Efficiently Locating Schema Incompatibilities in an eXtensible Markup Language

Roland Kaufmann
University of Bergen
roland.kaufmann@student.uib.no

May 30, 2003

Preface

This thesis is submitted as part of a cand. scient. graduate degree from the Department of Informatics under the Faculty of Mathematics and Natural Sciences at the University of Bergen.

Synopsis

The work presented here extends an existing algorithm for testing if an inclusion relation exists between two markup schemata, to only take into account the parts of the grammar that have been used in a given subset of its language. Statistics for this purpose are gathered in combination with validation when documents are entered and are stored along with them in the repository. This modified subtyping relation is used to determine compatibility with the current database when a schema is upgraded.

Candidate's background

As a student, the author has been associated with the Programming Theory group, which has a strong linguistic and algebraic focus besides design and tools development. This thesis is a culmination of experiences from working with the HyperEducator project, one of the ongoing research activities.

Acknowledgments

A work such as this thesis could not have been written in a vacuum without any aid, technically as well as socially.

The author would like to thank his advisor Khalid Azim Mughal for the interest he has shown in this work, and for the ability to ask the right questions at the right time, providing guidance, direction and inspiration. He has kept loose enough reigns to let the thesis take an organic direction yet tight enough to make sure it still was on track.

The author would also like to thank his family and friends for their support and encouragement while writing this thesis, and not at least his live-in girlfriend Jorunn for her enduring patience, compassion and love.

The time at the University of Bergen has been an unforgettable experience.

Colophon

A development license of Oracle's JDeveloper 9.0.3 IDE was used for editing and debugging and all code have been run on Sun's JDK 1.4.1. It should nonetheless run in any other environment that supports the Java2 runtime environment, version 1.4 or later. The project can also be assembled from the command-line using the build tool Ant 1.5.1, which is included.

No other third-party libraries have been used than JUnit, currently at version 3.8.1, which is employed as an aid for writing unit tests. The framework may be used without these unit tests, though.

The text of the thesis has been typed in jEdit 4.0 and prepared with Christian Schenk's MiKTeX 2.2 distribution of the LaTeX 2ϵ typesetting system before converted to PDF using Aladdin Ghostscript 7.04. The font is Palatino 11 pt for main text and mathematics while Lucida Sans have used in headings and Courier in source code listings. For graphics, Dia 0.90 has been used, with figures exported to the PSTricks format for inclusion in the main document.

All programs have been run at a Dell Inspiron 8100 workstation with an Intel Pentium III CPU running at 933 MHz and 512 MB RAM on the Windows XP 5.1 operating system. Online literature searches has been conducted at `http://citeseer.nj.nec.com/cs` and `http://www.google.com`.

Contents

Figures

Chapter 1

Introduction

This chapter will present the problems this thesis sets out to solve, give an overview of the goals that are targeted in that matter, and provide a roadmap for how they are intended to be reached.

1.1 Problems and goals

A problem may be defined as the difference between the actual and a desired situation [KT65]. Goals are the conditions that must be satisfied before the desired outcome can said to be reached. In this section, the current and wanted state of content integrity will be portrayed.

1.1.1 Background

Recent trends in content management systems have been to separate the data [1] representing information from the data telling how it should be presented, and furthermore employ structures reflecting the semantics in its encoding.

Schemata[2] describe what constitutes valid data, and they are used to perform validation as data is loaded into a *repository* where remaining for later use. When information eventually is to be displayed, a *stylesheet*[3] directs the transformation of data into a format that can be rendered by the client.

The power of this model lays in that multiple stylesheets can be used to render the same data in various ways, and that stylesheets can be changed to reflect artistic change to the presentation without modifying the underlaying data. Conversely, data can be edited without considerations of its ultimate appearance, as long as the data and the stylesheet are both in accordance with the schema. This process is illustrated in figure 1.1 on the following page.

Having data stored in a repository is valuable for at least three reasons:

[1] XML is an abbreviation for "eXtensive Markup Language"
[2] XSD is an abbreviation for "XML Schema Definition"
[3] XSLT is an abbreviation for "eXtensive Stylesheet Language Transformations"

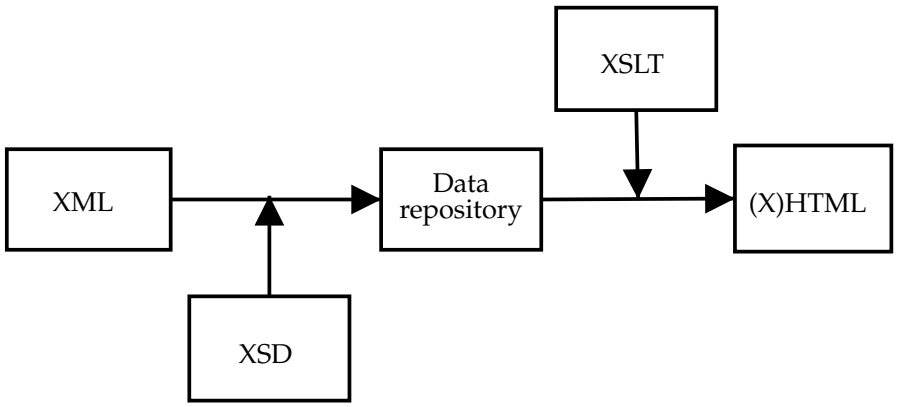


Figure 1.1: Content management system model

(i) If data is validated when entering the repository and non-valid data correspondingly rejected, then the user is assured that data retrieved from the repository is always valid and that no additional error checking is necessary while processing it.

(ii) Invalid data is caught upon entering the repository and can be fixed as part of the storing process. This is analogous to compile-time checking of computer programs.

(iii) By having a central repository, the client does not need to worry about locating the data.

Markup technologies are often used for these purposes because they offer a content-independent way to add structure to documents. The versatility and ubiquity of such standards makes a lot of tools and frameworks available upon which new and more specialized systems can be built.

1.1.2 Problem description

The above model works very well if the domain to which it is deployed is static. However, in reality a need to change the information will arise from time to time. This may not be a problem when it comes to the data itself, because the old information can be removed from the repository and then new data added. Neither does a change in the stylesheets pose a significant problem because nothing else depends on them (in the scope of this model at least). In fact, one can only look upon the new stylesheet as merely another way of presenting the data, something that the model actually was designed to facilitate.

The real problem lays in changing the schemata. If the schema is changed after data has been put into the repository, property (i) as described in the previous section — and in a sense also property (ii) — have been violated.

What is needed is an extension of the concept of data validation upon entrance towards also applying the schemata, by realizing that they are nothing but a special kind of data called *metadata*. The revised system is depicted in figure 1.2 on the next page.

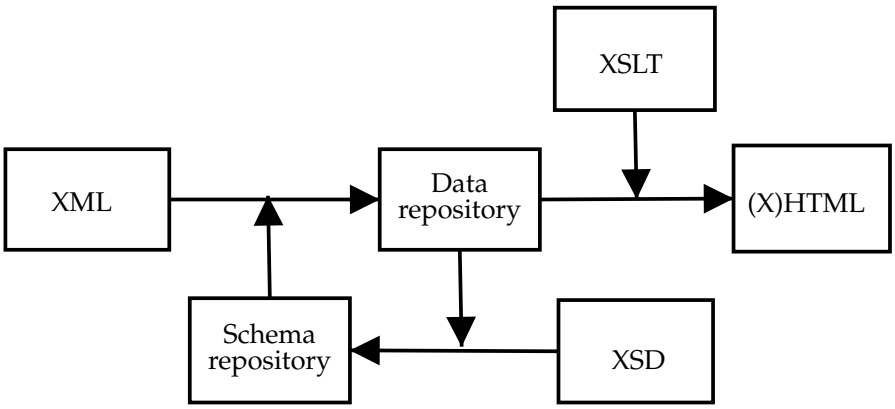


Figure 1.2: System with added feedback from data repository

When a schema is changed in the repository, all data that belongs to it are checked to determine whether they are still compatible. If there is a conflict, then the user must either:

(a) Resolve the conflict by changing the schema,

(b) Resolve the conflict by correcting the data, or

(c) Mark the data for deferred conflict resolution that effectively hides them from being used by the stylesheets.

Despite the perceived added value in such a system, there is to the authors knowledge no standard solution for such a setup and it is for that reason a problem considered worthy of further research. The task of this thesis is hence to solve this problem. Put another way, the task can be formulated:

> "Design and implement a system that directed by the repository of already validated documents can identify which parts of them are not compatible with a given change in the schema, using information about dependencies between the documents and their schemata already established in the database."

1.1.3 Analogy

It may be fruitful to look at a related problem familiar to software developers to illustrate the repository consistency problem further. In this analogy, a system corresponds to the repository, interfaces to schemata, classes implementing those interfaces to documents, and code using references to such classes to stylesheets. Compiling program code is the same as validating documents. Examples will be given in the computer language Java and assumed to be run in its associated runtime environment.

Imagine a program built up of classes that are under development and whose interfaces are hence still in flux. Say that there is a class named `Library`, given in listing 1.3. The semantics of this class is not important in this regard, only the contract it offers to its clients.

```
public class Library {
  public static int foo( int a ) { return a * a; }
}
```

Listing 1.3: Original library class

```
public class Program {
  public static void main( String[] args ) { System.out.println( Library.foo( 42 ) ); }
}
```

Listing 1.4: Program using the original contract

Listing 1.4 is an example of a small program that uses this class. This program compiles and runs just fine.

Consider then a change in the definition of the library class to the one given in listing 1.5, but without any corresponding changes done elsewhere, in particular not in `Program`. As can be seen, an extra parameter `b` has been added to the signature of the method `foo`, causing the contract of the class to change.

```
public class Library {
  public static int foo( int a, int b ) { return a * a + b; }
}
```

Listing 1.5: Altered library class

If only the class `Library` is compiled and the resulting object code deployed to the system, a run of the program will now terminate with a run-time error as shown in listing 1.6 since the main program still expects the library to adhere to the old contract which was effective when it was compiled. However, the part of the interface that was referred to earlier no longer exists, breaking backward compatibility.

```
Exception in thread "main" java.lang.NoSuchMethodError: Library.foo(I)I
        at Program.main(Program.java:2)
```

Listing 1.6: Run-time error in program after library change

```
Program.java:2: foo(int,int) in Library cannot be applied to (int)
  public static void main( String[] args ) { System.out.println( Library.foo( 42 ) ); }
                                                                      ^
```

Listing 1.7: Compile-time error in program after library change

Had the entire system including the `Program` class been recompiled, the message in listing 1.7 stating that the set of arguments given is not applicable to the current signature would be displayed at compile-time, and the error would have been detected prior to running the program.

However, it is debatable whether the fault is with the main program or if rather the new version of the library is to blame. At the time the class `Program` was compiled, it was correct and has not changed since. It was the change in the library class that provoked the error.

This corresponds to a document being invalidated by a change in the schema without any attempt to maintain integrity of the repository. If reverse dependencies was kept on the other hand, a smart editor would be able to alert the developer making the change to the library that this action would cause problems in the main program and that appropriate actions should be taken before being allowed to proceed.

1.1.4 Applicability

The problem described in the previous sections has been observed in practice by the author during work with the current generation of the HyperEducator system. The engine — called SOFU — maintains a repository of both schemata and documents, but does not provide for consistency checks between these two types of entities.

The strength of HyperEducator is the structured approach it takes to content management, supporting reuse and maintainability, and the author reckons that schema consistency checks will fit nicely into and add further to the values emphasized by this system, so the solution worked out through this thesis should be possible to integrate into the next version of HyperEducator without major obstacles.

1.2 Roadmap

A number of possible approaches can be taken to ensure integrity in the repository, and this section will indicate the direction chosen by the author for this thesis along with pointers to the areas to be treated in later chapters.

1.2.1 Alternatives

The diligent reader will perhaps already have identified that there are at least three principal ways to augment the system in figure 1.1 on page 16 to achieve the functionality of the one in figure 1.2 on page 17 upon entrance of new documents:

(A) Revalidate all existing documents against the new schema.

(B) Let existing documents use the old, and future documents use the new schema.

(C) Compare the new schema to the old schema, looking for conflicts.

Alternative (A) uses *brute force* to detect any conflicts, and although it will eventually provide the right result, this method is potentially very time-consuming and the author challenges the proposition that this is an *efficient* way to ensure consistency. In a real world scenario, the repository may contain a large number of documents while the change in the schema will typically only affect a relatively small part of them.

Furthermore, revalidating all documents makes incremental checking of the document while authoring a less viable option, since there is no information about the dependencies to the part being currently edited in particular, and the processor will have to perform all the work over again every time a change is made.

There are two fundamental weaknesses making solution (B), which exercises *version control* in the repository, less compelling than it may initially seem.

First, if new stylesheets are added, a version must be written for both the old *and* the new version of the schema as both may now exist in the repository and must be handled appropriately, and this increases development time. The amount of old stylesheets is however not an issue, as these must be ported to the new schema in any case.

Second, an administrative burden is put on the maintainer of the repository who must manually decide to submit changes to documents under the old schema or under the new schema, on a case-by-case basis since neither option is a natural candidate as the default. If always submitted under the old version of the schema, new features cannot be used, whereas submitting under the new version may yield the document invalid as it was not taken in consideration when accepting the schema.

Granted, both of these problems are solvable. Yet the thesis will not choose to explore this path further, but rather focus on option (C) instead due to its promise with respect to supporting data evolution over time.

1.2.2 Formal description

The purpose of this section is to give a more mathematically inclined formulation of the problem to facilitate identifying potentially helpful literature on the subject.

Let G be a grammar and $L(G)$ the language that this grammar generates, i.e. $L(G)$ is the set of sentences that are valid within this grammar. Let V be a set of documents that is in accordance to the grammar, i.e. $V \subseteq L(G)$. An illustration of this is given in figure 1.8.

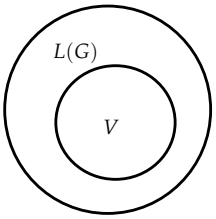


Figure 1.8: Valid documents are a subset of the language

Consider then a change in the grammar from G to the new grammar G'. The set V may now only partially also be a subset of $L(G')$ depending on how large the intersection between these two grammars is. Let the largest subset of V that is also a member of $L(G')$ be called V', i.e. $V' = \{v | v \in V \wedge v \in L(G \cap G')\}$. Let the subset of V that is *not* a part of V be called V^*, i.e. $V^* = V - V'$. All of this is shown in figure 1.9 on the next page.

The task is then to given the change from G to G' in the grammar to determine the set of documents V^* that is incompatible with it, i.e. the set of documents that was dependent on the part $G^* = G - G'$ of the old grammar.

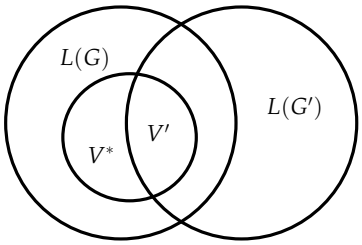


Figure 1.9: Subset of documents not in language of new grammar

1.2.3 Contribution

A literature search for a formal theory that contains a model for the intersection and difference between grammars describing markup languages reveals the existence of an active community working on the subject.

This thesis draws heavily on current research in the area of *regular tree grammars* and *subtyping*, and aims to make a contribution — albeit small — to advance the field by:

(I) Providing an implementation of regular tree grammars and subtyping in the Java host language.

(II) Finding possible ways to enhance the subtyping algorithm to only consider types in actual use.

(III) Discussing the considerations that must be taken when integrating the algorithms with a repository holding the data.

The novelty of the thesis lies in the combination of the repository and the grammar algorithms, as each of the individual components is already covered by existing work of others.

1.2.4 Outline

In order to develop a prototype and proof-of-concept for a content management system in which the algorithms for ensuring database integrity can be tested and experimented with, the following areas have been identified for exploration:

(1) Establish a formalism for describing grammars.

(2) Identify relations between grammars.

(3) Recognize the language of the grammar.

(4) Capture dependencies between documents and grammars.

(5) Store documents in a format suitable for interchange.

(6) Devise an implementation that uses the algorithms in combination with the database.

Each of these steps have been given attention corresponding roughly to a chapter in the thesis, in the listed order. Chapters 2, 3 and 4 are revolved around the foundation in formal language theory, chapter 5 is mainly about how the algorithms can be extended to provide the desired services, while chapters 6 and 7 are concerned about the practical aspect of integration with other components. The two latter chapters may be skipped if it is desirable to view the material only from a theoretical angle.

All chapters are written from the repeating formula of first introducing and discussing the theoretical aspects before moving on to extracting considerations and implementation techniques from that. Although the chapters are intended to be read sequentially, the structure should at the same time be modular enough to enable a reader with a thorough knowledge on a subject to skim or skip altogether some of them.

However, most of the text builds on results from previous chapters, creating a natural progression towards the ultimate goal. The dependencies among the chapters are illustrated in figure 1.10.

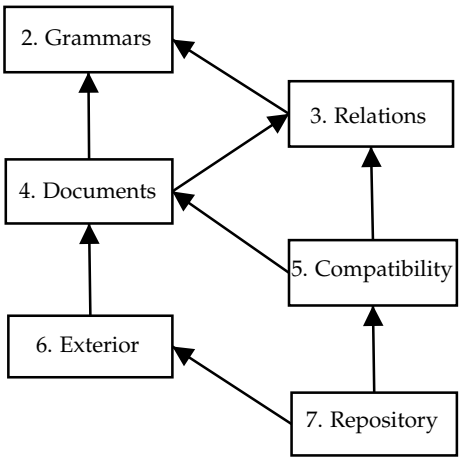


Figure 1.10: Logical dependencies between chapters

The framework developed in this thesis is written in the Java language and modeled as a library targeting systems using in the Java2 runtime environment; the availability and mindshare of the platform being a contributing factor in this decision. It requires no extensions beyond standard tools, and is intended for reuse at the class level. Other approaches have been to integrate the algorithms into the host language itself [Pie02, MS03].

Listings are presented directly in Java to avoid introducing a pseudo language for the purpose of discussing the code alone. Almost all of them are extracted from the source code, although some casts and exception handling have been omitted in some places for brevity. Comments are also removed as they are effectively replaced by the discussion in the text.

Chapter 2

Grammars

To model structured content, a theoretical framework is needed. One such framework is the theory of Regular Tree Grammars. This chapter introduces this framework and explains the rationale behind choosing to use that one in particular. The chapter then proceeds to define the constructs needed further in the thesis. It is assumed that the reader has basic knowledge of mathematical set theory.

2.1 Sequences

2.1.1 Strings

Information that is to be store must be encoded in some way. To encode documents *strings* are used. A string is a sequence of symbols from a particular *alphabet*. An alphabet is normally denoted with Σ. This thesis is only concerned about the characters from the English alphabet as content. However, there is nothing that hinders the usage of other alphabets.

Sequences can have an arbitrary, positive length including zero. A sequence with zero length simply does not have any characters in it and is called an empty string. Such a string is denoted with the special symbol *epsilon*, ϵ.

Some examples of strings are as follows. These examples are not exhaustive, but are rather meant as an illustration. The same character can occur multiple times in the string, in different positions and with other characters in between.

$$\epsilon \qquad ab$$
$$aa \qquad b$$

Strings are the foundation bricks of content, but they do not in themselves convey any meaningful information other than that they can be compared for equality and compared according to some lexicographical order.

Both of these definitions rely on similar definitions for individual characters in the alphabet. Regard the alphabet as a sequence of characters. Two instances of a character are equal if they match the character from the same position in the alphabet. A character

is said to be less than another if it occurs before in the alphabet, and greater if it occurs later.

$$a = a \qquad a < b \qquad b > a$$

Two strings are considered equal if and only if they contain the exact same number of characters and if the characters in each corresponding position are equal. A string is less than another if it contains a non-negative number of equal characters with the other followed by a lesser character or no character at all. A prefix is hence always shorter than the full string. Conversely, a string is greater than another if the other one is lesser than it according to the definition above.

$$\begin{array}{lll} \epsilon < a & a = a & a < aa \\ a < b & aa < ab & ab < ba \end{array}$$

While these definitions come in handy when inserting strings into and retrieving strings from collections, they are not helpful in categorizing strings further.

2.1.2 Languages

To structure a collection of strings, it is necessary to partition it into subcollections. This is done with a *language*. A language is a union of strings. For example a language that consists of all the strings mentioned in the first example of the previous section can be created:

$$\epsilon \quad | \quad aa \quad | \quad ab \quad | \quad b$$

In this thesis, the bar | is used to denote the union operator. An individual string may be seen as a set of strings containing one string, overloading the operator to work on both individual strings and a set of strings. The operator of course yield a set of strings as result.

A string is either in a language or it is not, dividing the space of string into two. Both of these parts may be empty, if for example the language contains none or all strings. Such an empty language is denoted with the symbol $\emptyset$.

In a language, more than one string may have the same prefix. Instead of writing out all the strings, the strings that have the same prefixes may be grouped into *sublanguages*. The suffixes of these strings constitutes a language of their own.

The notation of the main language may then be shortened by putting the prefix outside a parenthesis followed by the sublanguage that specifies the suffix inside. Applied to the example yields as follows. All the strings that begin with a have been grouped.

$$\epsilon \qquad a(a|b) \qquad b$$

A language is equal to another language if it contains the same set of strings. A partial order may be defined between languages using the notion of subsets.

2.1.3 Regular expressions

This thesis is interested in a class of languages called *regular languages*. These are languages that are defined using *regular expressions*[HMU01]. The language of such an expression E is denoted $L(E)$. Regular expressions are called a *meta-language* because it is a language used to describe other languages.

This class is interesting because languages in it can be recognized by an automaton without using a stack. The space complexity of the recognizer is then only dependent on the size of the language not of the input. In addition, there are well-known algorithms for translating a language described by a regular expression into such an automaton.

In particular, the intersection between two regular languages can be calculated by setting up the product of these two languages' automata and let the pairs that contains accepting states from both automata be accepting states in the product. The language of the resulting automaton will also be regular.

Next follows a definition of regular expressions. The reader is cautioned to note the difference between when a symbol is used as a literal in the definitions and when it is used as a semantic value.

Definition 2.1 (Regular expression) *A regular expression $RE(\Sigma)$ is defined recursively as*

- ϵ *is a regular expression with $L(\epsilon) = \{\epsilon\}$.*
- $\emptyset$ *is a regular expression with $L(\emptyset) = \emptyset$.*
- a *where a is a string from the alphabet Σ, is a regular expression with $L(a) = \{a\}$.*
- vw *where both v and w are regular expressions $RE(\Sigma)$, is a regular expression with $L(vw) = L(v)L(w)$, i.e. a cartesian concatenation of all the elements in the two sets.*
- $v|w$ *where both v and w are regular expressions $RE(\Sigma)$, is a regular expression with $L(v|w) = L(v) \cup L(w)$.*
- v^* *where v is a regular expression $RE(\Sigma)$, is a regular expression with $L(v^*) = \cup_{i \geq 0} L(v)^i$ where L^i is i instances of L concatenated to eachother. Note that $L^0 = \{\epsilon\}$ for all languages. This operator is called the* Kleene closure.
- (v) *where v is a regular expression $RE(\Sigma)$, is a regular expression with $L((v)) = L(v)$.*

No explicit operator is used for concatenation as is also the case in common text. Literal symbols from the alphabet may be read from left to right.

Parenthesises have the highest precedence due to the syntax. It is common to assign the Kleene closure operator higher precedence than the concatenation operator, and the concatenation operator higher than the union operator again. This thesis follows that convention. This means that $L(a|ab) = \{a, ab\}$, $L(aa|ab) = \{aa, ab\}$ and $L\Big((a|b)(a|b)\Big) = \{aa, ab, ba, bb\}$ but $L(a|ab|b) = \{a, ab, b\}$.

In addition, two shorthand suffix operators will be used:

v^+ is the same as (vv^*).

$v?$ is the same as $(v|\epsilon)$.

These operators are at the same precedence level as the Kleene closure, something that is indicated with the use of parenthesises in the definition.

The thesis will now proceed to look at the applicability of regular languages in structured content management systems.

2.2 Trees

2.2.1 Documents

A *tagged string* is a a string with a name attached to it. This name is called a *tag* or a *label*.

The tag is usually written in the same alphabet as is the string itself, so it is necessary to separate the two by the use of *markers*. The markers are symbols that is not a part of the original alphabet of the string. All the characters of the string that constitute the label is amalgamated into one unit. Prefixes and suffixes of the label is not interesting, only full equality to other labels.

In this thesis, the convention is to write tagged strings with the name in front of the content and use brackets ("[" and "]") as markers to signal the beginning and the end, respectively, of the string that is being tagged. An example of this is the tagged string $a[b]$ where a is the tag and b is the string being tagged.

An *element* is defined recursively as a tagged string which is not a string of characters, but a string of other elements. An element thus span out a tree, where elements that is non-empty make out the branches and empty elements make out the leafs. Elements can be written out using the notation described above, with content expanded recursively. The result would be string from the original string alphabet extended with marker characters. This thesis will also adopt a convention where the content are skipped if it only consist of the empty string. In those cases the markers are not written either. It then become ambiguous whether a string is one label consisting of several characters or several labels consisting of one character. To prevent such cases, all examples that are given in a mathematical context will use only one character. Some examples of elements are given below. The alphabet in this example are deliberately the same as in previous examples.

ϵ	$a[a]b$	$a[ab]b$
$a[a]a$	$a[ab]$	b

Parsers are software components that build the element tree from such a "flattened" string. However, this thesis will not contain a treatment of this functionality but instead refer the reader to a seminal text on the topic [ASU86].

The model can be further expanded by allowing elements where the content is a sequence of both other elements and character strings. This is called *mixed content*. If

an element with mixed content is expanded, *marked-up text* is the result. The diligent reader will have noticed that this thesis' choice of notation does not give an unambiguous way of writing such mixed text. The reason is that such mixed-in character strings is considered opaque and can be replaced by an element representing it if needed. This is indeed the technique used in later sections. The author has rather tried to make the notation shorter and more readable at the expense of expressiveness.

2.2.2 Schemata

Sequences of elements have a language, and thus the content of an element has a language too. But the language of the content need not be the same language as the one of the sequence that the element itself is in. Whenever a marker is found is an expanded element, a possibly entire new language must be recognized until the corresponding ending marker is found.

A *schema* is a union of elements. Informally, the schema makes out the language of the expanded element. As it is with prefixes and suffixes in languages, it is practical to factor out common labels and put the different content in a union. A schema that describes among other things the strings given as examples of elements in the previous section, is given below:

$$\epsilon \quad | \quad a\Big[(a|ab)\Big]\Big(a|b\Big) \quad | \quad b$$

Note that a union of sequences with a common prefix – such as $a[a|b]a$ and $a[a|b]b$ – can also be factored. The union branch operator may appear later in the sequence. An illustration of this is that the tree spanned out by the union temporarily grows together before splitting again.

Do not confuse schemata with *schema languages*. The latter are meta-languages used to describe schema. Real-world examples are DTD [BPSMM00] and XSD[TBMM01]. The focus of this chapter is to establish such a meta-language.

The find out if an element is part of a schema, an expression language is needed. It would be advantageous if an already known expression language could be employed since that would provide necessary tools as well a firm theoretical foundation.

2.2.3 The problem with regular string expressions

A recognizer for a regular expression can be built using a deterministic finite automaton [ASU86]. While these recognizers have the nice property of having a space complexity dependent only on the expression and not on the input, the same trait also limit the expressive power of this meta-language.

A deterministic finite automaton contains as the name implies a finite number of states. Assume that the number of states for a particular language is k. If given an input string with more than k symbols, one or more of the characters in the string must cause a transition to a state that has previously been visited. Otherwise, if each input character from the string gives a new state, there would obviously be more than k states,

invalidating the premises. Since each state in such a DFA is not allowed to keep any information other than what was given from the language, the state cannot know if it has been visited before or not. Hence, information has been lost when the state is reentered.

The *Pumping lemma* says that for strings that are larger than the number of states k in the automaton, a (non-empty) part of the string will be repetitive and this part will be within the first k characters. When the automaton comes to the repetitive part after say i transfers to unique states, it will "pump" it through the same state P_i before it continues with the rest of the string. This is illustrated in figure 2.2.

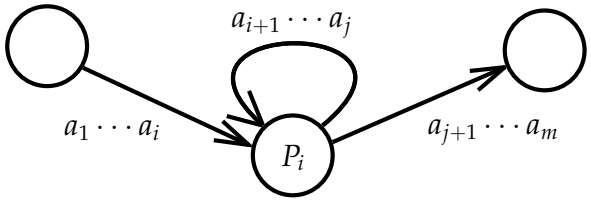


Figure 2.2: The pumping lemma

It can be shown that a language for a tagged string can be set up so that it contains a string where the start marker would be in the repetitive part whereas the end marker would not. This is possible due to the fact that the repetitive part must occur within the first k characters in the string. However, the pumping lemma states that the repetitive part could be removed and the remaining string would still be regular. Since the markers always occurs in pairs, this contradicts that the string would be in the language after all. Therefore, languages that contain this – or any other nested constructs for that matter – can not be regular. The reader is referred to [HMU01] for a proof of this assertion.

2.2.4 Context-free languages

A larger class of languages can be recognized by adding a stack to the recognizer. The result is called a *push-down automaton*. This stack gives the automaton the ability pump two substrings, as information can be stored on the stack for later retrieval. This means that nested constructs can be recognized. The class of languages that is covered by such an automaton is called *context-free languages* and a grammar that describes it a *context-free grammar*.

Most programming languages can be described by a context-free grammar and so can tagged strings. Indeed, markup is often used as an example of what can be done with recognizers based on context-free grammars.

However, a stack only allows information to be accessed from the top, and this impairs this class somewhat. Only one extra pump may be coordinated with the first, making it impossible to have more than two. Although nested constructs are recognized, repeatable constructs are not accepted.

The intersection of two nested expression where the end marker of the first is the start marker in the next would be a language where the second end marker functioned as an echo of the first. Generally, context-free grammars are not generally closed under the

intersection operator. If an algorithm was written to perform this operation, the return type could not be guaranteed to be the same as the two parameters.

That does not mean that context-free grammars cannot be used. In fact, in a marked-up string the start and end markers will always differ – by design. It might be possible to devise an algorithm that finds the intersection of two languages describing markup and treat the result as a context-free language. This algorithm may also always work. The consequence of the lack of a generalized closure property of intersection is simply that context-free grammars cannot be used as a theoretical framework to analyze and describe such an algorithm.

2.2.5 Regular tree grammars

What is needed is a class of languages that have similar powers as the context-free languages while retaining the closure properties of regular string languages. This class must employ the fact that markers always occur in pairs, and that the structure of the document is really a tree where each tagged string is a branch or a leaf as mentioned in section 2.2.1.

This class is called *regular tree languages* and consequently grammars that describe such languages are called *regular tree grammars*. This class of grammars are denoted *RTG*. Such grammars are also a schemata as defined in section 2.2.2 since they defines a sublanguage at each level. The applicability of such languages on markup is described in [Pre98].

It turns out that regular tree languages are closed under intersection much in the same way as regular string languages are [LMM00b]. Proofs of this can be found in [CDG+02].

Definition 2.3 (Regular Tree Grammar) *A regular tree grammar (RTG) is a 6-tuple* $G = RTG(L, T_E, T_C, P_E, P_C, S)$ *where*

- L *is a set with strings that will be labels. It is common to use lowercase letters for labels. Labels are* terminal symbols.
- T_E *is a set with symbols for "element types". It is common to use uppercase letters for types. Types are* non-terminal symbols.
- T_C *is a set with symbols for "complex types".*
- P_E *is a set with productions on the form* $e = l[t]$ *where* $e \in T_E$, $l \in L$ *and* $t \in \epsilon \cup T_C$. *The right side of these productions are tagged strings, as defined in section 2.2.1. These productions gives types for single elements. These are units that can be used to construct sequences of elements. Note that nothing precludes two elements to use the same label in this definition.*
- P_C *is a set with productions on the form* $c = RE(\epsilon \cup T_E)$ *where* $c \in T_C$ *and* $RE(\epsilon \cup T_E)$ *is a regular expression over the alphabet* $\epsilon \cup T_E$.
- S *is a set of start symbols for the grammar, where* $S \subseteq T_E$. *These are the possible roots of the tree.*

and there is a restriction that symbols that designate types can only appear once on the left side of the productions, i.e. $\forall e = l[t] \in P_E . \nexists e = l'[t'] \in P_E$ *and* $\forall c = RE(\epsilon \cup T_E) \in P_C . \nexists c = RE'(\epsilon \cup T_E) \in P_C$.

Although not strictly necessary it is common to restrict the usage of symbols so that a symbol can only identify a rule in either P_E or P_C but not both, i.e. $T_E \cap T_C = \emptyset$. It is assumed that neither T_E nor T_C contain unused symbols.

The roots of the tree must be single elements only. If all types (i.e. symbols from both T_E and T_C were allowed as roots, the grammar would describe a *hedge*[Mur99] instead of a tree.

Note that it is very important that only the symbols from T_E can occur on the right side of the productions in P_C. Each complex type must be "framed" by a label and a pair of markers before they can appear in other complex types. This is what makes the tree regular and not context-free.

This form of grammar is normalized. Any regular tree grammar can be converted into this form [LMM00b]. This thesis will later show the advantages of separating element types and complex types when an algebra is to be constructed for such grammars. Other forms of regular tree grammars have eliminated the notion of complex types and only operates with element types and "inline" expressions of those directly in content. The content is then said to have an *anonymous type*.

An alternative is also to implicitly assume that all symbols from T_E are "imported" into T_C with the corresponding rule $A = A$ in P_C for all such imported symbols A. Both element types and complex types can then be used in expressions.

Example 2.4 (Labels *a*, leafs *b*) $G = RTG(L, T_E, T_P, P_E, P_C, S)$ *where*

$$
\begin{aligned}
L &= \{a, b\} \\
T_E &= \{A, B\} \\
T_P &= \{A', B'\} \\
P_E &= \{A = a[A'], \\
&\qquad B = b[B']\ \} \\
P_C &= \{A' = (A|B)^+ \\
&\qquad B' = \epsilon\ \} \\
S &= \{A, B\}
\end{aligned}
$$

is a grammar that describes trees where a is the label of branches and b is the label of leafs.

The types A and B define two elements. Elements of type A may contain a variable non-zero number of elements as content, while those of type B can only have the empty string and must hence be leafs. The tree must have at least one element. The following are legal examples of markup that are in the language specified by this grammar. Recall from section 2.2.1 that the markers are not written when the content is empty.

$$
a[b] \qquad a\left[a\left[b\ a\left[a[b]\right]\right]\right] \qquad a[b\ b] \qquad b
$$

2.2.6 Subclasses

The class of regular tree grammars can be divided according to the uniqueness of labels in types. This property decides to which degree types can be inferred from the labels. This thesis will later employ this to store labels as an approximation to type names. The subclasses introduced in this section are the classes where such an approximation can be considered good.

To investigate what constitutes the uniqueness of a label in a grammar, it is necessary to define *competing types*[LMM00b].

Definition 2.5 (Competing element types) *1-lookahead competing element types in an RTG are types of the form $e = l[t], e \in T_E, l$ is a string, $t \in T_C$ that has the same label l.*

1-lookahead means that only the root element of the tree that can be spanned from the definition is inspected. By the definition of marked up text in section 2.2.1, this will also be the first label to appear in the resulting string. As this thesis is not interested in any other form of competition, 1-lookahead competing element types will simply be denoted as just "competing types".

