.

- Finally, the function $\mathcal{A}$ uses m to encode the query inductively:

$$
\begin{aligned}
\mathcal{A}(q, m) &= \bigvee_{i=1}^{|m|} \mathcal{A}(q, m_i) \\
\mathcal{A}((x, y, z), m) &= \mathit{lfp}\big(X, \langle \bar{s} \rangle d(m, x) \wedge \langle p \rangle d(m, y) \wedge \langle o \rangle d(m, z)\big) \\
\mathcal{A}(q_1 \text{ AND } q_2, m) &= \mathcal{A}(q_1, m) \wedge \mathcal{A}(q_2, m) \\
\mathcal{A}(q_1 \text{ UNION } q_2, m) &= \mathcal{A}(q_1, m) \vee \mathcal{A}(q_2, m) \\
d(m, x) &= \begin{cases} \varphi & \text{if } (x \mapsto \varphi) \in m \\ \top & \text{if } \mathit{unique}(x) \\ x & \text{otherwise} \end{cases}
\end{aligned}
$$

The disjuncts in the encoding guarantee that possible set of substitutions m capture the intended semantics of a cyclic query that contains distinguished variables and constants in its cyclic component.

Example 11 (SPARQL query encoding)**.** *Consider the encoding of $q_1 \sqsubseteq q_2$ of Example 5. To encode q_1, we freeze the variables and constants and proceed with $\mathcal{A}$ such that $\mathcal{A}(q_1) =$*

$$
\begin{aligned}
&\big(\mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge \langle p \rangle \mathit{translated} \wedge \langle o \rangle l\big) \\
&\quad \vee \mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge \langle p \rangle \mathit{wrote} \wedge \langle o \rangle l\big)\big) \wedge \\
&\mathit{lfp}\big(X, \langle \bar{s} \rangle l \wedge \langle p \rangle \mathit{type} \wedge \langle o \rangle \mathit{Poem}\big)
\end{aligned}
$$

To encode q_2, one first computes $m = \{m_1, \ldots, m_n\}$. Given a multiply appearing non-distinguished variable $l \in \mathit{ndvar}(q_2)$, we produce m as follows:

$$
\begin{aligned}
m_1 &= \{l \mapsto \alpha(l, (x, \mathit{translated}, l))\} = \{l \mapsto \langle \bar{o} \rangle \langle p \rangle \mathit{translated}\} \\
m_2 &= \{l \mapsto \alpha(l, (l, \mathit{type}, \mathit{Poem}))\} = \{l \mapsto \langle s \rangle \langle p \rangle \mathit{type}\} \\
m_3 &= \{l \mapsto \alpha(l, (x, \mathit{wrote}, l))\} = \{l \mapsto \langle \bar{o} \rangle \langle p \rangle \mathit{wrote}\}
\end{aligned}
$$

Using $m = \{m_1, m_2, m_3\}$, the encoding of q_2 is produced inductively:

$$\begin{aligned}
\mathcal{A}(q_2, m) = \bigvee_{i=1}^{|m|} \mathcal{A}(q_2, m_i) &= \mathcal{A}(q_2, m_1) \vee \mathcal{A}(q_2, m_2) \vee \mathcal{A}(q_2, m_3) \\
&= \big(\mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge \langle p \rangle \mathit{translated} \wedge \langle o \rangle \langle \bar{o} \rangle \langle p \rangle \mathit{translated}\big) \\
&\quad \wedge \mathit{lfp}\big(X, \langle \bar{s} \rangle \langle \bar{o} \rangle \langle p \rangle \mathit{translated} \wedge \langle p \rangle \mathit{type} \wedge \langle o \rangle \mathit{Poem}\big) \\
&\quad \vee \mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge \langle p \rangle \mathit{wrote} \wedge \langle o \rangle \langle \bar{o} \rangle \langle p \rangle \mathit{translated}\big)\big) \vee \\
&\quad \big(\mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge \langle p \rangle \mathit{translated} \wedge \langle o \rangle \langle s \rangle \langle p \rangle \mathit{type}\big) \\
&\quad \wedge \mathit{lfp}\big(X, \langle \bar{s} \rangle \langle s \rangle \langle p \rangle \mathit{type} \wedge \langle p \rangle \mathit{type} \wedge \langle o \rangle \mathit{Poem}\big) \\
&\quad \vee \mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge \langle p \rangle \mathit{wrote} \wedge \langle o \rangle \langle s \rangle \langle p \rangle \mathit{type}\big)\big) \vee \\
&\quad \big(\mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge \langle p \rangle \mathit{translated} \wedge \langle o \rangle \langle \bar{o} \rangle \langle p \rangle \mathit{wrote}\big) \\
&\quad \wedge \mathit{lfp}\big(X, \langle \bar{s} \rangle \langle \bar{o} \rangle \langle p \rangle \mathit{wrote} \wedge \langle p \rangle \mathit{type} \wedge \langle o \rangle \mathit{Poem}\big) \\
&\quad \vee \mathit{lfp}\big(X, \langle \bar{s} \rangle x \wedge \langle p \rangle \mathit{wrote} \wedge \langle o \rangle \langle \bar{o} \rangle \langle p \rangle \mathit{wrote}\big)\big)
\end{aligned}$$

So far we offered various functions to produce formulas corresponding to the encodings of queries and schema axioms. Hence, the problem of containment under a schema can be reduced to formula unsatisfiability in the μ-calculus as:

$$q \sqsubseteq_{\mathcal{C}} q' \Leftrightarrow \eta(\mathcal{C}) \wedge \mathcal{A}(q) \wedge \neg\mathcal{A}(q', m) \wedge \varphi_r \text{ is unsatisfiable.}$$

For the sake of legibility in writing, we use $\Phi(\mathcal{C}, q, q')$ to denote $\eta(\mathcal{C}) \wedge \mathcal{A}(q) \wedge \neg\mathcal{A}(q', m) \wedge \varphi_r$.

5.6.2 Reducing Containment to Unsatisfiability

We prove the correctness of reducing query containment to unsatisfiability test.

Lemma 3. *Given a set of $\mathcal{ALCH}$ schema axioms $\mathcal{C}$, $\mathcal{C}$ has a model iff $\eta(\mathcal{C})$ is satisfiable.*

Proof. ($\Rightarrow$) assume that there exists a model $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ of $\mathcal{C}$ such that $\mathcal{I} \models \mathcal{C}$. We build a restricted transition system $K = (S, R, L)$ from $\mathcal{I}$ using the following:

- for each element of the domain $e \in \Delta^{\mathcal{I}}$, we create a node $n^e \in S'$,
- for each atomic concept A, if $a \in A^{\mathcal{I}}$, then $(n^a, t) \in R(s), (t, n^{type}) \in R(p)$, $(t, n^A) \in R(o)$, $L(type), = n^{type}, L(A) = n^A$ and $L(a) = n^a$ where $t \in S''$,
- for each atomic role R, if $(x, y) \in R^{\mathcal{I}}$, then $(n^x, t) \in R(s)$, $(t, n^R) \in R(p)$, and $(t, n^y) \in R(o)$ such that $n^x, n^y, n^R \in S'$, $t \in S''$, and $L(x) = n^x$, $L(R) = n^R$, $L(y) = n^y$,
- $S = S' \cup S''$

To show that $\eta(\mathcal{C})$ is satisfiable in K. We proceed inductively on the construction of the formula. Since the axioms, $\{c_1, \ldots, c_n\}$, are made of role or concept inclusions or transitivity, we consider the following cases:

- when $\eta(c_i) = gfp\big(X, \omega(C_1) \Rightarrow \omega(C_2)\big)$. Since $C_1^{\mathcal{I}} \subseteq C_2^{\mathcal{I}}$, we get that $[\![\omega(C_1)]\!]^K \subseteq [\![\omega(C_2)]\!]^K$. And hence, $\omega(C_1) \Rightarrow \omega(C_2)$ is satisfiable in K. Besides, the general recursion ν guarantees that the constraint is satisfied in each state of the transition system. Therefore, $\eta(c_i)$ is satisfiable.
- when $\eta(c_i) = gfp\big(X, \omega(r_1) \Rightarrow \omega(r_2)\big)$. From $r_1^{\mathcal{I}} \subseteq r_2^{\mathcal{I}}$ we have that $\exists n^{r_1} \in L(r_1)$ implies $\exists n^{r_2} \in L(r_2)$ in K. Thus, $\exists s \in [\![\omega(r_1) \Rightarrow \omega(r_2)]\!]^K$. As K is a construction of $\mathcal{I}$, $\eta(c_i)$ is satisfiable in K.

Since K is a model of each $\eta(c_i)$, then $\eta(\mathcal{C})$ is satisfiable.
($\Leftarrow$) consider a transition system model K for $\eta(\mathcal{C})$. From K, we construct an interpretation $\mathcal{I} = (\Delta^{\mathcal{I}}, .^{\mathcal{I}})$ and show that it is a model of $\mathcal{C}$.

- $\Delta^{\mathcal{I}} = S$, $A^{\mathcal{I}} = [\![A]\!]^K$ for each atomic concept A,
- $\top^{\mathcal{I}} = [\![\top]\!]^K$, for a top concept,
- $r^{\mathcal{I}} = \{(s, s') \mid \forall t \in [\![r]\!]^K \wedge t' \in S \wedge (s, t') \in R(s) \wedge (t', t) \in R(p) \wedge (t', s') \in R(o)\}$ for each atomic role r,

Consequently, formulas such as $gfp\big(X, \omega(r_1) \Rightarrow \omega(r_2)\big)$ and $gfp\big(X, \omega(C_1) \Rightarrow \omega(C_2)\big)$ are true in $\mathcal{I}$. The first formula expresses that there is no node in the transition system where $\omega(r_1)$ holds and $\omega(r_2)$ does not hold. This is equivalent to $\omega(r_1) \Rightarrow \omega(r_2)$ and $[\![r_1]\!]^K \subseteq [\![r_2]\!]^K$ since r_1 and r_2 are basic roles. Thus, we obtain $r_1^{\mathcal{I}} \subseteq r_2^{\mathcal{I}}$ and $\mathcal{I} \models r_1 \sqsubseteq r_2$.

On the other hand, for the latter formula from above, one can exploit its construction. Note however that, similar justifications as above can be worked out to arrive at $\mathcal{I} \models C_1 \sqsubseteq C_2$ if C_1 and C_2 are basic concepts. Nonetheless, if they are complex concepts, we proceed as below. Consider the case when $C_1 = A \sqcap B$ and $C_2 = \exists R.C$, $[\![\omega(C_1) \Rightarrow \omega(C_2)]\!]^K$

$$
\begin{aligned}
&\Leftrightarrow [\![\omega(A \sqcap B)]\!]^K \subseteq [\![\omega(\exists R.C)]\!]^K \\
&\Leftrightarrow [\![A \wedge B]\!]^K \subseteq [\![\langle s\rangle\big(\langle p\rangle R \wedge \langle o\rangle(\langle s\rangle\langle o\rangle C)\big)]\!]^K \\
&\Leftrightarrow [\![A]\!]^K \wedge [\![B]\!]^K \subseteq \{s \mid \exists s'.s \in [\![\langle s\rangle\langle p\rangle R]\!]^K \wedge s' \in [\![\langle s\rangle\langle o\rangle C]\!]^K\} \\
&\Leftrightarrow A^{\mathcal{I}} \cap B^{\mathcal{I}} \subseteq \{s \mid \exists s'.(s, s') \in R^{\mathcal{I}} \wedge s' \in C^{\mathcal{I}}\} \\
&\Leftrightarrow (A \sqcap B)^{\mathcal{I}} \subseteq (\exists R.C)^{\mathcal{I}} \\
&\Leftrightarrow \mathcal{I} \models C_1 \sqsubseteq C_2
\end{aligned}
$$

Accordingly, from $\mathcal{I} \models c_1 \wedge \cdots \wedge \mathcal{I} \models c_n$, it follows that $\mathcal{I} \models \mathcal{C}$. □

Theorem 2 ([Hayes 2004]). *Given a query $q\{\overrightarrow{w}\}$, there exists an RDF graph G such that $[\![q\{\overrightarrow{w}\}]\!]_G \neq \emptyset$.*

Proof. (*Sketch*) From any query it is possible to build an homomorphic graph by collecting all triples connected by AND and only those at the left of UNION (replacing variables by blanks). This graph is consistent as all RDF graphs. It is thus a graph satisfying the query. □

Lemma 4. *Let q be a SPARQL$_{cdfc}$ query, for every restricted transition system K whose associated RDF graph is G, we have that $[\![q]\!]_G \neq \emptyset$ iff $[\![\mathcal{A}(q,m) \wedge \varphi_r]\!]^K \neq \emptyset$.*

Proof. ($\Rightarrow$) Assume that $[\![q]\!]_G \neq \emptyset$ and consider that G is a canonical instance of q (cf. Theorem 2). Using G, we construct a restricted transition system $\sigma(G) = (S, R, L)$ in the same way as it is done in Definition 6. To prove that $\sigma(G)$ is a model of $\mathcal{A}(q,m)$, we consider two cases:

(i) when q is cyclic-free, and

(ii) when q contains a cyclic component among its non-distinguished variables which is not distinguished variable-free or constant-free

Firstly, (i) if q is cycle-free, then encoding the non-distinguished variables with $\top$ suffices to justify that $\sigma(G)$ is a model of its encoding.