The first subclass is one where there is a one-to-one relationship between labels and element types. This is called a *local tree grammar*.

Definition 2.6 (Local Tree Grammar) *Local tree grammars are regular tree grammars where there are no competing element types, i.e. $\forall e \in P_E.\nexists e' \in P'_E.e$ competes with e'. The class of local tree grammars is denoted LTG.*

This class is called "local" because the type of an element can be inferred from the element alone with no need of context. By looking at the label, the correct type can be decided because there is no ambiguity. When defining grammars of this class, the notion of types is not needed and labels are used as type names. The languages defined by the meta-language DTD belongs to this class, but not all local tree grammars can be modeled using DTD [LMM00b].

The second subclass is one where several types can have the same label, but they cannot appear together. This is called a *single-type tree grammar*.

Definition 2.7 (Single-type Tree Grammar) *Single-type tree grammars are regular tree grammars where there are no competing types in the same expression on the right side of a complex type production, i.e. $\forall c = RE(\epsilon \cup T_E) \in P_C.\nexists e, e' \in T_E.e, e' \in RE(\epsilon \cup T_E)$ and e competes with e', and that starting symbols should not be competing, i.e. $\forall s \in S.\nexists s' \in S.s$ competes with s'. The class of single-type grammars is denoted STG.*

Note that the constraint says that competing symbols should not occur in the expression itself, not that they shouldn't be in the language of the expression (although the first precludes the other).

The name "single-type" stems from the fact that a label can only belong to one type within a given content model. If the parent type is known, then the label in conjuncture

with this parent type will determine the element's type. Since the starting symbols are not competing, types can be determined for all elements while reading markup by the use of a stack. The languages defined by the meta-language XSD belongs to this class, but similar to DTD not all single-type tree grammars can be modeled using it.

Since a grammar with no competing types cannot contain any expression with competing types either, it follows that all local tree grammars are also single-type tree grammars. By definition, both are regular tree grammars. The relationship of these three classes is therefore that $LTG \subset STG \subset RTG$.

If competing elements exists in the intersection of two grammars, then both the grammars must have contained those elements. If the result of an intersection is not an STG, it must have been because one of the terms wasn't STG either. Hence, the class STG is closed under intersection [LMM00a].

However, when performing a union operation, elements with the same label will be competing when these two are combined. For instance the union of a grammar G_1 that prohibits something and a grammar G_2 that requires the exact same thing, will be a grammar G_3 whose language will contain trees both with it and without it. A general regular tree grammar can solve this by introducing competing types, akin to the way polymorphism can be used in object-oriented programming languages [Boo94]. The union exists, but it is not a single-type tree grammar and hence the class STG is not closed under union [LMM00a]. A similar proof applies in the case of difference.

A regular expression is said to be *ambiguous* if a string can match its components in more than one way by dividing it differently. E.g. the string *aaa* will fit into the expression a^+a^+ as both $(aa)(a)$ and $(a)(aa)$. The middle a can fit into either the first or the second closure. *1-ambiguous* is the notion that at least one symbol in strings of this language has this property, as in the example above. Regular grammars where the right-hand side of the productions in P_C are *not* 1-ambiguous belongs to the class called $TDLL(1)$. The markup meta-languages DTD and XSD are the intersection of $TDLL(1)$ and respectively LTG and STG [LMM00b].

Purely theoretically, one of the subclasses could be used as foundation, aligning the implementation with one of the existing meta-languages. Since they are closed under intersection, finding schema incompatibilites become possible. However, it is impractical to construct grammars using these classes when the other operations are not available so this thesis will use general regular tree grammars as the base for the abstract data types that are presented next.

2.3 Specification

2.3.1 Algebra

An algebra for constructing regular tree grammars will now be devised. It will be specified in the syntax used in [LEW96]. The specification, however, is used as a tool to describe the design and not as a mean in itself. It will therefore not be treated with the same rigor as is necessary in mathematical texts. Note that the underscore character, _, is

used to denote explicit placement of the parameters to operations. (Another attempt to capture the semantics of regular expressions in a similar specification is [Wil01]).

From the definition of a regular tree grammar in section 2.2.5, it seems naturally to select the sets L, T_E and T_C as carriers for the algebra. Labels are character sequences, and most programming languages have such a type in their base library. It is therefore assumed present and imported into the algebra. The exact operations of this class are not important, as long as there is a comparison operator that can be used for ordering. The specification of such an operator is omitted from this specification for brevity and assumed a part of the import. Similarly is a type for performing boolean arithmetic included.

An element in the set T_C is constructed from a regular expression of elements in T_E, and the elements in T_E are in turn constructed by putting an element of T_C between a labeled set of markers. The modeling of productions from P_C which defines the elements of T_C is first considered. The algebra will mimic the constructions of a regular expression from section 2.1.3. Instead of creating a separate carrier that holds elements from T_E, the construction of these elements is integrated in the constructor that corresponds to the single symbol in the regular expression. Only one carrier is thus needed, and this carrier is capable of expressing both complex types and element types. The latter is modeled as an implicit complex type containing exactly one element type. Since any of this types are allowed as start symbols, the algebra actually models hedges instead of trees, cf. section 2.2.5. The resulting algebra is presented in figure 2.8. The carrier T represents an element type, and each of these element types is in themselves grammars.

Operations for the parenthesis construct in regular expressions are not needed, since the use of parenthesis in the specification language itself can be used to determine the grouping of symbols.

In the definition of RTGs, it was important that complex types (i.e. elements of the set T_C) did not occur in the definition of another complex type because that would make the grammar context-free. In the specification of the algebra, the lack of recursively constructed carriers prevent this from happening. Another implication of this is that the algebra must contain an explicit operation that can do the closure, and the specification of this closure operator uses the mathematical recursive operator , μ (similar to **`val rec`** in ML), even though it strictly violates the constructiveness of the algebra since no carriers can be created that way. Thus the closure could also be considered a constructor.

In addition to creating grammar types, operations that traverse the various subelements is needed for the algorithm that determines subtyping. To do this, the operation $\leq$ is introduced to create an order amongst the types and the operations *car* and *cdr* is used to find the first element and the following elements of a list, respectively. The specification is considered partial because the ordering of elements are incomplete from the rules that have been set up here. These rules rather are meant to illustrate the intention behind the operation.

$RTG =$	partial constructive spec	
	import $String, Boolean$	
	carrier $Type$	
	ops	
	ctor $\emptyset :\rightarrow Type$	empty set
	ctor $\epsilon :\rightarrow Type$	empty value
	ctor $\_[\_] : String \times Type \rightarrow Type$	label value
	ctor $\_,\_ : Type \times Type \rightarrow Type$	concatenation
	ctor $\_\|\_ : Type \times Type \rightarrow Type$	union
	$\_^* : Type \rightarrow Type$	Kleene closure
	$\_^+ : Type \rightarrow Type$	positive closure
	$\_? : Type \rightarrow Type$	optional
	$\_ \leq \_ : Type \times Type \rightarrow Boolean$	ordering
	$car : Type \rightarrow Type$	first
	$cdr : Type \rightarrow Type$	follow
	subject to	
	$\emptyset, x = \emptyset$	(I)
	$x, \emptyset = \emptyset$	(II)
	$\epsilon, x = x$	(III)
	$x, \epsilon = x$	(IV)
	$\emptyset \| x = x$	(V)
	$x \| \emptyset = x$	(VI)
	$x \| x = x$	(VII)
	$(x \| y) \| z = x \| (y \| z)$	(VIII)
	$x \| y = y \| x$	(IX)
	$(x, y), z = x, (y, z)$	(X)
	$(x \| y), z = (x, z) \| (y, z)$	(XI)
	$x^* = \mu y.(x, y) \| \epsilon$	(XII)
	$x^+ = \mu y.x, (y \| \epsilon)$	(XIII)
	$x? = x \| \epsilon$	(XIV)
	$\emptyset \leq \epsilon \leq l[x] \leq y \| z = true$	(XV)
	$s_1 \leq s_2 = true \Rightarrow s_1[x] \leq s_2[y] = true$	(XVI)
	$x \leq y = true \wedge y \leq z = true \Rightarrow x \leq z = true$	(XVII)
	$z \neq x \| y \Rightarrow car(z) = z$	(XVIII)
	$z \neq x \| y \Rightarrow cdr(z) = \emptyset$	(XIX)
	$z = x \| y \wedge x \leq y = true \Rightarrow car(z) = x$	(XX)
	$z = x \| y \wedge x \leq y = true \Rightarrow cdr(z) = y$	(XXI)

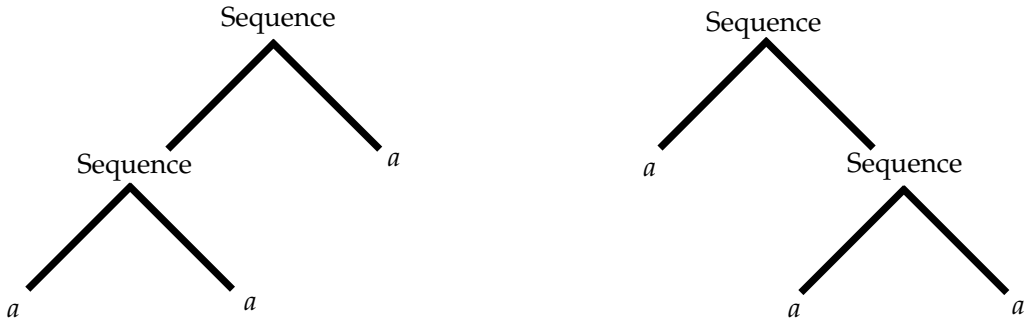
Figure 2.8: Specification for regular tree grammars

2.3.2 Design

To model this algebra in code, this thesis follows the design pattern that each carrier is an abstract class and each constructor defines a concrete subclass of this superclass. Operations whose first parameter is a carrier of the algebra, is turned into methods of the class for that carrier. Other operations are turned into class-wide methods.

A starting point is to model schemata the same way as they are written in the gram-

mar. Both the concatenation and the union operator would give a list of its operands, possibly implemented as a tree that would be traversed in infix order. However, this format is not very practical to work with since many isomorphic variants of the same grammar can be created. Take for instance the grammar a, a, a which can be created using the algebra as either $(a, a), a$ and $a, (a, a)$. Using a model as sketched above, this will generate the two trees in figure 2.9. Both of these two trees are *isomorphic*, i.e. they recognize the same language. Any equality relation now has the burden to recognize this and many other more elaborate constructs.



(a) Left-heavy (b) Right-heavy

Figure 2.9: Isomorphic element trees

Instead, a two-tier solution is employed. Unions are at the first level, and sequences at the second. The assignments to levels are consistent with the precedence between these two operators, as mentioned in section 2.1.3. The operator with the highest precedence is at the lowest level, and is grouped first. Each level keeps a list on which elements are combined using the same operation. Operations with arguments that is of a higher level is rewritten so that the operation is distributed over all the higher-level elements. This is done by rule (XI) in the algebra specification.

Each of the levels are coded as lists. The constants $\emptyset$ and ϵ are end-of-list sentinels that denotes the termination of the list. These constants are the unit element for the union and concatenation operation, respectively, as indicated by rules (VI) and (III). A single element may therefore be regarded as either a union with the empty set, or a concatenation with the empty value. This dual nature is reflected in the design, as the abstract type carrier can act as both a set and as an element by letting the operations change the concrete class polymorphically as needed.

Instead of translating between an "external" representation that is basically a parse tree of the terms of the algebra into an "internal" representation where the two-tier approach is used, this thesis will bring to the table an implementation where the translation is done implicitly in the construction. The advantage of this is that there is only one model to relate to. Note in this respect that both the union and the concatenation operators are constructors in the algebra, and indeed that is where this translation will take place.

These lists may be seen as trees which is rewritten to always be right-heavy, i.e. it is always the right operand that contains a non-basic type for this level.

An element is a set that contains itself. Informally, an expression may be seen as a zig-zag matrix where the rows are in union with each other and the columns are in sequence. To say that a particular sequence matches with this expression requires that there is at least one row where the columns (from left to right) matches this sequence exactly. Since unions are on a higher level than sequences, the matching of a sequence across columns is bounded to elements of the same row. The content model of each element is matched recursively.

Consider the example depicted in figure 2.10(a). This matrix corresponds to the expression $E_1 = a\Big(a|b\Big)\Big|b$ which gives $L(E_1) = \{aa, ab, b\}$. In this example, each possible sequence of elements have its own row (although that will not be the case generally, if closures are introduced).

(a) Without closure (b) With closure

Figure 2.10: Types as zig-zag matrices

To implement closures a loop must be introduced in the type carrier such that the type becomes reentrant.

This is perhaps best illustrated with an example. Take the expression $E_2 = a^*b$ which gives $L(E_2) = \{b, ab, aab, aaab, \ldots\}$. To derive the zigzag matrix for this type, rule (XII) is employed to expand the closure twice, and rule (XI) backwards to put the last term within the parenthesis. Notice the change of the bound variable from x to y as the part of the expression that is recursive changes.

$$\begin{aligned} a^*b &= (\mu x.ax|\epsilon)b \\ &= \Big(a(\mu x.ax|\epsilon)\Big|\epsilon\Big)b \\ &= a(\mu x.ax|\epsilon)b\Big|\epsilon b \\ &= \mu y.ay|b \end{aligned}$$

The last equation gives the matrix in figure 2.10(b). After an a has been matched, the next cell contains a pointer back to the very same expression meaning that the rest

of the sequence should be matched against this as well. In order to create such self-referencing loops before the type is fully constructed, a helper class is introduced that will later resolve the pointer. This helper class models an assignment to the internal pointers. Instead of exposing this assignment to clients directly, it is encapsulated in this class to allow it to be done once and only exactly once, making a missing assignment or a re-assignment illegal. The functional flavor of the algebra is hence preserved.

The resulting implementation class hierarchy derived from the algebra is shown in figure 2.11. *Leaf* and *EmptySet* are the constants with the corresponding name. The name *Leaf* reflects that this would be a leaf in the syntax tree of the grammar. *Label* is the class that models sequences of one or more elements (sequences that will always be prefixed by a label). *Union* models a union of two or more elements. An element may be regarded as a union in itself, but it will then always be represented by one of the other classes, never of *Union*. *Ref* is a reference to a type that has not yet been created and will be used to implement closures.

Although the sequence and union operations are denoted as constructors, there are no classes that directly models the result of these operations, cf. the discussion above. Note that these constructors takes a type carrier as their first parameter. They are rather modeled as operations based on one of the other classes that generates new objects based on the rules of the algebra. It is not so that being a constructor alone determines that there should be a corresponding concrete class. They are however constructors nonetheless since they generate object values that is not possible to obtain using any other operation.

The operations that are common to all concrete classes in this model is hence $\_$, $\_$, $\_|\_$, *car*, *cdr*, $\leq$ and $\_^*$ ($\_^+$ and $\_?$ can be implemented in terms of $\_^*$). These operations will now be discussed in terms of each of these concrete classes.

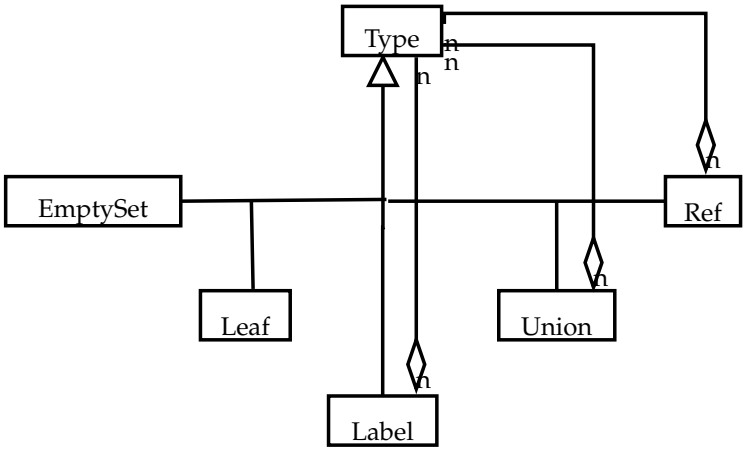


Figure 2.11: Class hierarchy

2.4 Implementation

A skeleton for the base class for the algebra is displayed in listing 2.12. The methods have been given names that is compatible with the Java syntax: *oe* and *eps* are the names of the $\emptyset$ and ϵ constructors respectively, while *label* is used to construct a new element type. *concat* creates a sequence of the object and the parameter and *union* creates a set of those two. The closure operations have all been reduced to a single method *decorate* whose name reflect that the expression is being "decorated" with a *, a $^+$ or a ? suffix.

```
public abstract class Type implements java.lang.Comparable {
  public static Type oe() {}
  public static Type eps() {}
  public static Type label( String tag, Type content ) {}
  public abstract Type concat( Type t );
  public Type union( Type t ) {}
  public Type decorate( boolean allowZero, boolean allowMore ) {}
  public int compareTo( Object o ) {}
  public Type car() {}
  public Type cdr() {}
}
```

Listing 2.12: Skeleton of Type.java

The ordering operator $\leq$ have been replaced with the standard Java method *compareTo*. The class notes to the environment that it is supports this method by implementing the marker interface *Comparable*. By doing this, the method promises to accept any object for comparison and the implementation will have to check that the object really is within the hierarchy of the *Type* class. The methods *car* and *cdr* keeps their name from the algebra.

Most of these methods can be implemented entirely, or at least give a default implementation, in the base class. The only method that does not have a natural default implementation is the *concat* method which is therefore declared abstract.

Binary operations in the algebra are defined in terms of the class of both its operands. In order to implement these in a language that does not support *multiple dispatch* [ADL91], type-testing is employed using the **`instanceof`** operator. This gives code in style of the *pattern matching* commonly found in programs written in ML [Pau91]. An alternative would have been to use the Acyclic Visitor pattern [Mar96]. While more "pure" object-oriented, this approach is not chosen on the grounds that it would complicate the code without giving any significant benefits since the type hierarchy is relatively static.

2.4.1 Empty set and empty string

Both *oe* and *eps* are functions that does not take any parameter. Neither does any of these functions have any side-effect. Therefore, the result must always be the same. The return value of these methods are static members that have the type (in a Java sense) *EmptySet* and *Leaf* respectively. Although the immutability of these classes ensure that these members could have been exposed directly, they are encapsulated in each their method so that later implementations may change. This implementation is a realization of the Singleton pattern [GHJV95]. The code for these methods is displayed in listing 2.13.

```
// class Type
static final Type OE = new EmptySet();
public static Type oe() { return OE; }
static final Type EPS = new Leaf();
public static Type eps() { return EPS; }
```

Listing 2.13: oe and eps methods

The classes *EmptySet* and *Leaf* can for the moment be assumed to be empty since they don't hold any state, only behavior.

2.4.2 Labeled elements

The *label* method provide the functionality to create a new labeled element. Recall from section 2.3.2 that not only is an element a set containing itself, but it will also be a sequence containing itself. Instead of coding the sequence explicitly, it will be integrated in the elements by the use of a "next" pointer. The class *Leaf* does not need such a pointer since it will always terminate the list. In addition to the next pointer, it will need members to hold the state associated with a single element, namely the tag of the element and its content type. The data members of the class *Label* is shown in listing 2.14.

The class is designed to be immutable, so only an readable accessor declared for *tag* and *content* so that this state may be inspected by client code. The naming of these "properties" have been chosen to mimic the name of the fields rather than the standard JavaBeans convention of prefixing the method name with *get−*. The member *next* does not have a corresponding accessor method as it is not desirable to expose the implementation of the sequence as a linked list.

Since only the constructor is able to set the semantic state of these objects once and for all, an evaluation of whether the fields could be declared **`final`** is in order. However, for reasons that will be explained in the next chapter (section 3.3.2), the two members that has the type *Type* can not formally be declared **`final`** (as a digression, they could have been declared **`mutable`** in C++).

```
class Label extends Type {
  final String tag;
  /* final */ Type content;
  /* final */ Type next;
  public String tag() { return tag; }
  public Type content() { return content; }
}
```

Listing 2.14: Data members of the Label class

The method *label* in the base class Type is only a wrapper for the constructor of the *Label* class. The parameter is sent forward as-is after a check for a null pointer. The null pointer does not have any semantics in the algebra, and no object of type *Type* should ever legally be **`null`**. In listing 2.15, the constructor is listed. The first form is the one

that is called by the method in the base class, and all it does is to create a single sequence consisting of this element alone. Note that the list is terminated with an ϵ.

The role of the constructor is simply to initialize the data members; no other behavior is done here. The label is *interned* so that it is replaced with its canonical representation. This ensures that semantic equality is aligned with reference equality, and comparisons after this point can be done in constant order relative to the length of the string.

```
// class Label
Label( String tag, Type content, Type next ) {
  this.tag = tag == null ? null : tag.intern();
  this.content = content;
  this.next = next;
}
```

Listing 2.15: Label constructor

2.4.3 Ordering types

Before unions can be discussed, an order must be introduced amongst types. A sequence has an implicit order from its composition, while in a union this is not the case. Hence an explicit order must be introduced so that the union can be kept sorted internally. If the list of union elements are always kept in this order, then the work is done upon construction and comparisons can be made more quickly. In a schema comparison, the schemata are only constructed once but may be compared several times, so this will amount to time savings.

Type carriers may be of very different classes so an artificial *rank* is introduced to make proverbial apples comparable to proverbial oranges. If the carriers are of a different class, the rank determines the ordering. The ranks are assigned as laid out in table 2.16.

Leaf	-1
EmptySet, *Label*, *Ref*	0
Union	1

Table 2.16: Ranks of type carrier classes

The rank has its origin in the cardinality of the union, i.e. it is vital to put all single element before composite elements. Since empty sets won't occur in the union (which by definition is two elements or more), it is lumped together with the rest. Leafs are made to sort below everything else so that they will be put at the front of the union. This has two advantages: First, if another leaf is inserted into the union, it can very quickly be detected whether it is already there. The probability of the same labeled element to be inserted twice is much lower. Second, having leafs first give better readable print-outs.

Furthermore, this thesis will use the term *basic type* for unions of a single sequence, and *non-basic* for unions with a cardinality different from exactly one.

The comparison operator may be coded as in listing 2.17. The method *rank* is a helper method that returns the rank, while *diff* is a helper method that simply returns the arith-

metic difference between two object pointers. This arithmetic difference is used to determine the order between two carriers with the same rank but different class. The comparison between two such objects may not matter, but it must always be consistent (within the same run at least). It is also used to determine an order between objects in the *Type* hierarchy and outside of it, in case any carriers are put in a heterogeneous collection.

Note that the result of a comparison method is always cached in a local variable to avoid making the same calls again later if the value should be returned. Although the value cannot have changed, considerable work may have been done which would be superfluous to do again. (This construct is akin to the **`let`** statement found in ML).

```
// class Type
public int compareTo( Other o ) {
  Type t = (Type) o;
  if( this.rank() != t.rank() )
    return this.rank() - t.rank();
  if( this.getClass() != t.getClass() )
    return diff( this, t );
  else
    return compareToSameClass( t );
}
```

Listing 2.17: compareTo method

If two carriers are of the same class, the class may define a more detailed ordering within itself. The role of the *compareToSameClass* is to determine this ordering. Since this method is virtual, it can only have the general *Type* as a parameter, but the various subclasses will assume that it is only called with an object of their own type. An example of an implementation of this method is the one for the *Label* carrier, as displayed in listing 2.18.

```
// class Label
int compareToSameClass( Type t ) {
  Label other = (Label)t;
  int c;
  if( ( c = tag.compareTo( other.tag ) ) != 0 )
    return c;
  else if( ( c = diff( content, other.content ) ) != 0 )
    return c;
  else if( ( c = diff( next, other.next ) ) != 0 )
    return c;
}
```

Listing 2.18: compareToSameClass for Label

The *compareToSameClass* method in *Label* uses the tag to further order the labels. If the labels are equal, it uses the object identity of the content and the rest of the sequence to determine the ordering. If both of these are the same object, then the two *Label*s must have been constructed from the same terms.

The opposite is not true, however. Not all objects that are created from semantically equal terms will be considered equal using the *compareTo* method. The structural comparison is only one level deep. If a call to *compareTo* was made on the *Type* components,

the result would be a recursion "pit" on a closure like the one in figure 2.10(b). An equality relation that does not have this problem is a topic in the next chapter. The conclusion that can be drawn at this point is that the *compareTo* method can be used for ordering, but not for testing equality.

2.4.4 Element unions

The _|_ operator may create a *Union* type if any sequence not already existing in the union is added, i.e. cases not covered by rules (VI) and (VII) in the algebra. A *Union* will thus consist of at least two non-empty sequences. As with the sequences, the elements of a union will be kept in a linked list. The list is built recursively so that the first element will always contain the base case, i.e. a single sequence, while the rest of the union is either a single sequence too or another union. Operations that is to be performed on the entire union can then be performed in an inductive manner. An artificial hierarchy could be used to introduce a difference between basic and non-basic type carriers, but this thesis has instead chosen to insert debug assertions to perform these checks, as only code internal to the package will set up the data structures. It is therefore not necessary to employ the type system to prevent the user from performing such a mistake.

```
class Union extends Type {
  final Type head;
  final Type body;
}
```

Listing 2.19: Union data members

The skeleton of the *Union* class is displayed in listing 2.19. Both fields are marked **`final`** as this structure is to be immutable. The disadvantage of having the list immutable is that insertions will have to create an entire new chain of elements. The old chain will be a candidate for garbage collection. As the cost of garbage collection is held to be proportional to the size of the memory being allocated [Boe95] and each element on the list is of constant size, this should not affect the time complexity of the insertion.

A helper method is needed to manage insertion of new elements. Due to the class' immutability, there is no need for any corresponding extraction operation. Use of the word "insertion" is actually somewhat wrong, because what happens is really that a new union that also contains the extra element is created. Code for this operation is displayed in listing 2.20.

First, an `assert`ion is made that only basic elements are inserted. Other methods will take care of the case where two unions are to be merged, as will later be shown. The right position of the new element is determined next. If it is order lesser than the existing head, it becomes the new head and the existing head is sent to the tail of the union. Otherwise, it must either be inserted into the tail, which is done recursively, or else in the case of the tail being a single element put in a union with the existing tail. The sort order between the existing tail and the new element is decided inline to make the constructor easier, although this job could be deferred to a sorting constructor.

```
// class Union
Union insert( Type t ) {
  assert !(t instanceof Union) && !(t instanceof EmptySet);
  int comp = head.compareTo( t );
  if( comp == 0 ) return this;
  if( comp  > 0 ) return new Union( t, this, true );
  if( comp  < 0 ) {
    if( body instanceof Union )
      return new Union( head, ((Union) body).insert( t ), true );
    else {
      comp = body.compareTo( t );
      if( comp == 0 ) return this;
      if( comp  > 0 ) return new Union( head, new Union( t, body, true ), true );
      if( comp  < 0 ) return new Union( head, new Union( body, t, true ), true );
    }
  }
}
```

Listing 2.20: insert method in Union

The third argument to the constructor instructs it to accept the parameters in their given position, i.e. the first parameter is the head and the second is the tail. This overload is not available to outside code.

2.4.5 List enumeration

To process a list, a scheme for enumerating all its elements is needed. This thesis employs the pattern of dequeuing the first element, treating the remainder of the list as a new list. In this context, operations for extracting the head and the tail of the list is defined. These operations are named *car* and *cdr*, respectively [Hol02]. An adapter can easily be created for the `java.util.Iterator` interface if desired.

Implementing these operations for a *Union* is trivial, since the unions by construction already keep the basic element that is the head in its own field and the insertion operator always ensures that the greater elements are propagated to the tail, keeping the least one at the front. Thus, these operations are reduced to simple accessors as shown in listing 2.21.

```
// class Union
Type car() { return head; }
Type cdr() { return body; }
```

Listing 2.21: car and cdr methods for Union

Note that when a basic element remains in the head, the class of the return value of the method *cdr* will no longer be *Union*. All the other classes simply return themselves as the head element of their implicit union and the empty set as their tail. This also works for the empty set itself, signaling the termination of the enumeration. The default implementation for all other classes than *Union* is displayed in listing 2.22.

The helper method *isEmpty* is defined to test if the class of the type carrier is the EmptySet, to avoid exposure of the implementation hierarchy. All ingredients is now

```
// class Type
Type car() { return this; }
Type cdr() { return Type.OE; }
```

Listing 2.22: car and cdr methods for Type

present to create a template of enumerations of union elements. This skeleton is presented in listing 2.23.

```
for( Type x = ···; !x.isEmpty(); x = x.cdr() )
  ··· x.car() ···
```

Listing 2.23: Template for foreach construct

The variable x local to the loop both holds the current value through the *car* method and acts as a counter that can be incremented through the *cdr* method.

Only the first tier of list elements are enumerable. A general enumeration of sequence elements is possible, but is not considered to be of any practical value to this thesis. An example to illuminate this is presented. Consider a union composed of two sequences, both of which have unions in their sequence list:

$$\Big(a, t|u\Big) \Big| \Big(b, x|y\Big)$$

A possible implementation of the hypothetical operations *sequence-car* and *sequence-cdr* is to extract elements from each of the sequences in the union, and then return the union of these elements again. Thus, *sequence-car* would return $a|b$ and *sequence-cdr* would return $t|u|x|y$ for the example above. However, the algorithms covered here has no use for these constructs.

Instead, this thesis will use the internal *next* field in the cases where the type carrier is known to be a single non-empty sequence, i.e. an instance of the class *Label*.

2.4.6 Union operation

The union operator joins two elements in a common set. This will usually cause the set to grow, but there are two exceptions. The empty set is the unit value for this operation, and adding it is an idempotent action with no effect. This is specified in rule (V) and (VI) from the algebra specification. The other exception is that making the union joining itself should also give the same set, i.e. the implicit set a single element. Rule (VII) governs this.

The default implementation of the *union* method is given in listing 2.24. It checks first for the two rules outlined above. If none of these apply, a new union is created. The second argument to the constructor instructs it to sort the arguments before creating the union, using the *compareTo* method, as was the case in the *insert* method covered in the previous section.

```
// class Type
public Type union( Type t ) {
  if( t instanceof EmptySet )
    return this;
  if( this.compareTo( t ) == 0 )
    return this;
  else
    return new Union( this, t, false );
}
```

Listing 2.24: Default implementation for union operator

To prevent empty sets being sent as an element to the *Union* constructor, the *union* method is overloaded for the *EmptySet* class. This is necessary since the *Union* constructor is obligated to create a *Union* object and cannot change the class if it determines that there is not enough basic elements amongst its parameters after all. The overloaded operator is listed in figure 2.25. It simply removes the empty set, returning the parameter as the new union. Recall that a single element has a dual role as a union containing itself. Observe that the operation works correctly, even if the union between two empty sets are attempted.

```
// class EmptySet
public Type union( Type t ) { return t; }
```

Listing 2.25: Union operator for an empty set

The job of merging two unions, can either be put off to the constructor of the *Union* class, or it can be handled by also overloading the *union* method in the *Union* class. The latter approach is chosen in this thesis, as it simplifies the constructor of the *Union* class as it can now assume that at least one of its arguments is a basic element. (If two non-basic elements is to be joined, one of the overloaded versions of the method is used).

```
// class Union
public Type union( Type t ) {
  Union u = this;
  for( Type x = t; !x.isEmpty(); x = x.cdr() )
    u = u.insert( x.car() );
  return u;
}
```

Listing 2.26: Merging two unions

The listing 2.26 shows how the merging of two unions is performed. Each element is simply *insert*ed into the existing union. The *insert* method will make sure that the elements are kept in order. This also works for empty sets and basic elements, as the enumeration will return zero and one elements, respectively.

2.4.7 Closures

Creation of closures require that type carriers can be referenced before they are fully defined. One way to do this is to create mutable structures which are initialized to some default value (commonly, **null**) and then corrected when the reference to the actual carrier is known. The downside of this approach is that it is hard to reason about whether the program is free from side-effects, due to this mutability.

Therefore, it is desirable to limit the span of the mutability to an explicit type carrier. Objects of this class works as a placeholder until the real target is known. The class *Ref* defines such placeholders for the type carrier from the algebra used in this thesis. An outline of this class is shown in listing 2.27.

```
class Ref extends Type {
  Type target;
  Type deref() {
    assert target != null;
    return target.deref();
  }
  public void assign( Type t ) {
    assert target == null;
    target = t;
  }
}

  // class Type
  Type deref() { return this; }
```

Listing 2.27: Backpatches

Note that *Ref* is not a public class. Like all the other type carrier classes, the client code will only make use of this through operations on the interface of the generic *Type* superclass.

Since *Ref* is a part of the *Type* hierarchy, it can be used as any other type. However, its use are in nature meant to be temporary. All operations could be forwarded to the target once the target was defined, creating a level of indirection. Instead, an operation *deref* is introduced to remove this extra layer. The default semantics of *deref* is simply to return itself, making the return value idempotent if called on objects from any other class than *Ref*.

For references, *deref* returns the target. The target itself is also *deref*ed, removing any number of layers of forward references that has been created. Hence, the return value of *deref* will never be a forward reference. Thus, the span of assignability to the reference is limited from its creation up to the point where *deref* is called.

References receive their target through the method *assign*. If it was possible to specify the target upon construction of the reference, the target itself could be used instead making the entire point of having a forward reference moot. It is only meant to defer having to know the actual reference to a later time, not to provide the ability to change an already initialize pointer. This is ensured by having checks that the target is assigned once (before it is used) and only once (upon assignment).

The process of allowing use of a symbol before it is defined and then completing its information at a later stage when this become known, is similar to the *backpatching* that is done in single-pass compilers.

The Kleene and positive closure operations ($\_^*$ and $\_^+$, respectively) are combined with the optional operation ($\_?$) to form a common method that controls the occurrence constraint of the type carrier. This property has two dimensions which are orthogonal; the symbol can be dropped, or it can be repeated, and all of these operations "decorate" the symbol with a marker accordingly. Hence a the common method is called *decorate* and takes two parameters that regulate whether the minimum number of occurrences is zero or one, and whether the maximum number of occurrences is one or many. The mapping between the decorations and the parameters is shown in table 2.28 and the method itself is defined in listing 2.29.

allowZero \ allowMore	**false**	**true**
false	X	X^+
true	$X?$	X^*

Table 2.28: Decorations and occurrence constraints

Allowing to specifying the two attributes of the constraint separately, is in line with the way this is done in XSD [TBMM01]. The advantage of this mapping will become appearent in later chapters where schemata will be read from this format.