Secondly, (ii) let us consider when q is cyclic, in this case, its encoding is $\bigvee_{i=1}^{|m|} \mathcal{A}(q, m_i)$. This disjunctive formula can encode multiply occurring non-distinguished variables using the distinguished variables and constants of q. Henceforth, creating a formula that is satisfiable in cyclic models.

It can be verified that $\sigma(G)$ is a model for one of the disjuncts $\mathcal{A}(q, m_i)$, this is because nominals encoding the constants and distinguished variables are true in $\sigma(G)$ as they exist already in G. In addition, the formulae encoding the non-distinguished variables are satisfied in $\sigma(G)$, since these variables are encoded in terms of the distinguished variables and constants of the query which are already shown to be true in $\sigma(G)$. Therefore, $\mathcal{A}(q,m)$ is satisfiable in $\sigma(G)$. To elaborate, if $(x, y, z) \in q$ and l is either x or y or z,

- for l either a distinguished variable or constant, the translation of l is satisfiable in $\sigma(G)$ since $[\![l]\!]^{\sigma(G)} \in L(l)$,
- for l a uniquely appearing non-distinguished variable, the translation of l is true in $\sigma(G)$ since its encoding $\top$ is true everywhere in the transition system,
- the translation l of a multiply occurring non-distinguished variable is true in $\sigma(G)$ since $\exists t \in S'.\ t \in L(c) \wedge t \in [\![c]\!]^{\sigma(G)}$, where c is a constant or distinguished variable in the triple (x, y, z).

Thus, since $\sigma(G)$ is a restricted transition system, we obtain that $[\![\mathcal{A}(q,m)\varphi_r]\!]_G \neq \emptyset$.
($\Leftarrow$) Assume that $[\![\mathcal{A}(q,m) \wedge \varphi_r]\!]^K \neq \emptyset$. From this we can derive that, $K = (S, R, L)$ is restricted transition system that satisfies $\mathcal{A}(q,m) \wedge \varphi_r$. We build an RDF graph G from K as follows:

- $\forall s_1, s_2, s_3 \in S' \wedge t \in S''.(s_1, t) \in R(s) \wedge (t, s_2) \in R(p) \wedge (t, s_3) \in R(o)$ and for each triple $t_i = (x_i, y_i, z_i) \in q$ if $s_1 \in L(x_i) \wedge s_2 \in L(y_i) \wedge s_3 \in L(z_i) \wedge z_i \in [\![\top]\!]^K$, then $(x_i, y_i, z_i) \in G$. This case holds if x_i, y_i and z_i are either distinguished variables or constants. Note here that if x_i or y_i or z_i appear in another triple

$t_j = (x_j, y_j, z_j) \in q$, then the equivalent item in t_j is replaced with the value of the corresponding entry in t_i.

- $\forall s_1, s_2, s_3 \in S' \wedge t \in S''.(s_1, t) \in R(s) \wedge (t, s_2) \in R(p) \wedge (t, s_3) \in R(o)$ and for each triple $t_i = (x_i, y_i, z_i) \in q$ if $s_1 \in L(x_i) \wedge s_2 \in L(y_i)$, then $(x_i, y_i, c_i) \in G$ where c_i is a fresh constant. This case holds if z_i is a non-distinguished variable. Similarly, the case when x_i or y_i or both are variables can be worked out.

- $\forall s_1, s_2, s_3 \in S' \wedge t \in S''.(s_1, t) \in R(s) \wedge (t, s_2) \in R(p) \wedge (t, s_3) \in R(o)$ and for each triple $t_i = (x_i, y_i, z_i) \in q$ and x_i is a non-distinguished variable that appears in a cycle and if $s_1 \in [\![\top]\!]^K \wedge s_2 \in L(y_i) \wedge s_3 \in L(z_i)$, then $(c_i, y_i, z_i) \in G$. Where c_i is a fresh constant and all such occurrences of the variable x_i in other triples are replaced by c_i's.

Since G is a technical construction obtained from a restricted transition system that associated with q, then it holds that $[\![q]\!]_G \neq \emptyset$. □

In the following, for the sake of legibility, we denote $\eta(\mathcal{C}) \wedge \mathcal{A}(q_1) \wedge \neg\mathcal{A}(q_2, m) \wedge \varphi_r$ by $\Phi(\mathcal{C}, q_1, q_2)$.

Theorem 3 (Soundness)**.** *Given a SPARQL query $q_1\{\overrightarrow{w}\}$, a SPARQL$_{cdfc}$ query $q_2\{\overrightarrow{w}\}$, and a set of $\mathcal{ALCH}$ axioms $\mathcal{C}$, if $\Phi(\mathcal{C}, q_1, q_2)$ is unsatisfiable, then $q_1\{\overrightarrow{w}\} \sqsubseteq_{\mathcal{C}} q_2\{\overrightarrow{w}\}$.*

Proof. We show the contrapositive. If $q_1 \not\sqsubseteq_{\mathcal{C}} q_2$, then $\Phi(\mathcal{C}, q_1, q_2)$ is satisfiable. One can verify that every model G of $\mathcal{C}$ in which there is at least one tuple satisfying q_1 but not q_2 can be turned into a transition system model for $\Phi(\mathcal{C}, q_1, q_2)$. To do so, consider a graph G that satisfies schema axioms $\mathcal{C}$. Assume also that there is a tuple $\overrightarrow{a} \in [\![q_1]\!]_G$ and $\overrightarrow{a} \notin [\![q_2]\!]_G$. Let us construct a transition system K from G. From Lemma 3, we obtain that $[\![\eta(\mathcal{C})]\!]^K \neq \emptyset$. Further, since K is a restricted transition system (cf. Definition 6), $[\![\varphi_r]\!]^K \neq \emptyset$. At this point, it remains to verify that $[\![\mathcal{A}(q_1)]\!]^K \neq \emptyset$ and $[\![\mathcal{A}(q_2, m)]\!]^K = \emptyset$.

Let us construct the formulas $\mathcal{A}(q_1)$ and $\mathcal{A}(q_2, m)$ by first skolemizing the distinguished variables using the answer tuple $\overrightarrow{a}$. Consequently, from Lemma 4 one obtains, $[\![\mathcal{A}(q_1)]\!]^K \neq \emptyset$. However, $[\![\mathcal{A}(q_2, m)]\!]^K = \emptyset$, this is because the nominals in the formula corresponding to the constants and non-distinguished variables are not satisfied in K. This implies that $[\![\neg\mathcal{A}(q_2, m)]\!]^K \neq \emptyset$. This is justified by the fact that if a formula φ is satisfiable in a restricted transition system, then $[\![\varphi]\!]^K = S$ thus $[\![\neg\varphi]\!]^K = \emptyset$. So far we have: $[\![\eta(\mathcal{C})]\!]^K \neq \emptyset$ and $[\![\varphi_r]\!]^K \neq \emptyset$ and $[\![\mathcal{A}(q_1)]\!]^K \neq \emptyset$ and $[\![\neg\mathcal{A}(q_2, m)]\!]^K \neq \emptyset$. Without loss of generality, $[\![\Phi(\mathcal{C}, q_1, q_2)]\!]^K \neq \emptyset$. Therefore, $\Phi(\mathcal{C}, q_1, q_2)$ is satisfiable. □

Theorem 4 (Completeness)**.** *Given a SPARQL query $q_1\{\overrightarrow{w}\}$, a SPARQL$_{cdfc}$ query $q_2\{\overrightarrow{w}\}$, and a set of $\mathcal{ALCH}$ axioms $\mathcal{C}$, if $\Phi(\mathcal{C}, q_1, q_2)$ is satisfiable, then $q_1\{\overrightarrow{w}\} \not\sqsubseteq_{\mathcal{C}} q_2\{\overrightarrow{w}\}$.*

Proof. $\Phi(\mathcal{C}, q, q')$ is satisfiable $\Rightarrow \exists K.[\![\Phi(\mathcal{C}, q, q')]\!]^K \neq \emptyset$. Consequently, K is a restricted transition system due to $[\![\varphi_r]\!]^K \neq \emptyset$ (cf. Proposition 1). Using $K = (S' \cup S'', R, L)$ we construct a model $\mathcal{I} = (\Delta^{\mathcal{I}}, \cdot^{\mathcal{I}})$ of $\mathcal{C}$ such that $q \not\sqsubseteq q'$ holds:

- $\Delta^{\mathcal{I}} = S'$, $A^{\mathcal{I}} = [\![A]\!]^K$ for each atomic concept A,
- $\top^{\mathcal{I}} = [\![\top]\!]^K$, for a top concept,
- $r^{\mathcal{I}} = \{(s, s') \mid \forall t \in [\![r]\!]^K \wedge t' \in S'' \wedge (s, t') \in R(s) \wedge (t', t) \in R(p) \wedge (t', s') \in R(o)\}$ for each atomic role r,
- for each constant c in q and q', $c^{\mathcal{I}} = [\![c]\!]^K$,
- for each distinguished and non-distinguished variable v in q, $v^{\mathcal{I}} = [\![v]\!]^K$, and
- for each distinguished variable v in q', $v^{\mathcal{I}} = [\![v]\!]^K$.

One can utilize Lemma 3 to verify that indeed $\mathcal{I}$ is a model of $\mathcal{C}$. Thus, it remains to show that $[\![q]\!]_{\mathcal{I}} \not\subseteq [\![q']\!]_{\mathcal{I}}$. From our assumption, one anticipates the following:

$$\begin{aligned}[\![\mathcal{A}(q) \wedge \neg\mathcal{A}(q')]\!]^K \neq \emptyset &\Rightarrow [\![\mathcal{A}(q)]\!]^K \neq \emptyset \text{ and } [\![\neg\mathcal{A}(q', m)]\!]^K \neq \emptyset \\ &\Rightarrow [\![\mathcal{A}(q)]\!]^K \neq \emptyset \text{ and } [\![\mathcal{A}(q', m)]\!]^K = \emptyset\end{aligned}$$

Note here that, if a formula φ is satisfiable in a restricted transition system K_r, then $[\![\varphi]\!]^{K_r} = S$. We use a function f to construct an RDF graph G from the interpretation $\mathcal{I}$. f uses assertions in $\mathcal{I}$ to form triples:

$$\begin{aligned} f(a \in A^{\mathcal{I}}) &= (a, \texttt{type}, A) \in G \\ f((a, b) \in r^{\mathcal{I}}) &= (a, r, b) \in G \\ f((a, b) \in (r^-)^{\mathcal{I}}) &= (b, r, a) \in G \\ f((x, y, z)) &= (x, y, z) \in G,\ \forall (x, y, z) \in q \end{aligned}$$

As a consequence, $[\![q]\!]_G \neq \emptyset$ and $[\![q']\!]_G = \emptyset$ because G contains all those triples that satisfy q and not q'. Therefore, we get $[\![q]\!]_G \not\subseteq [\![q']\!]_G$. Fundamentally, there are two issues to be addressed (i) when q' is not cyclic and (ii) when q' contains a cycle. (i) if there are no cycles in q', then replacing non-distinguished variables with $\top$ suffices (cf. the proof of Lemma 4). On the other hand, (ii) can be dealt with nominals, i.e., since cycles can be expressed by a formula in a μ-calculus extended with nominals and inverse, cyclic queries can be encoded by such a formula. Hence, the constraints expressed by $\neg\mathcal{A}(q', m)$ are satisfied in a transition system containing cycles □

5.6.3 Complexity

In the following, we establish the complexity of the containment problem under schema axioms. The schema axioms can be formed using the fragments of $\mathcal{ALCH}$. The expressiveness of the schema language is limited as such due to the expressive power of the logic used for the encoding: μ-calculus with nominals and converse becomes undecidable when extended with graded modalities [Bonatti *et al.* 2006].

Proposition 2 (Query satisfiability)**.** *Given an $\mathcal{ALCH}$ schema $\mathcal{C}$ and a query q, the complexity of satisfiability of q with respect to $\mathcal{C}$ is $2^{\mathcal{O}(|\mathcal{C}|+|q|)}$.*

Proposition 3. *SPARQL query containment under the fragments of $\mathcal{ALCH}$ schema axioms can be determined in a time of $2^{\mathcal{O}(n^2 log\ n)}$ where $n = \mathcal{O}(|\eta(\mathcal{C})|+|\mathcal{A}(q_1)|+|\mathcal{A}(q_2)|)$ is the size of the formula, and $\eta(\mathcal{C})$, $\mathcal{A}(q_1)$ and $\mathcal{A}(q_2)$ denote the encodings of schema axioms $\mathcal{C}$, and queries q_1 and q_2.*

Note that due to duplication in the encoding of q_2, the size of $|\mathcal{A}(q_2)|$ is exponential in terms of the non-distinguished variables that appear in cycles in the query. Hence, we obtain a 2ExpTime upper bound for containment. As pointed out in [Calvanese *et al.* 2008], the problem is solvable in ExpTime if there is no cycle on the right hand side query.

5.7 Experimental Investigations

To the best of our knowledge, no experimental evaluation of SPARQL query containment has been performed so far. Due to the relatively high complexity of this problem, it is important to know if the proposed solutions can be applied in practice. Moreover, well established benchmarks for query containment would help fostering the development and improvement of solvers.

The overall purpose of the remaining sections is to design a benchmark suite for testing SPARQL query containment and to evaluate the performance of current solvers. For that purpose, we analyze state-of-the-art solver capabilities as well as actual queries being asked on the web. This allows us to identify two classes of solvers addressing different types of problems: some solvers [Letelier *et al.* 2012] are restricted to conjunctive queries without projections, some other techniques based on query translations into the μ-calculus [Chekol *et al.* 2012b] are so far restricted to acyclic queries without OPTIONAL. The analysis of DBPedia query logs show that more than 90% of correct queries are acyclic, queries being distributed in large sets of DAG and tree queries. In the sequel, our experiments focus exclusively on acyclic queries.

We designed three test sets involving: conjunctive queries without projection, union of conjunctive queries with projection and union of conjunctive queries to be analyzed with respect to an RDF Schema. We also provide an evaluation protocol allowing to run the exact same queries through a specific interface and wrapped the three considered tools. We run these tests and observe the capabilities of different solvers in the different categories of tests.

Hence the contribution of this experimental part is threefold: (*i*) the analysis of the demographics of SPARQL queries actually produced on the web, (*ii*) the design of benchmark suites for evaluating query containment solver capabilities, (*iii*) the evaluation of three different solvers through these benchmarks. The proposed benchmark suites as well as our testing and analysis tools are available on the web and we expect that this will motivate others to produce even better query containment solvers.

Outline of the sequel: we present the state of the art in SPARQL query containment solvers and analyze the query landscape (§5.8). From this, we design our

benchmark suite (§5.9). Finally, we report evaluation experiments for the three available systems and discuss the results (§5.10).

5.8 Query Containment Solvers

We briefly present three state-of-the-art query containment solvers used in the experiments. Our goal is to characterize their capabilities in order to design appropriate benchmarks. In order to do so, we also analyze actual queries used on the semantic web.

Out of the three systems, SPARQL-Algebra is self contained whereas the other two are μ-calculus satisfiability solvers that need an intermediate query translation into formulas to determine containment.

5.8.1 SPARQL-Algebra

SPARQL-Algebra is an implementation of SPARQL query subsumption and equivalence based on the theoretical results in [Letelier *et al.* 2012]. This implementation supports AND and OPTIONAL queries with no projection. An on-line version of the solver is available at http://db.ing.puc.cl/sparql-algebra/.

5.8.2 AFMU

AFMU (Alternation Free two-way μ-calculus) [Tanabe *et al.* 2005] is a satisfiability solver for the alternation-free fragment of the μ-calculus [Kozen 1983]. It is a prototype implementation which determines the satisfiability of a μ-calculus formula by producing a yes-or-no answer.

To turn it into a query containment solver, it is necessary to turn the problem into a μ-calculus satisfiability problem.

We developed techniques for encoding queries into the μ-calculus ($\mathcal{A}$) in order to determine the containment of SPARQL queries [Chekol *et al.* 2012a, Chekol *et al.* 2012b]. Of the three approaches introduced in [Chekol *et al.* 2012a] to deal with RDFS, we have chosen the encoding of the schema (η) into the μ-calculus. We use these encoding schemes for deciding $q \sqsubseteq_{\mathcal{S}} q'$: it is necessary to check if the encoding of its negation, $\eta(\mathcal{S}) \wedge \mathcal{A}(q) \wedge \neg\mathcal{A}(q')$, is satisfiable. If this is the case, then containment does not hold, otherwise, it is established.

5.8.3 TreeSolver

The XML tree logic solver *TreeSolver*[4] performs static analysis of XPath queries which comprise containment, equivalence and satisfiability. To perform these tasks, the solver translates XPath queries into μ-calculus formulas and then tests the unsatisfiability of the formula. Unlike AFMU, the unsatisfiability test is performed in time of $2^{\mathcal{O}(n)}$ whereas it is $2^{\mathcal{O}(n \log n)}$ for AFMU, such that n is the size of the formula.

[4] http://wam.inrialpes.fr/websolver/

System	projection	UCQ	optional	blanks	cycles	RDFS
SPARQL-Algebra			√		√	
AFMU	√	√		√		√
TreeSolver	√	√		√		√

Table 5.5: Comparison of features supported by current systems.

5.8.4 Features supported by solvers

A summary of the features supported by actual implementations of query containment solvers is presented in Table 5.5.

Part of the query structures can be transformed into concept expressions in description logics and submitted to satisfiability (or subsumption) tests as well. So, in principle, query containment solvers based on description logic reasoners could be designed. However, we do not know any such solver.

5.8.5 State of the query landscape

To the best of our knowledge, no experimental work has been conducted to verify how many of real world queries are acyclic or cyclic. To answer this question, we analyzed DBpedia query logs[5]. We report on two log sets because there is a lot of variation between them. 2 905 035 queries from the logs were syntactically correct (90%). The results are given in Table 5.6. We tested the cyclicness of queries and found out that more than 90% of these queries are acyclic (94% of the small sample and 99% on the total). This justifies designing and evaluating acyclic queries. Projection is used in 11% of the large log and 22% of the smaller one, but such figures more than double if only "SELECT *" queries are counted as projection-free queries.

OPTIONAL are used in around 30% of queries, whereas UNION is used in 18% of those in the full log and 43% in the large one. Union of conjunctive queries with optional are 15 to 30% of the logs. This make them operators to be supported in query containment.

5.9 Benchmark Design

We first present the design of containment benchmark suites. Each test suite is made of elementary test cases asking for the containment of one query into another. We then introduce the principles and software used for evaluating containment solvers. The benchmark and software is available on-line at http://sparql-qc-bench.inrialpes.fr/.

[5]DBpedia 3.5.1 logs (ftp://download.openlinksw.com/support/dbpedia/) contain 3 210 368 queries between 30/04/2010 and 20/07/2010 and 378 530 queries of 13/07/2010 only.

	projection			no projection		
operator	tree	dag	cycle	tree	dag	cycle
none	175 220	562	1	1 534 150	1 761	1 748
union	9	26 625	547	24	29 629	1 166
opt	2 052	685	0	311 608	722	1
filter	7 912	711	6	264 821	340	1
un-opt	0	306	0	0	12 659	1
opt-filt	7 991	779	0	4 933	52 401	0
filt-un	2	183	0	23 802	12 286	0
un-opt-filt	0	102 765	0	0	302 657	23 969

Table 5.6: Query characteristics of the full DBPedia logs.

5.9.1 Structure of the benchmark

There are three qualitative dimensions along which tests can be designed: the type of graph pattern connectors (AND, UNION, MINUS, Projection, OPTIONAL, FILTER etc.), the type of ontology: (no schema, RDFS, SHI, OWL, etc.) and the query structure (tree, DAG, cyclic). In addition to these dimensions, quantitative measures are:

- the number of triple patterns,
- the number of variables,
- the number of triple patterns involving more than one variable (Tjoins),
- the size of the ontology.

We designed test suites of homogeneous qualitative dimensions selected with respect to the capacity of the current state-of-the-art solvers. The benchmark contains three test suites:

- **C**onjunctive **Q**ueries with **No Proj**ection (CQNoProj)
- **U**nion of **C**onjunctive **Q**ueries with **Proj**ection (UCQProj)
- **U**nion of **C**onjunctive **Q**ueries under **RDFS** reasoning (UCQrdfs)

The test suites are designed to model increasing expected difficulty by using more constructors. We did not provide tests of cyclic queries since only one solver is currently able to deal with them. However, this would be a natural addition to these tests.

Each test suite contains tests of different quantitative measures. Most of them are used for conformance testing, i.e., testing that solvers return the correct answer, but we also identify some stress tests trying to evaluate solvers at or beyond their limits. We discuss these test suites below.

5.9.1.1 CQNoProj

This test suite is designed for containment of basic graph patterns. It contains conjunctive queries with no projection. We have identified 20 different test cases (nop1–nop20),

Test case	Problem	AND	Vars	Tjoin
nop1	Q1a $\sqsubseteq$ Q1b	1	1	0
nop2	Q1b $\sqsubseteq$ Q1a	0		0
nop3	Q2a $\sqsubseteq$ Q2b	5	3	3
nop4	Q2b $\sqsubseteq$ Q2a	5		3
nop5	Q3a $\sqsubseteq$ Q3b	2	2	2
nop6	Q3b $\sqsubseteq$ Q3a	1		1
nop7	Q4c $\sqsubseteq$ Q4b	5	3	2
nop8	Q4b $\sqsubseteq$ Q4c	3		2
nop9	Q6a $\sqsubseteq$ Q6b	2	3	1
nop10	Q6b $\sqsubseteq$ Q6a	2		1

Test case	Problem	AND	Vars	Tjoin
nop11	Q6a $\sqsubseteq$ Q6c	2	3	1
nop12	Q6c $\sqsubseteq$ Q6a	0		1
nop13	Q6b $\sqsubseteq$ Q6c	2	3	1
nop14	Q6c $\sqsubseteq$ Q6b	0		1
nop15	Q7a $\sqsubseteq$ Q7b	9	10	9
nop16	Q7b $\sqsubseteq$ Q7a	10		9
nop17	Q8a $\sqsubseteq$ Q8b	3	4	3
nop18	Q8b $\sqsubseteq$ Q8a	2		3
nop19	Q9a $\sqsubseteq$ Q9b	4	3	2
nop20	Q9b $\sqsubseteq$ Q9a	4		2

Table 5.7: The CQNoProj testsuite. In the AND column, figures correspond to the number of AND in the left-hand side query of the test. Vars is the number of variables in each queries and Tjoin the number of triples in which occurs at least two variables.

each one testing containment between two queries. All the cases in this setting are shown in Table 5.7, along with the number of connectives and variables in the queries. The more difficult test used for stress testing are nop3, nop4, nop15, and nop16. The two former ones have a larger number of conjunction (and of Tjoin), while the two latter ones have an even larger number of conjunctions and variables. We have selected Tjoins (triples having two variables) as a measure of difficulty because simple triple joins may be compiled efficiently as tuples.

The worse case complexity of CQ without projection containment is PTIME [Chekuri & Rajaraman 1997].

5.9.1.2 UCQProj

This test suite is made of 28 test cases, each comprising two acyclic union of conjunctive queries with projection. In fact, 14 tests contain projection only, 6 tests contains union only and 2 tests contains both (see Table 5.8). The test cases differ in the number of distinguished variables (*Dvars*) and connectives (conjunction or union). Particular stress tests are p3, p4 (without union nor projection), p15, p16, p23, and p24.

The worse case complexity of UCQ with projection containment is NP-complete with cycles [Chandra & Merlin 1977], it is unknown for acyclic queries.

5.9.1.3 UCQrdfs

In query containment under RDFS reasoning, there are 28 test cases (Table 5.9). In comparison to the test cases in UCQProj and CQNoProj setting, the size of the queries is small. Each test case is composed of two acyclic UCQs and a schema. There are 4 different small schemas, C1–C4 whose characteristics, with respect to the type and number of axioms, are presented in Table 5.9. These small schemas are used for testing

Test case	Problem	AND	UNION	Dvars	Vars	Tjoin
p1	Q11a $\sqsubseteq$ Q11b	1	0	1	1	0
p2	Q11b $\sqsubseteq$ Q11a	0	0		1	0
p3	Q12a $\sqsubseteq$ Q12b	5	0	3	3	3
p4	Q12b $\sqsubseteq$ Q12a	5	0		3	3
p5	Q13a $\sqsubseteq$ Q13b	2	0	2	2	2
p6	Q13b $\sqsubseteq$ Q13a	1	0		2	1
p7	Q14c $\sqsubseteq$ Q14b	3	0	1	3	2
p8	Q14b $\sqsubseteq$ Q14c	5	0		3	2*
p9	Q15a $\sqsubseteq$ Q15b	0	0	2	3	1
p10	Q15b $\sqsubseteq$ Q15a	0	0		2	1
p11	Q16a $\sqsubseteq$ Q16b	2	0	1	3	1
p12	Q16b $\sqsubseteq$ Q16a	2	0		3	1
p13	Q16a $\sqsubseteq$ Q16c	2	0	1	3	1
p14	Q16c $\sqsubseteq$ Q16a	0	0		3	1

Test case	Problem	AND	UNION	Dvars	Vars	Tjoin
p15	Q17a $\sqsubseteq$ Q17b	9	0	10	10	9
p16	Q17b $\sqsubseteq$ Q17a	10	0		11	10
p17	Q18a $\sqsubseteq$ Q18b	3	0	4	4	3
p18	Q18b $\sqsubseteq$ Q18a	2	0		4	3
p19	Q19a $\sqsubseteq$ Q19b	4	0	2	3	2
p20	Q19b $\sqsubseteq$ Q19a	4	0		3	2
p21	Q19c $\sqsubseteq$ Q19b	4	0	2	4	3
p22	Q19b $\sqsubseteq$ Q19c	4	0		3	2
p23	Q20a $\sqsubseteq$ Q20b	2	7	10	10	9
p24	Q20b $\sqsubseteq$ Q20a	8	1		10	9
p25	Q21a $\sqsubseteq$ Q21b	6	2	2	4-6	5
p26	Q21b $\sqsubseteq$ Q21a	8	0		6	5
p27	Q22a $\sqsubseteq$ Q22b	3	1	2	2	2
p28	Q22b $\sqsubseteq$ Q22a	3	1		2	2

Table 5.8: The UCQProj test suite.

the correctness of solvers and are not realistic schemas. The most difficult tests are supposed to be rdfs23, rdfs24, rdfs25, rdfs26, rdfs27 and rdfs28 with both projection and union.