An extension is to allow the schema to specify not only from the predefined value sets $\{0, 1\}$ and $\{1, \infty\}$, but to also give any exact positive integer for the attributes. This thesis does not implement this extension as it only serves to complicate the code giving no change in the expression power. Furthermore; no schemata originally drafted in DTD will require it and thus the practical use is currently limited.

```
// class Type
public Type decorate( boolean allowZero, boolean allowMore ) {
  if( !allowMore ) {
    if( !allowZero )
      return this;                                  //X
    else
      return this.union( Type.EPS );                //X=X|ε
  }
  else {
    Ref r = new Ref();
    Type t;
    if( allowZero )
      t = this.concat( r ).union( Type.EPS );       //X* = (X,X*)|ε
    else
      t = this.concat( r.union( Type.EPS ) );       //X+ = X,(X+|ε)
    r.assign( t );
    return t/*.deref()*/;
  }
}
```

Listing 2.29: Wrapping closures

In the case of decorations that allow the symbol to repeat (*allowMore* is **true**), a ref-

erence r is first created, and this reference is then subsequently used in the construction of the type carrier t. First after the type t is completely created is the reference r resolved by the use of *assign* to the very symbol that is being defined, making the closure. Upon returning the newly created type, the structure could have been "compacted" by removing the reference using $deref$. However, this is not really necessary, it does not impact the time complexity of the program and it turns out that not doing so enables the printing algorithm to better identify loops. As not only closures create references, the algorithms have to be prepared to resolve any of these as needed anyway.

2.4.8 Sequence concatenation

Concatenation appends a new element to each basic element in the set unless the element is the empty sequence, in which case the resulting set will be the the new element alone as specified by rule (III). Correspondingly, rule (IV) specifies that appending the empty sequence (of basic elements) to a set does not change it, making the empty sequence the unit value of this operator.

Thus the concatenation is distributed over all elements in the first tier set, i.e. the union set, and can be reduced to operations on the second tier which consists of the basic elements. This can be expressed as:

$$X, t = \bigcup_{x \in X} x, t$$

Again, the technique of having basic elements behave as a union in the first tier containing themselves in the second tier is employed.

Appending something to each element in an empty set yields the empty set, as there is as many elements in the set before the concatenation as there is after. Only the union operation change the number of elements in the set; the concatenation changes each element individually. It is worth mentioning that appending the empty set to an element causes the element to "collapse" into the empty set itself, as there is no longer any (marked up) strings that will match this element. Nothing matches the empty set (whereas in comparison, an empty string will match the empty sequence), so even though the prefix up to the empty set can be matched, having an empty set in the tail will always cause the matching to fail. This property is given in the rules (V) and (VI).

Since the dispatch of the virtual method that makes out the operator will end up in the class of the left argument, the rules (III) and (V) will be coded in the $Leaf$ and the $EmptySet$ classes, respectively. The implementation of these methods are shown in listing 2.30. Note that rules (IV) and (VI) are also handled by these methods in the case where the left argument is a $Leaf$ or an $EmptySet$! However, as these rules specify the semantics of the operation based upon the right argument of the operator which is passed as a parameter to the virtual method, they must must be taken into consideration in each of the other classes as well.

Concatenation to unions that consist of more than one basic element can be reduced to a distributed operation, as pointed out above. The implementation of this method is

```
// class Leaf
public Type concat( Type t ) { return t; }

// class EmptySet
public Type concat( Type t ) { return this; }
```

Listing 2.30: Trivial concatenation rules

hence simply a codification of this formula, as shown in listing 2.31. Observe that if the parameter is the empty set, this enumeration will ultimately return the empty set if all the other classes obeys rule (VI). In other words, the return type of this method is not necessarily of the class *Union*.

```
// class Union
public Type concat( Type t ) {
  Type u = Type.OE;
  for( Type x = this; !x.isEmpty(); x = x.cdr() )
    u = u.union( x.car().concat( t ) );
  return u;
}
```

Listing 2.31: Distributing concatenation over (non-basic) unions

Like the *union* method, *concat* contains a hidden use of the **new** operation (through *insert*), which will potentially create a new *Union* in each iteration. This is necessary to preserve the immutability of the class. An alternative could have been to create a union that was mutable internally to the method, but this would have greatened its complexity. All temporary unions are eligible for garbage collection and will most likely be of the 0th generation, making the cost of reclaiming the memory low [App89].

```
// class Label
static java.util.Map/*<Label,Ref>*/ backpatches = new java.util.HashMap();
public Type concat( Type t ) {
  if( t instanceof EmptySet )
    return Type.OE;                                          // (†)

  Pair p = new Pair( this, t );
  Ref r = (Ref)backpatches.get(p);
  if( r != null )
    return r;                                                // (‡)

  r = new Ref();
  backpatches.put( p, r );
  Type n = new Label( tag, content, next.concat( t ) );      // (§)
  backpatches.remove( p );
  r.assign( n );
  return n;
}
```

Listing 2.32: Concatenation of labels

What remains is a discussion of concatenation to a non-empty basic element. The implementation of this method is shown in listing 2.32. It can be split in two parts based on the parameter that is to be added. The first part is marked with (†) and handles the

case of an empty set being added to the sequence. This will, as discussed above, give the effect that no marked up strings match the sequence and rule (IV) mandates that the empty set should be returned. All other parameters will return a non-empty sequence containing this sequence as a prefix (although not necessarily a proper prefix, in the case of an empty sequence being added), and this is handled in the second part of the method (in the **`else`** clause).

Concatenation to a sequence is done by appending the right argument at the end of the sequence. Construction of an entirely new sequence is necessary to preserve the immutability of the left argument. This is done by making a copy of the head of the sequence and then letting the remainder of the list handle the appending inductively. This is an application of the rule (X) from the specification. When the base case of an empty sequence end-of-list sentinel is reached, it will replace itself with the right argument containing what is to become the rest of the list. The memory occupied by the rest of the list is hence shared with the parameter, while the memory used by the original left argument is eligible for recycling if otherwise unused.

The line marked with (§) — still in listing 2.32 — handles the sequence insertion itself, while the rest of the else clause is a guard against endless recursion. It is possible to construct the recursive expression $\mu x.a, x$; a sequence where a labeled element has itself as the following tail. The code in figure 2.33 shows how this can be done using references.

```
Ref r = new Ref();
Type x = Type.label( "a", Type.eps() ).concat( r );
r.assign( x );
```

Listing 2.33: Recursive expression using references

The resulting object graph is displayed in figure 2.34. Note that this kind of loop constitute a kind of closure resembling a "bottom-less pit"; if first encountered, it will be repeated ad infinitum.

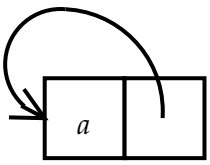


Figure 2.34: Concatenation of an element to itself

In such a construct, the expression can be replaced with itself since the number of iterations in both cases will be infinite. The code in listing 2.32 employs this rewriting. It creates a *Pair* consisting of the arguments of the concatenation operation, which is then placed on a call stack. If this pair is however detected to already be on the call stack, then this operation is deemed part of a loop (originally invoked from the line marked (§)) and should be short-circuited. To avoid synchronization, a thread-local variable may be used instead for the call-stack. Observe that concatenations to the closures created with the

decorate method will only short-circuit the repetitive parts of the expression, e.g.:

$$a^+, b = \Big(\mu x.a, (\epsilon|x)\Big), b = \mu x'.a, (b|x')$$

The diligent reader will perhaps also have interest in knowing that *Pair* uses the *compareTo* method for determining equality when searching the call-stack, which is recursion-safe as discussed in section 2.4.3.

As the result type of the concatenation is dependent on the class of the operands, it cannot yet be determined what to yield if a reference is used as the left operand. There are two ways to handle this situation. The first is to defer the concatenations to later when the target of the reference has become known. However, this require that the reference must maintain a queue of outstanding operations to be performed once it is to be resolved. The second approach is to resolve the reference before the concatenation is performed, restricting the usage of the reference.

```
// class Ref
public Type concat( Type t ) { return deref().concat( t ); }
```

Listing 2.35: Concatenation will resolve references

Choosing to defer the operation introduces more state to the object, and in order to preserve immutability each change of this state would have to result in a now clone, all of which would have to be associated with each other to coordinate the assignment of the reference. Due to this added complexity, the latter approach is the one that is chosen by this thesis.Preventing appends to forward references is however also a requirement for the type to retain its regularity [HVP00].

2.5 Further enhancements

In addition to the basic operations necessary for construction of type carriers, some auxiliary operations that ease programming can also be defined.

2.5.1 Traversal

It may be desirable to output a type carrier to a character stream for diagnostic purposes. This implicates serialization of the object graph. A naïve implementation where each object simply prints each of its fields may fall into an endless recursion, similar to the one that was described in section 2.4.8. An algorithm that detects loops and inserts symbolic links is therefore needed.

To facilitate such an algorithm, a special stream $RecursiveBuffer$ is designed. The responsibility of printing a carrier is split in two: The buffer receives a carrier to be printed, and a carrier must be able to render itself to such a stream. The loop detection is hence isolated within the buffer logic, while the appearance of each carrier is deferred to the correct type.

Listing 2.36 shows how these responsibilities are turned into code. Any object capable of writing itself onto the buffer must implement *Printable*. It may either render a field as a simple string, or as another *Printable*.

```
public interface Printable {
  void print( RecursiveBuffer buf );
}

public class RecursiveBuffer {
  public void append( String s ) {}
  public void append( Printable t ) {}
  public String toString() {}
}
```

Listing 2.36: Skeleton of recursive buffer

This yields two overloaded methods in the *RecursiveBuffer*. A simple string is added to the buffer right away, while an instance of *Printable* must be scrutinized for loops before the contents are put in the buffer. If a loop is indeed detected, then a symbolic link must be inserted instead of invoking yet another cycle. A variable is then assigned to this expression, and the designation for this variable is added to the stream. When the recursion unnests back to the first definition of this expression, then its variable declaration is also output. The implementation is similar to that of the *concat* method in *Label*, and will not be dwelled over again here.

```
  // class Label implements Printable
  public void print( RecursiveBuffer buf ) {
    buf.append( tag );       // String
    buf.append( "[" );       // String
    buf.append( content );   // Printable
    buf.append( "]," );      // String
    buf.append( next );      // Printable
  }
```

Listing 2.37: Example rendering routine

An example of the *print* method for a carrier is shown in listing 2.37. Note the similarities between this and the naïve implementation; the only difference is the buffer used. The method *toString* returns the entire buffer — including symbolic links — that has been printed to it up to this point, and can be used to output the expression on a console.

2.5.2 Typed content

Section 2.2.1 started by defining an element as a sequence of other elements. To allow not only structure of predefined tags as defined in the schema but also variable data entry, this definition was extended to include free-form text, i.e. content is a sequence of the union of both elements and characters, and section 2.2.1 named this *mixed content*. Text is allowed "in between" tags.

The meaning of these data are specific to the client application; the validator has no understanding of it as the schema only describes the structure and not the semantics. A

validator will treat the text as an opaque element and pass it on to the client.

However, the schema may be employed to specify a *datatype* of the text and the validator may check that the text is within the *value domain* of that datatype as an extra step in the validation process. Recent schema standards such as XML Schema enables the schema writer to not only specify a set of existing datatypes, but also to define new ones [BM01]. As this thesis is primarily concerned about document structure and not validation of data, mapping of free-form text to datatypes is outside its scope and will therefore be briefly discussed on a "hand-waving" basis only.

A continuous sequence of characters is modeled as an element with the specially recognized tag #PCDATA that has no further content. Instead of having this convention hard-coded into the clients, a method can be defined that encapsulates this convention. The definition of this method is shown in listing 2.38.

```
// class Type
public static Type text() { return new Label( "#PCDATA", EPS, EPS ); }
```

Listing 2.38: Free-form text elements

To extend the framework to handle datatype validation, a specialization of *Label* could be created, which aggregated the datatype specified. Upon matching an schema element to a document element, a polymorphic method in this associated datatype for validating the text further could be invoked if both elements had the tag that indicated free-form text, i.e. #PCDATA. The schema element would then contain the datatype and the document element would contain the value.

Built-in conversions is possible by creating a hierarchy amongst the datatypes, e.g. *Integer* could inherit *Double* which again could inherit *String*. When checking inclusion between two schema types, any datatype associated with a free-form text element must be castable from the "smaller" type into value domain of the "larger".

2.5.3 Wildcards — partial specification of schema

The need to embed subdocuments may arise. The outer document then specifies a structure, but another document is used to describe values instead of free-form text. The outer document is called an *envelope* and the inner document contained within it a *letter*.

A practical use for this feature is to include value data written in a presentation language such as HTML, in a structure specified by the schema.

One way of tackling this challenge is to have an import facility and use *namespaces* to keep the schemata apart. This require the schema of the inner document to be known at development time of the schema for the outer document. Another solution is to allow the outer schema to be only partially specified and allow arbitrary schemata for the inner document. The client must then resubmit the inner document to the verifier to have it checked. It is the latter solution that will be discussed here.

The basis of a template element is implemented by letting a **null** value for the tag act like a wildcard, which will match any other tag. The sorting order is altered to recognize

that a wildcard sorts before any other tag. (After any other tag would work just as well). The *any* element is a hedge consisting of a variable number of such elements again containing any hedge (hence the name). Listing 2.39 shows how this is implemented using a wildcard tag.

```
// class Type
public static Type any() {
  Ref r = new Ref();
  Type t = label( null, r ).decorate( true, true );
  r.assign( t );
  return t;
}
```

Listing 2.39: Wild-card element

Notice the resemblance between this setup and the one in section 2.4.8 where a loop was created. However, here the reference is used as content instead of as follower, making it possible for the hedge to have arbitrary depth. Since the wildcard element itself is wrapped in a Kleene closure, it may exit at any time (no pun intended).

2.6 Summary

This chapter reviewed the theory of strings, regular expression and context-free languages and discussed why these were inadequate or unsuitable for describing structured documents. It then presented regular trees as a proper framework for theoretical reasoning and introduced an algebraic specification for this grammar construct. A realization of this specification was next designed and various aspects about its implementation enlightened in order to give an understanding of the construction of element types. Provisions needed to extend the framework to not only handle structure but also value data is described.

Chapter 3

Relations

In order to reason about types it is necessary to be able to test which relations between them hold. This chapter extends the framework described in chapter 2 with the constructs necessary to determine selected relationships.

In section 2.4.3, the *compareTo* operator was defined to establish an ordering between type carriers. However, this method can only determine if two instances have the same literal content model. It cannot do a structural comparison over cyclic structures. In order to do this, a separate algebra that will recognize and keep track of the cycles is needed.

3.1 Top-down versus bottom-up

Semantic analysis of a tree can be performed in two ways: Top-down or bottom-up. The former start at the root working its way downward the branches and returning semantic information when popping the implicit stack. The latter starts out with the leafs, building and reducing an explicit stack. Often a bottom-up analysis is driven by a generic algorithm accompanied with a parsing table that holds the information about how tokens are reduced, while in the case of top-down this is hard-coded in the algorithm itself.

The approach chosen in this thesis will be that of a top-down algorithm, based upon the work found in [AM91] and [HVP00]. Schemata are already defined as tree structures and the algorithm employs these directly, avoiding the need to construct a separate parser table for each schema at run-time. (In most compilers the language is known before-hand and since a bottom-up parser table therefore can be constructed at compile-time it may hence be beneficial).

3.1.1 Alternatives

Another alternative is creating an automaton for each schema that recognize its language- and then use set-operations to compare two such automata. However, this approach has been shown to be unusable in practice as it too often causes worst-case time complexity [AM91]. Another more efficient algorithm to determine subtyping that has quadratic

complexity exists [KPS95], but it has to the authors knowledge not been adapted to untagged unions.

3.2 Rules

What constitutes a certain relationship will be defined by a set of *deduction rules*, and to determine whether a pair is in this relationship it is necessary to build a *proof "tree"* from these rules that gives a path from the expression to be checked to established axioms. Both the rules and an algorithm to pick rules to participate in the proof tree for a relation based on the type carriers to be tested will be given.

3.2.1 Caching

To avoid reevaluating a relation that have already been tested, a trick from dynamic programming called *memoization* will be used: Each time a relation is tested, the result of that test is recorded in a *cache*. When given a relation to evaluate, the records in this cache are first consulted to see if the operands have been evaluated before, and it that case the result is immediately returned (cache hit). Only a previously unseen pair of operands are put through a full proof (cache miss).

This technique is essentially helpful in testing cyclic structures. The relation to be tested is a priori assumed to be true and is added to the cache. If this pair also occurs later in the proof, the tree is not reentered but the branch is rather cut off with an immediate result. However, this presents a new problem in itself: If the proof is found to fail, the assumption does not hold. Not only must the record for this pair of operands be removed, but the records that is the result of all other subproofs based upon this assumption as well. Throwing away all results whenever a relation fails would be an overkill and counterproductive. What is needed is a concept of layering the results into generations where some results are "home-free" while others are on "probation".

3.2.2 Transactions

A concept that provides the ability to return to a previous state upon failure is the one of *transactions* [BN97]. A mark is set that indicates where the scope of the transaction should start. Thereafter, a number of elements are added to the set under the premise that their existence in it may be of temporary nature and is not yet determined. When the result value finally becomes determinable, the client can choose to either *commit* to keep all the values added to the set, or *rollback* to discard all changes and return the set to the state it had before the transaction was started.

Layering may be introduced so that a commit operation simply moves values from one generation to another that encompasses it. This is called *nested* transactions. The *root* transaction is the set of values that always hold and cannot be rolled back. All other transactions must be layered on top of this.

Transactions are normally regarded as having an imperative nature due to their use to upheld the properties atomicity, consistency, isolation and durability [BN97] (better known as ACID). For the scope of this thesis however, only the property of atomicity is interesting and a specification where the state change is modeled as functions is attainable.

A specification of an algebra for transactions is given in figure 3.1. It is parameterized on the values that can be put in the set, as these does not affect the operations of the transaction manager itself. As in the specification for regular tree grammars, it is assumed that an algebra for boolean logic is available.

$TX =$ partial constructive spec
import $Object, Boolean$
carrier $Transaction$
ops
ctor $root :\rightarrow Transaction$
ctor $\_\ add\ \_ : Transaction \times Object \rightarrow Transaction$
ctor $\_\ begin : Transaction \rightarrow Transaction$
$\_\ commit : Transaction \rightarrow Transaction$
$\_\ rollback : Transaction \rightarrow Transaction$
$\_\ lookup\ \_ : Transaction \times Object \rightarrow Boolean$
subject to
$t\ add\ x\ lookup\ x = true$ (I)
$root\ lookup\ x = false$ (II)
$t\ add\ x\ add\ y = t\ add\ y\ add\ x$ (III)
$t\ add\ y\ lookup\ x = t\ lookup\ x$ (IV)
$t\ add\ x\ commit = t\ commit\ add\ x$ (V)
$t\ add\ x\ rollback = t\ rollback$ (VI)
$t\ begin\ commit = t$ (VII)
$t\ begin\ rollback = t$ (VIII)

Figure 3.1: Specification for transactions

The carrier *Transaction* holds the state of each transaction. A further restriction may be put on this carrier that no instance is used other than once in the following operation with no change necessary in later presented algorithms and such a restriction would facilitate implementation. Operations are written in a post-fix notation and is left associative, to give the illusion of a pipe modifying the state at each point before passing it on to the next.

Naturally the operations *root* and *add* are constructors, but also *begin* must be classified as a constructor to indicate that transaction starting points are a part of its state. The result of a rollback would indeed be different if it was preceded by t than from $t\ begin$! The operation *rollback* only reverts the transaction back to a state that has been previously seen, and the carrier returned by the *commit* operation could be created be *add*ing directly to the parent transaction, so these are not constructors. The $lookup$ operation is simply an observer and does not change any state at all, something that can be inferred by the fact that it does not return a *Transaction* carrier at all.

Rules (I) and (II) says that an element must be added to the set before it can be found. The order in which they are added is irrelevant as so is what has been added other than the lookup target, states rules (III) and (IV) respectively. According to rules (V) and (VI), a value will only survive a commit and not a rollback. Note that this should not affect the presence of the value in an outer transaction, as the example term rewriting below shows. Rules (VII) and (VIII) allows inner transactions to be removed from the term once their contents has been rewritten into the outer transaction.

$$
\begin{aligned}
& root\ add\ x\ begin\ \underline{add\ x\ rollback}\ lookup\ x \\
\overset{(VI)}{=}\ & root\ add\ x\ \underline{begin\ rollback}\ lookup\ x \\
\overset{(VIII)}{=}\ & root\ \underline{add\ x\ lookup\ x} \\
\overset{(I)}{=}\ & true
\end{aligned}
$$

3.2.3 Implementation of the transaction manager

Every transaction needs to keep track of its parent transaction to which the values may be committed as well as the values themselves. This naturally leads to implementing the carrier as an object with two state members, shown in listing 3.2. The `Map` interface is reused from the language runtime library. It has operations to associate values with arbitrary keys, and then later retrieve values by those keys.

```
class Transaction {
  final Transaction parent;
  final Map locals;
}
```

Listing 3.2: Skeleton of a transaction

There are three classes of transaction carriers, all determined by the constructors; the root transaction, an empty nested transaction and transactions with elements added to them. The latter two classes can be modeled as one by realizing that a newly spawned transaction has an empty map of values. By using the convention that a **`null`** parent designates the root transaction, only one implementation class is needed to cover all algebraic classes.

New transactions are started with the constructors *root* or *begin*. Both of these need to create new object instances as they will need their own private map to add values. As an implementation detail, a private initializer that sets both state members upon allocating the instance is available as a helper method. The code of these algebraic constructor is then reduced to passing the correct initial state to this method. This scheme is shown in listing 3.3. The *root* method is **`static`** as it requires no previous transaction to work

upon (the parent argument to the initializer is **null**), while the *begin* method spawns a new transaction using an already existing one as basis.

```
// class Transaction
private Transaction( Transaction parent, Map locals ) {
  this.parent = parent;
  this.locals = locals;
}

static Transaction root() { return new Transaction( null, new WeakHashMap() ); }

Transaction begin() { return new Transaction( this, new WeakHashMap() ); }
```

Listing 3.3: Starting a new transaction

add and *lookup* takes as arguments a pair containing both a key and a value instead of just the generic *Object* from the specification. The motivation for this deviation is that the intended use of the transaction manager is to store whether there exists a relation or not. Instead of binding a boolean value to the pair of arguments and then lookup this tuple potentially twice (once for each boolean state), it is more efficient to perform the lookup directly on the pair and then return the tristate result code. This technique will be elaborated further in section 3.2.4; at this point it suffice to know that the signature of the methods have been modified to fit that use.

`WeakHashMap` is an efficient implementation of a dictionary in the host language Java. It holds *weak references* to the keys in the map, meaning that they become eligible for garbage collection if there are no other (strong) references to them even though they are still reachable from the map. In order to make a cache hit a reference to the key must be passed to the relation from the exterior, making it not only safe to use weak references but also necessary to avoid excessive caching of out-dated carriers that are no longer in scope. `WeakHashMap` uses the hash code of the key to look up the mapping, and this hash code is consistent with the *compareTo* method. An important but minute detail is that it is *Pair*s that are added to the transaction and in their *equals* method which the `WeakHashMap` uses to determine equality, they uses the *compareTo* method of their component *Type*s (see section 3.2.4 for an explanation).

Unfortunately, this component is mutable and does not support sharing of common state between instances. Consequentially, the new transaction state must create a new map initialized to be a clone of the old before adding the value to its local set. However simple the code in listing 3.4 may seem, the runtime costs associated with this copying is huge. The author has in practice found the algorithm to overall run approximately 10 times faster if the ownership of the map is rather transferred entirely to the new transaction and the old transaction is invalidated and its use prohibited, as mentioned in section 3.2.2.

Looking up values are done by checking all transactions that are pending, in opposite order of construction. The recursion terminates when either a value has been found, or the root set has been unsuccessfully checked and there is no more parents to be inspected. If a value is associated with a key more than once, only the last association is visible (unless the transaction is rolled back, of course). The same applies when a transaction is

```
// class Transaction
Transaction add( Object key, Object value )
  Map map = new WeakHashMap( locals );
  map.put( key, value );
  return new Transaction( parent, map );
}

Object lookup( Object key ) {
  Object found = locals.get( key );
  if( found == null && parent != null )
    return parent.lookup( key );
  else
    return found;
}
```

Listing 3.4: Insertion and lookup

committed: The values that are written to the parent must overwrite any already existing values for those keys. This is the semantics of the `put` method in `WeakHashMap`, so it can be employed directly without checking for this condition first. Listing 3.5 gives the implementation of the *commit* and *rollback* operations.

```
// class Transaction
Transaction commit() {
  assert parent != null;
  Transaction t = parent;
  for( Iterator i = locals.keySet().iterator(); i.hasNext(); ) {
    Object key = i.next();
    t = t.add( key, locals.get( key ) );
  }
  return t;
}

Transaction rollback() {
  assert parent != null;
  return parent;
}
```

Listing 3.5: Commit and rollback

3.2.4 Framework for a rule checker

With the aid of a transaction manager, a framework for determining relations using rules can be established. Testing a relation involves setting up a hypothesis of the form

$$\Phi \vdash T \,\square\, U$$

where Φ is a set of existing relations that is assumed a priori to hold and $\square$ is a placeholder for the relation that is claimed between T and U. Initially the set of assumption is set to $\emptyset$ at the top level. The framework will then try to prove the hypothesis by building a proof-tree from the set of induction rules that has been defined for the given relation.

Two rules are defined for all relations by the framework itself, and these are given in figure 3.6. The rule [HYP] checks whether the relation is already a part of the assump-

tions, working as a cache to avoid retesting. To determine if a set contains a given relation, it is sought for a pair whose elements are equal to the ones in the pair that is tested. It is only required that this comparison identifies whether two references are to the same instance, so the *compareTo* method may be used in this case. This observation is important, as structural equality has not been defined yet at this point. The notation in this thesis is that $\doteq$ indicates reference equality while $\simeq$ indicates structural equality to keep these two distinct.

[ASSUM] plays the part of a cycle-detector by putting the relation to be tested into the set of assumptions. A requirement that the relation is not already in the set is present to force the rule [HYP] to be used instead if that is the case, making the selection of rules at this level deterministic. It then uses a relation-specific satisfaction operator ($\vdash_\square$) to further test the relation. If the general operator ($\vdash$) was used, then the rule [HYP] would immediately cause this condition to return true in any case and the prover would be broken. By using a separate operator, the prove must go through some of the relation-specific induction rules before returning to either [HYP] or [ASSUM].

$$\frac{(t \,\square\, u) \in_{\doteq} \Phi}{\Phi \vdash t \,\square\, u} \quad [HYP]$$

$$\frac{(t \,\square\, u) \notin_{\doteq} \Phi \qquad \Phi \cup (t \,\square\, u) \vdash_\square t \,\square\, u}{\Phi \vdash t \,\square\, u} \quad [ASSUM]$$

Figure 3.6: General rules for relation framework

A superclass *Relation* is created that models the set theory concept of a relation, namely all known pairs that are in it. The method *isIn* takes two arguments and return whether these are in the relation or not. Since the code cannot possibly start out by finding all such pairs upon construction, it will have to rely on *lazy evaluation*, i.e. it proves relations only as needed.

Upon successful proof, the pair of arguments is added to the set and used as part of the assumptions the next time another pair should be evaluated. Once a relation is proved to hold, it cannot be "unproven". The opposite result may be stored too. If a relation has been found *not* to hold, that will not change either. Also, if a relation hold upon a set of assumptions, it should also hold upon a superset of those relations as well. That the client cannot override the induction rules by explicitly adding or removing pairs from the relation, is a crucial supposition for these properties to hold. The rules only assumes that Φ is a set of pairs that if encountered indicates a cycle, while in practice it will also contain a history of what has been proved.

The set of assumptions floats in a depth-first manner through the rules, similar to the evaluation of semantic attributes in an abstract syntax tree as described in [EMRS97].

An implementation for *Relation* is given in listing 3.7. Notice the use of aggregation of *Transaction* instead of inheritance due to the client not being supposed to have explicit access to the pairs that has been tested. The relation starts out with the root transaction.

```
// class Relation
Transaction currentScope = Transaction.root();

public boolean isIn( Type left, Type right ) {
  Pair pair = new Pair( left, right );                          // (1)
  Boolean found = (Boolean) currentScope.lookup( pair );
  if( found != null )
    return found.booleanValue();

  currentScope = currentScope.begin();                          // (2)
  currentScope = currentScope.add( pair, Boolean.TRUE );
  if( rules( left, right ) ) {
    currentScope = currentScope.commit();
    return true;
  }

  currentScope = currentScope.rollback();                       // (3)
  currentScope = currentScope.add( pair, Boolean.FALSE );
  return false;
}
```

Listing 3.7: Relation membership (rules [HYP] and [ASSUM])

This represents using the empty set as an initial set of assumptions.

A pair have a ternary state to the relation; proven to hold, proven not to hold and not yet determined. The wrapper class `Boolean` in the runtime library represents such a tristate with its three legal values; `TRUE`, `FALSE` and **`null`**. The work of the *isIn* method is to map this ternary state onto a binary that tells for certain if the pair is in the relation or not. If the state is already determined (either `TRUE` or `FALSE`), this is simply returned. Otherwise, the case is undetermined. It must be resolved and the set updated before the new state is returned.

The first block, marked with (1), performs the lookup and returns the value directly if it is already determined. This corresponds to the rule [HYP]. The second block, marked with (2) adds the pair to the set and then uses the method *rules* to perform the work of the relation-specific satisfaction operator ($\vdash_{\square}$). This corresponds to the rule [ASSUM]. This test must run within its own transaction so that results that stems from this assumption is not used if the hypothesis is proven not to be true, as explained in section 3.2.1. If that case, the transaction must be rolled back and the faulty assumption replaced. This happens in the block marked with (3).

By extending this class, a relation can employ the framework to easily implement the specific rules in a straight-forward manner from their description. The method *rules* delegates its responsibility to further methods based on the class of the left argument, as this turns out to be the primary factor that distinguish the rules.

3.3 Equivalence

The ordering provides a way to differentiate between carriers, and can be used to see if two references are in fact pointing to the same object. Thus, it is feasible to use the ordering to define a limited variant of equality:

Definition 3.8 (Ordering equality) *Two regular tree expressions are considered identical if they are indistinguishable for the ordering, i.e.* $T \doteq U \Leftrightarrow T \leq U \wedge U \leq T$.

However, the ordering is only partially defined and will not identify all carriers the specification rules equal as such. (Remember that the specification is only partial, and a full specification of the ordering is what is missing). This is intentional. The ordering is meant as a internal primitive operation in order to overcome the limitation of the computer architecture that it is not efficient to let a term always refer to the same object. An object is more like an avatar of a carrier. Hence, it is worthwhile to observe that the same term may give objects that is not equal according to the ordering. This introduces a discrepancy between the mathematical theory and the implementation. Different notations will be used to designate different concepts of equality to avoid confusion.

An interesting problem may be to see if the creation of types from two regular tree language expressions was done in the same manner, which is more in line with what is normally associated with equality. To do so, they must be tested for *structural equivalence*, which is defined as follows:

Definition 3.9 (Structural equivalence) *Two regular tree expressions t and u are* structurally *equivalent if their terms are equal according to the specification, i.e.* $t \simeq u \Leftrightarrow T(RTG)(t) = T(RTG)(u)$. *(Notice that the equality used here are from the specification language, not the ordering).*

Do not confuse structural equivalence with semantical equivalence, defined later in definition 3.17. Using algebraic theory for the isomorphism between two terms, the following properties can be "lifted" to hold for the structural equivalence relation also:

$$\begin{array}{rcl} \text{Reflexive} & : & t \simeq t \\ \text{Symmetric} & : & t \simeq u \quad \Rightarrow \quad u \simeq t \\ \text{Transitive} & : & t \simeq u \quad \wedge \quad u \simeq v \quad \Rightarrow \quad t \simeq v \end{array}$$

By regarding the constraints for the specification of regular tree languages defined in figure 2.8 on page 34 as a term rewriting system (since it is intended to be a constructive specification, cf. [LEW96]), the terms that are "melted" together to the same carrier by the algebra may in conjuncture with these properties (since term rewriting only goes one way) be proven to be structural equivalent. E.g. using rule (IV) it can be shown that $x \simeq x, \epsilon$ and according to rule (II), $\emptyset \simeq x, \emptyset$.

This has the nice implication that two terms deemed equal by the method *compareTo* will always be structural equivalent as well. A new rule called [TAUT] will reflect this. Since using that method is faster than employing the framework, the rule [HYP] can be extended (for the equivalence relation only) to not only check if $T \simeq U$ is in the existing set but also $U \simeq T$ as well, to catch such cases earlier. In particular, it might be observed that different instances of ϵ and $\emptyset$ has no difference in state, so the equivalence within these classes are tautologies.

Figure 3.10 list the additional deduction rules necessary for the structural equivalence relation. Note that the relation-specific satisfaction operator ($\vdash_{\simeq}$) is used in the conclusion

$$\frac{t \doteq u}{\Phi \vdash_{\simeq} t \simeq u} \quad [TAUT]$$

$$\frac{l = l' \qquad \Phi \vdash t \simeq r \qquad \Phi \vdash u \simeq s}{\Phi \vdash_{\simeq} l[t], u \simeq l'[r], s} \quad [LAB]$$

$$\frac{\forall i \,.\, \exists j \,.\, \Phi \vdash t_i \simeq r_j \qquad \forall j \,.\, \exists i \,.\, \Phi \vdash t_i \simeq r_j}{\Phi \vdash_{\simeq} \Big|_i t_i \simeq \Big|_j r_j} \quad [BIJ]$$

Figure 3.10: Rules for *structural* equivalence

of the rules, while the general satisfaction operator ($\vdash$) are used in the premises. This ensures that the set of assumptions are always updated by the rule [ASSUM] and that [HYP] will prevent reproof of an already seen branch. Only an informal explanation will be given for these rules. Do not fall into the trap of thinking that these rules will also suffice for checking *semantic* equivalence. The reason why is discussed further in section 3.3.3.

The rule [LAB] states that for two non-empty basic sequences to be structurally equivalent, both their labels and their contents must be so. The rest of the sequence is then tested inductively. The base case for this induction is when the relation between the terms is a tautology, for instance $\epsilon \simeq \epsilon$. In practice, the rule [TAUT] will use the *compareTo* method to find a broader set of terms that can be considered equal using term rewriting.

Equivalence between two unions is more tricky, since the order of the elements in the union is not significant. The unions have only to be *isomorphic* to be equivalent. A union may be considered an unordered set, and equivalence will be determined by testing for the presence of a *bijection* between two such sets, i.e. if each element in one of the sets has a correspondent element in the other set and vice versa. As the elements are not necessarily considered equal by the ordering, but may still be equivalent, all possible combinations from the two sets must be checked. (A double loop is the easiest although not the most efficient way to test this).

Observe that the rules [LAB] and [BIJ] are related to the implementation of the *compareTo* method but differs from it in that they allow recursion, cf. section 2.4.3 on page 41.