The worse case complexity of UCQ under RDFS has an ExpTime upper bound [Chekol *et al.* 2012a].

5.9.2 Benchmarking software architecture

For testing containment solvers, we designed an experimental setup which comprises several software components. This setup is illustrated in Figure 5.6. It simply considers a containment checker as a software module taking as input two SPARQL queries (q and q'), eventually an RDF Schema ($\mathcal{S}$), and returning true or false depending if q' is entailed by q (under the constraints of $\mathcal{S}$).

This has been provided as a Java interface using Jena to express queries and RDF Schema. We have developed three wrappers implementing this interface for the three tested systems. Other systems may be wrapped in the same interface (dashed rectangles in Figure 5.6) and tested in the same conditions. This platform may also be used for providing non regression tests for containment solvers.

Tests proceeds by providing test cases to the interface, timing the execution of the containment test around this common interface call. So timing occurs at the frontier of the dashed box of Figure 5.6, i.e., after query and schema parsing. This advantages

Schema	Axiom types
C1	subclass (2)
C2	domain (1) and range (1)
C3	subclass (1), subproperty (2) and domain (1)
C4	subclass (1)

Test	Ontology	Problem	AND	UNION	Dvars	Vars	Tjoin
rdfs1	C1	Q39a $\sqsubseteq$ Q39c	0	0	1	1	0
rdfs2		Q39c $\sqsubseteq$ Q39a	0	1		1	0
rdfs3	C1	Q39a $\sqsubseteq$ Q39b	0	0	1	1	0
rdfs4		Q39b $\sqsubseteq$ Q39a	0	0		1	0
rdfs5	C1	Q39b $\sqsubseteq$ Q39c	0	0	1	1	0
rdfs6		Q39c $\sqsubseteq$ Q39b	0	1		1	0
rdfs7	C1	Q39d $\sqsubseteq$ Q39e	4	0	1	3	2
rdfs8		Q39e $\sqsubseteq$ Q39d	4	0		3	2
rdfs9	C2	Q40b $\sqsubseteq$ Q40d	0	0	1	2	1
rdfs10		Q40d $\sqsubseteq$ Q40b	0	0		1	0
rdfs11	C2	Q40e $\sqsubseteq$ Q40b	1	0	1	2	2
rdfs12		Q40b $\sqsubseteq$ Q40e	0	0		1	0
rdfs13	C3	Q41b $\sqsubseteq$ Q41c	0	0	1	2	1
rdfs14		Q41c $\sqsubseteq$ Q41b	0	0		2	1
rdfs15	C3	Q41b $\sqsubseteq$ Q41d	0	0	1	2	1
rdfs16		Q41d $\sqsubseteq$ Q41b	0	0		2	1
rdfs17	C3	Q41c $\sqsubseteq$ Q41d	0	0	1	2	1
rdfs18		Q41d $\sqsubseteq$ Q41c	0	0		2	1
rdfs19	C3	Q41b $\sqsubseteq$ Q41a	0	0	1	2	1
rdfs20		Q41a $\sqsubseteq$ Q41b	0	0		1	0
rdfs21	C3	Q41e $\sqsubseteq$ Q41a	0	1	1	2	2
rdfs22		Q41a $\sqsubseteq$ Q41e	0	0		1	0
rdfs23	C4	Q43a $\sqsubseteq$ Q43b	3	1	2	2	2
rdfs24		Q43b $\sqsubseteq$ Q43a	3	1		2	2
rdfs25	C4	Q43a $\sqsubseteq$ Q43c	3	1	2	2	2
rdfs26		Q43c $\sqsubseteq$ Q43a	3	1		2	2
rdfs27	C4	Q43b $\sqsubseteq$ Q43c	3	3	2	2	2
rdfs28		Q43c $\sqsubseteq$ Q43b	3	1		2	2

Table 5.9: The UCQrdfs test suite.

SPARQL-Algebra, because it works directly on the ARQ representation, whereas the two other solvers have first to translate the ARQ representation into a μ-calculus

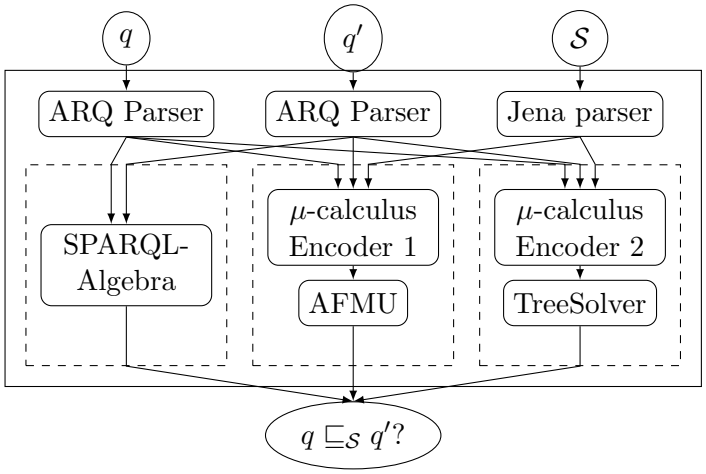


Figure 5.6: Experimental setup for testing query containment. The tester (plain rectangle) parses queries and schemas and passes them to a solver wrapper (dashed rectangle).

formula which is then parsed and transformed in each solver's internal representation.

5.10 Experimental Results

We evaluated the three identified query containment solvers with the three test suites. Rather than a definitive assessment of these solvers, our goal is to give first insights into the state-of-the-art and highlight deficiencies of engines based on the benchmark outcome. None of these systems is sharply optimized. However, their behavior is sufficient for highlighting test difficulty.

We run the experiments with an ordinary laptop computer running Mac OS X (specifically a 2.7 Ghz MacBook Pro with 16GB RAM).

The solvers were not genuinely reentrant. Hence, each test case has been run in a separate process after that the first case of each suite has been run as a warm up.

All solvers are Java programs. The Java virtual machines were run with maximum heap size of 2024MB and a timeout at 20s (20000ms). Raising memory size to 1GB and timeout to 40s does not change timeout results. The μ-calculus solvers take advantage of a native BDD library. Using the native implementation doubles the speed of these solvers, however, it also brings large initialization time (in spite of warm-up set up).

Reported figures are the average of 5 runs (we run the tests 7 times and ruled out each time the best and worse performance).

5.10.1 CQNoProj Results

On the conjunctive queries without projection, the SPARQL Algebra implementation is at least 10 times faster than the μ-calculus implementations (Figure 5.7). This comes as no surprise, since the latter are exponential time solvers whereas the former is a polynomial time solver.

AFMU times out on stress tests (nop3, nop4, nop15 and nop16). This happens whenever containment is determined between queries that contain more than 10 joins, such as in test cases nop15 and nop16. TreeSolver is able to deal with such cases albeit at the price of long response times. Overall, TreeSolver outperforms AFMU.

The fact that SPARQL-Algebra does not suffer from these sets, shows that the encoding of the μ-calculus solvers can be improved for such practical cases.

SPARQL-Algebra responded incorrectly, in test case nop7 (cf. Table 5.7), when blank nodes are used in the queries. It is not expected to deal with blank nodes. The other solvers are able to take them into account.

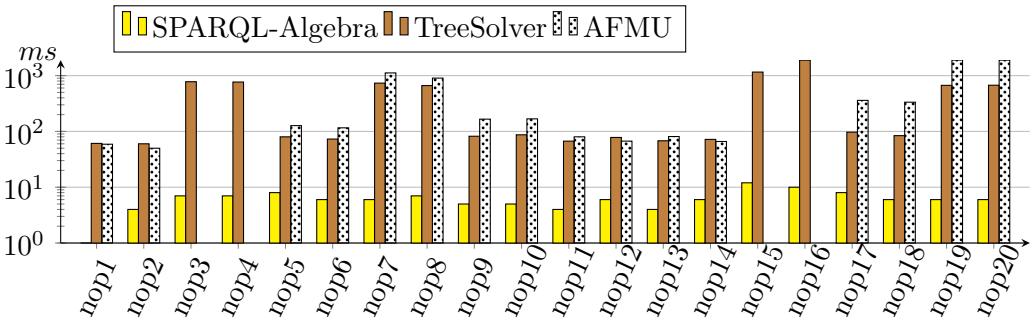


Figure 5.7: Results for the CQNoProj test suite (logarithmic scale).

5.10.2 UCQProj Results

On the UCQProj test suite (see Section 5.9.1.2), we compared the two systems able to deal with UNION: TreeSolver and AFMU. Figure 5.8 shows that the performances of AFMU and TreeSolvers are roughly comparable with the notable exception that TreeSolver answers for cases where AFMU fails. Specifically, TreeSolver times out only on test p24, while AFMU cannot deal with all stress tests: p3-p4, p15-p16, p23-p26. For this test suite, the necessary run time tends to be far longer as it often ends up in filling the available heap. For some of these tests (p15-16), performances could certainly be improved by adopting a better encoding of triples.

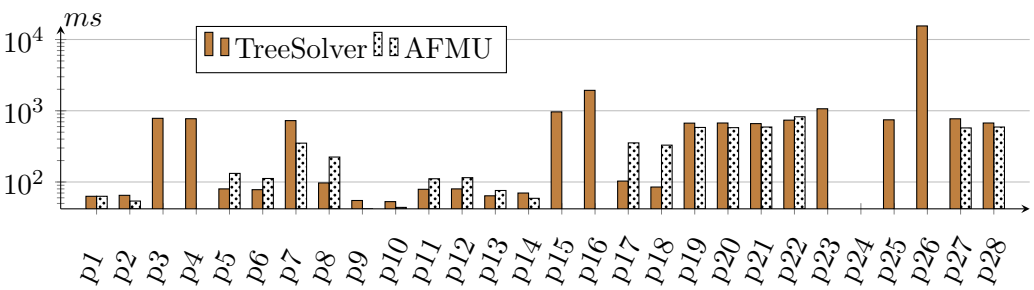


Figure 5.8: Results for the UCQProj test suite (logarithmic scale).

5.10.3 UCQrdfs Results

The results for containment of acyclic UCQs under RDFS (cf. Section 5.9.1.3) are given in Figure 5.9. They show that both solvers answer containment queries within a few hundreds of milliseconds. In this tests, TreeSolver clearly outperforms AFMU. Out of 28 tests, TreeSolver is much faster than AFMU in 23 tests, whereas AFMU is slightly faster than TreeSolver in 5 tests.

For both solvers, tests rdfs7 and rdfs8 as well as tests rdfs23-28 have been significantly more difficult than the other tests. This should not be due to the C4 ontology which is reduced to only one subsumption assertion, but rather to the presence of UNION and projection.

AFMU returns an incorrect answer for rdfs9 which seems to be a bug in the solver.

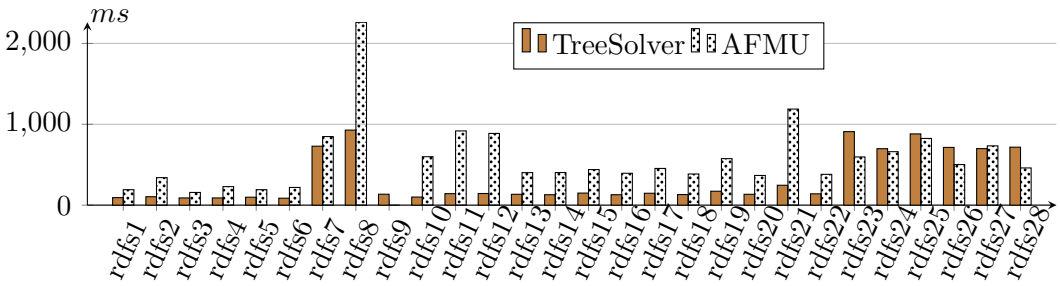


Figure 5.9: Results for the UCQrdfs test suite.

5.10.4 Discussion

In summary, all solvers under all experimental settings responded positively i.e., they all determined containment correctly under their stated application limits (we tested this independently). However, from these experiments, a lot remains to be done in order to alleviate the shortcomings of the current systems.

SPARQL-Algebra is faster on its domain of application. The advantages of this solver compared to the others are that it supports subsumption of OPTIONAL query patterns and also cyclic CQs. However, blank nodes are not supported.

AFMU is able to determine containment of acyclic UCQs under ontological axioms. For queries of reasonable size, the solver determined their containment correctly. The problem is that when queries have a larger size, e.g., more than 8 joins, the solver saturates memory. This is shown for test cases nop15 and nop16 (Figure 5.7) as well as for test cases p15 and p16 (Figure 5.8). However, the implementation of this solver is not optimal: the authors have documented improvements. Moreover, determining containment of general UCQs (beyond the acyclic ones) will require extending the solver.

TreeSolver has globally similar limitations as AFMU: no support for cyclic queries and difficulty with queries of large size (as we can expect from worse case com-

plexity), such as nop16. However, TreeSolver globally outperforms AFMU: TreeSolver is most often much faster, and, moreover capable of successfully dealing with more tests. This can probably be explained by the fact that the TreeSolver's algorithm [Genevès *et al.* 2007] is based on a least-fixpoint computation, whereas AFMU's algorithm is based on a greatest-fixpoint computation [Tanabe *et al.* 2005]. By nature, AFMU's algorithm starts from all possibilities and repeatedly removes inconsistencies until a fixpoint is reached. In sharp contrast, TreeSolver's algorithm basically starts from the emptyset and repeatedly tries to prove new relevant branches until it finds a fully proved model. A major consequence is that AFMU is required to compute a whole fixpoint each time before concluding about the existence (or inexistence) of a model. The situation is very different with TreeSolver, that can conclude as soon as it finds a (fully proved) satisfying model, without necessarily having to build a fixpoint completely before concluding about satisfiable formulas.

Determining the type of queries to compare (cyclic, disjunctive, with blank nodes, with projections, etc.) is easy. Hence, it is possible to build a system assembling these solvers and providing the best possible performances for each case.

5.11 Related Work

Query optimization has been the subject of an important research effort for many types of query languages, with the common goal of speeding up query processing. The works found in [Stocker *et al.* 2008, Groppe *et al.* 2009, Schmidt *et al.* 2010] considered the problem of SPARQL query optimization. Results presented in this Chapter can be used to prove the correctness of query rewriting techniques. In the following we briefly review works that previously established closely related results for related query languages.

An early formalization of RDF(S) graphs has been presented in [Gutierrez *et al.* 2004], in which the complexity of query evaluation and containment is also studied. The authors investigate a datalog-style, rule-based query language for RDF(S) graphs. In particular, they establish the NP-completeness of query containment over simple RDF graphs, this result is also published in the RDF semantics document [Hayes 2004]. The query language is rather simple compared to SPARQL and no constraints were assumed for the problem. [Serfiotis *et al.* 2005] provides algorithms for the containment and minimization of RDF(S) query patterns utilizing concept and property hierarchies for the query language RQL (RDF Query Language). The NP-completeness is established for query containment concerning conjunctive and union of conjunctive queries. In line with this, a recent work in [Polleres 2007] shows how to translate SPARQL queries into non-recursive Datalog with negation. The paper does focus on query evaluation (not on query containment).

The work in [Groppe *et al.* 2009], investigated static analysis of SPARQL queries that are embedded in a Java program. It checks the correctness of the syntaxes of RDF data, SPARQL queries, and SPARUL update queries. Beyond this, their system

called SWOBE (Semantic Web Objects Database Programming Language), detects if a query has a non-empty result set. Most recently, static analysis and optimization of OPTIONAL graph patterns is studied in [Letelier *et al.* 2012] where they proved the Π_2^p-completeness of query subsumption and NP-completeness of query equivalence.

Besides works that focus on querying RDF graphs, in the following, we explore the relations and containment problem between SPARQL and query languages from other domains.

SPARQL vs. Relational Algebra It has been shown that SPARQL is equally expressive as relational algebra (RA) [Angles & Gutierrez 2008]. It is easy to see that relational algebra with SPJUD (Selection, Projection, Join, Union and Difference) [Abiteboul *et al.* 1995] operators is equivalent to that of SPARQL with SELECT, AND, UNION, OPTIONAL and FILTER. The algebraic operators that are defined in SPARQL resemble the algebraic operators defined in relational algebra; in particular, AND is mapped to the algebraic join, FILTER is mapped to the algebraic selection operator, UNION is mapped to the union operator, OPTIONAL is mapped to the left outer join (which allows for the optional padding of information), and SELECT is mapped to the projection operator. As opposed to the operators in relational algebra, which are defined on top of relations with fixed schema, the algebraic SPARQL operators are defined over so called mapping sets, obtained when evaluating triple patterns. In contrast to the fixed schema in relational algebra, the "schema" of mappings in SPARQL algebra is loose in the sense that such mappings may bind an arbitrary set of variables. This means that in the general case we cannot give guarantees about which variables are bound or unbound in mappings that are obtained during query evaluation.

Studies on the translation of SPARQL into relational algebra and SQL [Cyganiak 2005, Chebotko *et al.* 2006] indicate a close connection between SPARQL and relational algebra in terms of expressiveness. In [Polleres 2007], a translation of SPARQL queries into a datalog fragment (non-recursive datalog with negation) that is known to be equally expressive as relational algebra was presented. This translation makes the close connection between SPARQL and rule-based languages explicit and shows that RA is at least as expressive as SPARQL. Tackling the opposite direction, it was recently shown in [Angles & Gutierrez 2008] that SPARQL is relationally complete, by providing a translation of the above-mentioned datalog fragment into SPARQL. As argued in [Angles & Gutierrez 2008], the results from [Polleres 2007] and [Angles & Gutierrez 2008] taken together imply that SPARQL has the same expressive power as relational algebra. From early results on query containment in relational algebra and first-order logic, one can infer that containment in relational algebra is undecidable. Therefore, containment of SPARQL queries is also undecidable. Hence, in this Chapter, we considered various fragments of SPARQL to study containment.

Query Entailment is the decision problem associated with query answering. For CQs, query answering and containment are equivalent problems. In fact, query con-

tainment can be reduced to query answering [Calvanese *et al.* 1998]. In this regard, conjunctive query containment under the description logic $\mathcal{DLR}$ is studied in [Calvanese *et al.* 2008]. CQ query answering in the presence of simple ontologies (fragments of DL-Lite) has been studied [Calvanese *et al.* 2007, Lutz *et al.* 2009]. For expressive ontology languages, query entailment (and hence containment) in DLs ranging from $\mathcal{ALCI}$ to $\mathcal{SHIQ}$ is shown to be 2ExpTime in [Lutz 2008, Glimm *et al.* 2008, Ortiz *et al.* 2008a, Eiter *et al.* 2009]. See Table 5.3 of Section 5.2.4 for a partial summary of the studies on query answering.

In this study we do not deal with the same query language as the one dealt with in [Glimm *et al.* 2008]. In fact, the supported SPARQL fragment is strictly larger than the one studied in [Glimm *et al.* 2008]. Specifically, UCQs in [Glimm *et al.* 2008] are made of $C(x), R(x, y)$ for an atom C, a role R, and variables x and y, whereas we do also support queries capable of querying concept and role names at the same time, such as $q(x) = (x, y, z)$. Further, the purpose of reducing the problem to the μ-calculus is exactly about extending query containment to even more features (such as SPARQL 1.1 paths with recursion, entailment regimes, and negation). For instance, it is known that recursive paths can be easily supported in μ-calculus (using fixpoints). Beyond this, the novelty of the study is the reduction of the SPARQL containment problem to μ-calculus satisfiability, and the advantages of using such a logic: expressivity, good computational properties, extensibility. The main focus of the contribution is not the complexity bound by itself but rather a new approach with a broader logic, paving the way for future extensions as it was never done before.

Finally, with an implicit goal of minimizing query evaluation costs, in [Pichler *et al.* 2010] comprehensive complexity results were obtained for the problem of redundancy elimination on RDF graphs in the presence of rules (RDFS or OWL), constraints (tuple-generating dependencies) and with respect to SPARQL queries.

Semistructured data In line with CQs in databases and description logic worlds, we have regular path queries— languages that are used to query arbitrary length paths in graph databases — in semi structured data. Like CQs, they have been used and studied widely. They are different from CQs in that, they allow recursion by using regular expression patterns. The problem of containment has been addressed for extensions of this language. In this regard, a prominent language used in semi-structured data is XPath. This language has been studied extensively over the last decade. These studies range from extending or reducing to static analysis. Static analysis of XPath queries has been studied in [Genevès *et al.* 2007], encompassing containment, equivalence, coverage, and satisfiability of XPath queries. In fact, this Chapter is motivated by [Genevès *et al.* 2007] in that the approach to study these problems uses a graph logic and provides a working implementation.

Other notable results come from the study of Regular Path Queries (RPQs). RPQs are extremely useful for expressing complex navigations in a graph. In particular, union and transitive closure are crucial when we do not have a complete knowledge of the structure of the knowledge base where this is the case for RDF graphs. Containment

of (two-way) regular path queries (2RPQs) have been studied extensively [Calvanese *et al.* 2000, Calvanese *et al.* 2003, Barceló *et al.* 2010]. These languages are used to query graph databases and containment has been shown to be PSPACE-complete, this complexity bound jumps to EXPTIME-hard under the presence of functionality constraints. On the other hand, the containment of conjunctive 2RPQs is EXPSPACE-complete, this bound jumps to 2EXPTIME when considered under expressive description logic (DL) constraints [Calvanese *et al.* 2011]. However, it is exponential if the query on the right hand side has a tree structure (cf. for example, [Calvanese *et al.* 2008]). Further, paths are being included in the new version of SPARQL, thus this work can be used to test containment of path SPARQL queries under the RDFS entailment regime.

Containment under constraints Query containment has also been studied under different kinds of constraints. Results in this setting include, decidability of conjunctive query containment under functional and inclusion dependencies is studied in [Johnson & Klug 1984], also [Aho *et al.* 1979] proved decidability of this problem under functional and multi-valued dependencies. Further, decidability and undecidability results are proved in [Calvanese *et al.* 2008] for non-recursive datalog queries under expressive description logic constraints. Moreover, the undecidability is proved in [Calvanese & Rosati 2003] for recursive queries under inclusion dependencies.

The most closely related work is [Calvanese *et al.* 2008] in which query containment under description logic constraints is studied based on an encoding in propositional dynamic logic with converse (CPDL). They establish 2EXPTIME upper bound complexity for containment of queries consisting of union of conjunctive queries under $\mathcal{DLR}$ schema axioms. Our work is similar in spirit, in the sense that the μ-calculus is a logic that subsumes CPDL, and may open the way for extensions of the query languages and ontologies (for instance OWL-DL). Besides, the two languages are different since SPARQL allows for predicates to be used as subject or object of other triple patterns and can be in the scope of a variable. This is not directly allowed in $\mathcal{DLR}$ (union) of conjunctive queries. Our encoding of RDF graphs and SPARQL queries preserves this capability.

The evaluation of SPARQL queries under schema constraints is considered by W3C under the entailment regime principle. In this case, SPARQL queries are evaluated by taking into account the semantics of a schema language [Kollia *et al.* 2011]. It is possible to define query containment under such entailment regimes. We show how this can be done in this Chapter.

Benchmarking Recently, static analysis and optimization of SPARQL queries has attracted widespread attention, notably [Chekol *et al.* 2011, Letelier *et al.* 2012, Chekol *et al.* 2012a, Chekol *et al.* 2012b] for static analysis and [Stocker *et al.* 2008, Groppe *et al.* 2009, Schmidt *et al.* 2010, Letelier *et al.* 2012] for optimization. These studies have grounded the theoretical aspects of these fundamental problems. However, to the best of our knowledge, there is only one implementation from [Letelier *et al.* 2012] and it supports only conjunction and OPTIONAL queries with no projection (containment

of basic and optional graph patterns).

On the other hand, in databases, containment of union conjunctive queries (UCQs) is well studied and has a well know NP-complete complexity. The importance of the study of this problem goes beyond the field of databases, it has its fair share from the description logic community. Many of the works, from description logics, concentrated on the problem of query answering as containment follows from it. These works, have sound theoretical proofs, algorithms, and mathematical explanations. However, they lack an implementation (or experimentation) of their approaches.

Finally, various SPARQL query evaluation performance benchmarks have been proposed [Bizer & Schultz 2008, Bizer & Schultz 2009, Schmidt *et al.* 2009], but no SPARQL query containment benchmark to our knowledge.

5.12 Conclusions

We have introduced a mapping from RDF graphs into transition systems and the encodings of queries and schema axioms in the μ-calculus. We proved that this encoding is correct and can be used for checking query containment. We have provided implementable algorithms, as a consequence, this work opens a way to use available implementations of μ-calculus satisfiability solvers from [Genevès *et al.* 2007] and [Tanabe *et al.* 2008] as introduced in [Chekol *et al.* 2013]. Beyond this, we have established a double exponential upper bound for containment test under $\mathcal{ALCH}$ axioms. The presented encoding is sound (in the sense that whenever the algorithm detects that q_1 is contained in q_2, the containment holds), however it is not complete (i.e., it may happen that the algorithm fails to detect an existing containment relationship) if q_2 is not a SPARQL$_{cdfc}$ query (Definition 10). Interestingly, the cases where the algorithm might be incomplete can be detected, since those cases correspond to the queries of type SPARQL$_{cdfc}$, hence the user can be warned about such a potential risk. These cases have been dealt with in Chapter 6 of [Chekol 2012]. We would like to emphasize that, in addition to the complexity bound we provide, no implementation has been reported in previous works.

The evaluation of SPARQL query containment should help developers to produce more and better solvers. Our experiments contributed to the evaluation of SPARQL query containment in several ways:

- We have studied the demographics of SPARQL queries on a large example and found that (1) a large part of these queries are acyclic, and (2) those parts that either contain projections (effective SELECT) or not, are significant;
- From this study and the state of the art in query containment solvers, we have designed a benchmark suite made of three suites testing conjunctive queries without projection, union of conjunctive queries with projection and with RDFS reasoning;
- We have proposed and implemented a methodology for evaluating this problem;
- Finally, we have applied these to existing containment solvers (SPARQL-Algebra, AFMU and TreeSolver) and we can report the following lessons:

- All tested solutions perform correctly with respect to their declared applicability limits (which are easily testable);
- SPARQL query containment can be practically performed, in spite of its complexity, in a reasonable time with respect to network communication costs,
- the current state-of-the-art is at its early stage and requires improvement and new ways to determine containment and equivalence of queries, in order to become a useful tool for query optimizers.