An example of the use of these deduction rules within the framework is shown in figure 3.11. Although this relation is stated as a shorthand in section 2.1.3, it is not obvious from the term rewriting system in specification alone whether it is possible to back up this statement or not. Closure of a single labeled element is used to illustrate how all the rules that makes out the structural equivalence work. To simplify the appearance of the proof, the usage of rule [ASSUM] is dropped where there is no corresponding [HYP] that relies on the relation added. Neither is the set of assumptions shown for every rule. When reading the proof, the terms x^* and x^+ could be read as variable names instead of

closures, if the reader finds the latter interpretation to inhibit the comprehension.

$$\dfrac{\dfrac{l = l \qquad \dfrac{\cdots}{\epsilon \simeq \epsilon}\,[TAUT]}{l[\epsilon], x^* \simeq x^+ \vdash_{\simeq} l[\epsilon], (\epsilon|l[\epsilon], x^*) \simeq l[\epsilon], (\epsilon|x^+)}\,[LAB]}{l[\epsilon], (\mu x^* \,.\, \epsilon|l[\epsilon], x^*) \simeq \mu x^+ \,.\, l[\epsilon], (\epsilon|x^+)}\,[ASSUM]$$

$$\dfrac{\dfrac{\cdots}{\epsilon \simeq \epsilon}\,[TAUT] \qquad \dfrac{\cdots}{l[\epsilon], x^* \simeq x^+}\,[HYP]}{\epsilon|l[\epsilon], x^* \simeq \epsilon|x^+}\,[BIJ]$$

Figure 3.11: Proof that $x, x^* \simeq x^+$ where $x = l[\epsilon]$

Creating an implementation of the equivalence relation consists mostly of overriding the methods in *Relation* that handles non-empty sequences and unions with code for the rules [LAB] and [BIJ], respectively. The first one is shown in listing 3.12 and is considered straight-forward. The reader may want to compare this with listing 2.18 on page 41 to study the similarities.

```
// class EqRel extends Relation
boolean ruleLabel( Label left, Label right ) {
  return        ( left.tag    == right.tag   )
         && isIn( left.content, right.content )
         && isIn( left.next,    right.next );
}
```

Listing 3.12: Testing structural equality between labels

```
// class EqRel extends Relation
boolean ruleUnion( Union left, Union right ) {
  return isSurjective( left, right ) && isSurjective( right, left );
}

boolean isSurjective( Union left, Union right ) {
  boolean foundForAll = true;
  for( Type x = right; !x.isEmpty(); x = x.cdr() ) {
    boolean foundForThis = false;
    for( Type y = left; !y.isEmpty(); y = y.cdr() )
      foundForThis |= isIn( x.car(), y.car() );
    foundForAll &= foundForThis;
  }
  return foundForAll;
}
```

Listing 3.13: Testing structural equality between unions

The second, defined in listing 3.13, is slightly more intricate. Each of the conditions in the premises are coded with the help of the auxiliary routine *isSurjective*, which determines if there is at least one corresponding element in the "right-most" set for each and every element in the "left-most" set, using the equivalence relation to determine correspondence. Then this test is turned around and checked the other way too see if there is any remaining elements in the left-most set that has no equivalent (and if a mapping was to be defined between these two sets, it would for these arguments have to elect targets already "taken" by being "hit" by some of the elements that *do* corresponds, breaking the

injection).

3.3.1 Exposing equivalence

The client code can test for equivalence at any time using the relation explicitly. However, sometimes there is an implicit need to test equivalence between two carriers imposed by other components that can not be retrofitted to use the relation. Neither is the explicit use of a relation always practical as it may reduce the readability of the code.

Java has a method `equals` that is provided to expose equivalence between two arbitrary objects. Although specified to take any other object in order to allow for heterogeneous collection, only the case where the other object is a type carrier will be discussed here. It is trivial to extend the code to handle classes outside of the *Type* hierarchy, and this is left as an exercise for the reader.

```
// class Type
public boolean equals( Object other ) {
  return this.compareTo( other ) == 0;
}
```

(a) Using the ordering alone

```
// class Label
public boolean equals( Object other ) {
  return new EqRel().isIn( this, (Type) other );
}
```

(b) Using the structural equivalence relation

```
// class Label
private static EqRel eq = new EqRel();

public boolean equals( Object other ) {
  return eq.isIn( this, (Type) other );
}
```

(c) Using global caching

Listing 3.14: equals() for labeled elements

It remains to be discussed which kind of equivalence should be exposed through this method; the "shallow" comparison using the ordering or the "deep" comparison using the relation. The former does not expose structural equivalence deeper than one level while the latter do. That means that there will be a difference between these two methods only in the carrier classes that references other carriers, such as *Label* and *Union*, while for *EmptySet* and *Leaf* they will yield the same result. As there is a significant cost combined with using a full-scale relation, this implies that it should only be used where actually providing additional benefits.

This thesis opt to provide the most functionally complete implementation, which means that the `equals` method should test for structural equivalence. The ordering method *compareTo* can still be used if a more lightweight comparison is needed. As a detail it should be noted in that respect that clients should put the carriers in a `HashMap`

or `HashSet` if structural equivalence is desired (as these classes uses `equals`), while `TreeMap` or `TreeSet` (which uses `compareTo`) is suitable for ordering.

Figure 3.14(a) give the general implementation of the *equals* method that makes it aligned with the ordering. It must then be overridden in *Label* and *Union*, for which an example in the case of *Label* is depicted in figure 3.14(b). This variant uses a local relation and consequently everything learned in this relation is forgotten upon return.

A more efficient method is the one in figure 3.14(c). Due to the immutability of the carriers, it can store all previously compared elements in a class-wide relation. It is particularly suitable to be used in a dictionary, where it is typically the same elements that are compared many times.

The method is not synchronized as any concurrent access should be handled by the relation. (For the method in figure 3.14(b), no synchronization was needed because the relation was a local variable). Furthermore, the relation should be private to the class, as no carriers from any other classes would ever be equivalent to a labeled element, causing nothing else than more objects in the map to inspect.

3.3.2 Congruence

Having two different concepts of equivalence can lead to confusion about which one is proper to use when. This problem can be removed if the two concepts in some way can be aligned with one another. A way to do this is through creating a *quotient algebra*.

In a quotient algebra, all carriers that are *congruent* with each other in the are mapped into a single, *canonical* carrier that represents them all. In addition to terms that evaluate to the same carrier, the case here will be to also map different objects that represents the same term into a reference to a single object.

The congruence criteria that will be used is structural equivalence. All objects that are structurally equivalent will form their own little sets and elect a "representative" for this set. By replacing objects with their representatives, they will after the mapping not only be considered equal by the "smallest" equality definition which is the one defined by the ordering, but also using the reference equality of the host language. This can be expressed with the following statement:

$$t \simeq u \quad \Rightarrow \quad RTG/_{\simeq}(t) \doteq RTG/_{\simeq}(u)$$

The mapping from RTG to $RTG/_{\simeq}$ will be done by a new method called *intern*, which returns the representative for the carrier at hand. In order to optimize repeated comparisons, this method may be called in advance (for *all* objects) as it will ensure that using simple reference comparison will suffice later.

In classes where the objects hold no state, all of them will naturally be mapped to a common, class-wide representative. This applies to the classes *EmptySet* and *Leaf*. Evidence of this is that the *equals* method already uses *compareTo*. The interning of the carriers can then be done by mapping back onto the *flyweight element* that is used by the constructor to represent this class. Listing 3.15 shows how this is done in practice.

```
// class Leaf
Type intern() { return Type.eps(); }

// class EmptySet
Type intern() { return Type.oe(); }
```

Listing 3.15: Interning to flyweight elements

Mapping the "heavier" classes *Label* and *Union* requires that a set of canonical elements are maintained so that interning is done once and for all for each element. It is not intended that it should be possible to retrieve the originating term for the representative, so it can be selected arbitrary from the set of congruent elements as long as they all select the same. One such implementation — for the class *Label* — will be presented, in listing 3.16. The implementation for *Union* follow along the same lines.

```
// class Label
static Map canonicals = new WeakHashMap();

Type intern() {
  Label representative = (Label) canonicals.get( this );
  if( representative == null ) {
    canonicals.put( this, this );
    content = content.intern();
    next = next.intern();
    return this;
  }
  else
    return representative;
}
```

Listing 3.16: Interning to canonical elements

A set that will hold all objects deemed canonical are declared class-wide. Upon interning, this set is first inspected to see if there is an element in it which is structurally equal to this one. This is done by the dictionary itself, as it uses the *equals* method to find a matching element from the set. If no canonical is found, this object is elected the representative as it is the first (and at the time the only) to appear in its congruence set, and the dictionary is updated. The key and the value is the same so that the map works like a set that returns the found element upon lookup.

After the set is updated, the element itself is "compacted" by replacing the content with its canonical representation. This must be done *after* it has been appointed the representative, in order to avoid a recursion trap if the content turns out to have a reference back to its parent. The replacement of a field with its canonical representation is safe as these will always be observed as equal (regardless of the equality concept used!). Although not strictly necessary to map further congruent elements, it must be done in order for the *compareTo* method to realize that the content of two labels are the same so that the mapping can be done more efficiently later.

3.3.3 Prefixing

Structural equivalence allows reasoning based on the algebraic specification. However, the rules in this specification is currently limited to recognizing isomorphy between parse tree of the terms, which has a limited application in itself. More interesting is checking if two carriers may be used as substitutes for one another:

Definition 3.17 (Semantical equivalence) *Two regular tree language expressions are* semantically *equivalent if and only if they recognize the same (tree) language, i.e.* $t \equiv u \Leftrightarrow L(t) = L(u)$.

Using set theory, it can be inferred that the above definition also constitutes an equivalence relation. The difference between this and the other definition of equivalence arises because the rules of the specification only ensure distributivity in the first tier (consisting of unions of elements) and not between the tiers. Unions inside sequences can not be moved up to the top level. What is missing is the rules:

$$l[t], \Big(u|s\Big) = l[t], u \Big| l[t], s \qquad \text{(A)}$$

$$l\Big[t|r\Big], u = l[t], u \Big| l[r], u \qquad \text{(B)}$$

Incorporating these rules in the constructors of the algebra would make all unions "surface" to the top level causing the resulting type to be a union of all documents of its language. The increase in problem size renders this approach unusable in practice, and this is partially why these rules have been let out of the algebra.

It is not the only reason however, because the are other problems as well. Upon construction of a closure, the constructors does not yet have information to determine if a forward reference is to a union or not, let alone which elements will participate in the union if it is. Unwinding the union when the reference is resolved is not an unproblematic option either, as it will possible change the type of the carrier (from a single labeled element to a union) and hence restrict such operations to be done only within the framework itself.

Another alternative is to reverse the rules and make the rewriting the other way, collecting unions with a common prefix and put them inside a single labeled element. While it seems desirable to let the algebra clean up the term by identifying parts that perhaps should have been put together in the first place, it would require very elaborate logic to determine these parts, and all this logic has of course an expense.

To grasp the complexity involved, consider that the two terms following is *not* equal, and that such a rewrite would be overzealous prefixing. Rather, elements with the same label and following sequence should be collected as a separate step from elements with the same label and contents.

$$l[t], u \Big| l[r], s \neq l\Big[t|r\Big], \Big(u|s\Big) \qquad \text{(C)}$$

Rewriting at the insertion into a union should be done by matching the element up with other elements with either the same content or the same following sequence. The

element that is selected for a merge must be temporarily removed from the union and recursively inserted again to allow both rules to apply. The rewriting of a cartesian product should be done as illustrated below.

$$\begin{aligned}\Big(\Big(l[x], a\Big|l[x], b\Big)\Big|l[y], a\Big)\Big|l[y], b &= \Big(l[x], \Big(a|b\Big)\Big|l[y], a\Big)\Big|l[y], b \\ = l[x], \Big(a|b\Big)\Big|l[y], \Big(a|b\Big) &= l\Big[x|y\Big], \Big(a|b\Big)\end{aligned} \tag{D}$$

Note however, that this alone does not solve the problems of forward references nor the inability to do deep comparison due to recursive structures.

A trade-off that performs well in practice is to perform the expansion of a label into the cartesian product of its contents and following sequence locally (i.e. only one level at the time) in the relation data structure where the deduction are under transactional control and closures will be handled correctly.

A third option is to employ the anti-symmetry of the subtyping relation defined in the next section, and reuse the code develop there. If desired, any of the two last approaches presented can be chosen and the implementation of `equals` changed accordingly. (One can also argue in that case that the current *compareTo* method should be renamed and that the subtyping relation should be used to provide the `Comparable` interface).

3.4 Subtyping

While it may be interesting to see if two types describe the exact same set of documents, it may be more beneficial to look into the more general, one-sided relation that checks if one of the types give at least the same set of documents as the other:

Definition 3.18 (Inclusion) *A regular tree language expression t is a* subtype *of u if and only if all trees recognized by t is also recognized by u, i.e. $t \sqsubseteq u \Leftrightarrow L(t) \subseteq L(u)$. Conversely, u is called a* supertype *of t.*

This relation is useful since it provides the ability to check if a change to a type makes it an *extension* of the original, or if it leads to a new, incompatible type. The supertype extension will — by definition — recognize all the documents of the subtype.

The direction of the inclusion may be confusing for day-to-day programmers. Type systems for most of the present generally used object-oriented programming languages usually only allow for *restriction* of an existing type although this form of inheritance often is given the name "extension". (As a digression, it should be mentioned that extensions can also be implemented using this mechanism since there are no checks that the so-called subtype actually adheres to the Liskov Substitution Principle [LG00] other than what the programmer has manually added of pre- and postcondition invariants). However, regardless of the name the concept still smells as sweet: The subtype is a more restrictive set of objects than is the supertype (considering that all of the instance objects

of the subtype is also of the supertype). Neither is the notion of subtyping that is used here interchangeable with using the terms to mean a branch of the parse tree.

Notice that this relation constitutes a partial ordering of types, as it fulfills the following properties:

$$\begin{array}{rcl} \text{Reflexive} & : & t \sqsubseteq t \\ \text{Anti-symmetric} & : & t \sqsubseteq u \quad \wedge \quad u \sqsubseteq t \quad \Leftrightarrow \quad t \equiv u \\ \text{Transitive} & : & t \sqsubseteq u \quad \wedge \quad u \sqsubseteq v \quad \Rightarrow \quad t \sqsubseteq v \end{array}$$

3.4.1 Complexity considerations

The subtyping relation could theoretically be realized by creating automata for each of the arguments and then checking the following equality, as both intersection and complementation are operations that can be performed on recognizable tree languages [CDG+02].

$$t \sqsubseteq u \quad \Leftrightarrow \quad L(t) \cap \overline{L(u)} = \emptyset$$

The Venn diagrams in figure 3.19 illustrates both a case where the two types are in the subtype relation and one where they are not. The shaded area denotes the complement of the right argument. If the left argument is not a subset of the right but falls (partially) into this area, the relation does not hold.

(a) $t \sqsubseteq u$ (b) $t \not\sqsubseteq v$

Figure 3.19: Subtyping modeled with Venn diagrams

It has been shown that the subtype relation has a worst-case exponential time complexity [Sei90] as the automaton must be made deterministic using subset construction to find the complement, and algorithms that are based on automata construction have turned out to be impractical due to the immediate realization of the entire node graph [AM91, HVP00].

3.4.2 Algorithm

What will be used in this thesis is the algorithm of Hosoya et al. presented in [HVP00], around which the framework presented earlier has been created to fit. The worst-case complexity of this algorithm is still exponential, but it can recognize subtyping between patterns without the need to expand them. As a result, it performs much better in practice.

In short, this algorithm works by breaking a type into its ground term components and testing them piece-wise through the framework, depending on the class of the carrier. All possible constructions of the left side have associated rules and these rules also covers any combinations on the right side. For non-basic elements on the left side, each subelement is recursively tested against the entire right side, and success is reported if and only if *all* of them matches. The outcome for basic elements depends on whether they are within the sum of cross-product from the right side. Sequences that are clearly disjoint with eachother can be optimized away before the cross-products are enumerated. Each of these steps will be explained and covered in detail in the following sections.

Merely a presentation of the rules and their implementation is done here, and will be restrained to an overview of their rationale only. For a rigid proof of soundness, completeness and termination the reader is referred to [HVP00]. For the algebraically inclined, their explanation may be better than a geometric representation.

The algorithm is well-suited for processing regular (tree) languages since it can handle untagged unions, a feature used in the types with which this thesis is concerned. For determining inclusion between two types from lambda calculus, a more efficient algorithm that runs in quadratic time exists [KPS95]. Programming languages usually falls into this category.

3.4.3 Non-basic elements

The first test handles the elements that are not basic, whose cardinality are different from one. The following deduction rules make use of transitivity amongst subsets, i.e. a subset of a subset will also be a subset. Each of the parts can hence be tested separately. This only works with the left argument, as the subset relation is anti-symmetric, i.e. a subset of a set is not necessarily still a subset of a subset of that set.

For such a composite type to be a subtype, *all* of its parts must be within the supertype. Two rules that processed these types are given in figure 3.20.

$$\frac{}{\Phi \vdash_{\sqsubseteq} \emptyset \sqsubseteq t} \quad [EMPTY]$$

$$\frac{\forall i \,.\, \Phi \vdash t_i \sqsubseteq u}{\Phi \vdash_{\sqsubseteq} \Big|_i t_i \sqsubseteq u} \quad [SPLIT]$$

Figure 3.20: Subtyping rules for unions

Notice that the rule [EMPTY] is not strictly needed, as the case of an empty set is also covered by the rule [SPLIT]. If the union is empty, then there are no elements that is not a subtype and therefore all (zero) of them are. This is also intuitive, as the empty set is logically a subset of any set. However, it is advantageous to do this as a special case since the empty sets are represented by their own carrier class and the framework dispatches

to various method based on the class. Doing the rule in its own method is just as short as the code needed to forward it the the more general one.

```
// class SubRel
boolean ruleEmptySet( EmptySet left , Type right) { return true; }
```

Listing 3.21: Subtyping for the empty set (rule [EMPTY])

The implementation of the rule [EMPTY] is as shown in listing 3.21 trivial. It always returns true as there are no premises to verify. Rule [SPLIT] consists of a loop that updates a flag using logical-AND. The flag starts out being set since an empty union would give that result as discussed above. It will be cleared if any of the individual parts fail. Code for this method is given in listing 3.22. Not shown here is the possibility of *short circuiting* the evaluation; if the flag has been cleared then the enumeration may as well be aborted as there are no other elements that can bring it back to the set state.

```
// class SubRel
boolean ruleUnion( Union left , Type right ) {
  boolean matchesAllTested = true;
  for( Type x = left; !x.isEmpty(); x = x.cdr() )
    matchesAllTested &= isIn( x.car(), right );
  return matchesAllTested;
}
```

Listing 3.22: Subtyping for a union with more than one element (rule [SPLIT])

3.4.4 Empty sequences

Empty sequences have no attached state and hence all of them look the same, and will be indistinguishable to the ordering. Such basic elements are atoms, and are used as leaves in the parse tree of a term. Any set of items that at least contains the empty sequence will be a superset of it. To test if the right side of the subtype relation is a supertype of the empty sequence it will therefore suffice to enumerate any of the parts (union-wise) to see if a leaf is found. If the right side is another basic element, it will be viewed as a union containing only itself and the basic element would then have to be a leaf too. The rule [LEAF] in figure 3.23 expresses this.

$$\frac{\exists i \,.\, \epsilon \doteq t_i}{\Phi \vdash_{\sqsubseteq} \epsilon \sqsubseteq \left|_i t_i\right.} \quad [LEAF]$$

Figure 3.23: Subtyping rules for leafs

The premise does not contain any recursive calls, so the leaf is a base case for the deduction, as would be expected.

The leaf will have to be at the top-level union in order for the code to flag that it has been found. It will be the first and only element in its sequence element. It cannot be

```
// class SubRel
boolean ruleLeaf( Leaf left, Type right ) {
  boolean foundALeaf = false;
  for( Type x = right; !x.isEmpty(); x = x.cdr() )
    foundALeaf |= ( x.car().compareTo( Type.eps() ) == 0 );
  return foundALeaf;
}
```

Listing 3.24: Subtyping for the empty sequence (rule [LEAF])

inside a sequence since a label then would have to occur in the markup stream contradicting it being an empty string.

The code in listing 3.24 shows how this is implemented using a loop that searches the union using logical-OR to update a flag that tells if a leaf has been found. Similarly to the processing of the union, it is possible to short-circuit the evaluation by terminating the enumeration (with a successful result code) once the flag has been set.

3.4.5 Disjoint sequences

Testing a labeled element against a union with high cardinality can be a very expensive operation, as will be shown in the next section. Therefore, it is interesting to find optimizations that can avoid or mitigate that situation. One such optimization is to "trim" the right argument (containing the union) of elements that only contribute to increasing the problem size but which is not part of the solution, before sending the modified union on to the main deduction rule.

Any sequence that starts with another label than the one on the left side (and there is only one label on the left side since the case of two or more should be handled by the rule [SPLIT]) cannot span a language that contains any of the documents that is described by that element. Neither can sequences that are empty (which must pass through the rule [LEAF] to succeed). Hence, only labeled sequences starting with the same label that is present on the left side need to be passed on to the recognizer. Figure 3.25 does just that.

$$\frac{\Phi \vdash l[t], u \sqsubseteq \left|_{j \in J} r_j, \quad J = \{i\ ,\ r_i \doteq l[t'], u'\}\right.}{\Phi \vdash_{\sqsubseteq} l[t], u \sqsubseteq \left|_{i} r_i\right.} \quad [PRUNE]$$

Figure 3.25: Subtyping rules for pruning

This framework will not dispatch directly to this rule since it has the same signature as the recognizer. It has the form of a helper method that is used by the recognizer as a preprocessing step. The code for this method is depicted in listing 3.26.

Again the right side is enumerated, but in contrast with the other rules a success code is not returned but rather a new type. This type is a union consisting of all elements from the right side that matches the selection criteria; that they are a labeled element with the same tag as the left side.

```
// class SubRel
Type prune( Label left, Type right ) {
  Type result = Type.oe();
  for( Type x = right; !x.isEmpty(); x = x.cdr() )
    if( x.car() instanceof Label && left.tag() == ((Label) x.car()).tag() )
      result = result.union( element );
  return result;
}
```

Listing 3.26: Removing anything but relevant labels (rule [PRUNE])

The returned carrier is not passed through the framework, and it might seem like an oversight that it doesn't get cached by [ASSUM]. In reality the rule is only used as a front-end and the parameters — which has a greater chance of appearing again, and do not have to be pruned in order to match — will get cached upon call to the outer recognizer.

3.4.6 Non-empty basic elements

At this point rules have been given for the constructors that generates unions and empty sequences. What is missing is a rule to handle the case where the left side consists of a single sequence starting with a labeled element and the right side is one or more of those. This is where the heavy lifting is done and henceforth this part of the algorithm is referred to as the *recognizer*.

An initial approach could have been to follow along the lines of [SPLIT] and test if any of the elements on the right side single-handedly matched the left side, by recursively inspecting whether the content and the rest of the sequence was a subtype of its corresponding counterparts. This would result in the rule [WEAK-REC] shown in figure 3.27.

$$\frac{\exists i \,.\, \Phi \vdash t \sqsubseteq r_i \quad \wedge \quad \Phi \vdash u \sqsubseteq s_i}{\Phi \vdash_{\sqsubseteq} l[t], u \sqsubseteq \Big|_{i=1}^{n} l[r_i], s_i \quad , \quad n \geq 1} \quad [\mathit{WEAK\text{-}REC}]$$

Figure 3.27: Insufficient rule for testing labels

However, this rule does not fit the bill. In order to visualize this, it might be fruitful to find an analogy in ordinary set algebra. As sequences are distributive over unions, they can be pictured as products and unions as additions, respectively. Although sequences are additions in some sense, they are represented as products here because the resulting language domain is the cross-product of the domain of the arguments. For the case of $n = 3$, the conclusion to be verified can be written:

$$t \times u \subseteq r_1 \times s_1 + r_2 \times s_2 + r_3 \times s_3$$

If $t \times u$ is a subset of only one of the terms — say the first term $r_1 \times s_1$ as is illustrated

in figure 3.28 — the rule [WEAK-REC] will indeed yield the correct result. Each term on the right side is represented by a solid square, and the dashed rectangle represents the left side which is embodied in the upper-left square. These are the cases where the right side is the same as the left side apart from being extended with some new terms which has nothing to do with the old ones.

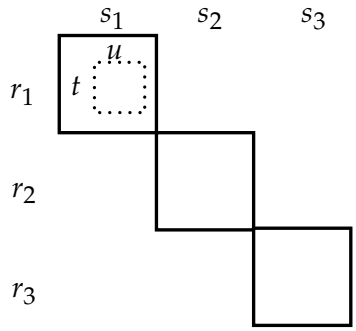


Figure 3.28: Subtype of only one term on the right side

But if the left side stretches over more than one term as in figure 3.29, then it will no longer suffice. What has happened here is that no individual element on the right side covers the entire left side, but the *sum* of them do.

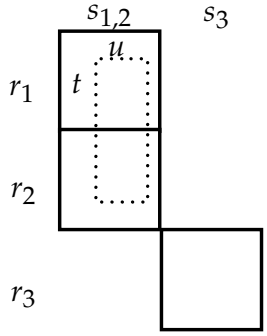


Figure 3.29: Subtype of more than one term on the right side

The reason this is possible — that the rectangle stretches over several squares, but is still not outside them — is of course that the squares are made adjacent by letting the argument in one of the dimensions overlap. In this case it is because $s_1 = s_2$. This is not such an uncommon incident that it may seem to be at first.

Consider rewriting rule (B) from section 3.3.3. According to it, the following regular tree relation and the corresponding classic set relation should hold:

$$\begin{array}{rcl} l\left[r_1|r_2\right], s_{1,2} & \sqsubseteq & l[r_1], s_{1,2} \Big| l[r_2], s_{1,2} \\ (r_1 + r_2) \times s_{1,2} & \subseteq & r_1 \times s_{1,2} + r_2 \times s_{1,2} \end{array}$$

Yet this is not recognizable to [WEAK-REC]. To overcome this limitation, Hosoya et al. develop in [HVP00] a rule [REC] shown in figure 3.30 that takes into consideration *all* of the terms while still breaking the problem in smaller components.

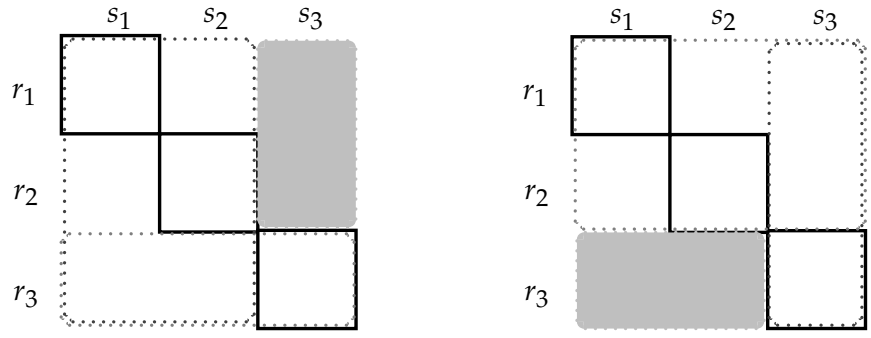
$$\frac{\forall I \in \wp(N) \,.\, \Phi \vdash t \sqsubseteq \Big|_{j \in I} r_j \quad \vee \quad \Phi \vdash u \sqsubseteq \Big|_{k \in \wp(N) \setminus I} s_k \quad , \quad N = \{i,\, 1 \leq i \leq n\}}{\Phi \vdash_{\sqsubseteq} l[t], u' \sqsubseteq \Big|_{i=1}^{n} l[r_i], s_i \quad , \quad n \geq 1} \; [REC]$$

Figure 3.30: Correct rule for testing labels

To do its work, this rule enumerates all possible combinations of terms from the right side (the *powerset*) and for each of these inspects the dimensions of those combination separately against the corresponding factor from the left side. This is what causes the exponential time complexity for the algorithm, as the powerset has 2^n elements. For instance if $n = 3$ then $N = \{1, 2, 3\}$ and $|\wp(N)| = 2^3 = 8$.

The interesting thing to observe here is that for each combination the first dimension is tested against, the other is done so against the *complement*, i.e. all the elements from the second dimension that did *not* have the same indices as the ones that were used from the first. Also note that the relationship is logical OR and not logical AND. Checking the other dimension against the same combination and requiring them both to hold is too lenient as it will only determine if the left side is inside the product of the sums and not the sum of the products. It cannot tell if the left side is only partially inside both terms (cf. the product $\{r_1, r_2\} \times \{s_1, s_2\}$ in figure 3.34 on page 79).

Hence the argument is turned around: If the left side is not in this combination for the first dimension, then it must at least be in the some of the *other* components for the other dimension. This leaves out the space that are in the combination's second dimension but is not in the combination itself (since if it were it would be matched by the condition that it should also be in the combinations first dimension). It includes space that are not in any of the terms (the intersection of the first dimension of the combination and the second dimension of those that are not), but this space is excluded by the combination that is the complement of this one (where the role of the dimensions are reversed). Each combination's complement will always be present in the powerset.



(a) $\{r_3\} \times \mathcal{T} + \mathcal{T} \times \{s_1, s_2\}$ (b) $\{r_1, r_2\} \times \mathcal{T} + \mathcal{T} \times \{s_3\}$

Figure 3.31: A combination exclude its complement's intersection part

An illustration may be in order to imagine this, and the reader is called upon to review the explanation above with the visual aid of figure 3.31. As $u \subseteq \mathcal{T}$ for any u where

$\mathcal{T}$ denotes the *maximal set* (i.e. "everything" in that dimension), $t \subseteq r$ is equivalent to $t \times u \subseteq r \times \mathcal{T}$ and vice versa. The squares represents the terms as usual while the dashed rings spans out everything that is in the same first dimension as the chosen combination and everything that is in the same second dimension as its complement, respectively. The gray area is what is excluded from this particular combination.

A point worth mentioning here is that each of the areas that are spanned out by taking the product of a combination in one dimension and its complement in the other (e.g. $\{r_3\} \times \{s_1, s_2\}$) is a "tile" that covers a part of the space that the squares would not if there were no overlap. (Actually the tile is what is excluded by the combination's complement. The intersection of the dashed rings in figure 3.31(a) is the gray area of figure 3.31(b)).

An intuitive explanation of the algorithm is that it checks if the left side is not within any of these tiles, but is rather outside each of them (i.e. inside the squares). If it is outside *all* of the tiles, then it must be inside *some* of the squares. Testing if the left side is not inside a tile (in both dimensions) can be rewritten to testing if it is outside it (in any dimension) using DeMorgan's law. Be aware that in this scheme, the outcome for the tile is determined by the combination's complement since that is the one that does the exclusion of the product.

To see how this works out in practice, consider table 3.32 that displays the steps involved in running the algorithm in the premise on the case presented in figure 3.28 on page 76. In each row, there is always one of the two result columns that indicate success because a dimension from the first term must always belong in one of the two sets.

$R = \Big\vert_{j \in I} r_j$	$t \sqsubseteq R$	$S = \Big\vert_{k \in \bar{I}} s_k$	$u \sqsubseteq S$
$\{\}$	False	$\{s_1, s_2, s_3\}$	True
$\{r_1\}$	True	$\{s_2, s_3\}$	False
$\{r_2\}$	False	$\{s_1, s_3\}$	True
$\{r_3\}$	False	$\{s_1, s_2\}$	True
$\{r_1, r_2\}$	True	$\{s_3\}$	False
$\{r_1, r_3\}$	True	$\{s_2\}$	False
$\{r_2, r_3\}$	False	$\{s_1\}$	True
$\{r_1, r_2, r_3\}$	True	$\{\}$	False

Table 3.32: Run of $l[t], u \sqsubseteq \Big\vert_i l[r_i], s_i$ with no overlapping terms)

By focusing on keeping the left side out of the area that is excluded instead of inside any of the one that is included, this recognizer handles overlapping too. Table 3.33 on the facing page lists the evaluations done to determine the outcome in the case of figure 3.29; the one where the rectangle stretched over two squares.

The lines that are different from figure 3.32 are highlighted. The interpretation of these lines are that the combination containing only r_1 alone is not enough anymore; having both r_1 and r_2 is required to succeed in that dimension. However, having s_2 (which also implies s_1) in the other dimension saves the day.

If there are no common components in any dimensions of the terms on the right side,

$R = \Big\vert_{j \in I} r_j$	$t \sqsubseteq R$	$S = \Big\vert_{k \in \bar{I}} s_k$	$u \sqsubseteq S$
$\{\}$	False	$\{s_1, s_2, s_3\}$	True
$\{r_1\}$	False	$\{s_2, s_3\}$	True
$\{r_2\}$	False	$\{s_1, s_3\}$	True
$\{r_3\}$	False	$\{s_1, s_2\}$	True
$\{r_1, r_2\}$	True	$\{s_3\}$	False
$\{r_1, r_3\}$	False	$\{s_2\}$	True
$\{r_2, r_3\}$	False	$\{s_1\}$	True
$\{r_1, r_2, r_3\}$	True	$\{\}$	False

Table 3.33: Run of $l[t], u \sqsubseteq \Big\vert_i l[r_i], s_i$ where terms are overlapping ($s_1 = s_2$)

then the left side will either have to be a subset of only one of them or it will be of none. Put another way: If a dimension of the left side is covered by more than one term on the right side then the other dimension in these terms must overlap in order for one of the sides of the squares to be common (and not just a corner) so that they are adjacent. If there are no adjacent sides, then the left side cannot stretch over two squares without also being outside of them.

To see this, remember that the rewriting rule (C) from section 3.3.3 on page 69 does not hold since the product of sums is larger than the sum of products (as the latter does not include the products of the non-corresponding terms):

$$\begin{array}{rcl} l\Big[r_1 | r_2\Big], \Big(s_1 | s_2\Big) & \not\sqsubseteq & l[r_1], s_1 \Big| l[r_2], s_2 \\ (r_1 + r_2) \times (s_1 + s_2) & \not\sqsubseteq & r_1 \times s_1 + r_2 \times s_2 \end{array}$$

There cannot be a union on the left side without having either the union expanded to two terms with one equal component (in the other dimension — like rules (A) or (B) do), or of course that the union is entirely matched by a single term.

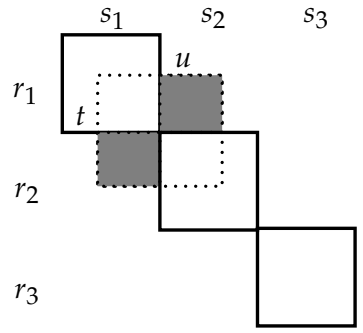


Figure 3.34: Non-overlapping terms on the right side

Figure 3.34 is a graphical illustration of this, and table 3.35 on the following page contains a display of how each of the combinations in the powerset and their complements are evaluated. The lines that make the check fail (both of the result columns are *False*) have been highlighted.

$R = \Big\vert_{j \in I} r_j$	$t \sqsubseteq R$	$S = \Big\vert_{k \in \bar{I}} s_k$	$u \sqsubseteq S$
$\{\}$	False	$\{s_1, s_2, s_3\}$	True
$\{r_1\}$	False	$\{s_2, s_3\}$	False
$\{r_2\}$	False	$\{s_1, s_3\}$	False
$\{r_3\}$	False	$\{s_1, s_2\}$	True
$\{r_1, r_2\}$	True	$\{s_3\}$	False
$\{r_1, r_3\}$	False	$\{s_2\}$	False
$\{r_2, r_3\}$	False	$\{s_1\}$	False
$\{r_1, r_2, r_3\}$	True	$\{\}$	False

Table 3.35: Run of $l[t], u \sqsubseteq \Big\vert_i l[r_i], s_i$ where terms are not overlapping

The gray areas in figure 3.34 on the page before are the parts of the left side that is not matched by any term on the right side; these are the parts of the dashed rectangle that fall outside of the squares. Observe that all of the tiles from the lines that fails in table 3.35 cover these gray areas. The first and third line cover the lower left gray area, while the second and fourth line cover the upper right (exactly opposite of what the intersections do).