These benchmark suites are well-suited for pinpointing the theoretical shortcomings of containment solvers.

Chapter 6

Conclusion and Perspectives

6.1 Summary of Contributions

This document presents an excerpt from the research results that I have obtained since I received a PhD in December 2006. I chose to focus on a few contributions. While all of them aim at building safer and more efficient web applications, each one deals with a different particularity of web applications. These contributions thus concern 4 subtopics: (i) the evolution of schemas, (ii) functions and polymorphism, (iii) automated analysis of layouts, and (iv) containment for queries over graphs. Chapter 1 explains the originality of my research approach and summarises these contributions. Each contribution is presented in further details in a dedicated Chapter of this document.

6.2 Perspectives

Perspectives of my research are drawn here under the form of a summary of the research program of the Tyrex team-project, in which I lead the activities on modeling and static analysis for web applications.

6.2.1 Motivations: social and economic challenges

The web represents the biggest mass of information that mankind has ever gathered. Furthermore, during the last two decades, the web became crucial in our daily life activities (work, banking, shopping, education, administration, leisure, social networking). This revolution is continuing its path toward a more compelling user experience through richer content (such in HTML5, multimedia, 3D audio and graphics) and ever increasing web applications via their reconversion through services. The future of the web will be influenced by our ability to leverage this unprecedented potential and to accomplish the successful synergy of applications and content.

This proposal aims at developing a vision of a web where content is enhanced and protected, applications made easier to build, maintain and secure. We seek at opening new horizons for the development of the web, enhancing its potential, effectiveness, and dependability.

6.2.2 Scientific goals and research directions

The main open problem that we can observe today is a lack of formalisms, concepts and tools for reasoning simultaneously over documents, data and communication aspects

in programs. The scientific challenge that we face is to establish such a unifying framework in the context of the web. This is a difficult problem that we propose to address along two main directions:

- **modeling**, which consists in capturing various aspects of document, data and communication in a unifying model, and whose hard part consists in taking into account the peculiarities of the web that require new theoretical tools that do not exist today.
- **analysis, verification and optimization**, which consist in guaranteeing safety and efficiency properties of information systems, and whose hard part consists in dealing with very complex problems close to the frontier of decidability, and therefore in finding useful balances between expressivity, complexity, succinctness, algorithmic techniques and effective implementations.

This research proposal aims at developing formalisms, languages, concepts, algorithms, and tools for building a unifying framework, along the two above interconnected directions. The overall goal is to enable more reliable, secure, and efficient systems. We give more details on each direction below.

6.2.2.1 Modeling documents, data and communications

Specificity of web documents Web documents provide a new field of study, for which it is not possible to use already existing techniques without substantial modifications. The peculiarity of web documents originates from their ordered tree structure. These structures can for example be seen as a relaxation of the classical relational model, one of the foundations of traditional databases, where less rigid and homogeneous "data fields" are allowed. This data model has proven to be very useful for representing various families of documents: multimedia, hypertext, news articles, scientific documents, etc. It is therefore necessary to develop new theoretical foundations, possibly drawing on methods used in other domains of computer science.

One essential concept consists in describing classes of documents that share the same requirements (e.g. web pages through XHTML, or mathematical formulas through MathML). Mastering such representations is also crucial for reasoning over sets of documents. From a theoretical point of view, this modeling task constitutes a renewal for the study of tree automata and logical theories introduced in the late 1960's. These theories are currently rapidly evolving to support the new features provided by web documents, requiring more and more expressiveness and succinctness. We intend to contribute to this modeling effort, especially through contributions on modal logics such as the modal μ-calculus, introduced more recently.

Universal content models and formats Models and formats used for sharing multimedia content on the web must represent the many facets of multimedia documents. Their richness and versatility determine how multimedia content can be processed and used in various contexts. During the last decade, content has shifted from mainly static

pages to highly dynamic and programmable ones. However, content was massively produced in a hackish manner and very basic document features have been subcontracted to scripting which became the "Jack of all Trades" technology in browsers. In addition, a huge portion of the content on the web is today not well-formed, nor valid, severely compromising their long-term access and their automatic processing.

We believe that it is vital to design rigorously documents to outlive any particular piece of hardware or system where they may reside. Furthermore, we seek to build advanced document models that allow describing the increasing variety of modalities such as 3D sound, augmented reality and dynamic content (e.g. data streams) which are becoming a commodity on mobile platforms and applications. The difficulty here is to be able to create models and formats that combine these aspects by declarative means in a consistent manner both at syntactic and semantic level. They should be able to both enhance user experience and facilitate their manipulation by programs.

Specificity of the web: integrating documents, data and communication aspects The web is traditionally composed of resources (data and documents) and services (applications) that exchange resources. The frontier between the two becomes fuzzy as more and more scripting occurs in web pages. However, scripting is currently done at very low level (e.g. similar to an assembly language) and this prevents many sorts of analysis and processing. If we consider XML programming at a higher level then, in the same way as XML documents are twofolds — the raw content and its type — we can consider two aspects of a programming language, with respect to XML: whether or not it provides syntactic support to process XML documents (content side), whether or not it can enforce document constraints (type side) and finally, whether or not it offers the means to integrate smoothly with external services (communication side). We believe that there is a need for higher level abstractions, that make machine processing possible or easier, and that integrate/encompass all these aspects. Current programming technology is still very limited from this perspective. For example, XQuery, which represents a good candidate for a uniform and high level language, has a very imprecise type system and has no communication facilities.

More generally, representing in a uniform way data and programs is a first step toward higher order programming, as noticed by Luca Cardelli in his work on semi-structured computation when he remarks: "if we can take advantage of the similarities [between mobile computation and semi-structured data] and generalize them, we may obtain a broader model of data and computation on the Internet." [1] We believe that this important step can be investigated along several directions. One direction consists in extending expressive modal logics with, for instance, function types for representing programs. Another direction consists in considering process calculi as the missing part for, e.g., extending the XQuery data model with a broader computation model.

[1] Luca Cardelli. "Semistructured Computation". In Research Issues in Structured and Semistructured Database Programming. Lecture Notes in Computer Science, Volume 1949, 2000.

6.2.2.2 Analysis by Reasoning

Type-checking web applications Developing safer web applications depends on the quality of the methods, that we will be able to produce in order to enable the correct manipulation of data in applications. Classical program verification techniques fail to extend to rich data manipulations which are the core of web programming. We propose to develop web program analysis techniques (verification and optimization) which allow detecting errors and enhancing performance in data manipulation. We will concentrate on techniques based on type checking by introducing appropriate type systems and reasoning techniques on programs. The main challenge here is to find decidable methods whose complexity do not preclude their practical applicability. To reach this goal we intend to use logical methods such as modal logics and satisfiability solvers where we gained significant experience.

Global verification of data manipulation and exchange We seek to build global analysis and verification techniques encompassing errors in data manipulation, data exchanges in communication protocols and the interactions between application components. The type systems approach seems particularly appropriate since they allow, by construction, to ensure global properties of a web application by a local and modular verification of its components. As such, they will constitute an important object of investigation. Specifically, we will focus on type systems based on the modeling described in Section 6.2.2.1 (modal logics extended with higher order capabilities such as polymorphism, or process calculi, etc). The expected benefit of such a formalization is to leverage on the extension of the large tool box of proof methods and theoretical results that equip existing calculi. Ideas, concepts, and techniques from lambda-calculi and from process calculi have been successfully applied to the study of behavioral properties of distributed systems, and of type systems for concurrent (functional and/or object-oriented) languages. Overall, we seek to integrate all these aspects in a uniform and soundly based type system.

Designing for evolution In the ever-changing context of the web, XML schemas continuously change in order to cope with the natural evolution of the entities they describe. A change in the schema may require updates of programs that must cope with the newly described set of valid documents. We propose to introduce new methods and tools for determining and facilitating program updates resulting from these changes. Similarly, web services evolve over time, through the modifications of their interfaces. We intend to develop reasoning techniques capable of analyzing programs, schemas and communications in order to automate and efficiently guide these unavoidable updates. Such methods are crucial to enforce quality assurance in web applications and to help tackling forward/backward compatibilities issues.

Supporting integrated, rich, dynamic and augmented content Until now, content rendering on the web was mainly based on supporting media formats separately. It is still notably the case in HTML5 where vector graphics, MathML content,

audio and video are supported as isolated media types. With their increasing support in browsers together with others such as 3D audio and graphics, we need more than ever methods to integrate them tightly in applications and in particular in browsers. In addition, with the increasing use of web content in mobile terminals, we need to take into account highly dynamic information flowing from sensors (positioning and orientation) and camera. These information need to be captured and efficiently combined with content in web browsers. To reach that goal, we need to ease the manipulation of such content with carefully designed programming interfaces and by developing supporting integrative methods. The challenge is to find appropriate abstractions while hiding the increasing complexity of such content.

Bibliography

[Abiteboul *et al.* 1995] S. Abiteboul, R. Hull and V. Vianu. Foundations of databases, volume 8. Addison-Wesley, 1995. (Cited on page 143.)

[Aho *et al.* 1979] A. V. Aho, Y. Sagiv and J. D. Ullman. *Equivalences Among Relational Expressions.* SIAM J. Comput., vol. 8, no. 2, pages 218–246, 1979. (Cited on page 145.)

[Alkhateeb *et al.* 2009] F. Alkhateeb, J.F. Baget and J. Euzenat. *Extending SPARQL with regular expression patterns (for querying RDF).* J. Web Semantics, vol. 7, no. 2, pages 57–73, 2009. (Cited on pages 9 and 102.)

[Angles & Gutierrez 2008] R. Angles and C. Gutierrez. *The Expressive Power of SPARQL.* The Semantic Web-ISWC 2008, pages 114–129, 2008. (Cited on page 143.)

[Baader & Nutt 2003] F. Baader and W. Nutt. *The description logic handbook.* pages 43–95. Cambridge University Press, New York, NY, USA, 2003. (Cited on page 107.)

[Baader *et al.* 2007] F. Baader, D. Calvanese, D. McGuinness, D. Nardi and P. F. Patel-Schneider, editeurs. The description logic handbook: Theory, implementation, and applications. Cambridge University Press, 2007. ISBN 9780511717383. (Cited on pages 107 and 109.)

[Baget 2005] J.F. Baget. *RDF Entailment as a Graph Homomorphism.* The Semantic Web–ISWC 2005, pages 82–96, 2005. (Cited on page 119.)

[Barceló *et al.* 2010] P. Barceló, C. Hurtado, L. Libkin and P. Wood. *Expressive languages for path queries over graph-structured data.* In PODS'10, pages 3–14. ACM, 2010. (Cited on page 145.)

[Barcenas *et al.* 2011] Everardo Barcenas, Pierre Genevès, Nabil Layaïda and Alan Schmitt. *Query reasoning on trees with types, interleaving and counting.* In IJCAI'11 : Proceedings of the 22nd International Joint Conference on Artificial Intelligence, pages 718–723, 2011. (Cited on page 10.)

[Barton & Odvarko 2010] John J. Barton and Jan Odvarko. *Dynamic and graphical web page breakpoints.* In Proceedings of the 19th international conference on World wide web, WWW '10, pages 81–90, New York, NY, USA, 2010. ACM. (Cited on page 75.)

[Benedikt & Cheney 2010] Michael Benedikt and James Cheney. *Destabilizers and Independence of XML Updates.* Proceedings of the VLDB Endowment, vol. 3, no. 1, pages 906–917, 2010. (Cited on pages 44 and 45.)

[Benedikt & Koch 2009] Michael Benedikt and Christoph Koch. *XPath leashed.* ACM Comput. Surv., vol. 41, pages 3:1–3:54, January 2009. (Cited on page 39.)

[Benedikt *et al.* 2005] Michael Benedikt, Wenfei Fan and Floris Geerts. *XPath satisfiability in the presence of DTDs.* In PODS '05, pages 25–36. ACM Press, 2005. (Cited on page 39.)

[Benzaken *et al.* 2003] Véronique Benzaken, Giuseppe Castagna and Alain Frisch. *CDuce: an XML-centric general-purpose language.* In Proceedings of the 8th international conference on functional programming (ICFP '03), pages 51–63, Uppsala, Sweden, 2003. (Cited on pages 7, 39, 42, 45 and 70.)

[Beyer *et al.* 2005] Kevin Beyer, Fatma Özcan, Sundar Saiprasad and Bert Van der Linden. *DB2/XML: designing for evolution.* In SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pages 948–952. ACM, 2005. (Cited on page 38.)

[Bierman *et al.* 2010] Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu and David Langworthy. *Semantic subtyping with an SMT solver.* In Proceedings of the 15th international conference on functional programming (ICFP '10), pages 105–116, Baltimore, MD, USA, 2010. (Cited on pages 44, 45 and 71.)

[Bizer & Schultz 2008] C. Bizer and A. Schultz. *Benchmarking the performance of storage systems that expose SPARQL endpoints.* In Proc. 4 th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS), 2008. (Cited on page 146.)

[Bizer & Schultz 2009] C. Bizer and A. Schultz. *The Berlin SPARQL benchmark.* International Journal on Semantic Web and Information Systems (IJSWIS), vol. 5, no. 2, pages 1–24, 2009. (Cited on page 146.)

[Blackburn *et al.* 2007] P. Blackburn, J. van Benthem and F. Wolter. Handbook of Modal Logic. Elsevier, 2007. (Cited on page 115.)

[Boag *et al.* 2007] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie and Jérôme Siméon. *XQuery 1.0: An XML Query Language, W3C Recommendation*, January 2007. (Cited on page 43.)

[Bonatti & Peron 2004] P.A. Bonatti and A. Peron. *On the undecidability of logics with converse, nominals, recursion and counting.* Artificial Intelligence, vol. 158, no. 1, pages 75–96, 2004. (Cited on page 117.)

[Bonatti *et al.* 2006] P. A. Bonatti, C. Lutz, A. Murano and M. Y. Vardi. *The Complexity of Enriched μ-calculi.* Automata, Languages and Programming, pages 540–551, 2006. (Cited on pages 117 and 131.)

[Bos *et al.* 2011] Bert Bos, Tantek Çelik, Ian Hickson and Håkon Wium Lie. *Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification.* W3C recommendation, World Wide Web Consortium, June 2011. (Cited on pages 78 and 86.)

[Bosch *et al.* 2014] Martí Bosch, Pierre Genevès and Nabil Layaïda. *Automated Refactoring for Size Reduction of CSS Style Sheets.* In Proceedings of the 2014 ACM symposium on Document engineering (to appear), DocEng '14. ACM, 2014. (Cited on page 8.)

[Bryant 1986] Randal E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation.* IEEE Trans. on Computers, vol. 35, no. 8, pages 677–691, 1986. (Cited on page 64.)

[Calvanese & Rosati 2003] D. Calvanese and R. Rosati. *Answering Recursive Queries under Keys and Foreign Keys is Undecidable.* In Proc. of the 10th Int. Workshop on Knowledge Representation meets Databases (KRDB 2003), volume 79, pages 3–14, 2003. (Cited on page 145.)

[Calvanese *et al.* 1998] D. Calvanese, G. De Giacomo and M. Lenzerini. *On the decidability of query containment under constraints.* In Proceedings of PODS, pages 149–158. ACM, 1998. (Cited on pages 111 and 144.)

[Calvanese *et al.* 2000] D. Calvanese, G. De Giacomo, M. Lenzerini and M. Y. Vardi. *Containment of Conjunctive Regular Path Queries with Inverse.* In Proc. of the 7th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR 2000), pages 176–185, 2000. (Cited on page 145.)

[Calvanese *et al.* 2003] D. Calvanese, G. De Giacomo, M. Lenzerini and M. Y. Vardi. *Reasoning on Regular Path Queries.* SIGMOD Record, vol. 32, no. 4, pages 83–92, 2003. (Cited on page 145.)

[Calvanese *et al.* 2007] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini and R. Rosati. *Tractable Reasoning and Efficient Query Answering in Description Logics: The DL-Lite Family.* Journal of Automated Reasoning, vol. 39, no. 3, pages 385–429, 2007. (Cited on page 144.)

[Calvanese *et al.* 2008] D. Calvanese, G. De Giacomo and M. Lenzerini. *Conjunctive Query Containment and Answering under Description Logics Constraints.* ACM Trans. on Computational Logic, vol. 9, no. 3, pages 22.1–22.31, 2008. (Cited on pages 119, 123, 132, 144 and 145.)

[Calvanese *et al.* 2011] D. Calvanese, M. Ortiz and M. Simkus. *Containment of Regular Path Queries under Description Logic Constraints.* In Proc. of the 22nd Int. Joint Conf. on Artificial Intelligence (IJCAI 2011), 2011. (Cited on page 145.)

[Castagna & Nguyen 2008] Giuseppe Castagna and Kim Nguyen. *Typed iterators for XML.* In ICFP '08: Proceedings of the ACM SIGPLAN international conference on Functional programming, pages 15–26, 2008. (Cited on page 39.)

[Castagna & Xu 2011] G. Castagna and Z. Xu. *Set-theoretic Foundation of Parametric Polymorphism and Subtyping.* In Proceedings of the 16th international conference on functional programming (ICFP '11), Tokyo, september 2011. (Cited on pages 7, 42, 43, 45, 55, 57, 58, 60, 68, 70 and 71.)

[Çelik *et al.* 2011] Tantek Çelik, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss and John Williams. *Selectors Level 3*. W3C recommendation, World Wide Web Consortium, September 2011. (Cited on pages 76 and 83.)

[Chandra & Merlin 1977] A. K. Chandra and P. M. Merlin. *Optimal Implementation of Conjunctive Queries in Relational Data Bases*. In Proceedings of the ninth annual ACM symposium on Theory of computing, pages 77–90. ACM, 1977. (Cited on page 136.)

[Chebotko *et al.* 2006] A. Chebotko, S. Lu, H.M. Jamil and F. Fotouhi. *Semantics preserving SPARQL-to-SQL query translation for optional graph patterns*. Rapport technique, Technical Report TR-DB-052006-CLJF, 2006. (Cited on page 143.)

[Chekol *et al.* 2011] M. W. Chekol, J. Euzenat, P. Genevès and N. Layaïda. *PSPARQL Query Containment*. In DBPL'11, August 2011. (Cited on pages 9 and 145.)

[Chekol *et al.* 2012a] M. W. Chekol, J. Euzenat, P. Genevès and N. Layaïda. *SPARQL Query Containment under RDFS Entailment Regime*. In IJCAR'12, pages 134–148. Springer, 2012. (Cited on pages 9, 133, 137 and 145.)

[Chekol *et al.* 2012b] M. W. Chekol, J. Euzenat, P. Genevès and N. Layaïda. *SPARQL Query Containment under SHI Axioms*. In AAAI'12, volume 1, pages 10–16, 2012. (Cited on pages 9, 132, 133 and 145.)

[Chekol *et al.* 2013] Melisachew Wudage Chekol, Jérôme Euzenat, Pierre Genevès and Nabil Layaïda. *Evaluating and Benchmarking SPARQL Query Containment Solvers*. In ISWC'13: Proceedings of the 12th International Semantic Web Conference, pages 408–423, 2013. (Cited on pages 9 and 146.)

[Chekol 2012] Melisachew Wudage Chekol. *Static Analysis of Semantic Web Queries*. Phd, Université de Grenoble, Dec 2012. (Cited on page 146.)

[Chekuri & Rajaraman 1997] C. Chekuri and A. Rajaraman. *Conjunctive query containment revisited*. Database Theory–ICDT'97, pages 56–70, 1997. (Cited on page 136.)

[Clark & DeRose 1999] James Clark and Steve DeRose. *XML Path Language (XPath) Version 1.0, W3C Recommendation*, November 1999. (Cited on pages 16 and 45.)

[Colazzo *et al.* 2004] Dario Colazzo, Giorgio Ghelli, Paolo Manghi and Carlo Sartiani. *Types for path correctness of XML queries*. In ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming, pages 126–137, New York, NY, USA, 2004. ACM Press. (Cited on page 39.)

[Colazzo *et al.* 2006] Dario Colazzo, Giorgio Ghelli, Paolo Manghi and Carlo Sartiani. *Static analysis for path correctness of XML queries*. J. Funct. Program., vol. 16, no. 4-5, pages 621–661, 2006. (Cited on page 39.)

[Cyganiak 2005] R. Cyganiak. *A relational algebra for SPARQL.* Digital Media Systems Laboratory HP Laboratories Bristol. HPL-2005-170, 2005. (Cited on page 143.)

[de Moura & Bjørner 2008] Leonardo Mendonça de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver.* In Proceedings of the 14th international conference on tools and algorithms for the construction and analysis of systems (TACAS '08), pages 337–340, Budapest, 2008. (Cited on pages 44 and 45.)

[Eberlein *et al.* 2010] Kristen James Eberlein, Robert D. Anderson and Gershon Joseph. *Darwin Information Typing Architecture (DITA) Version 1.2.* Oasis standard, OASIS, December 2010. (Cited on page 90.)

[Edmund M. Clarke *et al.* 1999] Jr. Edmund M. Clarke, Orna Grumberg and Doron A. Peled. Model checking. MIT Press, Cambridge, MA, USA, 1999. (Cited on page 64.)

[Eiter *et al.* 2009] T. Eiter, C. Lutz, M. Ortiz and M. Šimkus. *Query answering in description logics with transitive roles.* In Proc. of IJCAI, pages 759–764, 2009. (Cited on pages 110, 111 and 144.)

[Engovatov & Robie 2010] Daniel Engovatov and Jonathan Robie. *XQuery 3.0 Requirements, W3C Working Draft*, September 2010. (Cited on pages 7 and 43.)

[Frisch *et al.* 2008] A. Frisch, G. Castagna and V. Benzaken. *Semantic Subtyping: dealing set-theoretically with function, union, intersection, and negation types.* Journal of the ACM, vol. 55, no. 4, pages 1–64, 2008. (Cited on pages 42, 43, 44, 46, 47, 48, 55 and 70.)

[Gapeyev *et al.* 2006] Vladimir Gapeyev, François Garillot and Benjamin C. Pierce. *Statically Typed Document Transformation: An Xtatic Experience.* In PLAN-X 2006: Proceedings of the International Workshop on Programming Language Technologies for XML, volume NS-05-6 of *BRICS Notes Series*, pages 2–13, Aarhus, Denmark, January 2006. BRICS. (Cited on page 39.)

[Genevès & Layaïda 2006] Pierre Genevès and Nabil Layaïda. *A system for the static analysis of XPath.* ACM Trans. Inf. Syst., vol. 24, no. 4, pages 475–502, 2006. (Cited on pages 4 and 120.)

[Genevès & Layaïda 2010] Pierre Genevès and Nabil Layaïda. *Eliminating dead-code from XQuery programs.* In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE '10, pages 305–306, New York, NY, USA, 2010. ACM. (Cited on page 10.)

[Genevès & Layaïda 2011] Pierre Genevès and Nabil Layaïda. *Inconsistent path detection for XML IDEs.* In Proceeding of the 33rd international conference on Software engineering, ICSE '11, pages 983–985, New York, NY, USA, 2011. ACM. (Cited on page 10.)

[Genevès & Layaïda 2014a] Pierre Genevès and Nabil Layaïda. *Equipping IDEs with XML Path Reasoning Capabilities*. ACM Trans. Internet Technol., 2014. (Cited on page 10.)

[Genevès & Layaïda 2014b] Pierre Genevès and Nabil Layaïda. *The XML Reasoning Solver Project*, June 2014. http://wam.inrialpes.fr/xml. (Cited on pages 14, 16 and 38.)

[Genevès *et al.* 2007] Pierre Genevès, Nabil Layaïda and Alan Schmitt. *Efficient Static Analysis of XML Paths and Types*. In Proceedings of the 28th conference on programming language design and implementation (PLDI '07), pages 342–351, San Diego, CA, USA, 2007. (Cited on pages 7, 8, 9, 16, 17, 19, 21, 22, 29, 39, 44, 45, 50, 52, 55, 61, 62, 63, 71, 81, 83, 91, 97, 102, 142, 144, 146 and 160.)

[Genevès *et al.* 2009] Pierre Genevès, Nabil Layaïda and Vincent Quint. *Identifying query incompatibilities with evolving XML schemas*. In ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming, pages 221–230, New York, NY, USA, 2009. ACM. (Cited on pages 6, 13 and 64.)

[Genevès *et al.* 2011] Pierre Genevès, Nabil Layaïda and Vincent Quint. *Impact of XML Schema Evolution*. ACM Trans. Internet Technol., vol. 11, pages 4:1–4:27, July 2011. (Cited on page 6.)

[Genevès *et al.* 2012] Pierre Genevès, Nabil Layaïda and Vincent Quint. *On the analysis of cascading style sheets*. In WWW'12: Proceedings of the 21st World Wide Web Conference, pages 809–818, April 2012. (Cited on page 8.)

[Genevès *et al.* 2014] Pierre Genevès, Nabil Layaïda, Alan Schmitt and Nils Gesbert. *Efficiently Deciding μ-Calculus with Converse over Finite Trees*. Long version of [Genevès *et al.* 2007], Research Report hal-00868722, CNRS & Inria, June 2014. http://hal.inria.fr/hal-00868722/en/. (Cited on pages 14, 23 and 86.)

[Genevès 2006] Pierre Genevès. *Logics for XML*. PhD thesis, Institut National Polytechnique de Grenoble, December 2006. (Cited on pages 22, 62, 63, 81 and 83.)

[Gesbert *et al.* 2011] Nils Gesbert, Pierre Genevès and Nabil Layaïda. *Parametric Polymorphism and Semantic Subtyping: the Logical Connection*. In ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming, pages 107–116, 2011. (Cited on page 7.)

[Glimm *et al.* 2008] B. Glimm, I. Horrocks, C. Lutz and U. Sattler. *Conjunctive query answering for the description logic $SHIQ$*. J Artif Intell Res, vol. 31, pages 157–204, 2008. (Cited on pages 110, 111 and 144.)

[Groppe & Groppe 2008] Jinghua Groppe and Sven Groppe. *Filtering unsatisfiable XPath queries*. Data Knowl. Eng., vol. 64, no. 1, pages 134–169, 2008. (Cited on page 39.)

[Groppe *et al.* 2006] Sven Groppe, Stefan Bottcher and Jinghua Groppe. *XPath Query Simplification with regard to the Elimination of Intersect and Except Operators.* In ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops, page 86, Washington, DC, USA, 2006. IEEE Computer Society. (Cited on page 39.)