Implementing the rule is much easier than comprehending it. Listing 3.36 displays the necessary code. The helper class *PowerSet* implements an enumeration over the powerset of a set of n elements by creating a binary counter that goes from 0 to $2^n - 1$ using each bits to indicate whether an element should be included in that particular combination or not. The test is otherwise modeled straight-forward after the rule, requiring all combinations to hold for the method to succeed.

```
// class SubRel
boolean ruleLabel( Label left, Type right ) {
  right = prune( left, right );

  boolean overAll = true;
  for( PowerSet p = new PowerSet( right ); p.hasNext(); p.next() )
    overAll &=    isIn( left.content, unionOf( p.subset(),content   )
               || isIn( left.next,    unionOf( p.complement(),next );

  return overAll && !right.isEmpty();
}
```

Listing 3.36: Subtyping for a single label (rule [REC])

All the labels on the right side in the conclusion of the rule are the same as on the left side in order to make it necessary to do pruning before using the rule in a proof tree. The method *ruleLabel* on the other hand, will be invoked by the framework regardless of the labels in the union. Hence, pruning must be invoked explicitly at the top of the method to eviscerate any irrelevant elements and the argument representing the right side is replaced by a "liposucked" copy.

It is also implicit stated that the union must not be empty by letting the counter be restricted to strictly positive numbers. As there is no deduction rule that cover the case of

the right side being an empty union, an appearance of a such should make the proof fail. This check must be done explicitly in the code since having no combinations to enumerate (and therefore none to contradict) would otherwise erroneously indicate a success.

The special case of a union with a cardinality of one can be (and has been in the program) optimized to do the evaluation in the style of rule [LAB] (from section 3.3) to eliminate the superfluous comparisons against the empty set.

Notice finally in the code the detail that `content` and `next` used on the subsets are not static properties of the type (which would be a union), but rather dynamic properties (function objects) named after them that fetches that field from the labeled element passed. They are used to combine the fields of each element in the subset union into forming a new union which is subsequently passed to the recursive relation check against that field from the left side.

3.5 Summary

In this chapter a framework for evaluating deduction rules upon structures where recursion may be present were established, using a transactional cache. In the context of this framework, the following relations

- Ordering equality, $\doteq$, which identifies equal instances
- Structural equality, $\simeq$, which identifies equal terms
- Semantical equality, $\equiv$, which identifies equal languages
- Subtype, $\sqsubseteq$, which identifies sufficient languages

were in order of "broadness" defined for the regular tree types set forth in chapter 2. The difference between them, their value and application were explained. A description of the relations' inner workings was given and code to implement them presented.

Chapter 4

Documents

The thesis has up until this point had its focus on types. It will now turn to the entity that made types interesting in the first place, namely *documents*. Documents are vastly more simpler to reason about than types. A type has a corresponding language that may even be infinite, whereas a document only refers to one, finite instance of this language.

Documents are values of the types that are being described. Types describe the structure of the data, while the data itself is located in the documents. Before the interaction between types and documents can be looked into, the way that documents are built must be examined.

4.1 Anatomy

A document consists of data that is already stored in a way that indicate its structure. The structure is embedded in the document together with the data.

Contrast this with *free-form* data, which is nothing more than a flow of single atom values without any relation between them. Structure does not come with the data, but must rather be imposed onto it later. The process of reading through the data and determine its structure is known as *parsing*. The result will be a document.

There reason why it may sometimes be advantageous to represent data in free form instead of as documents, is that the underlaying storage may not know of any larger entities than the most general ones such as characters. It is thus necessary to represent the structure as a stream of those and later rebuild the document from that stream. However, this thesis will not concern itself with the steps prior to achieving a document as there is available literature that explains this in detail, e.g. [ASU86].

4.1.1 Trees

A very common, versatile way of structuring data is in a *tree*. A tree is a set of objects (called *nodes*) that have either no or one *parent* and may have several *children* of the

same kind. An object that is the ancestor of another may not simultaneously be a descendant. The object that does not have any parent is called the *root* of its tree.

Trees are usually finite, and they can be nicely mapped into a free-form notation. Recall from section 2.2.1 that an element of information could be represented as a tree as well as flattened to a marked-up string of characters. Just like the difference between context-free languages and regular tree languages are in the use of markers, the one between documents and streams is also.

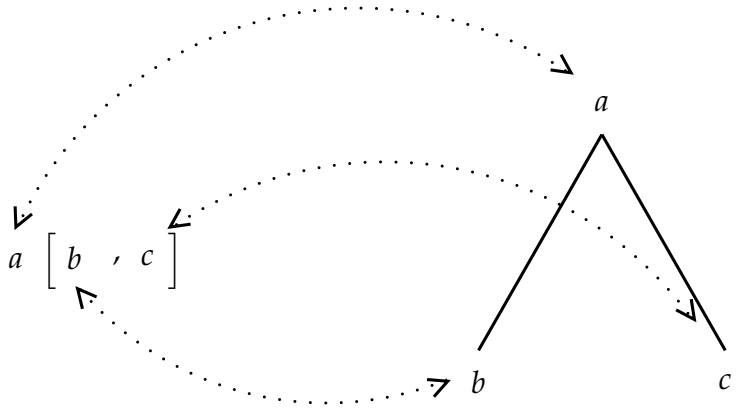


Figure 4.1: Mapping between a tree and a marked-up string

An example of this mapping is shown in figure 4.1. All elements that are within the markers are made children of a new node which gets the label attached to the markers. The order of the children is important as the sequence is not commutative. This allows the positioning of elements to have semantics, and it also makes it much easier to match a document up against its type as will later be shown.

The outlook this thesis has on documents are that they hold data in a structure and does not have any semantics of their own. A document carrier contains the structure in its construction and there are no other operations on these data. The framework does not assume in any way what the client code intend to do on this data.

Nodes in the tree are built out of strings and other data in elements besides labels must have a valid conversion with that type, thus ensuring that the tree is serializable to and from a character stream.

A specification for such value-holding carriers is displayed in figure 4.2. It is modeled after the construction of labeled type elements from the regular tree grammar in section 2.3.1. The only rules are the ones that states that the empty model is the additive element for sequences and that sequences are associative, respectively.

An algebra that satisfies this specification is one where the carriers are hedges of trees. Each carrier is a sequence of elements that in themselves may be carriers. This may be realized in Java as a linked list of the class in listing 4.3. Since each object has a reference to the next sibling in sequence through the field called *next*, all the children of a node can be pointed to by simply aggregating the first in the field *content*. The label associated with the node is stored in *tag*.

Only this one class represents all the carriers in the algebra, as the language itself

$VAL =$	initial spec	
	import $String$	
	carrier $Value$	
	ops	
	ctor $\bot :\rightarrow Value$	empty model
	ctor $\_[\_] : String \times Value \rightarrow Value$	labeled content
	ctor $\_ : String \rightarrow Value$	character data
	ctor $\_\,,\,\_ : Value \times Value \rightarrow Value$	hedge
	subject to	
	$\bot\,,\, x = x$	(I)
	$x\,,\, \bot = x$	(II)
	$x\,,\,(y\,,\,z) = (x\,,\,y)\,,\,z$	(III)

Figure 4.2: Specification for document values

```
public class Value {
  final String tag;
  final Value content;
  final Value next;
}
```

Listing 4.3: Skeleton of a document

contains a notion of the empty model $\bot$ in the form of the special reference **`null`**. Hence, there is no need to create a derivative class for this constructor. Rule (II) allows for a single labeled element to be represented as a sequence containing this element followed by the empty model. This leads to an opportunity to represent all of the carriers of type *Value* by a single base class. As before, *String* is reused from the runtime library of the platform.

Notice that there are no reference to the parent in a carrier. An object can be used as a child but is never associated with the parent explicitly. As the object is immutable (all fields are marked as **`final`**) the parent cannot influence its state. Since the algorithms work in a top-down fashion there is no need to maintain the parent reference and although not leveraged, it gives the side effect of also allowing two identical branches to share object representation.

By having immutable objects, trees created is always finite. A tree cannot be constructed where one of its descendants is itself because the reference to itself would then have to be exist before it was created! A similar problem was handled in section 2.4.7, but unlike types there are no provisions for forward references in values.

```
// class Value
public static Value label( String tag, Value content ) {
  return new Value( tag, content, null );
}
```

Listing 4.4: Hiding object allocation

The implementation fact that a labeled value is also a sequence of one should be hid-

den. Also, it is not desirable to expose the explicit allocation of a new object to represent the carrier. Both of these goals can be met by moving the construction of labeled elements into a static method as shown in listing 4.4. This passes the empty model as a the parameter that represents the *next* field, terminating the sequence. This could also be seen as a default value for that field, while the other two must always be specified.

The client code should use this method as a factory. This reduces dependencies between modules and will allow the implementation to alter the representation scheme at a later point if desirable while introducing as few changes as possible in the interface.

4.1.2 Leaves and sequences

Selecting **`null`** to represent the termination sentinel in sequences aligns it with the leaves in the trees. There is no difference between an empty content model and the residual part of a sequence after all the children has been enumerated.

Listing 4.5 contains the code that performs the operation of the sequence constructor in the algebra. If the end-of-sequence marker is encountered, then this is the right place to insert the value to be appended. The new tail is responsible of terminating the sequence, and it is not possible to create any carrier to pass to this method that is not properly terminated. If the end of the sequence has not been reached yet, the sequence must be traversed using rule (III) until that is the case, pushing the insertion further down an induction chain where the empty model is the base case.

```
// class Value
Value concat( Value v ) {
  return clone( next == null ? v : next.concat( v ) );
}

protected Value clone( Value next ) {
  return new Value( tag, content, next );
}
```

Listing 4.5: Planting a hedge

Immutability of the values are guaranteed by creating new values as they are "modified" by replacing the rest of the sequence with one that has been modified (an element has been added). The code is functional: A new value is returned instead of modifying the one the operation is invoked upon. The original value may be seen as merely a template on which the new sequence is generated (but with a changed tail).

The allocation of a new object is deferred to the helper method named *clone* instead of calling the operator **`new`** directly, for reasons that will be explained in the next section. The method *clone* creates a new object which has the same label and content model as this one (thus the name), but which may occur in another context.

However, observe that *concat* is not a **`public`** method. The cost of this gained simplicity is that **`null`** must be treated like any other valid value object. More specifically it must be legal to use the empty model as a starting point of a sequence. The implementation of the sequence constructor cannot be realized through *concat* alone, but must be

filtered through a static method such as the one in listing 4.6 that checks for the occurrence of **null** on the left side.

```
// class Value
public static Value combine( Value a, Value b ) {
  return a == null ? b : a.concat( b );
}
```

Listing 4.6: Performing concatenation on **null** references

If the left side is the empty model, then the sequence is replaced with the carrier to add in accordance with rule (I). Otherwise an initialized object is present and the method *concat* can be used on this.

4.1.3 Character data

Relying on structure alone may be an insufficient or impractical mean to convey information in a document. Many applications use data that is not available for enumeration at the time the schema is written, or whose format is not easily expressed as a regular tree. Examples of these are names of persons, quantities and measurement data.

Such entities are considered opaque to the structure and is better rendered in its own format, encoded as a sequence of characters embedded in the document's stream. When the stream is parsed, these characters must still be scanned in order to determine their boundary. They are therefore referred to as *parsed character data*.

This thesis will treat character data as an element with no content model, labeled by the special token #PCDATA. Occurrence of data may then be modeled in the type by using this element. The characters themselves are attached to the element out-of-band so that they are available to the application, but no interpretation of them are done. Further specification of the allowed syntax requires the client to do its own specialization of the code. Mapping the semantics of specialized data to a type will be discussed in section 4.2.4.

```
class Pcdata extends Value {
  final String text;

  Pcdata( String text, Value next ) {
    super( "#PCDATA", null, next );
    this.text = text;
  }
}
```

Listing 4.7: Character data elements

Specialized elements are handled by subclassing the carrier class *Value*. Listing 4.7 contains the general specialization for all character data. The structure of the element is fixed, and the text is stored in a field that is available for later introspection. As before, direct exposure of the carrier is not desirable, so the algebra is extended with a method that

performs the construction of a single, stand-alone such item that can later be combined with other branches into a larger document.

This method is the remaining constructor from the specification; the one that converts strings into value carriers. Its implementation is shown in listing 4.8.

```
// class Value
public static Value data( String text ) {
  return new Pcdata( text, null );
}
```

Listing 4.8: Constructing value carriers from strings

With the introduction of a derivative carrier class, the framework cannot use one specific constructor to create a clone of an object instance as this operation has become polymorphic. Remember from section 4.1.2 that cloning is necessary to obtain a duplicate that can be used in another context, due to the fact that the object instance itself is immutable. Hence the method *clone* must be reimplemented in *Pcdata* to return an object of the same type. The code for this is in listing 4.9. Observe that in contrast with the implementation from the base class in listing 4.5, the fields unique to this class is passed to the constructor.

```
// class Pcdata
protected Value clone( Value next ) {
  return new Pcdata( text, next );
}
```

Listing 4.9: Polymorphic copying

If this is not done, instances of the subclass will be *sliced* as the superclass only copy the fields that it know of and not the new that is introduced in the subclass. (Unfortunately there is no way in Java to specify that a subclass *must* customize a method in order to prevent slicing from a base class that is not abstract).

4.1.4 Attributes

Little has been said so far about *attributes*. Attributes in documents can be seen as semantic "constants" that are applied to the parse tree. They would be constants as they are specified together with the data and is thus static. A broader concept of attributes includes dynamic ones, i.e. variables that are evaluated upon traversal of the tree in the semantic passes of the application [EMRS97]. The attributes mentioned here are for instance suitable as initializers for those variables.

Attributes formally introduce other properties than the label into the element, integrating them with the markup language itself instead of leaving it to the application to define how multiple values are encoded in the character data children. Attributes are used to explicitly state the presence of those values in the schema, enforcing that the document writer is aware of them. Indeed can both the label and the character data of the element be regarded as nothing more than predefined attributes [McG01].

In present markup languages such as SGML and XML attributes can only hold flat contents in form of a single string, whereas character data can be interspersed with other element children making it possible to create a composite structure. Traditionally, the convention has been to use attributes to hold *meta-data*, which is information about *how* the document should be processed, while character data was used to hold the payload of "business" information [Hol99]. The connotation has been that attributes and tags are the "envelope" and the character data is the "letter".

Another difference was that in the schema language DTD only attributes could be strongly typed, albeit the selection of types were fairly limited. Character data was always seen as an opaque stream. In newer schema languages such as XML Schema and RELAX, it is possible to specify the (simple) type of character content too, blurring the distinction between them. Attributes are being reduced to a short-hand notation of children that cannot have complex types, i.e. a hierarchical structure.

The approach used by this thesis will be along the lines of this development. Attributes (in terms of semantic constants) does not seem to add any power of expression and will therefore be implemented by using existing parts of the framework, keeping its weight down. They will be created as child elements containing only text. The value of the attribute is kept in the text whereas the name of the attribute is put in the label. To prevent clashes with real child elements as defined in the schema, the names of the attribute will be prefixed by "@". Listing 4.10 contains the appropriate code.

```
// class Value
public static Value attr( String name, String value ) {
  return new Value( "@" + name, new Pcdata( value, null ), null );
}
```

Listing 4.10: Attributes as character data elements

The type of attributes is restricted to a simple character string only. If further refinement is desired, a subclass of *Pcdata* that handles the (simple) datatype should be created and a method that created attributes using this subclass instead added to the algebra. A general solution would involve that the class factory representing the datatype of the attribute was passed as a parameter.

Although it is possible to model the tag of an element as an attribute, this is usually not done and the libraries for parsing markup do not encourage it either. Elements must still have a field for names if they are used in the implementation of attributes, so no savings are possible. Hence, the label is still internalized in the object as a field and this field is used as storage for the name of the attribute.

By convention there is an informal requirement that attributes are put before any other child elements and that there are no two attributes with the same name. However, these conventions are not enforced by the framework in any way, partly because it does not need them to be fulfilled. If the usual attribute semantics is necessary, it is up to the client application to overhold them.

4.2 Validation

The purpose of schemata is to describe a set of documents without having to list them all. In order for a schema to not refer to all documents but rather just some of them, there must be a way to identify those documents that are in the schema's language. As a language may be infinite, it is futile to attempt to list all documents that are in the language. Instead, the test will be done the other way around: A single, concrete document can be checked to be in the language of the schema. This is called the *language relation* and it is is formally stated in definition 4.11:

Definition 4.11 (Language relation) *A type t is a model for a document v, written $t \models v$, if and only if $v \in L(t)$. Conversely, $L(t) = \{v \mid t \models v\}$.*

The process of determining whether a document is in the language of a schema is called *validation*. It is then assumed that the document was written with a particular schema in mind, and that it should be verified to (still) be in compliance with it. Often this information follow along with the document in the form of a processing instruction or can be inferred from the context that the document was retrieved, much like type information is attached to references in strongly-typed managed languages (e.g. Java).

4.2.1 Data integrity

Validity can be checked on two levels. The first concept is *well-formedness*, which treat of validity towards the markup language itself. A character stream is said to be well-formed if it is in within the classes that the markup language spans out, i.e. if it can be recognized as marked-up text. Any such stream can be read into a document, and in fact is this the very criteria for parsing documents from streams: That they are well-formed.

On the other hand, the second concept of validation is about validity according to a type. Since a type only describes documents, valid markup must also be well-formed. A document will always be valid according to *some* type as it is possible to construct a type that has been tailored to it [GGR+00], but there is of course no guarantee that it can be shoe-horned into an arbitrary target type. Validation can be regarded as the process of ensuring that the implicit type that can be inferred from the document is a subtype of the one desired. This is revealed by the following rephrase of the validation problem: If u is a type (that is inferred) such that $L(u) = v$, then $u \sqsubseteq t \Rightarrow v \in L(t)$.

Checking for well-formedness is within the problem domain of the parser, and is consequently outside the scope of this thesis. Validation however, is performed by a component known as the *validator* and such a module is what this thesis will present.

Type validity hold an important rule in ensuring a certain level of quality in the data. If the document is known to adhere to the structure set forward in the type specification, the client application can access its various parts in a safe manner, assuming that they will be present as expected. Error handling can be done up-front instead of being tangled together with the business logic that determines the document's semantics. Moreover, the explicit specification of a structure through the use of a type makes it possible to

do validation as an entirely separate step from the other processing. Validation can be done once upon storing the document initially, enabling the client to assume that the documents that a drawn from the repository to meet the set standard.

4.2.2 Matching

A natural way of doing validation seems at first to be that each type determines if the value is within its language, and let this test be done recursively in children and tails. One would perhaps expect that the problems with comparing types — namely that it is possible to fall into a recursion trap if there exists a forward reference to the very type that is being checked — is not present in this scenario since values cannot contain loops and enumeration therefore must be finite as the check terminates when it hits the base case of a leaf. An imaginary method *match* could return the presence of a document in the type's language.

For labeled elements the reasoning in the above paragraph is is correct. Since this class is what (primarily) creates problems in type-to-type comparison, it might be tempting to draw the erroneous conclusion that type-to-value checking will work fine when this obstacle is removed. However, it is a fallacy for unions which also is prone to this problem due to the implementation artifact of forward references needed for closures.

Because of the flexibility given in the extra degree of freedom from the union tier of types, which does not exists in values, it is the type that is the effective unit of dispatch in the validation, as will explained shortly. The language of a union contain a document if it is in the language of each part of the union. The rule [TRY] in figure 4.12 describes the formal relationship between union types and documents, and the reader is encouraged to explore the similarities with rule [SPLIT] from section 3.4.3 on page 72.

$$\frac{\exists i \,.\, t_i \models v}{\Big|_i \, t_i \models v} \quad [TRY]$$

Figure 4.12: Validation is distributive over unions

If one of the elements in the union is a forward reference that point to the union itself again, this rule will cause a recursive comparison to enter an infinite loop. Although value carriers have countable lengths, there is no part of the rule that reduces the structure of the value on the right side, only the type on the left side.

Even a simple check for forward references to the same union is not sufficient, as a cycle may be comprised by more than one union. In order to properly handle such cycles, all references must be resolved using the *deref* method from section 2.4.7 on page 46. Mutually referenced unions will then be combined into one. Another approach is to treat the language relation the same way as the subtype relation; through a transactional caching rule-deduction so that the rule [TRY] is not re-entered upon the same pair. The language relation can then either be implemented as an explicit relation containing the

matching methods for each carrier class as rules, or as an application of the subtyping relation through the conversion of the document to a type carrier.

The framework contains all of these implementations and enables switching between them with a compile-time parameter, but the thesis will only provide an in-depth review of the latter. The diligent reader who wants to inspect the other two are referred to the source code.

4.2.3 Inferring types

Every document has a corresponding type that validates the exact structure of it and nothing more. If anything is removed or changed from the type the document will no longer be in its language, and everything that is added will be superfluous. Because of these properties, this type is called the *minimal* or *initial* type of the document.

The rules for inferring a minimal type from a value are displayed in figure 4.13. What they do is to build a tree with the same labeled element that is in the document, putting leaves at the end of each branch. Written in this format, the application of the rule may seem somewhat backwards: The right side of the conclusion is the pattern that matches the value passed as parameter, giving the resulting type on the left side. The premise of the rule shows the intermediate calculations necessary to arrive at that result. Note that the two rules explore all possible value constructors.

$$\frac{}{\epsilon \models \bot} \quad [NIL]$$

$$\frac{l = l' \qquad t \models v \qquad u \models w}{l[t], u \models l'[v], w} \quad [VAL]$$

Figure 4.13: Mapping values to types

No unions are created from these rules, which is of course due to the deterministic nature of documents. If these rules are to be used in an explicit language relation, they must also include the rule [TRY] from the previous section which not only covers unions with cardinality larger than one but also the empty set. The empty set should fail to validate anything as its language contains no documents.

Implementing these rules is relatively straight-forward from the rules, as displayed in listing 4.14 on the facing page. Unfortunately, the since the **`null`** reference was chosen to represent the sequence terminator for value carriers it cannot be implemented as a regular method of the algebra but must be a static method, giving a procedural flavor to the mapping that does not explore polymorphism. However, the author regard this as a minor trade-off. It can be remedied by putting the label construction into a factory method of the value carrier, akin to *clone* (cf. section 4.1.2 on page 86).

A front-end that performs the validation in form of an internalized method in the type can now be written as is done in listing 4.15. This code uses the subtype relation to

```
// class Value
public static Type infer( Value v ) {
  if( v == null )
    return new Leaf();
  else
    return new Label( v.tag, infer( v.content ), infer( v.next ) );
}
```

Listing 4.14: Implicit type of documents

check if the type is capable of holding the structure inferred from the document. Since the minimal type to which it is compared is a tree of temporary objects, there is no point in caching the relations deduced from this for later nor is it productive to employ in this comparison relations cached elsewhere.

```
// class Type
public boolean match( Value v ) {
  return new SubRel().isIn( Value.infer( v ), this );
}
```

Listing 4.15: Front-end for language relation

Regarding the complexity of the validate operation, the worst case scenario of runtime exponential to problem size will not be encountered due to the fact that there are no unions in the minimal type. Although it is possible that unions in the schema will have to be investigated for every element, the length of the term provide an upper limit since union cycles are cut off. Sequence cycles are terminated when the structure of the value carrier is exhausted, making runtime dependent on the size of the document as well as the size of the schema.

While the use of the subtype relation to do validation may seem elegant, there is some overhead associated with maintaining the cache and creating temporary type carriers, which can be avoided by using the recursive algorithm. However, with the recursive algorithm not being as configurable as the relation and of approximately the same order, the proper implementation to select should be determined based on the requirement profile of the client.

4.2.4 Custom datatypes

Matching the character data to a datatype other than string poses some extra problems. In essence, a subclass of the value carrier that represents the datatype must be created and a token that can represent such elements selected. Section 4.1.3 contains a template for such specialization through the use of *Pcdata* to represent strings. A hierarchy of datatypes must be created and the subtype relation enhanced correspondingly to detect subtyping within this hierarchy instead of doing simple tag comparison, where applicable in the rules (cf. rule [PRUNE] in figure 3.26 on page 75).

Doing validation in terms of checking the inferred type will no longer be a viable option in the presence of custom datatypes, as the actual content is lost when rephrasing

the value to a type. Any simple datatype should inherit from string and will therefore turn out to "match" it even when the contents does not. Hence, matching must be done with either the explicit relation or the recursive algorithm, that both work of bona fide value carriers. In these cases, the checking of the actual contents against the datatype must be done in the rule [VAL] from the previous section. An actual such implementation is however considered to be outside of the scope of this thesis.

4.3 Paths

In conjuncture with the discovery of certain properties or facts about a document, the need may arise to address only a branch of it during processing as the result of the algorithm is often caused by only a certain subset of elements. For the writer, the document does not appear as an opaque blob but rather as a structure of smaller entities strung together. If given an indication as to which of these was the origin of the result, it becomes easier to pinpoint any flaws or deviations from the original intent and identify the necessary changes to make amends. This is analogous to the way a programmer is given line and column number in the source file when the compiler detects an error.

At runtime, the various parts of a document is available as references to carrier instances. If the document is kept loaded in memory and an editor is integrated with the framework, such a reference may be sufficient to use to address the branch. Unfortunately, instance references cannot be persisted and activated again later. The memory that backs the object may be recycled by the garbage collector and there is no guarantee that the object will appear at the same memory position if revived from secondary storage. Hence, references are short-lived and cannot provide the identity of an object in the long run.

To address parts of a document in a location-independent manner, a construction called a *path* is used. Paths does not tell where the branch exist at any point, but rather how to get there from a well-known point. In this thesis, it is assumed that the client has a mean to locate a specific document from the repository and because the entire document can always become bootstrapped from that location, the start of the document then becomes an *anchor* from which one can navigate. Usage of paths relative to other anchors are of course also possible, but that aspect is not investigated here.

4.3.1 Determinism

Paths specify the navigation necessary to arrive at a particular branch by listing the tags encountered and the direction taken at each junction. However, listing everything up to the point where the branch starts would amount to a lot of needlessly echoing of the document. Therefore, a path is built up by including only those elements that is actually entered, represented by their tag. Any sibling element that does not have this tag is implicitly skipped. Since more than one element may carry the same tag, it is also necessary to denote in some way how many elements with this tag that is to be skipped. This latter number is called the *index* or sometimes the *position*. The index number will indicate

which occurrence of a labeled element in a sequence that is to be entered. The children of the ones before it need not be examined at all, which may speed up the process of retrieving a branch.

An example of a path is:

$$/a[2]/b[3]$$

which identifies the third occurrence of b within the children of the second occurrence of a relative to the start of the document. For this path to be a branch at all (and not an empty model), there must be at least one other occurrence of a at the top level, and two other occurrences of b at the next.

Branches in documents are identified uniquely by the path, i.e. a path will only identify one particular part of the document. The reason for this is that the path describes the subterms necessary in the construction of the document for it to contain this branch. The building blocks of a path may be seen as descriptions of value constructors, and these operations only creates one carrier (which will contain the branch in question). For two branches to have the same path, the same carrier would have to be constructed from two different constructors.

This does not mean that the path determines the document, because the construction may also contain several other subterms that is not mentioned in the path. It does for instance not say what elements there should be in elements that are skipped, nor how many and what kind these elements should be — merely that they should not be entered if they do not match the pattern in the path. From this follows that the same path may exists in more than one document even though the documents are not alike, since the position of the branch relative to other parts may change. Also notice that that the path does not make any statements of the contents of the branch which may also be different from document to document.

Paths are not limited to documents only but can be "lifted" to work on schemata as well, since a schema is a set of documents. A path in the schema would then denote the set of branches that is identified by it in all of the documents in the language of the schema. In this context, it is very likely that the path addresses more than one value carrier. With regards to schemata, paths are *non-deterministic*.

What is interesting is then to attempt to use the path as a filter for subterms in *type* construction in the same manner as in value construction, in order to determine if the language of a type carrier contains the (sub)set of value carriers that the path describes. That a type matches a path is used to mark that documents that also matches the path are in the language of that type.

Since it is possible in types to construct unions in addition to sequences and hierarchy, a type carrier that matches a path will potentially generate more than one value carrier in a document due to this extra degree of freedom. It is also common to sometimes drop indices in schema paths because closures can make the point of having them somewhat moot. The type carrier a^* for example, will match both paths $a[1]$ and $a[2]$ or any other index for that matter.

4.3.2 Incarnation

Locating a branch in a document is traditionally done in two ways: Either by navigating an already-built tree structure [ABC+98] or by filtering a stream of events that is generated as the document is traversed once [Bro01]. The former uses a top-down approach while the latter has a bottom-up flavor. The algebra for a path that is presented here will be compatible with both.

A path is built by stating how subterms that appear in the construction of the value is treated. The path always start from the root, which can be thought of as an empty model to which the document is then appended. A labeled element can either be skipped entirely if the branch is not within its descendants, or it can be entered. Skipping an element moves processing to the next value in the sequence, while entering it brings the child value sequence into focus.

In a processing model based on event streams, the events will indicate the beginning and the end of an element. These events are generated when the parser discovers the markers in the character stream from which the document is built. Such events are in their nature imperative, while the specification presented here is intended to be functional. This is solved by letting the discovery of an element imply that all events until the end marker are ignored unless the next action is to explicitly enter that element. The tree model is likewise handled by simulating events by a depth-first traversal.

$PTR =$ constructive spec	
import $String, Int$	
carrier $Path$	
ops	
ctor $/ :\rightarrow Path$	starting anchor
ctor $\_\triangledown : Path \rightarrow Path$	enter branch
ctor $\_\triangleright\_ : Path \times String \rightarrow Path$	discover tag
$parent : Path \rightarrow Path$	previous level
$count : Path \times String \rightarrow Int$	get index
$xpath : Path \rightarrow String$	serialize
subject to	
$parent(/) = /$	(I)
$parent(p\triangledown) = p$	(II)
$parent(p \triangleright s) = parent(p)$	(III)
$count(/) = 0$	(IV)
$count(p\triangledown, s) = 0$	(V)
$s = s' \Rightarrow count(p \triangleright s', s) = count(p, s) + 1$	(VI)
$s \neq s' \Rightarrow count(p \triangleright s', s) = count(p, s)$	(VII)
$xpath(/) = ""$	(VIII)
$xpath(p\triangledown) = xpath(p) + "/"$	(IX)
$xpath(p \triangleright s) = xpath(parent(p)) + "/"s +$ $"["+count(p \triangleright s, s) + "]"$	(X)

Figure 4.16: Specification for paths

Figure 4.16 contains the specification for a path. As usual is a specification for the stan-

dard carriers *String* and *Int* assumed readily available from elsewhere. The constructors but the constant represents the various filters that can be applied. In addition, *xpath* is an "observing" operation that renders the path into an externalizable format that can be displayed to the user or be dealt with by a component in the application that is not part of the framework. This format is a subset of the standard path language for markup documents, XPath [CD99]. A slash is used to separate one level from the next, and indices are put in angle brackets behind the appropriate label. The two other operations are helper methods that is used by *xpath* and is not imminently needed.

Informally, a path is implemented as a stack of maps, where each level in the stack corresponds to a level in the structure tree and the map contains the number of elements of each label kind that has been processed so far. The state necessary to hold this carrier is outlined in listing 4.17. As will be explained shortly, the map is implemented in terms of a stack.

```
class Path {
  Stack/*<Stack <String>>*/ tags = new Stack();
}
```

Listing 4.17: Skeleton of Path carrier class

When an element is entered by the $\triangledown$ operation, a new level with an empty map in it is added to the stack, and processing continues at that level. Each of the elements in this level that is discovered by the $\triangleright$ operation updates the map by incrementing the counter for the label passed as an argument, keeping track of the number of such elements encountered. The root / simply consists of a stack with a single element where no labels have been encountered yet. This is shown in listing 4.18 below.

```
  // class Path
  Path() { tags.push( new Stack() ); }
```

Listing 4.18: Constructing a root path

If the runtime had provided immutable versions of Stack and Map, for example implemented as a linked list, the remaining two constructors could have been implemented almost right out of the description of the previous paragraph, returning a new path with a stack of maps that was updated accordingly.

However, the stacks and maps in the runtime are mutable. Cloning the entire stack for each operation is considered impractical. Likewise is a custom implementation of stacks and maps that has the immutability property. Rather, this thesis finds it desirable to reuse the data structures provided in the runtime due to their efficiency. As will be strived to show, this can be done without compromising the intent behind the specification although it will not be functional anymore.

Instead of having each constructor generating a new path carrier, they are transformed into "mutating" operations that changes the already instantiated object to the new state. Each such operation is then accompanied by an operation that can undo this

effect so that the carrier returns to the old state. The operation $\triangledown$ is implemented as shown in listing 4.19 as a pair of methods that adds a new level to the outer stack and removes it again, respectively.

```
// class Path
void increaseLevel() { tags.push( new Stack() ); }

void decreaseLevel() { tags.pop(); }
```

Listing 4.19: Depth of the path

The reason why the map is implemented in terms of a stack will now be revealed: The stack is the natural data structure to use if there is a need to remove the item that was just put into it, which is precisely the task of the undo operation. Instead of keeping a counter associated with each label, the number of times the label appear on the stack is its corresponding index. Hence, by using a stack, insertion becomes efficient at the expense of lookup, which now have to walk the stack in order to count the elements. However, this trade-off is deemed acceptable since the observing of the path is to be done only when the path is finally rendered to the client as a one-time operation, while updating of the path will be done inside of the algorithms, multiplying the complexity of the update operation with the complexity of the algorithm.

Listing 4.20 contains the code necessary to implement the $\triangleright$ operation. The methods respectively inserts and remove label occurrences from the top-most level.

```
// class Path
void enterSequence( String tag ) { tags.peek().push( tag ); }

void leaveSequence( String tag ) { tags.peek().pop(); }
```

Listing 4.20: Handling of tag occurrence

In addition to these constructors, there is a *clone* method added to the algebra that performs a deep copying of the outermost stack in order to create a replicate of the path. (The inner-stack need only be copied shallowly since the strings on it are immutable). If a snapshot of the object representing the path at its current state is needed, this method can be used to create a copy that will not be modified through the original reference.

Rendering of the path is done through the method *toString* that fulfills the duty of the *xpath* operation, as this is the customary way in Java to name an operation that prints the object to a character stream. The *count* helper operation of the specification walks the inner-most stack and counts the elements on it with the matching label. The *parent* helper methods is present to provide a way to inductively recurse through the elements of the stack. Both of these operations can be inlined into *toString* method and rewritten as loops.

They are present in the specification as separate operations simply because the specification language is strictly functional and there are no other convenient ways to model loops. While perhaps theoretically elegant to implement them using recursion in the code

too, it can be argued that it is also the only gain relative to an imperative implementation and will perhaps add verbosity at the expense of clarity since the implementation otherwise is not suited to follow the functional treatment. For instance is it hard to retrieve the parent path without copying data. On these grounds, the imperative version is presented here.

```
// class Path
public String toString() {
  StringBuffer buf = new StringBuffer();
  for( int level = 0; level < tags.size(); level++ ) {                    // (†)
    Stack tagsInLevel = tags.elementAt( level );
    String tag = tagsInLevel.peek();                                      // (‡)

    int count = 0;
    for( Iterator iter = tagsInLevel.iterator(); iter.hasNext(); )        // (§)
      if( tag.equals( iter.next() ) )
        count++;

    buf.append( '/' ).append( tag ).append( "[" ).append( count ).append( "]" );
  }
  return buf.toString();
}
```

Listing 4.21: Serialization of a path

The outer loop started at (†) represents the use of the *parent* operation. Instead of starting the rendering with a recursive call, each of the levels are processed in order from bottom to top. Operation *count* is performed by the loop that starts at (§) and the label is found at the line marked with (‡). The line at the end outputs the label and index together with the delimiters of the XPath language. The code accompanying the thesis includes more corner cases and error handling, and the reader is referred to this for details.

4.3.3 Traversal

Gathering statistics about the usage of tags in a document can be done by looking at all the paths that exists in it. Any information or counters could then be inferred from these paths and then stored in for example a database that kept record of which documents matched which properties in a mapping table. Later in the thesis, such a system will be developed.

Finding all the paths in a document is a matter of traversing it and track the path to each of the elements as they are visited. Callbacks before and after the elements are visited enables the application to hook into the path enumeration and perform special processing at each point.

A class called *Traverser* will encapsulate the functionality of the traversal, and to hook into this process the client should specialize this class and provide the concrete methods necessary for callback. At each callback, the current path is available through a protected member, which is used to reflect the state of traversal. It should be regarded as read-only by the callback methods, i.e. they should only use its observing method to extract the state, in order to not disrupt the flow.

Listing 4.22 shows the *Traverser* class and its main method *traverse* that takes as input the document value tree upon which the operation is to be performed. The client should override the methods *preProcess* and *postProcess* to perform actions before and after a node in this tree is discovered, respectively. The branch which is rooted in this node is passed as a parameter to these callbacks.

```
// class Traverser
protected Path path = new Path();

void traverse( Value v ) {
  if( v != null ) {
    path.enterSequence( v.tag );                          //(†)
    preProcess( v );

    path.increaseLevel();
    traverse( v.content );                                //(‡)
    path.decreaseLevel();

    postProcess( v );                                     //(§)
    traverse( v.next );
    path.leaveSequence();
  }
}
```

Listing 4.22: Traversing a structure tree

The first thing that happens is that the element is discovered, and this is recorded in the path in the line marked with (†). The preprocessing hook is then called with the path that now includes the element, before any of the children are considered.

Before entering the children, the level of the path is increased so that all children rightfully will appear as subelement of this one. The line at the note (‡) then process all the children by calling the *traverse* method recursively on the first child in the sequence, before the path is then restored to the parent level.

The tail is then handled by calling *traverse* recursively at the parent level with its first node. The rest of the sequence is processed in an inductive manner with the **`null`** sentinel being the base case. In this case it is actually more convenient to use recursion than a loop, since there are further actions to be done after the tail has been traversed.

In order to only have one node at a given level "active" at a time, postprocessing must however be done *before* the recursive call so that it is completed when the next sibling is entered for preprocessing, as can be witnessed at the line marked with (§).

After the sequence is handled, the front tag is removed from the stack returning the path to its original position. Although superfluous in traversal of value carriers alone, this step should generally be done so that if traversal is done in conjuncture with types, it returns to the original state allowing the value to be retested against another type union element.

Reversal must not be done straight after postprocessing before the recursive call to the tail, because the label of the element must be on the stack for the index count to be correct. In contrast to the processing events, the trails of the sequences on the stack is hence nested.

4.3.4 Use in relations

The framework should keep track of how the type carrier is traversed so that in case it determines that a given relation between two types does not hold, the location of the erroneous construct can be reported for the user to inspect. The reader should recall from section 4.3.1 the discussion of path indices validity in schemata. Generally however, the error will be reported at the first incident in the schema due to the loop detection, and this facilitates bug hunting. A field is hence added to the relation to hold this current path, as is shown in listing 4.23 below.

```
// class Relation
Path currentPath;
```

Listing 4.23: Current position of traversal in a relation

To update this path when a type carrier is decomposed and its various parts are invoked for relation testing, the code must be changed to use wrapper methods around *isIn* that performs this task. The signature of these methods are the same as the original, so they are interchangeable at source level. This thesis will only describe these methods and outline their use. For a full examination of how the framework is changed, the reader is invited to review the code itself.

The first wrapper is called *isInForRoot* and is used to determine the relation between two top-level type carriers, i.e. this method should be invoked from outside the framework to bootstrap the testing and is not used by any of the internal code. Listing 4.24 presents this method. It creates a new path that is initialized to the root of the type structure tree. (Recall from listing 4.18 on page 97 that the default constructor of the *Path* class represents the *root* constant). After this new object has been assigned to hold the current path, regular traversal is started at the first element by calling *isInForSeq*, which is another wrapper described in a moment.

```
// class Relation
boolean isInForRoot( Type t, Type u ) {
  currentPath = new Path();
  return isInForSeq( t, u );
}
```

Listing 4.24: Wrapper for testing root elements

This method may also be renamed to *isIn* to preserve source compatibility since it is now intended as the external interface of the relation.

If the framework decompose a labeled element and is about to perform testing on any of the child element, the path must first be updated to indicate that another level has been entered. This is done by employing the *isInForChild* wrapper displayed in listing 4.25, which simply increase the level before handling the request to the child sequence, and restores the old level afterwards. The child elements are also handled by the *isInForSeq* wrapper, which deals with sequences.

```
// class Relation
boolean isInForChild( Type t, Type u ) {
  currentPath.incrementLevel();
  boolean result = isInForSeq( t, u );
  currentPath.decrementLevel();
  return result;
}
```

Listing 4.25: Wrapper for testing child elements

When an element that might be a basic non-empty sequence is encountered, the path is changed to reflect that this tag has been seen. This task is performed at the line marked with (†) in *isInForSeq*, shown in listing 4.26. Notice that one of the interpretations of the name *inInForSeq* is a misnomer. It is so called because it handles sequences, but this is not the only thing on which it operates. It accepts any element, which is why the call to *enterSequence* must test if the type carrier class is *Label* first.

```
// class Relation
int currentDepth = -1;

boolean isInForSeq( Type t, Type u ) {
  currentDepth++;
  if( t instanceof Label )                                    //(†)
    currentPath.enterSequence( ((Label) t).tag() );

  boolean result = isInOrig( t, u );
  checkForErrors( result );                                   //(‡)

  if( t instanceof Label )                                    //(§)
    currentPath.leaveSequence( ((Label) t).tag() );
  currentDepth--;

  return result;
}
```

Listing 4.26: Wrapper for testing sequence elements

The instance field *currentDepth* indicates the depth of the recursion, since it is incremented at the start of the method and testing of every element eventually flows through *isInForSeq* (either directly or through the other two wrappers *isInForRoot* or *isInForChild*). Observe that the recursion depth is not the same as the number of levels in the path, as the wrappers are invoked on other elements than labeled sequences, such as unions and leafs. This field is initialized to -1 so that it will be bumped up to 0 at the first invocation and incremented from there.

The effect of the this bookkeeping is undone in the block marked with (§). In between, the call to the original relation framework from listing 3.7 on page 62 is executed, which has been renamed here to *isInOrig* in order to prevent a potential name clash with the new external interface *isInForRoot* should the latter be renamed to reflect that, as mentioned above. A method *checkForErrors* that does error handling has been inserted immediately after the outcome of the test is available, in line (‡), before the old path is restored further down.

The result of the test is given back to the caller through the return value, and ultimately this will end up in the application code that made the first call to the framework. However, the return value is only a thumb-up/thumb-down response that does not give any further information about the cause of failure. To alleviate this, the path to the first part of the type carrier where non-compliance to the requirements set forward by the relation is detected, is also saved. If the relation returns **`false`**, this path can then be retrieved through a property.

Hence, two fields called *failurePath* and *failureDepth* are added to the relation, and these fields will hold copies of the value the progress indicators *currentPath* and *currentDepth* (from listings 4.23 and 4.26) had at the time of the failure. The fields are initialized to default variables indicating that no errors has happened yet. Listing 4.27 holds the code that handles failure reporting, and the line marked with (†) shows what the state of the error tracking is before entering this routine.

```
// class Relation
Path failurePath = null;                                    // (†)
int failureDepth = -1;

void checkForErrors( boolean outcome ) {
  if( !outcome && failureDepth < 0 ) {                      // (‡)
    failurePath = currentPath.clone();
    failureDepth = currentDepth;
  }

  if( outcome && currentDepth < failureDepth ) {            // (§)
    failurePath = null;
    failureDepth = -1;
  }

  if( !outcome && currentDepth < failureDepth ) {           // (¶)
    failureDepth = currentDevel;
  }
}
```

Listing 4.27: Error handling

As mentioned before, this method is designated to inspect the result from the relation testing and update the error tracking accordingly. The parameter *outcome* indicates if the relation was found to hold or not between the two type carriers passed to the relation. If this variable is **`false`**, an error occurred.

If in addition, *failureDepth* is simultaneously still at its default value no error is currently registered, so this would be the point of origin of a new error. The error tracking variables are then assigned to the state of traversal at this point, which is done by the block at (‡).

Registering an error does not necessarily imply that the entire relation check will ultimately fail. The outcome on one level may only be one of many checks performed to test various premise conditions and will eventually be overturned if another branch is able to fulfill the rule. Consider the rule $[TRY]$ (cf. figure 4.12 on page 91) where a value only have to validate against *one* of the elements in the union and may test false against the rest.

The expression $currentDepth < failureDepth$ is true if traversal have reached a parent level of where an error was registered. (Note that if an error is not registered, this test will never yield true). A positive outcome on such a lower level should clear the error from upstreams, and the block (§) will do this by resetting the error tracking fields to their default values again.

However, the error cannot be eradicated by any positive outcome at a lower level. Only direct recursion ancestors should be able to do this. Otherwise, a "cousin" test that is just not as deep could inadvertently reset the tracking variables since its level counter happens to be lower than the branch where the error first appeared, with the result that they are lost after the parent rule have evaluated all its premises. Figure 4.28 illustrates this.

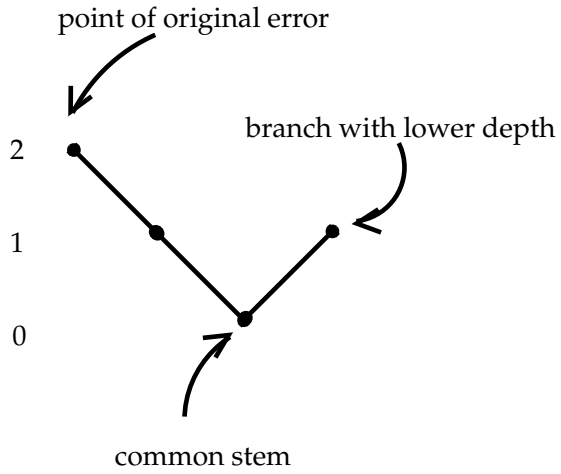


Figure 4.28: Need for error level escalation

To prevent this, the error must be *escalated* to the parent level when the rule return. The failure level is then bumped down to reflect that the error has now caused this level to fail too since the rule did not find any other premise that could be used instead.

The last test in the error handler — the block marked with (¶) in listing 4.27 — does this escalation as negative outcomes trickles down the recursion ladder. $failureDepth$ are the place in the *code* which has failed due to the error, whereas $failurePath$ points to the corresponding location in the (document structure) *data*. When an error is encountered, the code recurses back to the origin but traversal of the data halts. Therefore, only the depth variable are updated in an escalation and not the path.

4.4 Summary

Documents are trees where each internal node is labeled with a tag. Character data and attributes, entities that are common in standard markup languages, can also be modeled with these basic elements. Documents structure data.

Validation is the process of determining whether a document is within the language described by a schema. It can be done in three ways: with an explicit language rela-

tion, by a recursive algorithm with cycle detection or by reusing the subtype relation in conjuncture with the minimal type inferred from the document. Validation tells if the document has the correct structure.

A path is a way of addressing a branch of a document in an instance-neutral manner. Paths are built by specifying how the document should be decomposed and navigated to retrieve the desired node. They can be used to pinpoint the location of where the document break the structure, or to express which parts of a structure that has been encountered.

Chapter 5

Compatibility

In the initial design process, schemata are usually written prior to the introduction of documents in the system, as the schemata are meant to provide structural descriptions and must be available at the time of document validation.

However, deployment of the system and its use give feedback to another round of design in which the assumptions have changed somewhat from the initial phase. Hence, a desire to change the schemata arise, but a total redesign and redeployment of the system which involves reviewing all documents may be infeasible from an organizational and economic viewpoint.

This chapter explore the options available and the considerations necessary — both in the framework and in the schemata — to do such incremental upgrades to the content structure.

5.1 Schema-oriented solutions

This thesis will approach the problem of structural changes with a focus on reasoning upon the schema types, and attempt to limit the number of constraint put on the writing or the usage of the schema. In other words, it will seek to obtain a solution that involves little manual labor from neither the end users nor the users of the frameworks.

The problem is thus to ensure that the new language is a superset of at least the subset of the old language that was actively used, i.e. that the new language still contains all documents entered. Future documents must then relate to the new schema, so the parts of the old schema that is not used is no longer of any concern.

5.1.1 Extension and removal

If the modifications to a schema consists of extending it to accommodate new uses that was originally overlooked, the subtyping relation from section 3.4 can be used to find out if the type carrier that represents this new schema still will validate all the existing documents, by simply comparing it to the old. If the old schema is a subtype of the new, only additions have been made, allowing a larger set of documents in the language.

But, if the action taken is not just to extend but also to remove elements, then this simple procedure will no longer be sufficient, as the full language of the old schema is no longer a subset of the new even though the documents in the residual has never been entered into the system.

Since a modification can be seen as first a removal and then an extension in a single operation, any operation that allows an existing part of the schema to be altered should be prepared for the possibility of rendering the new schema incompatible.

Destructive changes may be necessary on several grounds: Parts of the schema could have been included at the wrong places because the understanding of the problem domain was not good enough, or it might be that use has shown that they were never necessary and it is now desirable to clean up the schema from old cruft. The design process can never be expected to find the perfect match at first try, so the framework must give provisions to rectify such errors.

5.1.2 Brute force

Evidently, one possible solution to the problem is to use brute force: When a new schema is proposed, all documents that was entered into the system (for the old version of that schema) is completely revalidated. If all documents passes this test, then the language of the new schema at least contains what was actively used from the old and can take its place.

This thesis does not hold brute force to be a viable option due to that the complexity of the algorithm will increase with a magnitude. The order of the validation procedure will incur for each and every one of the documents. In a real system, there is also typically many more documents than there are schemata.

On these grounds, this approach is dismissed and it is mentioned here merely to show that it is in fact possible to ensure history adherence, and to establish an upper bound on the runtime.

5.1.3 Type-based statistics

A more suitable option seems to be to leverage the procedure used for extension and make it also handle reduction of the schema. Hence, the subtyping relation is modified to take into account only the parts that has actually been used by documents, when testing for schema upgradeability. Instead of always terminating the deduction when a mismatch between the old and the new type carrier is found, it should be done only if there exists a document branch that is in the language of the type carrier in question.

If no such branches are present, then this mismatch could safely be ignored because none of the old documents would fail validation because of it, and all new documents are validated according to the new schema from this point on anyway.

Usage of unions in the schema is what makes part of it optional for documents. When the validator enters the [TRY] rule (cf. figure 4.12 on page 91), only *one* of the elements in the union have to be satisfied in order for the value to pass. Each of these branches

identifies a subset of the type's language, and the documents in such a subset are the ones that require the corresponding branch of the union to be true during validation. Conversely, if a branch never yield true for any of the documents that are added to the repository, then no documents in the subset of the language that it represents are present.

Note that more than one branch may give a positive result for a given document, meaning that their subsets are overlapping. For instance, the example

$$a[\epsilon|x] \quad \Big| \quad a[\epsilon|y]$$

will match in both branches for the document $a[\epsilon]$, meaning that this document is in the language of both branches and that these consequently must be (partly) overlapping, whereas the following type

$$a[\epsilon] \quad \Big| \quad a[x] \quad \Big| \quad a[y]$$

will only match the document through the first union element. The languages of these two examples are the same.

Each branch will therefore be evaluated individually for inclusion regardless of the others, and the rule [SPLIT] (cf. figure 3.20 on page 72) modified accordingly to no longer require the right side (the new schema) to match all the elements on the left side (the old schema) but only those that were invoked during validation. The union elements that are essential for the history of documents to match are in some way marked while the others are discarded. A related method is described in [BCF97] where only the automaton states that has been visited in a bottom-up validation are kept.

The approaches discussed in the rest of this chapter all consider how in various ways to mark the parts of the schema that is used by documents. The question is always how to identify the type carrier in order to know that it should be included in a check for upgradeability. In this context the modified subtype relation may also be referred to as the *compatibility* relation or the *upgrade* relation.

An alternative is to create the minimal type necessary to describe all the documents in the repository. Upon insertion of a new document, its minimal type is inferred and then put in union with the minimal type of the schema that is being built incrementally. This is perhaps the most theoretically sound method, it has yet the deficit that the type algebra used in the framework does not compact the carrier according to rule (D) on page 70. As a result, this approach may end up with an inflated minimal type that grows linearly with the number of documents and would give the same runtime performance as revalidating all the elements brute force.

5.2 Name-based approaches

At first glance, the intuitive solution seems to be to give each type carrier some kind of identifier. Records are then kept of which types are "touched" during validation, by collecting the names of these types into a list that can be stored. When the upgrade check

is done, all type carriers that are *not* on this list may be ignored. However, the problem has now been rephrased into how a name of a type should be chosen.

5.2.1 Explicit naming

The easiest solution for the framework is to put the burden of selecting names on the user. Every time a type is created, it must also be given a name. This name then becomes the permanent identity of the type and the instance that is created is only an *incarnation* of an object for this particular run of the program.

In many modern schema languages such as XML Schema and RELAX [TBMM01, CM01], the user is given the opportunity to define and label types explicitly. The type can then be reused by referring to it by this name. This is in contrast with the older language DTD [BPSMM00] where types are always defined inline.

However, not all types are accessible for naming. Only some of the carriers of which the schema is made are created explicitly by the user, while the rest are the results of operations performed on the others. The group of carriers that is specified inline without being tagged is called *anonymous* types.

For example, consider the composite type (To refresh the definition of the question mark, the reader is referred to section 2.1.3 on page 2.1.3):

$$a,\ b?,\ c$$

As $b? = \epsilon|b$, the last concatenation will be distributed over this union. The expression will therefore result in the internal representation (observe that the language of the expression is $\{ac, abc\}$):

$$a,\ \Big(\ c\ \Big|\ (\ b,\ c\ )\ \Big)$$

If no documents has the branches that is defined the optional schema element b, the last union element above (the inner parenthesis) is not needed during validation. That element was created as a result of the concatenation operation though, and is not assigned any name. Hence, there is no key under which to record its presence or lack of it.

Additionally, the existence of such accounts in the schema indicates that there may be carriers beneath the level of abstraction on which the user operates, making it hard to come up with meaningful names for them anyway.

5.2.2 Generated names

At a glance, it is tempting to argue that it is conceivable to create a solution where the name of the stem being decorated is combined with a tag indicating the decoration created. For every operation that creates a new type carrier, the name of the argument(s) must be combined with information about the action taken, e.g. if a type called “b” is decorated to be optional, then the resulting type could be named “b?”.

If the system has to generate some of the names it might as well generate them all, removing the burden from the user. Naming of types can then be made an optional feature, and when a types name is reported from the framework, the internal name is first attempted mapped to the user-specified label if present.

The name of the initial type carriers — the types that are the result of a constructor being invoked by the user — must then be generated from some external measure since constructors of course do not take any arguments from which the name can be based. Two candidates for such automatically generated names are (i) Globally Unique IDentifiers (GUIDs) and (ii) definition paths.

GUIDs are essentially serial numbers [LS98] usable in distributed systems. Definition paths can be used in systems where the schema themselves are also being described using structured documents. The path in this document to the branch that defines the type can then function as its name.

The advantages of these two methods are that they work with anonymous types. The downside are that they are both *volatile* in that a small change in the schema may cause many types to get a different generated name even though they were not directly affected by the change otherwise, as both methods deal out names in a sequential manner. When the schema is changed, types may get a different name from what they had in the original.

5.2.3 Change management

A third option is to employ the user-defined names whenever possible and then use automatically generated names only for anonymous types. This alleviate the problem somewhat since the manually given names are typically more *robust* and name generation can be framed to within the nearest such labeled block. Although it lessens the impact by making changes affect only local anonymous types, it is still short of being a general solution.

If the user-defined names are changed, the statistics that are gathered on the old names would have to be updated correspondingly. Otherwise, the first upgrade would still work since the original names would be used by the compatibility relation to determine which parts to ignore, but any further upgrades could not be based on the statistics because it would then be out of touch with the names that were used in the new schema. Such a hybrid approach will thus require that the names picked by the user must remain fixed for the lifetime of the schema, restraining flexibility in schema authoring.

Furthermore, even though the names are held constant, there may not necessarily be a homomorphism for each type in the old schema to a corresponding type in the new schema, even though the schemata are compatible. Consider an example where an element is "split" by lifting a union to a higher level:

$$a\Big[\ b_1 \mid b_2\ \Big] \quad \sqsubseteq \quad a[\ b_1\ ] \ \Big|\ a[\ b_2\ ]$$

Upgrading the statistics would involve transferring the counter for the element on the left side to the counters for the two elements on the right side. Alas, the distribution of

this counter cannot be decided without taking into account the number of times the types b_1 and b_2 have been used within the type a, respectively. Upgradeability of counters based on types can therefore not be regarded as transitive.

Being unable to perform upgrades like this severely impairs the ability to refactor the schema, and this thesis therefore considers it unacceptable to put these constraints on the user. Conclusively, types can be addressed but recording schema usage this way is insufficient to determine upgradeability by modified subtyping.

5.3 Path-based approaches

Another way to take actual schema use into account than identifying the type carriers employed in validation is to look at the other side of the proverbial coin and do this based on the document branches used, represented by their paths. This works by first recording all paths of the documents in the repository and then in the compatibility check only invoke the parts of the grammar that covers branches with the same paths. As will become apparent in a later section, this can actually be more flexible that basing the cut-off on type identity since the same type may be reused in other, unrelated places in the schema.

When a rule is invoked with a left-hand side argument (the old schema) whose path is determined not to be used, the right-hand side type (the new schema) is said to be *acquitted*, and the return value will then be positive as if the pair did satisfy the subtype relation. This will work for recursive carriers too, since the outcome for them will be determined at the first encounter. If the path $/a/b$ is not present, then the path $/a/b/a/b$ will not be either. If the result for the pair is subsequently cached, the acquittal is said to be *global* since it then applies to all instances of that pair, whereas if it only applies in this particular evaluation it is *local*.

Due to the loop detection built into the framework, the gap between type and path determinability (i.e. types can be reused while a path only identifies one branch in the document, even if the value carrier is reused) turns out to be an obstacle that hinders this solution from being effective.

5.3.1 Global acquittal

If the relation between two arguments is reported to hold based only on their lack of occurrence in one particular path, the acquittal is over-generalized as the type may also occur other places in the schema not related to that path. Take for example the schema

$$a[\ \epsilon \mid b\ ],\ (\ \epsilon \mid b\ )$$

Observe that the type carrier inside the bracket parenthesis is the same as the one in round parenthesis. Even if the same instance object was not used for these two types, they will be considered equal by the framework.

If there are no documents containing a branch with the path $/a[1]/b[1]$, the type carrier b will be acquitted when the contents of the first element is checked. However, the not-in-use status does not necessarily apply to the path $/b[1]$. Upon return to the top-level, the check for presence of a corresponding type to the second sequence term in the new schema would then faultily be left out. As a result, the new schema would pass even though it was not compatible with the documents in the repository, which is clearly unacceptable.

5.3.2 Local acquittal

Since global acquittal is unsuitable, the other option is to filter out the types that is not used in any document before they are cached. The deduction framework will then ignore them as if they never existed in that particular path, but they will be revisited if they occur later, in another.

The problem with doing this is that although acquittal is local for this particular reference to the type in the schema, the outcome will propagate to the parent if it is only performed isolated in a level with no further book-keeping. The following example illustrates this:

$$a[\,\epsilon \mid b\,],\ a[\,\epsilon \mid b\,]$$

Notice the resemblance between this example and the one from the previous section, only that the union appears as a child of the second sequence element just like the first. The content of both elements are optional, so it may or may not appear in documents, independently.

If there are no paths $/a[1]/b[1]$ registered in the repository, then the type b will first be locally acquitted at that point during deduction and the outcome is as said not cached. However, this may cause the deduction rule that is evaluating the parent type a to have all its premises fulfilled and consequently the result for a is put in the cache, even if the replacement type for a's content in the new schema does not allow b at all.

At the path $/a[2]$, the cache will "remember" that the type a already cleared out and acquit it at this point too. However, a check for the existence of the path $/a[2]/b[1]$ has now been forgotten.

Although the error in the previous paragraph lays in the non-deterministic nature of type paths (cf. section 4.3.1), dropping the indices from the path does not help either, because a prefix in the path may still change. E.g. if each element of the sequence is again wrapped in elements, but this time with different labels:

$$x\Big[\,a[\,\epsilon \mid b\,]\,\Big],\ y\Big[\,a[\,\epsilon \mid b\,]\,\Big]$$

Now there might be that there (disregarding indices) exists no path $/x/a/b$ but that a path $/y/a/b$ is present. With local acquittal and no feedback, the type representing a would be cached in this example also, since the prefix of the path now functions as a discriminator instead of the index.

Feedback could have been introduced by combining the result value with a flag that told whether it was caused by lack of relevant document (branches) instead of a deduction rule. If a mandatory premise returned with this flag set, the parent rule would become "tainted" and would have to set the flag itself to avoid being erroneously cached. Such a design could jeopardize the loop detection if the argument pair is removed from the cache before the framework is entered; it must rather be entered in advance (as is done by the rule [ASSUM]) and removed afterwards if the evaluation has been tainted.

This feature still requires that a large number of paths to be recorded, and it somewhat deprive the framework of the ability to cache relationships unless the test is done *after* the deduction rules have otherwise failed (so that the lack of presence overturn a failed result rather than preemptively mask it out before testing). It requires an extensive overhaul of the framework core to accommodate the extra flag in the return values and for that reason the code that is affected will not be reiterated here. Diligent readers interested in details are referred to the source code. For a lighter treatment, the discussion in section 5.4.3 can be followed analogously with necessary adjustments for cache effects.

Future work should further explore ways to increase the time and space efficiency of this method, as it looks very viable for the task at hand.

Instead of employing the full path of all branches, both the amount of history stored and the problem of identical types being present at branches with different prefixes can be partially solved by looking at *suffixes* instead, which is the topic of the next section.

5.4 Context-based approaches

A *context* is loosely speaking a description of the environment in which a given element is present. Approaches based on contexts are hybrids that combine type-based and path-based compatibility checking by generating a context representing the type, from parts of the path. The aim is to achieve the efficiency of a type-based check, and the robustness of a path-based check.

The context is built from a suffix of the path. Each term constraints the kind of element that has occurred at the corresponding level by applying the restriction that it must have the given label. Hence, the labels are used as an estimator of the type. A parent narrows the available candidates in which a child can take place.

The *granularity* of the context is determined by the *depth* of this suffix, i.e. how many terms that is included from the end of the path. The more constraints, the fewer branches in the document will actually match the suffix. Optimally, a context should describe all the branches that are validated by a type, and only that type. This thesis will only consider contexts with depths of one and two.

5.4.1 Aptness

Using a context with a depth of one is the same as recording all the tags that are used in the documents, and then only acquit a labeled element if its label has not been seen at all.

This works very well if there is close correspondence between the label and the type, and will be a perfect match if there is only one type for each label.

Recall from section 2.2.6 (on page 31) that grammars of the subclass LTG possesses this property. The type is defined in terms of the label. This implies that for schema that is written in the meta-language DTD [BPSMM00], recording a counter for each possible label suffices.

However, for grammars that are outside of this class, the chance of getting a *spurious miss* is high. A spurious miss is when the type is reported to be in use and can therefore not be acquitted whilst there in reality are no documents that would contradict the new schema, i.e. a branch has marked more than its one real type. Using only one term from the suffix gives very coarse contexts so the chance of this happening increases.

For meta-languages that has more expressive power, a context of larger depth should thus be considered. Note that if only a subset compatible with DTD is used of the grammar, the resulting schemata will still be within the subclass LTG, and this may therefore be the case for schemata originally defined in DTD.

Another subclass of interest is STG (cf. section 2.2.6 on page 31), where a label is guaranteed to be unique within the same type. In other words can the type be inferred from the label if the parent type is known, in grammars of this class. This class is interesting because it is the foundation of the meta-language XML Schema [TBMM01] which despite its drawbacks has gained industry support and is poised to overtake DTD in usage.

The problem of finding the parent type still remains though, and the solution is to attempt to estimate it from a context. Hence, more terms are drawn from the suffix so that information about the parent is also included.

Inductively, this reasoning can be repeated until the context consists of the full path, reverting to local acquittal as described in section 5.3.2. In practice, most of the time the accuracy of the parent type can be traded off for a short context since overloading of labels is not that common. Estimating the parent using a single level may be enough, meaning that the context has a depth of two terms: The parent label and the current label.

Context-based testing is no panacea. If for example path $/x/a/b$ is neither used nor supported in the new schema (and therefore due for removal) but path $/y/a/b$ is, then a context (a, b) will be recorded and cause a spurious miss in the resulting failure to acquit the former type.

5.4.2 Recovery

Spurious misses are errors that are the result of being over-cautious. If the framework is in doubt, it should rather dismiss a schema that could have been entered on the grounds that it has not enough evidence rather than taking the chance of allowing an incompatible schema to enter the repository, potentially corrupting its integrity.

This section explore the options for amending the algorithm in order to recover from spurious hits without manual intervention from the user. When a spurious miss occurs, the documents that contains the context in question must be validated brute force against the new schema and the compatibility testing must then resume as if this error never

happened. To determine the subset of documents that must be checked, a *reverse lookup* is performed on the context, assuming that the repository not only contains an inventory of the contexts that has been seen upon entering documents but also a mapping from each context to the document in which it was found as well. Since the context was marked as being in use in the first place, such a reverse lookup will always return a non-empty set of documents.

Attempting to do validation at the spot where the error first arose is not very efficient. Many errors are premises that fail at one level but from which a rule at a lower level recovers as another premise fulfills its requirements (cf. section 4.3.4 on page 103). Hence, performing validation once a rule fails will do much more work than is really necessary.

To determine if the error will make the subtyping algorithm ultimately fail, it must be allowed to propagate to the top level. At this point the documents that are identified by the reverse lookup can be validated, and if all of them is accepted by the new schema the compatibility test should resume from where it was first stopped. However, restarting the framework from a point deep inside the recursion involves capturing the memento [GHJV95] of all objects in the deduction to restore this state later, which is a non-trivial task that this thesis makes no attempt to accomplish due to the amount of effort entailed.

An easier implementation is to record the path of failure in a *blacklist*, to which the algorithm has access. The compatibility relation will ignore errors that happens on paths that are found in this list. After a successful validation of all document that has the context built from this path, the compatibility relation is restarted from scratch again, working its way back to the point where the previous error occurred. This time it will be ignored since it is on the blacklist, and the upgrade check can proceed.

While relatively straight-forward to code by simply letting the presence in the blacklist also yield a positive result, this method is far from ideal as it may require several restarts, having to perform multiple passes through the schema. Nonetheless, it does not perform more validation than is required (by a context-based approach) if a list of the statuses for all documents that has been validated is kept and consulted before attempting to do so again, to prevent documents containing more than one doubtful context being tested more than once. A snag is that it suffers from the deficiency of erroneously propagating success to the parent, unless a flag indicating locality is passed together with the result value.

5.4.3 Integration

In order to integrate context-based acquittal with the compatibility checking, the framework must be modified to also take the statistics from the repository into account when determining the result value of a deduction. Rules that would normally return `false` will have their outcome overturned if the context is not marked as used.

The best place to do this seems to be immediately upon return from the cache wrapper around the deduction rules. At this point, the path is still current for the type being tested and all the rules have been evaluated so the outcome of the subtyping is known.

Listing 5.1 on the next page shows how the code for performing acquittal is injected

into the framework by rewriting the method *isInForSeq*. The result value is no longer determined by the deduction alone but also by the statistics gathered from the repository. If the context that this type is used does not exist in any document, the compatibility check should succeed regardless of what the rules deemed. Similarly, paths that has been entered on the blacklist could be acquitted because all documents containing their context have as suggested by the previous section been verified separately, allowing the client to override any error and let the framework continue.

```
// class Relation
boolean isInForSeq( Type t, Type u ) {
  ...
  boolean result = !existsInContext( t ) || isInOrig( t, u );
  ...
}
```

Listing 5.1: Only considering previously seen contexts

Note that the order of these two tests are significant as the compiler will short-circuit the evaluation if possible, meaning that the rule deduction routine that alters the cache may or may not run depending on the result of the other test. It is done in advance so that the subtyping will not be affected by acquittal based on contexts. Performing a lookup in the context database is about as quick as consulting the cache for types, so it is not necessary to enter acquittals into the cache. (Context lookup is however only a supplement and can not comprise a compatibility relation substitute).

If interaction with the cache is desired — for instance to roll back failures that are acquitted locally — the logic must be moved inside the *isInOrig* method itself. A full treatment of this variant is considered too extensive to review, and is available in the code only.

Deciding if a context exists in the repository is done here by querying a collection, which can either be loaded in advance or be a dynamic view to the database. The context sought is created on the fly using the current path. Listing 5.2 depicts this setup.

```
// class SubRel
boolean existsInContext( Type t ) {
  if( t instanceof Leaf || t instanceof Label )
    return contexts.contains( new Context( currentPath ) );
  else
    return true;
}
```

Listing 5.2: Query context dictionary

Only labeled and leaf elements are tested for their context, as types of these classes are the only ones that will add something to the path. Unions on the other hand, do not contribute to the path themselves but rather functions like containers for other types that do. A union could be thought of as containing a *set* of paths. They are considered to always be present, to force introspection of each of their elements. Therefore, **`true`** is unconditionally returned in this case.

Context encapsulates the code that retrieve proper information about the environment from the path. It can be considered a view of the necessary parts. Additionally, contexts are unlike paths immutable, so they can be passed around easily like value objects. To accommodate changes in the depth of the context, only the implementation of this class will have to be modified. Listing 5.3 shows a realization of a context with a depth of two where the parent and the current term is extracted from the path to build a suffix string.

```
class Context {
  final String suffix;

  public Context( Path path ) {
    suffix = blankIfNull( path.getParent() ) + "/" + blankIfNull( path.getCurrent() );
  }

  private static final String blankIfNull( String s ) {
    return s == null ? "" : s;
  }
}
```

Listing 5.3: Anatomy of a context

In addition to the method of construction, it must also provide the standard methods of comparison, *equals* and *hashCode*, in order to do efficient lookup as well as methods for storing the context externally. These can trivially be implemented based on the only instance field.