[Groppe *et al.* 2009] J. Groppe, S. Groppe and J. Kolbaum. *Optimization of SPARQL by using coreSPARQL.* In ICEIS (1), pages 107–112, 2009. (Cited on pages 142 and 145.)

[Gutierrez *et al.* 2004] C. Gutierrez, C. Hurtado and A. O. Mendelzon. *Foundations of Semantic Web Databases.* PODS '04, pages 95–106, New York, NY, USA, 2004. (Cited on page 142.)

[Hayes 2004] P. Hayes. *RDF Semantics.* W3C Recommendation, 2004. (Cited on pages 103, 104, 105, 106, 128 and 142.)

[Horrocks & Patel-Schneider 2010] I. Horrocks and P.F. Patel-Schneider. *Knowledge Representation and Reasoning on the Semantic Web: OWL*, 2010. `http://www.cs.ox.ac.uk/ian.horrocks/Publications/download/2010/HoPa10a.pdf`. (Cited on pages 109 and 110.)

[Horrocks *et al.* 2006] I. Horrocks, O. Kutz and U. Sattler. *The even more irresistible SROIQ.* In Proc. of KR 2006, pages 57–67, 2006. (Cited on page 107.)

[Hosoya & Pierce 2003] Haruo Hosoya and Benjamin C. Pierce. *XDuce: A statically typed XML processing language.* ACM Transactions on Internet Technology, vol. 3, no. 2, pages 117–148, 2003. (Cited on pages 39, 42, 45 and 70.)

[Hosoya *et al.* 2005] Haruo Hosoya, Jérôme Vouillon and Benjamin C. Pierce. *Regular expression types for XML.* ACM Transactions on Programming Languages and Systems, vol. 27, pages 46–90, January 2005. (Cited on pages 14, 16, 45 and 70.)

[Hosoya *et al.* 2009] H. Hosoya, A. Frisch and G. Castagna. *Parametric Polymorphism for XML.* ACM Transactions on Programming Languages and Systems, vol. 32, no. 1, pages 1–56, 2009. (Cited on pages 42, 56, 57 and 70.)

[Johnson & Klug 1984] D. S. Johnson and A. C. Klug. *Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies.* Journal of Computer and System Sciences, vol. 28, no. 1, pages 167–189, 1984. (Cited on page 145.)

[Kazakov 2008] Y. Kazakov. *SRIQ and SROIQ are Harder than SHOIQ.* In Description Logics, 2008. (Cited on page 107.)

[Keller & Nussbaumer 2009] Matthias Keller and Martin Nussbaumer. *Cascading style sheets: a novel approach towards productive styling with today's standards.* In Proceedings of the 18th international conference on World wide web, WWW '09, pages 1161–1162, New York, NY, USA, 2009. ACM. (Cited on page 75.)

[Keller & Nussbaumer 2010] Matthias Keller and Martin Nussbaumer. *CSS Code Quality: A Metric for Abstractness.* In Seventh International Conference on the Quality of Information and Communications Technology (QUATIC), pages 116–121, October 2010. (Cited on page 75.)

[Kollia *et al.* 2011] I. Kollia, B. Glimm and I. Horrocks. *SPARQL Query Answering over OWL Ontologies.* In Proc. 8th ESWC, Heraklion (GR), volume 6643 of *LNCS*, pages 382–396, 2011. (Cited on page 145.)

[Kozen 1983] D. Kozen. *Results on the propositional μ-calculus.* Theor. Comp. Sci., vol. 27, pages 333–354, 1983. (Cited on pages 9, 101, 115, 116, 117 and 133.)

[Kupferman *et al.* 2002] O. Kupferman, U. Sattler and M. Vardi. *The complexity of the graded μ-calculus.* Automated Deduction—CADE-18, pages 173–206, 2002. (Cited on page 117.)

[Letelier *et al.* 2012] A. Letelier, J. Pérez, R. Pichler and S. Skritek. *Static analysis and optimization of semantic web queries.* In PODS'12, pages 89–100. ACM, 2012. (Cited on pages 132, 133, 143 and 145.)

[Lie 2005] Håkon Wium Lie. *Cascading style sheets.* Phd thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2005. (Cited on page 74.)

[Lutz *et al.* 2009] C. Lutz, D. Toman and F. Wolter. *Conjunctive Query Answering in the Description Logic EL Using a Relational Database System.* In IJCAI, pages 2070–2075, 2009. (Cited on page 144.)

[Lutz 2008] C. Lutz. *The complexity of conjunctive query answering in expressive description logics.* Automated Reasoning, pages 179–193, 2008. (Cited on pages 110, 111 and 144.)

[Marden & Munson 1999] Philip M. Marden and Ethan V. Munson. *Today's style sheet standards: the great vision blinded.* Computer, vol. 32, no. 11, pages 123–125, nov 1999. (Cited on page 74.)

[Mateescu *et al.* 2009] R. Mateescu, S. Meriot and S. Rampacek. Extending SPARQL with Temporal Logic. INRIA Research Report, RR-7056, 2009. (Cited on page 118.)

[Møller & Schwartzbach 2005] Anders Møller and Michael I. Schwartzbach. *The Design Space of Type Checkers for XML Transformation Languages.* In Proc. Tenth International Conference on Database Theory, ICDT '05, volume 3363 of *LNCS*, pages 17–36, London, UK, January 2005. Springer-Verlag. (Cited on page 39.)

[Moon *et al.* 2008] Hyun J. Moon, Carlo A. Curino, Alin Deutsch and Chien-Yi Hou. *Managing and Querying Transaction-time Databases under Schema Evolution.* In VLDB '08: Proceedings of the 34nd international conference on Very large data bases, pages 882–895. VLDB Endowment, 2008. (Cited on page 39.)

[Moro *et al.* 2007] Mirella M. Moro, Susan Malaika and Lipyeow Lim. *Preserving XML queries during schema evolution.* In WWW '07: Proceedings of the 16th international conference on World Wide Web, pages 1341–1342. ACM, 2007. (Cited on page 39.)

[Muñoz *et al.* 2007] S. Muñoz, J. Pérez and C. Gutierrez. *Minimal Deductive Systems for RDF.* volume 4519 of *LNCS*, pages 53–67, 2007. (Cited on page 105.)

[Murata *et al.* 2005] Makoto Murata, Dongwon Lee, Murali Mani and Kohsuke Kawaguchi. *Taxonomy of XML schema languages using formal language theory.* ACM TOIT, vol. 5, no. 4, pages 660–704, 2005. (Cited on pages 12, 15 and 30.)

[Ortiz *et al.* 2008a] M. Ortiz, D. Calvanese and T. Eiter. *Data Complexity of Query Answering in Expressive Description Logics via Tableaux.* Journal of Automated Reasoning, vol. 41, no. 1, pages 61–98, 2008. (Cited on pages 110 and 144.)

[Ortiz *et al.* 2008b] M. Ortiz, M. Šimkus and T. Eiter. *Worst-case optimal conjunctive query answering for an expressive description logic without inverses.* In Proc. of AAAI, volume 8, 2008. (Cited on page 111.)

[Pan *et al.* 2006] Guoqiang Pan, Ulrike Sattler and Moshe Y. Vardi. *BDD-Based Decision Procedures for the modal logic K.* Journal of Applied Non-classical Logics, vol. 16, no. 1-2, pages 169–208, 2006. (Cited on page 64.)

[Pérez *et al.* 2009a] J. Pérez, M. Arenas and C. Gutierrez. *Semantics and complexity of SPARQL.* ACM Transactions on Database Systems (TODS), vol. 34, no. 3, page 16, 2009. (Cited on pages 111 and 114.)

[Pérez *et al.* 2009b] J. Pérez, M. Arenas and C. Gutierrez. Semantics of SPARQL. Unpublished Manuscript, 2009. (Cited on page 111.)

[Pichler *et al.* 2010] R. Pichler, A. Polleres, S. Skritek and S. Woltran. *Redundancy elimination on rdf graphs in the presence of rules, constraints, and queries.* Web Reasoning and Rule Systems, pages 133–148, 2010. (Cited on page 144.)

[Pietriga 2005] Emmanuel Pietriga. *MathML Content2Presentation Transformation*, May 2005.
http://www.lri.fr/~pietriga/mathmlc2p/mathmlc2p.html. (Cited on page 36.)

[Polleres 2007] A. Polleres. *From SPARQL to rules (and back).* In WWW '07, pages 787–796, 2007. (Cited on pages 142 and 143.)

[Polleres 2012] A. Polleres. *How (well) do Datalog, SPARQL and RIF interplay?* Datalog in Academia and Industry, pages 27–30, 2012. (Cited on page 114.)

[Prud'hommeaux & Seaborne 2008] E. Prud'hommeaux and A. Seaborne. *SPARQL Query Language for RDF.* W3C Rec., 2008. (Cited on pages 111 and 114.)

[Quint & Vatton 2007] Vincent Quint and Irène Vatton. *Editing with style.* In Proceedings of the 2007 ACM symposium on Document engineering, DocEng '07, pages 151–160, New York, NY, USA, 2007. ACM. (Cited on page 75.)

[Reynolds 1983] John C. Reynolds. *Types, Abstraction and Parametric Polymorphism.* In IFIP Congress, pages 513–523, 1983. (Cited on page 57.)

[Rose 2004] Kristoffer H. Rose. *The XML world view.* In DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering, pages 34–34, New York, NY, USA, 2004. ACM. (Cited on page 39.)

[Sattler & Vardi 2001] U. Sattler and M. Y. Vardi. *The Hybrid μ-Calculus.* In IJCAR, pages 76–91, 2001. (Cited on page 117.)

[Schmidt *et al.* 2009] M. Schmidt, T. Hornung, G. Lausen and C. Pinkel. *SP^2Bench: A SPARQL Performance Benchmark.* In ICDE'09, pages 222–233. Ieee, 2009. (Cited on page 146.)

[Schmidt *et al.* 2010] M. Schmidt, M. Meier and G. Lausen. *Foundations of SPARQL Query Optimization.* In ICDT '10, pages 4–33, New York, NY, USA, 2010. ACM. (Cited on pages 142 and 145.)

[Sedlar 2005] Eric Sedlar. *Managing structure in bits & pieces: the killer use case for XML.* In SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data, pages 818–821. ACM, 2005. (Cited on page 39.)

[Serfiotis *et al.* 2005] G. Serfiotis, I. Koffina, V. Christophides and V. Tannen. *Containment and Minimization of RDF/S Query Patterns.* In The Semantic Web - ISWC 2005, volume 3729 of *LNCS*, pages 607–623, 2005. (Cited on page 142.)

[Serrano 2010] Manuel Serrano. *HSS: a compiler for cascading style sheets.* In Temur Kutsia, Wolfgang Schreiner and Maribel Fernández, editeurs, PPDP, pages 109–118. ACM, 2010. (Cited on page 75.)

[Stocker *et al.* 2008] M. Stocker, A. Seaborne, A. Bernstein, C. Kiefer and D. Reynolds. *SPARQL Basic Graph Pattern Optimization Using Selectivity Estimation.* In Proceeding of the 17th international conference on World Wide Web, WWW '08, pages 595–604, New York, NY, USA, 2008. ACM. (Cited on pages 142 and 145.)

[Tanabe *et al.* 2005] Y. Tanabe, K. Takahashi, M. Yamamoto, A. Tozawa and M. Hagiya. *A Decision Procedure for the Alternation-Free Two-Way Modal μ-calculus.* In TABLEAUX, pages 277–291, 2005. (Cited on pages 9, 102, 133 and 142.)

[Tanabe *et al.* 2008] Y. Tanabe, K. Takahashi and M. Hagiya. *A Decision Procedure for Alternation-Free Modal μ-calculi.* In Advances in Modal Logic, pages 341–362, 2008. (Cited on pages 9, 101, 102, 118 and 146.)

[Ter Horst 2005] H.J. Ter Horst. *Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary.* Web Semantics: Science, Services and Agents on the World Wide Web, vol. 3, no. 2-3, pages 79–115, 2005. (Cited on pages 105, 106 and 107.)

[Thomas 1990] Wolfgang Thomas. *Automata on infinite objects.* In Handbook of theoretical computer science (vol. B): formal models and semantics, pages 133–191. MIT Press, Cambridge, MA, USA, 1990. (Cited on page 16.)

[Vouillon 2006] Jérôme Vouillon. *Polymorphic regular tree types and patterns.* In Proceedings of the 33rd symposium on principles of programming languages (POPL '06), pages 103–114, Charleston, SC, USA, 2006. (Cited on pages 67 and 70.)

[Wadler 1989] Philip Wadler. *Theorems for free!* In Proceedings of the 4th international conference on functional programming languages and computer architecture (FPCA '89), pages 347–359, London, 1989. (Cited on page 60.)

[Walsh 1999] Leonard Muellner Norman Walsh. Docbook: The definitive guide. O'Reilly & Associates, 1999. (Cited on page 90.)

[Werntges 2011] Heinz Werntges. *A CSS for DocBook*, November 2011. http://www.cs.hs-rm.de/ werntges/proj/wysiwyg-dbk01.html. (Cited on page 90.)

[Yu & Popa 2005] Cong Yu and Lucian Popa. *Semantic adaptation of schema mappings when schemas evolve.* In VLDB '05: Proceedings of the 31st international conference on Very large data bases, pages 1006–1017. VLDB Endowment, 2005. (Cited on page 39.)