5.5 Summary

Upgrading a schema does not need to involve revalidating the entire repository of documents, but can instead be done by reasoning about the relationship between the old and the new type. The subtyping relation will detect if the new schema is an extension of the old. To allow reductions, it must be verified that no document depends on the parts being removed. This is done by modifying the subtype relation to acquit types that are unused, regardless of their presence in the new schema.

A type-based approach for identifying unneeded branches is theoretically correct, but will put unreasonable restrictions on the changes that can be made. A path-based method will work provided that the effect is kept local although it will lead to data inflation and lower efficiency. Context- based acquittal is not perfect as it may give spurious misses, but constitutes an acceptable compromise for the schemata encountered in practice.

Chapter 6

Exterior

Conversation between two systems require that they both use the same format for interchange of data. Standardization of this format facilitates interoperability amongst a variety of solutions without creating tight bindings, and lowers the cost of integration as tools are generally available instead of having to be custom-made.

This chapter explore the integration of the framework presented in this thesis with such a standard format for both documents and schemata, and the host language's runtime library package associated with it. The format selected is the industry-wide supported eXtensive Markup Language (XML) [BPSMM00].

6.1 Syntax

XML is devised to outline how a document should be parsed from a character stream, as discussed in section 4.1 on page 83. It does not describe a concrete set of documents itself, but is rather a meta-language that sets forward a common, basic syntax shared by a family of languages. In conjuncture with a schema that provides the valid tags and structure, a complete grammar is formed. The separation of work-load between syntactic and semantic processing allows components to be better specialized to their task and users to leverage modularization.

6.1.1 Elements

Up to this point, this thesis has employed the notation that was introduced in section 2.2.1 on page 26, where square brackets have been used as markers. XML on the other hand, uses angle brackets and encapsulates the tag rather than the content. Thus, the tag is integral to the marker. Instead of ending the element with a simple character, the tag is repeated in the end marker to ease detection of an imbalanced tree, but with a slash prefix indicating that the element is being closed. Hence, an element with the tag a will look like the following:

$$< a > \cdots < /a >$$

where the three dot ellipsis is a placeholder for the content of the element. If there is no content, the element can be abbreviated to a single tag but with a slash *suffix* to specify that this is a start and end marker in one. E.g.:

$$< a / >$$

The advantage of specifying elements this way is that tag characters can never be mistaken to be free-form data, as they are always delimited by the angle brackets. This could however also be accomplished by starting tags with an escape character so it is not particularly space efficient. It has also the same short-coming as most other formats that some characters have a special use and are prohibited from occurring within the data.

6.1.2 Attributes

Besides putting child elements between the start and the end marker, XML also allows a particular form of children called *attributes* to appear inside the start marker with an assignment expression syntax. The tag of the child is then used on the left side of an equal sign and the value of the child is put inside quotes on the right. Here is an example where the element labeled a has two attributes named b and c respectively:

$$< a \;\; b = " \cdots " \;\; c = " \cdots " > \cdots < / a >$$

The content of an attribute is called its *value*, and attribute values are limited to only contain character data. No structure is allowed, and it is therefore an error if an angle bracket appears inside the value of an attribute. Attributes are always terminal nodes in the structure tree.

6.1.3 Special characters

In order to use the special characters that are part of the language syntax inside free-form data, an *escape* character is employed. When the parser encounters this character, the next token is used to determine the replacement of this *entity*.

In XML, the escape character is the ampersand, and the next token is terminated with a semi-colon. This implies that the ampersand character must itself be escaped to appear in the text. Table 6.1 contains a list over the special characters which are essential for the basic syntax and their corresponding escape codes. Note that none of the characters that ends the commands such as the right angle bracket and the semi-colon) *must* be escaped.

<	<
"	"
&	&

Table 6.1: Escape sequences

Other entities may also be declared, but this feature is not necessary in the context of the framework presented here and is therefore considered to be outside the scope of this thesis.

6.2 Meta-schema

Schemata are anticipated to be written using the XML Schema Description (XSD) language [TBMM01]. Although neither as powerful as the academically founded RELAX NG [CM01] nor as widespread as the older DTD [BPSMM00], it is the recommendation from the standard organization W3C and is rapidly being picked up by the industry. The selection of this schema language does not exclude other adapters to be written for the framework.

XSD is — unlike DTD — of the XML family and a schema can therefore serve a dual role as both a document and a schema. It can almost be used to describe itself, and some of its quirks make it indeed necessary to use a language more powerful to capture it completely. It is in this author's opinion quite intricate and not for the faint of heart.

Implementing the entire standard would be a Herculean effort, and this thesis will therefore focus on providing support for a selected subset only. It contains only the basic components (elements and attributes) and the provisions necessary for recursion (named types) and rudimentary support for reuse (named groups).

Funnily, a feature of regular tree grammars that is *not* found in XSD is required to describe the subset to compensate for some parts that have been removed. Consequently, not all XSD schemata will be understood by the code presented here and some grammars that are not proper XSD will be understood, but it is attainable to write schemata within the subset that is usable with this as well as other validators. Specifically, this applies to grammars converted from the older language DTD.

The schema is presented in its own syntax, implying that either prior knowledge of XSD is required or the reader must possess the intuition needed to "bootstrap" an understanding of it. Only listings that illustrates a concept in the explanation is selected, while parts that are redundant for that purpose are not included. The walk-through of its various parts also serves as examples of its use. This section is not intended to be a general tutorial of XSD however, but rather to serve as a reference of which constructs that is carried on to the subset. For a more comprehensive introduction, [vdV01] can be read. The purpose is to deliver an insight of the schema that will be useful in comprehending the workings of the semantic builder that is shown in a later section.

The inexperienced reader is advised to follow the explanation while only paying loose attention to the listings first, and then review the listings more rigorously afterwards. It may also help to take a glance at section 6.2.4 before proceeding. Figure 6.2 on the following page provides a logical overview of relationships between the units in the subset.

All schemata must have the root element `<schema>`, and this is the only element that is allowed at the top level. The namespace of the element should be set to signal that a schema is being defined, and an attribute called `targetNamespace` is used to hold the namespace in which documents of this schema should be put. Note from listing 6.3 on the next page that the namespace of the meta-schema is the same as is being defined, i.e. the schema defines itself.

A schema may contain a variable number of type definitions that define content models which can be reused elsewhere, and global elements that are start productions for

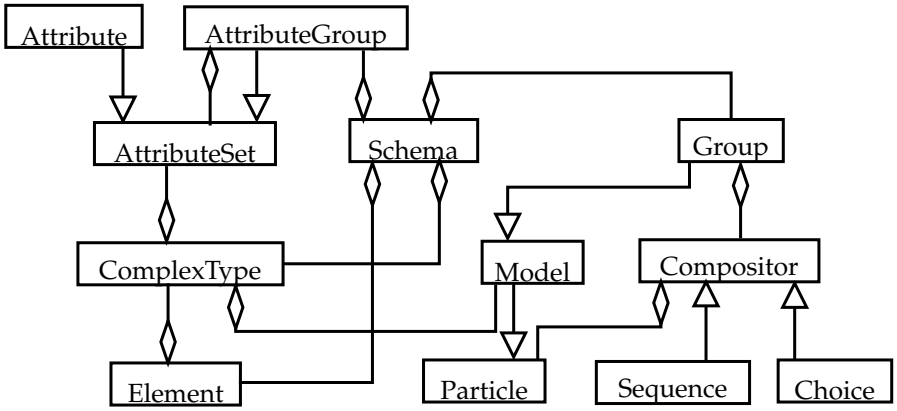


Figure 6.2: Logical view of XML Schema subset

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.w3.org/2001/XMLSchema">
  ...
</schema>
```

Listing 6.3: Root node of a schema

the grammar. As listing 6.4 displays, the starting element in XSD is labeled `schema` and its contents is an unbounded closure of a union between `complexType` and `element`, which represents type definitions and global elements, together with `group` and `attributeGroup`, which allow element and attribute entities, respectively. Each of these will be reviewed in the following sections.

```
<element name="schema">
  <complexType>
    <choice minOccurs="0" maxOccurs="unbounded">
      <element name="attributeGroup" type="attributeGroup-def"/>
      <element name="complexType"    type="complexType-def"/>
      <element name="group"          type="group-def"/>
      <element name="element"        type="element-def"/>
    </choice>
    <attribute name="targetNamespace" use="required"/>
  </complexType>
</element>
```

Listing 6.4: Schema top-level definitions

In addition, `schema` elements have an attribute called `targetNamespace` which holds the identifier of the root type that is being defined. This should be a Uniform Resource Identifier [BLFM98], although a generic string datatype is used here. As this serves only as an identifier, the actual content of the attribute does not matter beyond being unique.

6.2.1 Types

Complex types are regular tree expressions that can be used as content for elements. Elements that are not global, i.e. should not be valid start symbols, must be defined inline. To avoid copying and pasting definitions for these elements, a separate type should be defined and this type referred to in the declaration of the element.

Recall from section 4.1.4 on page 89 that a *complex* type allows a hierarchical structure whereas a *simple* type only defines the textual content. In this thesis, the only simple datatype available is a character string, and thus no provisions are made for defining other simple types in the meta-grammar. Mixed content, i.e. elements and free text interspersed, is in XSD (unlike e.g. in RELAX NG) specified as an indicator for the entire type, which will require text elements to be inserted in the content model a posteriori. However, this can be laborious and problematic and the implementation here is abridged to support only mixed content in the style of DTD, i.e. in unbounded closures, so to cover at least hypertext.

A top-level `complexType` can be seen as a named type carrier. There is however a limitation in XSD that complexType cannot stand alone in definitions in the same way as elements, but only be used as a complete content model. A related concept called a *group* is intended as a replacement for entities. However, groups can be abused in attempts to degenerate the grammar into a context-free language [LMM00b] and should therefore be treated with caution as this is generally undetectable [HMU01], but still prevented by the framework through not allowing appending operations on forward references (in accordance with the notion of wellformedness of types in [HVP00]).

```
<group name="type-def">
  <sequence>
    <group ref="model" minOccurs="0"/>
    <group ref="attributes" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
</group>
```

Listing 6.5: Type definitions

```
<complexType name="complexType-def">
  <group ref="type-def"/>
  <attribute name="name" use="required"/>
  <attribute name="mixed" use="optional"/>
</complexType>
```

Listing 6.6: Complex types

Listings 6.5 and 6.6 show the definition of `complexType`s. The definition of the the content has been factored out to a separate group which will be reused later. In addition to its content, it has two attributes on top-level: One that defines the name of the type and another that tells if mixed content is allowed. The name must always be present at top-level; otherwise the type cannot be referred to. The flag on the other hand is optional

and has a default value of `false` if not specified, meaning that the content of the type being defined must strictly consist of elements alone and no free text.

A complex type contains a *model* that defines the content model and then a set of attributes, neither which has to be present and therefore both the `attributes` group and the `model` group must explicitly be set to optional. A group on the other hand can contain one and only one compositor. Being present at the top level, they must also be given names to be referable.

Even though there are no practical obstacles, a group cannot be used as an implicit type with the group's content embedded. (However, the framework does not discriminate between how the type carrier was originally declared and will allow it without issuing a warning, if a schema that is not validated is entered).

Compositors are entries that represent a regular tree expression that may contain multiple terms, and is defined in listing 6.7. A `sequence` is a list of particles that must *all* be present, while `choice` is a list where only *one* can be, indicating a union. In regular expression syntax (cf. section 2.1.3 on page 25), sequences were specified by separating the particles by commas (,) and choices by separating them with bars (|).

```
<group name="compositor">
  <choice>
    <element name="sequence" type="particles"/>
    <element name="choice"   type="particles"/>
  </choice>
</group>
```

Listing 6.7: Terms for composite types

XSD contains a quirk in that only compositors and group references can occur at the top level in a model. Single elements are not implicitly seen as model of their own, but must be wrapped in a compositor or a group. Neither can a group reference occur within the body of a group declaration, although the only effect this would have would be to create an alias for the first group. Nevertheless, that is the reason for having `compositor` specified as a separate group that is later reused in both `group` and `model`. The latter extends the set of compositors with references to already defined groups (see listings 6.8, and 6.9 on the facing page).

```
<complexType name="reference">
  <attribute name="ref" use="required"/>
  <attributeGroup ref="occurs"/>
</complexType>
```

Listing 6.8: References to globally defined entities

Note that the group reference in a model has not only an attribute that points to the identifier, but that there also is a group of attributes from listing 6.10 on the next page that controls the decoration (cf. section 2.4.7 on page 47). The compositors also have these attributes, meaning that a closure can be specified for any model.

```
<group name="model">
  <choice>
    <group ref="compositor"/>
    <element name="group" type="reference"/>
  </choice>
</group>
```

Listing 6.9: Allowed model for a type

```
<attributeGroup name="occurs">
  <attribute name="minOccurs" use="optional"/>
  <attribute name="maxOccurs" use="optional"/>
</attributeGroup>
```

Listing 6.10: Decoration of model terms

The attribute `minOccurs` specifies how many instances of the term that *must* be found, and is used to indicate whether it is optional or not. The only allowed values in the framework are `0` and `1`, with `1` being the default. The number of instances that *can* be found is regulated with the attribute `maxOccurs`. The valid domain for this attribute is in this thesis constrained to `1` and `unbounded`. Again is `1` the default, meaning that exactly one instance is required if both of these attributes are omitted. The value `unbounded` means that there is no limit on the times the term can occur, and designates a closure. For an overview of how these attributes map to the decorations used in regular expressions, refer to table 2.28 on page 47.

Strictly, the compositors in groups cannot contain occurrence attributes, but as this would complicate the schema definition considerably, any specifications of these attributes are propagated to the instantiation of the group, where they are allowed. Custom schemata should refrain from exploiting this fact.

```
<complexType name="particles">
  <choice minOccurs="0" maxOccurs="unbounded">
    <group ref="model"/>
    <choice>
      <element name="element" type="element-inline"/>
      <element name="element" type="reference"/>
      <element name="element" type="element-type"/>
    </choice>
  </choice>
  <attributeGroup ref="occurs"/>
</complexType>
```

Listing 6.11: Particles

The contents of compositors and groups are made up of *particles* as defined in listing 6.11 which represents a regular expression of the elements that will occur in the documents. A particle can be thought of as a type carrier. This can be given as a list of elements or nested models. The nested models act like parenthesis would do in the regular expression syntax. Any number of particles may occur in the list, and each of these particles can be decorated separately be using the attributes `minOccurs` and `maxOccurs` mentioned

before.

It this definition, the subset deviates from the standard in the way element particles are described. Here, a choice between three element types with the same label ("element") is given. These forms describe elements that are defined inline, elements that are referred to, and elements whose type is referred to, respectively. To be a single-type grammar (cf. section 2.2.6 on page 31) only one content model can be specified for an element with a given label. Since attributes are put on the type itself and not as part of its model group, a choice cannot be used to express different mutually exclusive compositions of them (for a related discussion, see [SR01]). Yet, this is the prime distinctions for element declarations. Thus, a non-deterministic model is employed in the meta-schema. This feature will however not propagate into the schemata defined, i.e. it is still possible to define single-type grammars using the XSD subset.

A more lax meta-schema that in itself is compliant with XSD can be defined by adding the attributes `type` and `ref` to the complex type `element-inline` and mark both these and the existing attributes as optional, enabling this type to be assume the role as the only content model for `<element>`s. However, any inconsistent use the attributes for this element can no longer be detected by the validator, but must be done in code. Also, by increasing the number of optional attributes, validation will incur a performance hit.

During design of custom schemata, it is recommended that co-constraints like the one above is avoided and rather resolved by using different labels instead of relying on a polymorphic definition. Attributes that deals with the context of an element (such as `minOccurs`) instead of the element itself should be shunned and replaced with wrapping elements.

6.2.2 Elements

Elements can be defined globally in the schema for use as start symbols, or locally within a particle list. XSD has the ability to lift a local element to the global level through the attribute `form`, but this feature is not supported here.

A global element must have its content model defined inline. If it is desirable to enable reuse of types for global elements, a choice similar to the one in listing 6.11 on the page before could be added in the definition for `schema`, with no extra source code needed.

Models can be specified inline by a `complexType` element. If this element is omitted, then the element will get an empty model, meaning that it cannot contain any subelements nor any text and must be closed immediately using the short form described in section 6.1.1 on page 120. The label of the element will always be the name given to definition. For this reason, not only global elements but also elements that are defined inline must be named. As with other particles, it must contain the attributes `minOccurs` and `maxOccurs` to enable decorations to be specified.

Listings 6.12 and 6.13 on the next page contain the definition of inline elements. The definition of the optional content `complexType` reuses the group `type-def` from listing 6.5 on page 123. Although it may seem like a good candidate, the type `complexType-def` is not reused for this element, as it contains the attribute `name` which

cannot be put on inline type definitions. Note that the inline type defined for `complexType` within an element and the type defined for a global `complexType` is identical apart from this particular attribute.

```
<group name="content">
  <sequence>
    <element name="complexType">
      <complexType>
        <group ref="type-def"/>
        <attribute name="mixed" use="optional"/>
      </complexType>
    </element>
  </sequence>
</group>
```

Listing 6.12: Content models

```
<complexType name="element-inline">
  <group ref="content" minOccurs="0"/>
  <attribute name="name" use="required"/>
  <attributeGroup ref="occurs"/>
</complexType>
```

Listing 6.13: Inline element definitions

The definition for the type `element-def` that describes global elements is practically the same as `element-inline`, but without the attribute group `occurs`. The main body of the definition is put in the group `content` so it can then be reused by referral and the appropriate attribute mix applied to the type afterwards.

References to globally defined elements are made by specifying the name of the element in the `ref` attribute, just like references to groups. Consequently, the type `reference` from listing 6.8 on page 124 is used to these.

An element may be specified locally but reuse a pre-defined type. This has the advantage that the element will not be visible at the top level. Instead of using the type `reference`, a new type must be defined that contains not only the name of the type that will constitute the element's contents, but also an attribute for its label. Instead of using `ref` for the identifier of the type referenced, `type` is used to designate that it is a type and not a global element that is pointed to. As usual, `name` labels the element. Listing 6.14 shows such a definition.

```
<complexType name="element-type">
  <attribute name="name" use="required"/>
  <attribute name="type" use="required"/>
  <attributeGroup ref="occurs"/>
</complexType>
```

Listing 6.14: Using a type defined elsewhere

6.2.3 Attributes

Normally, attributes are defined in conjuncture with types, but they can also be specified in a reusable group to avoid repeating their definition needlessly, akin to what can be done with element groups. Where attributes can be put (inside `type-def`s), either complete definition of the attributes must be presented, or a reference to a group containing them. Definitions and references can be mixed freely. Listing 6.15 displays the group for attributes.

```
<group name="attributes">
  <choice>
    <element name="attribute">
      <complexType>
        <attribute name="name" use="required"/>
        <attribute name="type" use="optional"/>
        <attribute name="use"  use="optional"/>
      </complexType>
    </element>
    <element name="attributeGroup">
      <complexType>
        <attribute name="ref"  use="required"/>
      </complexType>
    </element>
  </choice>
</group>
```

Listing 6.15: Individual attribute entries

Every attribute must be given a `name`, which is the key in the expression within the opening marker of the host elements that sets this attribute's value. The datatype allowed for the text that makes up the value is given by `type`. If this is omitted, the default of `string` is used, denoting a character string with no restrictions. No other datatypes are implemented by the framework, but this attribute is still included in the subset since it is sometimes explicitly set in custom schemata. Occurrence is not specified with the usual attributes `minOccurs` and `maxOccurs` as for particles, but rather with `use`, which is here limited to the values `optional` and `required`. Even though it has the same semantics as `minOccurs` within the subset, this is not the case in the full XSD standard, and hence is different names and values used. An attribute cannot occur more than once in an element, so there is no attribute corresponding to `maxOccurs`.

The definition of an `attributeGroup` reference resembles that of the other references defined by `reference` except for the lack of the occurrence specification, which is propagated from the attributes within the group individually instead of being put on the group as a whole. Attribute groups at the top level consists of simply a list of the attributes that it should contain as well as the name of the group, much in the same way as listing 6.6 on page 123 does.

6.2.4 Simpler variant

XSD can be pretty daunting at first sight, mostly because of the various context-dependent constraints that are put on its elements. Therefore, it might be clarifying to create a simpli-

fied definition of the grammar where no considerations to these idiosyncrasies are made, in order to get an overview of the structure of a schema.

Listing 6.16 contains the meta-schema converted into DTD in its entirety. It contains the same elements and attributes as the XSD version, but with a different composition. This meta-schema is a local type grammar, meaning that each label has only one content model associated with it. Hence, it will recognize some schemata that is not legal XSD as it is unable to distinguish between the various contexts. However, all of the schemata that is allowed by the framework will successfully be validated by it. Thus, it can be used as a negative test; schemata that fails by it will not be allowed in the repository either.

```
<!ELEMENT attribute          EMPTY >
<!ATTLIST attribute          name        CDATA #REQUIRED
                             type        CDATA #IMPLIED
                             use         CDATA #IMPLIED >

<!ENTITY % attributes        "(attribute|attributeGroup)*" >
<!ENTITY % named             "name        CDATA #IMPLIED
                              ref         CDATA #IMPLIED" >

<!ELEMENT attributeGroup     %attributes; >
<!ATTLIST attributeGroup     %named; >

<!ENTITY % occurs            "minOccurs CDATA #IMPLIED
                              maxOccurs CDATA #IMPLIED" >

<!ELEMENT element            (complexType)? >
<!ATTLIST element            %named;
                             type        CDATA #IMPLIED
                             %occurs; >

<!ENTITY % compositor        "sequence|choice" >
<!ENTITY % model             "%compositor;|group" >
<!ENTITY % particles         "(%model;|element)*" >

<!ELEMENT sequence           %particles; >
<!ATTLIST sequence           %occurs; >

<!ELEMENT choice             %particles; >
<!ATTLIST choice             %occurs; >

<!ELEMENT group              (%model;)? >
<!ATTLIST group              %named;
                             %occurs; >

<!ELEMENT complexType        ((%model;)?,%attributes;) >
<!ATTLIST complexType        name        CDATA #IMPLIED
                             mixed       CDATA #IMPLIED >

<!ELEMENT schema             (attributeGroup|complexType|group|element)* >
<!ATTLIST schema             xmlns       CDATA #FIXED "http://www.w3.org/2001/XMLSchema"
                             targetNamespace CDATA #REQUIRED >
```

Listing 6.16: Meta-schema in DTD

It was created by collecting all models for elements with the same label and amalgamate them into one. The attributes that are not omnipresent are made optional in order to allow them to be missing in some context. This has of course the drawback that they can also be left out from vital places without being detectable by the meta-schema. Generally, a local tree grammar whose language is a superset of a regular tree grammar's can

always be created [Mur99].

Entities have the same names as the groups in the XSD subset to which they correspond, to ease comparison between the two grammars.

6.2.5 Unsupported features

To arrive at a manageable subset, not only obscure features but also more common facets were discarded from the meta-schema. This section enumerates some of these aspects to explain why they were not included as well as pointing to the direction future work should take in order to reincorporate them. None of the suggestions mentioned here have been incorporated into the existing code. Knowing the limitations will also be of help in designing custom schemata that is intended for use in the framework.

Restrictions and extensions. Inheritance from other types is not supported on two grounds. First, it adds a lot of extra syntax for the meta-schema to handle and second, it is difficult to build an extension sequence on top of a forward reference. Removing and constraining type carriers is even more cumbersome. Solving these problems can be done by introducing a front-end that does a topological sort of type definitions and then construct the corresponding type carriers piece-wise afterwards. A workaround in this version, is to use aggregation of groups instead of extension, as is exemplified by listing 6.9.

Simple datatypes. Allowing constraints on free-form text adds an entire new dimension in matching. As this thesis is concerned mainly of structure and not really of content, this is omitted. Section 4.2.4 on page 93 discusses how simple types can be added to the framework in general.

Permutations. Only the compositors `sequence` and `choice` is supported, whereas `<all>` is not. An unordered set of particles would amount to a union between all possible permutations, inflating the size of the type carrier. This issue can be resolved by a front-end which generates them all, replacing the set with the resulting union.

Free units. Global attributes and multi-schema specification are not available as units defined this way will be pervasive. They would have to be added as an optional feature of every element, adding further to the space and time complexity of the carriers being created. However, this cost would only incur for schemata that actually utilize the feature.

Declaration order. Groups must be defined before they are employed. Recall from section 2.4.8 on page 51 that forward references cannot be used as the head of a sequence.Types are not subject to this limitation because they are only allowed as complete and not partial content and will hence be encapsulated between labels. Usually this constraint is of a more practical nature as circular groups are not allowed anyway. Thus, there will always be a well-defined order between them and it boils down to a matter of where the definition is physically located in the file.

Attribute order. For the same reason that permutations are not supported, documents must contain attribute expressions in the same order of appearance as in the schema definition for the element on which they are applied. Like many of the other items, it can if found to be a major obstacle be remedied by introducing a front-end that both sorts the

attributes on name upon reading the documents and rearranges the attribute order in the schemata correspondingly.

6.3 Processing model

Java contains a package for XML reading and processing called JAXP [MDB01], and this section discusses the usage and minor adjustments needed to align it with the framework presented in this thesis.

6.3.1 Streams

The traditional model for input/output has been to pull a chunk of (character) elements from a stream. Instead of reading characters, JAXP is instead based on getting parsing *tokens* from the source. Opening and closing markers and free text are the main constituents of these tokens, along with some more esoteric ones that are not interesting in this context.

Unlike the pull-based model where elements are typically read in a loop and then acted upon, JAXP employs a push-based model where the loop is hidden inside the framework in a manner similar to what is common in window handling packages such as e.g. Swing, and the elements are dispatched to a *handler* in the form of an *event*. ContentHandler is the interface for such handlers, and the relevant events are partially listed in listing 6.17. In order to handle these events, the receiving class must implement this interface and override the methods.

```
// package org.xml.sax
interface ContentHandler {
  void startElement( String namespace, String localName, String qName, Attributes atts );
  void endElement( String namespace, String localName, String qName );
  void characters( char[] ch, int start, int length );
  //...
}
```

Listing 6.17: Callbacks for parsing events

The events `startElement` and `endElement` notifies the handler that an element has been opened and closed, respectively. The first three parameters describe the tag of the element. The namespace identifies the schema in which it was defined and the local name is the text that makes out the tag itself. The collection `atts` contains all the attribute assignment expressions that was found in the opening marker.

The event `characters` method will be called every time free-form text is found between markers, including indentation whitespace that is normally not intended to be a part of the value carrier for the document. It may be called several times for the same element, even without the occurrence of elements in between text. Entities are expanded and translated into their definition before this method is called. Note that the array passed as parameter may also contain other data than the text; only the range specified by the two last parameters are considered valid for inspection.

A call to `endElement` with the same parameters as `startElement` is said to be *matching* if there has been an even number of opening and closing events between them, i.e. they are for the same element. All events that occurs between these two such matching events are on the contents of the element.

Custom semantic properties is not provisioned for in the generic handler. Information that needs to be exchanged between events must be stored as fields in the handler object to be available to all callback methods. Also missing is a well-defined way of switching handlers during processing based on the events received, characteristic of a finite state machines. This must be done by either exposing the input source to the handler or by introducing a delegating layer which can perform the switch. The latter approach is the one that will be taken by a later section in this chapter.

6.3.2 Flow

Parsing the character stream into markup events are modeled by the interface `XMLReader`. As the excerpt in listing 6.18 shows, it consists of a property `setContentHandler` that should be set to the handler that will receive the events, and an action `parse` that will trigger the processing of a given document. The events will be called during invocation of this method. It is synchronous, meaning that the thread will block until the entire document is done or an error has occurred.

```
// package org.xml.sax
interface XMLReader {
  void setContentHandler( ContentHandler handler );
  void parse( InputSource input );
  //···
}
```

Listing 6.18: Source of events

`InputSource` is an indirection to the underlaying stream from which characters are fetched. It may seem like a superfluous wrapper, but its added value is in the ability to specify the encoding of the stream and to identify from which entity it was loaded, neither of which are utilized by this thesis. It would also have been an excellent place to integrate retrieval from diverse sources, but this opportunity has not been seized.

```
XMLReader r = SAXParserFactory.newInstance().newSAXParser().getXMLReader();     // (1)
r.setFeature( "http://xml.org/sax/features/namespaces", true );
r.setFeature( "http://xml.org/sax/features/namespace-prefixes", false );
r.setContentHandler( /* ··· */ );                                               // (2)
r.parse( new InputSource( new StringReader( /* ··· */ ) ) );                    // (3)
```

Listing 6.19: Instantiating a parser

Reading a document hence takes form of the three steps in listing 6.19. First, a parser must be requested. This is done through a factory object, to obtain interchangeability between parsers by altering a configuration setting. Here, a SAX parser is requested as this is appropriate for push-based processing. To handle namespaces correctly, some flags

must be set. The first regulates whether namespaces are supported, and the second if a blank attribute (!) will be set to the current namespace. Strangely, these values are the default of the SAX specification, but are not set correspondingly in the Java package!

Second, this parser must be initialized with the handler that will receive the events, and third is the parsing started. In this example, both the objects specifying the handler and the input stream has been elided for simplicity.

By creating a handler *chain*, more than one handler can operate on the same document without doing a complete reparse for each of them. This means that large documents can be processed in one pass without claiming temporary storage, easing the space requirement burden for the server running the application. A chain is set up by letting the first *hook* receive the event and then pass it on to the next in the chain. If so is desired, it may even change the events in transit, filtering the information. The parser is unaware that more that one handler is subscribing to events, as the first hook is the only one it relate to.

The interface for frontends that do preprocessing of the events before they are forwarded to the ultimate handler is a descendant of `XMLReader` called `XMLFilter`. Inheriting the reader makes the frontend indistinguishable from it, requiring no changes at the client side. In addition to `setContentHandler` that let one specify the target, filters also contains the method `setParent` which is used to set the source from which events are delivered. (The filter will install itself as the source's handler, thus creating a chain).

To assist in creating filters, JAXP provides the class `XMLFilterImpl`. It implements both `XMLFilter` and `ContentHandler`, directing events to itself instead of a specified external handler. It provides a default implementation for each event that will forward the event, so only the methods for the events of interest must be explicitly overridden. To prevent the event from being terminated, the base class implementation should be called at the end of each callback, and it will pass it on to the next handler in the chain. The relationship between these classes and interfaces is shown in figure 6.20.

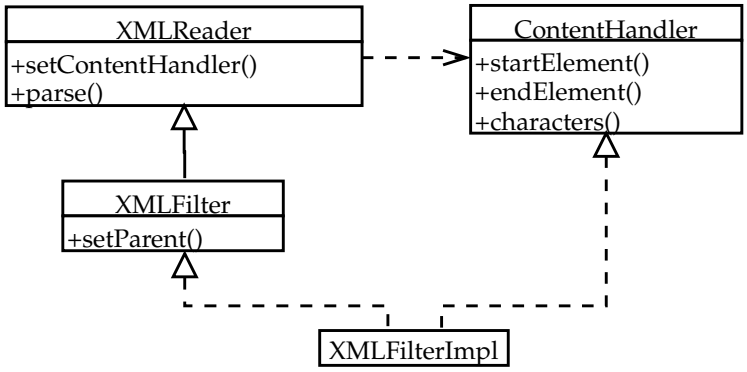


Figure 6.20: JAXP package

Handlers based on this class turns out to be very versatile as they can be used both stand-alone or as part in a chain, and this is therefore recommended design. When developing with a pull-based model, all logic should be implemented in filters and the message pump be a separate component that is not integrated with any of them, maximizing the

flexibility in combining these in various settings.

Events originate from the "uppermost" of parents, i.e. the filter that does not have any parent itself, and then trickles down through the chain to the handler that is not parent of anyone. Because of the dual role of a filter as both a source and a target, either `setContentHandler` or `setParent` can be used to add handlers to the chain.

Listing 6.21 illustrates how this is done. Both of these examples create a chain that invokes handlers *a*, *b* and *c* in that order. If `setParent` is used as in listing 6.21(a), filters are added at the end and `parse` is called on the last filter, pulling event through the chain. Contrast this with listing 6.21(b) where a filter is added in front of a handler and then the first filter is activated to push events.

```
b.setParent( a );
c.setParent( b );
c.parse( /* ··· */ );
```

(a) Downwards

```
b.setContentHandler( c );
a.setContentHandler( b );
a.parse( /* ··· */ );
```

(b) Upwards

Listing 6.21: Creating a parsing chain

6.3.3 Extensions

While JAXP contains wide-ranging support for manipulating XML document elements through the DOM, it makes no attempt to unify this with the streaming model, resulting in a lack of ability to persist parsing tokens. This section introduces two helper classes that has been added by this thesis to fill this gap. They are completely generic and not bound to any other part of the framework otherwise presented.

XMLWriter. Not all handling of the document may be within the same process, so therefore the need to serialize the document back to a character stream arise. The class `XMLWriter` does the opposite of an `XMLReader`; instead of being an event source, it is an event *sink*. As events are received, corresponding text is being written to a given stream.

Being filters, objects of this class can either be the end-point of a chain, sending the document to a descriptor such as a socket or a file, or they can work as a log that traces the document that is sent through the chain.

SAXEvent/SAXBuffer. In addition to transporting events between different process spaces, it may be necessary to move them across the time dimension as well. An example of this is when the header of the document determines how the body should be processed, it cannot be implemented with a switch of handlers as the header must be included in processing too. The solution is then to buffer the events while filtering them through detection logic. When enough has been read, the correct handler is installed and the hold buffer is replayed onto it before commencing further processing where parsing previously left off.

Every event is modeled as an object that is based on the abstract class `SAXEvent` stating that it must be able to replay itself to a given handler. On receiving an event, `SAXBuffer` will store all the parameters into such an object and add these to a list.

Hence, the events are transformed into function objects. For instance is `startElement` turned into an instance of the class `SAXStartElementEvent`.

The difference between this approach and building the Document Object Model (DOM) is that here a tree is not created, but the events are rather stored linearly and is more akin to the character stream from which they originated. The buffer can instead be seen as a binary representation of a tokenized but yet unparsed document.

6.4 Attributed grammars

A SAX "parser" is not really a parser in the traditional sense of the word as used in compiler theory. It is more like a *scanner*, whose task is to do preprocessing to remove comments and to transform the stream from raw characters into tokens that can more easily be worked on by the parser.

The job of the (real) parser is to convert syntax objects into semantic objects [ASU86], on which semantic calculations can be performed to retrieve the meaning of the text [EMRS97]. Input objects to such a calculation are called *inherited* while output objects from it are called *synthesized*. Semantic objects also go by the name *attributes* since they are attached to symbols of the grammar. However, to avoid confusion with the markup counterparts, this thesis will stick to the former term. To perform the analysis brought forward by previous chapters, the semantic objects that are interesting to retrieve are value carriers from documents and type carriers from schemata.

In this section, a method of extracting these objects from the syntax that has been laid out in sections 6.1 and 6.2 using the processing model from section 6.3 will be presented. Since this describes how the semantical objects are built from the external syntax, it is called a *syntax directed definition*.

6.4.1 Semantic stack

Normally, a syntax directed definition describes the flow of semantic objects between the grammar productions. As the algebras in this thesis are built in a functional matter, each object can be synthesized from the objects that makes out its immediate children, with the exception of a *symbol table* that is inherited from its ascendants and siblings to accommodates references declared earlier. This kind of flow belongs to a class called *L-attributed* definitions, where the evaluation of semantic objects can be done with a single depth-first traversal of the syntax tree [ASU86].

Being able to perform evaluation in this manner fits well with processing the document using a stream-based method such as SAX. Instead of building a parse tree first, virtual nodes can be visited upon events indicating that the element is entered or left, skipping that intermediate form completely. An auxiliary stack is needed to hold the semantic objects that are currently in scope.

When an element is opened, a construct known as a *builder* is created. This builder will accumulate information from the contents that is needed to create a semantic object for this element. If the semantic objects had been mutable, the information could have

been passed directly to them, but in order to provide all arguments to the constructor simultaneously, a temporary storage is needed in time till they are all completed. A builder is roughly equivalent to a grammar production in traditional compiler generation tools.

The builder is then put on the parse stack, where it will receive objects from all child elements. At the end of an element, the builder is popped from the stack and realized into a semantic object which is then sent to the builder of the parent, which is now the new top of the stack. The root element will thus eventually realize into an object that represents the entire document. Table 6.22 exemplifies the parse of a document into a value.

Document	parseStack[0]	parseStack[1]	parseStack[2]	parseStack[3]
<a>	$\perp$	$\perp$		
	$\perp$	$\perp$	$\perp$	
	$\perp$	b		
<c x="···">	$\perp$	b	$@x$	
</c>	$\perp$	$b,\ c[@x]$		
<d>	$\perp$	$b,\ c[@x]$	$\perp$	
<e>	$\perp$	$b,\ c[@x]$	$\perp$	$\perp$
</e>	$\perp$	$b,\ c[@x]$	e	
</d>	$\perp$	$b,\ c[@x],\ d[e]$		
	$a\left[b,\ c[@x],\ d[e]\right]$			

Table 6.22: Parsing stack during document build (after each event)

In the following sections, specialized builders for value and type carriers will be presented. For a more general approach that uses reflection to automate the builder, relieving the developer from hand-coding it, see [Zuk01]. The downside is however that it does not allow any mismatch between the schema and class model.

6.4.2 Value building

Generating value carriers from documents is a relatively straight-forward task since the mapping from one to another is more or less one-to-one as the algebra was designed to fit exactly that purpose. Due to this simplicity, building values serves as a nice example of the technique. As the builder always operate on the same carrier class and the operations of the builder correspond to the ones in the algebra, the parse stack can contain the carriers directly and the building operations made static in respect of it. The builder inherits from `XMLFilterImpl` so it can be used as a handler receiving scanner events.

```
public class ValueBuilder extends XMLFilterImpl {
  Stack/*<Value>*/ parseStack = new Stack();
  public ValueBuilder() { parseStack.push( null ); }
  //···
}
```

Listing 6.23: Parser stack setup

Listing 6.23 on the facing page displays the setup of the parser stack. A new stack is created and filled with the *Value* equivalent of an empty document, namely the **null** reference. This is done to ensure that the stack is always filled with valid content, so testing for boundary conditions all the time is avoided. The **null** reference is the additive value for the concatenation operation and is therefore the ideal candidate for a neutral starting point.

The same thing is done locally within the construction of a value element, as can be witnessed in listing 6.24. The stack is prepared with a carrier that will hold the content of the element. An accumulator value is initialized to **null** in the line marked with (1). This temporary carrier embodying the builder is then pushed to the parse stack, so that child elements (other than attributes) will be directed at it. This happens in the line marked with (2). If any attributes are present, they will be passed to the start event as a collection parameter and not as separate events. The code iterate through this collection and add each of the attributes to the accumulator in (3). Finally, the event is in (4) forwarded to the super class where it would propagate through the rest of the handler chain.

```
// class ValueBuilder
public void startElement( String uri, String local, String qualified, Attributes a ) {
  Value content = null;                                          // (1)
  parseStack.push( content );                                    // (2)
  for( int i = 0; i < a.getLength(); i++ )
    add( Value.attr( a.getLocalName( i ), a.getValue( i ) ) );   // (3)
  super.startElement( uri, local, qualified, a );                // (4)
}
```

Listing 6.24: Adding a new builder to the parse stack

Adding an element to the contents is done with the helper routine displayed in listing 6.25. It first pops the stack for the current element and then uses the `combine` method to append the element is question to its tail. The value carrier representing the new sequence is the added back to the parse stack, effectively replacing the old sequence.

```
// class ValueBuilder
void add( Value v ) {
  Value siblings = (Value) parseStack.pop();
  parseStack.push( Value.combine( siblings, v ) );
}
```

Listing 6.25: Sending content to the current builder

When the closing marker of the element is encountered, a sequence corresponding to the content will be at the top of the stack. It is dequeued and combined with the label provided as an parameter to the event, to a new carrier which represents the element. This value is then added to the parent sequence. After the content was popped, the content of the parent remains on the stack, and this is the sequence in which this element resides. Listing 6.26 on the next page shows how this is implemented.

Free-form text is handled just like an element, but instead of inspecting the stack for any child elements, the text is extracted directly from the parameter that is passed to the

```
// class ValueBuilder
public void endElement( String uri, String local, String qualified ) {
  Value content = (Value) parseStack.pop();
  add( Value.label( local, content ) );
  super.endElement( uri, local, qualified );
}
```

Listing 6.26: Converting the current builder into a semantic object

event and a special element that is intended for holding parsed character data is created (cf. section 4.1.3 on page 87) as illustrates by the snippet in listing 6.27. Indentation is deemed insignificant and removed from each of the sides of the text.

```
// class ValueBuilder
public void characters( char[] ch, int start, int length ) {
  String text = String.copyValueOf( ch, start, length ).trim();
  add( Value.data( text ) );
  super.characters( ch, start, length );
}
```

Listing 6.27: Handling free-form text

The bottom of the parse stack holds not the content of an element, but the content of the entire document. Another way to see this is to pretend that there is a virtual element that contains the root element. (The DOM for instance, contains such a notion). Thus, when the document has been processed, there will only be a single element on the stack and this would be the value carrier for the root element in the document. Listing 6.28 depicts how this ultimate element is publicized for the application code to retrieve after the processing is complete.

```
// class ValueBuilder
public Value get() {
  assert parseStack.size() == 1;
  return (Value) parseStack.peek();
}
```

Listing 6.28: Retrieving the semantic root object

6.4.3 Environment

Documents are strict tree structures, where nodes do not occur more than at most once in any path. Types on the other hand, may form graphs with cycles, due to the possibility of forward references. In code, such references can easily be set up imperatively using the host language's support for object references (see for example listing 2.33 on page 50). To do the same in schema declarations, there must be a way to convert names into references.

Units that are reachable are said to be within the *environment* of the scope, and the mapping between the name and the reference is stored in a *symbol table*. A lookup in the environment is done to resolve a referenced unit from its name. Since all declarations are

global in the meta-schema subset selected, there are no practical difference between the environment and the symbol table in this thesis.

The code to find a type carrier from its name is presented in listing 6.29. A map is first consulted to see if an entry has been defined for this name. If that is the case, this reference is returned directly and the environment acts like nothing more than a wrapper of the symbol table.

```
// class Environment
Type lookup( String name ) {
  Type t = (Type) symbolTable.get( name );
  if( t == null ) {
    t = new Ref();
    symbolTable.put( name, t );
  }
  return t;
}
```

Listing 6.29: Lookup in the symbol table

More interesting is the case of the name not being found. The environment will then create a forward reference that will later be bound to this unit when (and if) it is defined, and this reference is returned. The reference is also added to the symbol table so that other lookups will return the same reference, to avoid having more than one outstanding references to backpatch when the identifier is later discovered.

Hence, the `lookup` operation will always return a legal type carrier, even though the unit that was referenced has not been seen yet during parse of the schema! Error handling that checks for an invalid carrier being returned, is not necessary in the builders. Any missing references will be detected upon the first use of the carrier, although this can also be provoked with a simple call to `deref` after the parse is completed.

When a named unit is defined (at the top level), it must be entered into the symbol table. This is performed by the `bind` method in the environment, and the code for this operation can be found in listing 6.30.

```
// class Environment
void bind( String name, Type t ) {
  Type old = (Type) symbolTable.get( name );
  if( old != null ) {
    assert old instanceof Ref;
    ((Ref) old).assign( t );
  }
  symbolTable.put( name, t );
}
```

Listing 6.30: Adding to the symbol table

A check is first performed to determine whether a new symbol is being entered or if an old symbol is being replaced. Only forward references can be replaced; anything else indicates a duplicate definition, which is not permitted. Regardless of the old content, it is the new carrier that is bound to the name after the `bind` operation.

In the code accompanying the thesis, a specialized reference that also hold the name of the unit is used, to simplify debugging of missing references. Even other references can be assigned to a reference, chaining them, but although supported by the framework this feature is never employed by the meta-schema as aliases are not supported.

6.4.4 Type building

Compared to values, types are harder to build. There is not a direct and trivial mapping from the schema to the type carrier, although the semantic discrepancy is admittedly low. Instead of operating directly on the resulting values on the parse stack, the translation between the schema and the type carrier is done with a set of builders. However, as all of these builders are focused on producing type carriers as a result, a generic builder model like the one in the previous section can still be conceived, with the core evaluation delegated to mini-builders. Each such mini-builder is called a semantic *action*, which is the equivalent of a grammar production translation scheme. Such actions are modeled with the interface shown in listing 6.31.

```
interface Action {
  void setScope( Environment env );

  Environment getScope();
  NamedType getType();

  void onAttribute( String name, String value );
  void onChild( String name, NamedType value );
}
```

Listing 6.31: Semantic actions

This interface specifies that an action in the type builder has one inherited method `setScope` that receives the symbol table upon entry, and two synthesized methods `getScope` that can be used to pass this symbol table onto child elements, and `getType` that yields the resulting type of the production upon completion. An action is a function object where all the arguments are set as properties.

Output objects from each child are delivered to the action through the methods `onAttribute` and `onChild`, depending on whether these were defined with an attribute or a child element respectively. Each action will define its own set of valid properties (right-hand-side symbols), whose names are not hard-coded in the definition of the method but rather passed in the argument `name` for each invocation. Hence, these two methods handle a range of dynamic properties. The builder can then be designed in a generic fashion independent of the content being processed. The responsibility of checking if a valid property was attempted set thus falls on the action, not on the builder. Note that a property may be set more than once, indicating that it is has an (multi-value) array type.

The only thing that ties the action interface to the builder is the class of the resulting synthesized object. `NamedType` is a plain old data structure that simply aggregates a `String` and a `Type` in order to treat those two as a pair. If the type carrier is labeled,

its name will follow it to the parent in whose environment it is subsequently added. The name is optional, enabling types to be *anonymous* if the name is not given but set to the **null** reference. This cumbersome method is imposed by XSD which does not have a separate construct to bind type expressions to identifiers but rather does this through labeling the expression itself.

Each kind of element in the XSD subset is a *non-terminal* that will have a corresponding grammar *production* associated with it, describing its valid content symbols. The definition in section 6.2.4 on page 128 is reminiscent of such a grammar. To associate each schema element with the action that handles this production, the builder maintains a mapping of their names to a factory that can instantiate the appropriate implementation. Listing 6.32 shows how this map is filled by an anonymous inner class with an instance initializer.

```
// class TypeBuilder
Map/*<String, Class>*/ actions = new HashMap() { {
  put( "group", GroupAction.class );
  put( "element", ElementAction.class );
  put( "sequence", SequenceAction.class );
  put( "choice", ChoiceAction.class );
  put( "complexType", ComplexTypeAction.class );
  put( "attribute", AttributeAction.class );
  put( "attributeGroup", AttributeGroupAction.class );
  put( "schema", SchemaAction.class );
} };
```

Listing 6.32: Mapping actions to schema elements

When an element is found in the schema, the type builder will locate the appropriate action factory (1) and use it to create a new instance (2) that will be handling the evaluation of this particular element. The scope in then retrieved from the parent (3) and sent to the new production (4) to initialize the environment before the invoked production is put on the parse stack (5). Observe that since the environment is a mutable object, any changes will propagate to the ancestors. Now, the action is ready to receive its properties and the builder sends the attributes first (6) before returning to the scanner letting the child elements apply themselves. Listing 6.33 contains this procedure.

```
// class TypeBuilder
public void startElement( String uri, String localName, String qName, Attributes a ) {
  Class builder = (Class) actions.get( localName );                    // (1)
  Action current = (Action) builder.newInstance();                     // (2)
  Action parent = (Action) parseStack.peek();                          // (3)
  current.setScope( parent.getScope() );                               // (4)
  parseStack.push( current );                                          // (5)
  for( int i = 0; i < a.getLength(); i++ )
    current.onAttribute( a.getLocalName( i ), a.getValue( i ) );       // (6)
  super.startElement( uri, localName, qName, a );
}
```

Listing 6.33: Invoking a new production

Recall that the parse stack contains the (mini-)builders for the elements in the path down to the current one, and this stack must be updated every time an element is opened

or closed. The evaluation of the productions will follow the structure of the schema document, as the abstract syntax tree of a grammar is a regular tree [CDG+02, HMU01], effectively meaning that a markup document is essentially a parsed grammar ready to be evaluated. The production is evaluated when the element ends, and listing 6.34 displays the steps necessary.

```
// class TypeBuilder
public void endElement( String uri, String localName, String qName ) {
  Action current = (Action) stack.pop();                          // (7)
  NamedType type = current.getType();                             // (8)
  Action parent = (Action) stack.peek();
  parent.onChild( localName, type );                              // (9)
  super.endElement( uri, localName, qName );
}
```

Listing 6.34: Evaluating the production into a semantic object

All child elements has been evaluated and aggregated into the action at the time the element is closed. The action is popped from the parse stack (7) and then a type carrier is extracted from it (8). This semantic object representing the element, is sent to the parent which is now the current top of the parse stack (9). The name of the element available in the parameter `localName`, is used as the property to which the semantic object is communicated.

The builder is not dependent on any details of the schema language selected as the translation scheme is left at the discretion of the actions. If support for another schema language such as RELAX NG [CM01] is desired, it is a matter of changing the available set of productions to another that comprehend its syntax.

6.4.5 Productions

Action objects promote a design where each semantic expression is dealt with isolated, enabling focus to be kept locally within the production and selectively ignore considerations from other parts of the grammar. It can be inductively assumed that properties are initialized correctly due to the processed schema being valid according to the meta-schema, and each production generating the applicable type from these.

This section will review some of the core actions of the XSD subset in order to give an impression of how the model works. All actions inherit from `DefaultAction`, which centralized common chores such as reporting assignment to non-existent properties as errors.

Schemata are containers for the elements and other constructs that are defined at the top level. Its responsibilities are to register these global units in the symbol table and to provide a reference to a valid starting symbol for the grammar.

Listing 6.35 on the next page illustrates how this task is accomplished in the `SchemaAction` action, which is handling the `<schema>` element. Upon being notified of a child element, the tag is inspected to determine if it represents a labeled element or a type construct and branch into the correct logic correspondingly. The label must be used

```
// class SchemaAction extends DefaultAction
public void onChild( String name, NamedType value ) {
  if( name.equals( "element" ) ) {
    Label label = (Label) value.model;
    env.bind( "/" + label.tag(), label.content() );
    start = start.union( label );
  }
  else if( name.equals( "complexType" ) || name.equals( "group" ) ) {
    env.bind( value.tag, value.model );
  }
}
```

Listing 6.35: Registering top-level units in the symbol table

as a differentiator since a complex type may consist of only a single labeled element and hence be indistinguishable from an element in itself by looking at the resulting semantic object.

Global elements are referred to using the label prefixed by a slash, which is otherwise not allowed to occur in names and can therefore be used as a special token to prevent clashes with other units in the unified symbol table as types and elements are normally regarded to be in two different namespaces. They are also added to a field that holds the union of all legal start symbols, while free types are only bound directly into the symbol table by the name given to them.

```
// class ElementAction extends ParticleAction
public void onAttribute( String name, String value ) {
  if( name.equals( "name" ) )
    tag = value;
  else if( name.equals( "type" ) ) {
    content = env.lookup( value );
  }
  else if( name.equals( "ref" ) ) {
    tag = value;
    content = env.lookup( "/" + value );
  }
  else
    super.onAttribute( name, value );
}
```

Listing 6.36: Aggregating element attributes

Basic types representing labeled elements receive the information necessary to build the carrier through a set of attributes put on the `<element>` that defines them. The action `ElementAction` is the target of these events, and listing 6.36 shows how it aggregate content for the fields `tag` and `content` representing constructor arguments, depending on the information that is passed to it. The error code handling inconsistency between the combinations of attributes allowed have been omitted for brevity.

If a pair of `name` and `type` attributes are set, then the label is given directly while the type must be looked up in the symbol table, both from the value of the attribute. Although they work as a pair, the attributes are set independently. The attribute `ref` on the other hand, refers to a global element providing both the label and the content model and hence set both of these fields at the same time. Observe that a slash is prepended to

the label to select the namespace of elements instead of complex types.

When the element is to be realized, the fields are combined into a type carrier using the appropriate constructor, as can be seen in listing 6.37. Complex types are combining their model and their attributes in a similar manner.

```
// class ElementAction extends ParticleAction
Type getInternalType() {
  return Type.label( tag, content );
}
```

Listing 6.37: Evaluating the labeled element production

Other actions keep a field that represents the resulting object rather than the ingredients and keep this updated at all times. This approach is selected when the element has a variable number of children. Instead of keeping a separate list, the type carrier itself is used as a collection.

Compositors such as `<sequence>` and `<choice>` exhibit this behavior. The implementation of `ChoiceAction` which handles unions, is displayed in listing 6.38. The type carrier for the union is stored in the field called `model`. This field is initialized to an empty set, which is the default model for a union with no elements. All particles that are encountered inside this compositor will be added to the union, whereas children of unknown type are ignored and sent to the base class for error handling.

```
class ChoiceAction extends ParticleAction {
  Type model = Type.oe();

  public void onChild( String name, NamedType value ) {
    if( name.equals( "sequence" ) || name.equals( "choice" ) ||
        name.equals( "element" )  || name.equals( "group" ) ) {
      model = model.union( value.model );
    }
    else
      super.onChild( name, value );
  }

  Type getInternalType() { return model; }
}
```

Listing 6.38: Computing semantic expression during property assignment

Since `model` always contains the current contents of the union, evaluating the semantic object consists simply of returning this field with no further ado, as is done in the method `getInternalType`. This does not expose the internal state of the action, as the *Type* carrier is immutable. Hence, there is no chance that a returned object will be changed through the reference that the action holds. Anyway, it is assumed that the builder will only call `getType` once for each builder, and that no properties will be set after this is done.

Notice that neither `ElementAction` nor `ChoiceAction` implement `getType` from the interface `Action`, but rather the method `getInternalType` which is defined in their superclass `ParticleAction`. This class is the foundation for all schema constructs

that can act as particles and thus be decorated with a cardinality different from one, indicating that they are optional, repeatable or both.

Handling of the attributes `minOccurs` and `maxOccurs` that controls the occurrence of the particle is common functionality for all particles and therefore most rationally placed in a base class which is sandwiched in between the ultimate ancestor `DefaultAction` and the concrete class for the production.

This class overrides the semantic evaluation with the code in listing 6.39 to decorate the type that is returned from the particle's content, according to the property settings. As only particles can have occurrence constraints, it is assumed that subclasses that override `getInternalType` will yield anonymous types and thus have no name with which to tag the resulting semantic object. Occurrence constraints and labels are mutually exclusive: Particles that are not at top-level cannot be labeled and decorations at the top-level makes no sense.

```
// class ParticleAction extends DefaultAction
public final NamedType getType() {
  Type t = getInternalType();
  return new NamedType( null, t.decorate( allowZero, allowMore ) );
}
```

Listing 6.39: Decorations handled in common base class

Since the `ElementAction` production descends from `ParticleAction`, the schema must upon entering it in the symbol table therefore bind it to the label of the element and not the label from the semantic object, as the latter will be set to the **`null`** reference.

6.5 Summary

A general syntax for markup languages is defined by the standard eXtensive Markup Language, also known as XML, which describes how tokens in a character stream can be interpreted as a node tree. Constraining the structure of this tree can be done with the standard XML Schema Definition, also known as XSD.

Together, instances of these two can define a grammar for a language which belongs to the class that can be handled by the framework of this thesis. The meta-schema that is supported is an XSD subset, where provisions for reusing type is included. As the meta-schema is a schema too, it can be defined using itself.

Java's runtime library contains a package for push-based reading of XML documents. Transformation of external documents to an internal representation is done in one pass over the source using depth-first evaluation of semantic objects. Handling of productions is delegated to separate modules whose implementation is exchangeable if other meta-schemata are desired.

Chapter 7

Repository

In order to efficiently reason about the integrity amongst the stored documents, the framework must be able to assume that they are not altered without its knowledge. Hence, they must be stored in a strictly controlled repository where the framework can monitor any changes that are made.

This chapter will discuss the requirements a storage system must meet to be usable for this purpose, and then how a layer can be formed to integrate it with the rest of the framework. A consideration of the effects this will have on the repository layout will also be presented.

7.1 Back-end

A *back-end* is the storage system that assumes the responsibility of ultimately providing physical storage for the document. This need not be the only task it does — it is often combined with other functionality as well — but this action is almost always deferred until all other processing has completed, persisting the most current version of the document.

There is a variety of standard products available to fulfill this role, and a goal of this thesis is to employ to the extent possible only features common between them so as to make them interchangeable, in order to have the option of picking the solution best suited for the project overall and not create unnecessary dependencies on a particular third-party product.

7.1.1 Catalog

For a document to be reachable, it must be identifiable, meaning that every document must be associated with some information that makes it unique. The collection of such data is called the *catalog* of the repository, because it holds the inventory.

The catalog keep track of two kind of entities; documents and schemata. A member of the latter group is naturally identified by the namespace it defines, whereas the former exhibits no such property and must have another method of identification imposed on it.

Each document will be named by a Universal Resource Identifier (URI), but only exclusive use within those that belongs to the same schema will have to be warranted. Thus, the primary key of the document is a pair consisting of the schema namespace and the URI of the document.

Linking the document to the schema this way is done to relieve the catalog from maintaining a mapping from values to their types behind the scenes, instead making it explicit, partitioning the complete set of documents into smaller and more manageable subsets which are type-wise internally consistent. Many relational storage engines also demand the data to be strongly typed.

A front-end can if desired be devised that perform the lookup before relaying the request, to make only one truly unique URI suffice. The schema would then have to be inferred when validating the document upon entry to the repository, and this could for instance be done through use of a `schemaLocation` attribute [TBMM01]. Another solution is to simply let parts of the URI designate the schema, e.g. by mapping specific path components to it.

Intuitively, the user should be able to retrieve information through a *virtual file system* gateway, where the identifier of a document contains all necessary information to locate it. Such an identifier is thus called a *locator*. If the organization has registered the domain name `acme.com` and set up a repository that can be reached through the service `repos`, a document with type `foo` and name `bar` could be addressed like follows:

```
http://acme.com/repos/foo/bar.xml
```

The individual components of the identifier does not have to be the resource itself, as long as the token provides enough information for the system to locate it under the hood. Not exposing the actual name itself enables the system to encapsulate certain services such as for example load balancing and location independence.

However, the above implementation is only one of many possible ways to invoke the repository, as the framework itself does not read anything into identifiers. On the contrary may the name of a document be in a totally opaque (regarding the location of the document at least [LS98]) and mechanical manner, as the following example from [MLS02] shows:

```
urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6
```

Client-specified document identities creates a separation of roles between the writer that authors the data and the maintainer that structures the repository, in contrast to the situation where the identity is hard-coded in a `id` attribute on the root node. Whether this is an advantage or a disadvantage depends on the organizational structure and is therefore considered outside the scope of this thesis.

Selecting the option of identifiers being specified explicitly will instead be done based on its technical merits: It does not prevent the other alternative to be later supplied by a front-end, whereas the opposite would arguably be somewhat more involving. Also, it fits best with the push-based streaming model in which algebra carriers are read, since

there is no need to set up a temporary buffer to preliminary scan the document for the identity as described in section 6.3.3 on page 134.

7.1.2 Meta-data

Due to the fact that schemata really are documents too, albeit with a different role, they can be stored in the repository in the same way as regular data, so as to avoid any special tailoring for meta-data and enable reuse of the generic processing capabilities. Just like a document relates to the schema, the schema on the other hand relates to the meta-schema. This world-view allows both targets to be interfaced in a uniform manner.

The repository must still do some extra-ordinary handling when updating schemata in order to ensure that the entire information set that belongs to it is revalidated (cf. chapter 5), and this is done by recognizing the name of the meta-schema and intercepting modifications to any documents of this schema behind the scenes.

7.1.3 Functionality

Every storage system must at least support a set of fundamental operations for putting data into it and getting the data back out again. To avoid that old data linger on when there are no longer any use for them, destructive functionality is also essential.

Additionally, a distinction between destructive and non-destructive updates is often made. The difference is that a destructive update may overwrite some of the existing data, while the non-destructive may not. Having these two notions as separate commands enables the client to do partial updates by only supplying the parts of the data that is to be replaced without touching any of the remaining items, whereas creation of a new document on the other hand, always must supply a full batch of all mandatory information. Hence, an partial update may be done more efficiently than a delete–create combination.

This amounts to what in common jargon is called “CRUD” functionality, named so after the initials of the four basic operations:

- Create; insert a new document into the repository
- Read; get the contents of a document that is present
- Update; alter the data in an existing document
- Delete; remove a document from the repository

However, this thesis will only consider atomic updates, i.e. if a change is to be made to the document, its entire contents must be replaced with a new version where the appropriate modifications are done. The impedance mismatch between the streaming model selected and the imperative model needed for partial updates is a contributing factor to this decision. A future version should consider adding provisions for update hints to allow the back-end room to do optimizations by selecting better suited native commands for the task.

Besides altering the database set by manipulating the documents, the back-end must also have provisions for inspecting the set by querying the catalog, i.e. list the documents available. Indeed do the operations on the catalog resemble the ones for the documents but lifted to a higher level, as the creation and destruction of documents are updates to the catalog.

Uploading a document will have to occur at a pace in which both the client can deliver data and the back-end can accept it. Bottleneck in the back-end usually implies full processor utilization and will cause the operation to block the current thread. Conversely, being equally vulnerable for delays originating at the client side is not as acceptable, since these are outside the framework's sphere of control and it lacks any domain-specific knowledge that may be needed to resolve them.

For this reason, the data pump will not be located inside the framework. Instead, the upload operation is instead split into two method that indicate initiation and completion respectively, and the client will impel the data into a designated handler in between calls to these two.

Downloading is still done synchronously so as to not require the client to specify a callback for transaction control, since data is assumed to be readily available from the repository at all times and can be spooled to a temporary buffer to insulate from blocking if the client does not want to perform this kind of management.

Listing 7.1 summarizes the interface through which a back-end adapter must communicate with the framework. The first four methods performs actions on the documents in the repository, while the last three provide access to the catalog.

```
interface BackEnd {
  ContentHandler startPost( String schema, String id );
  boolean endPost( ContentHandler handler, boolean commit );

  void get( String schema, String id, ContentHandler target, boolean forEdit );

  void delete( String schema, String id );

  Iterator/*<String>*/ startList( String schema );
  void endList( Iterator/*<String>*/ it );

  boolean exists( String schema, String id );
}
```

Listing 7.1: Operations required by a storage back-end

Identification of documents are done by a pair of character string arguments containing the name of the schema and the name of the document within that schema, respectively. As each of these parts plays an individual role in the catalog, the pair is not encapsulated in its own a value type. Neither is the requirement that the names must be legal URIs enforced: Technically, any string will fit although the use of URIs is recommended to avoid any name clashes and to ensure compatibility with other external tools.

Calls to the methods prefixed with `start-` and `end-` should always be performed in tandem, where the invocation of one matches a corresponding of the other. The `start-` method that initiates the action not only return a value which is an access point to the data

for the client, but which also acts as a *cookie* that turns the `end-` form into its continuation, resuming work after the client has done its part of the job. Hence, thread causality can shift between the client and the framework without having the former adhere to a specific interface. (Listing 7.16 in section 7.3.4 on page 163 contains an example of this technique's usage).

Uploading is completed by calling `endPost` with the `commit` argument set to **`true`**, which signals to the back-end that the data sent should be moved into persistent storage. If this argument instead is set to **`false`**, the update is canceled and no change in the document is made. Thus, errors that occur while writing new information need not propagate into the repository, but can at least be recovered from by reverting the document to its state prior to the operation. The return value of the method in turn tells whether the document was accepted by the back-end or not.

A document is represented by an event handler for its stream (cf. section 6.3.1 on page 131), on which a representation of the state transfer of the repository can be read or written. All operations are independent of the documents' contents, and embodies rather generic verbs [Fie00].

While the `startList/endList` pair follows the same pattern as `startPost/endPost`, the return value is a read-only enumerations of the documents available for a given schema, and it is not intended for the client to alter. Consequently, no flag to indicate the success of the operation is needed, but the client must still call the `endList` operation to give the back-end an opportunity to release any associated resources.

Searching for the existence of a given document can be done without traversing the entire list, since the `exists` method provides a specialized shorter version, which can employ better suited commands in the back-end for this purpose.

Retrieving documents require the client to pass an event handler to which the document will be streamed, exposing either a builder for a semantic object or another destination capable for chaining. Observe that in the latter context, the upload handler of another repository to which the document is intended copied may very well be used.

The flag `forEdit` disables the execution of all transformations that should otherwise be applied to the stream before presenting it to let the client get back the "raw" representation that was originally uploaded instead of a "cooked" one that has been massaged by various filters, enabling the application to differentiate between document authoring and document presentation.

Future work should consider the addition of a log property to the repository (or an argument to the operations) to which messages from the back-end could be sent while sending or receiving data, so as to allow non-fatal notifications through another mean than exceptions and false boolean return values.

7.1.4 Simultaneous access

Intended as central storage that will keep all documents under integrity control, the repository must handle access by multiple users. Not only is it probable that several presentations of the material will be requested at the same time by for instance a web

server, but on a production system it may also be conceived that editing of the very same documents is concurrently attempted.

When a thread updates the database simultaneously with another thread requesting conflicting items, the reader may experience various data anomalies stemming from unfinished or uncommitted actions taken by the writer, depending on the level of isolation that is ensured [BN97]. Each level that give a potentially different result than the reader would otherwise get, is named by the incidents in which the corruption may occur. They are presented here in decreasing order of severity (lack of isolation):

Dirty reads. Data that has been written by another thread but not yet committed is called *dirty*, and should not be exposed to any readers. This anomaly is on document level, and in order for it to happen both the reader and the writer must be addressing the exact same schema and name. It is most often caused by back-ends that give the writer direct access to a shared data area.

Phantom reads. Enumerations of a schema that include documents which a writer has started uploading but not yet committed is said to contain *phantom* records, because these will disappear if the other thread experiences an error and the updates it performed must be reverted. Readers will be affected not only by accessing the catalog directly, but also implicitly when locating documents to download. The common reason for this to happen is that the back-end propagates changes from the writer's local copy to the global catalog at the start of (or continuously from) the update instead of at the end.

Non-repeatable reads. A reader that attempts to request a previously read document once again but find that it has now been changed by another thread have done a *non-repeatable* operation, because although the command is the same, the result is not. The two queries are not recognized to be in the same scope, and the source of this inconsistency is that the back-end does not keep a cache for each reader that is separate from the global data set.

Each of these effects is undesirable, and it is hence necessary to evaluate efforts that can counter them. The most popular mechanisms for ensuring isolation are *locking* and *multi-versioning* [BN97]. Locking works by prohibiting access to an item while another thread is operating upon it, whereas multi-versioning ensure that all uncommitted work is done on local copies. Both topics have a vast amount of considerations associated with them, but due to the lack of space and time only a surface treatment of the actual solution selected will be given here.

Congestion may be reduced by differentiating between inspective and mutative access to allow multiple threads to read at the same time [Lea97]. This certainly sounds attractive as some characteristics of the system indicate that some operations may be parallelized. For instance, uploading of new data will indeed require write access to the document itself, but only read access to the schema (for validation).

For its purpose, this thesis nevertheless selects an approach where updates are written to temporary storage and not exchanged with the content in the main database until the operation commits, requiring only synchronization to avoid collision at that particular moment. Snapshots are not identified with version numbers as in relational databases [Mit01], but instead with plain object references.

Using this methodology, dirty and phantom reads are prevented but in its current incarnation non-repeatable reads will still be possible. Arguably, this shortcoming is inherit in the interface which does not provide any way to demarcate the boundaries of related operations. The seriousness of this limitation is debatable however, given that the data returned after all is committed, and it is left as future work to introduce a layered approach for handling transactions along the lines of section 3.2.2 on page 56.

7.2 Databases

The interface is designed so that adapters can be developed for a diverse assortment of back-ends, to accommodate changing needs from a variety of projects in which the framework may be deployed. This section will introduce the types of databases most likely to be used, and how adapters can be molded to fit them.

Only one of these — the in-memory database — has been implemented and included in the source code. It is selected on the grounds of being extremely light-weight by not requiring any external third-party resources and can therefore be used in all the same settings that the rest of the framework can, facilitating further development and experimentation of features and extensions.

7.2.1 In-memory database

Having data stored *in-memory* means that the entire object graph is always kept active and no attempt are ever made to swap parts of it out to secondary storage such as a hard-drive.

Because of this, the database is not durable, as all its contents is lost and must be loaded again every time the program restarts. It is also very limited, as the amount of primary storage such as RAM memory constrains the number of objects that can exist at the same time. Although modern operating system are very good at providing virtual memory, the algorithms does not exhibit any control over the placement of objects in the address space and thus do not make any efforts to minimize thrashing in a strained situation [SG98].

Performance and latency is generally good for repositories containing only a small number of objects, and the footprint of the code itself is low. This kind of database is suitable for situations where the contents can initially be fetched from a master copy and is not considered critical, such as in an information kiosk set up for demonstration purposes.

Implementation will be comprised of a simple two-layered hierarchy where all the documents that belongs to the same schema are kept together in a container, which are again put in a root collection for all schemata. Each of these layers are indexed by the names of the items. This layout resembles a file system, where each schema is a directory and the all the corresponding documents are files within it. A durable implementation using the file system can easily be realized by exploiting this similarity and using the in-memory database as a foundation.

Use of this setup will be illustrated through the operations that deal with up- and downloading of documents. However, an understanding of the methods for querying and modifying the catalog should also be attainable from the basis laid forward here.

```
// class InMemoryDB implements BackEnd
Map/*<String, Map<String, Document>>*/ schemata = new HashMap();

Map/*<XMLReader, Document>*/ sessions = new WeakHashMap();
```

Listing 7.2: Permanent and temporary storage containers

Temporary storage is provided by a separate area where data is associated with the handler for the operation rather than the name of the document. Listing 7.2 shows the declarations for these two maps. Upon startup of the back-end, they are initially empty. Observe that the permanent storage uses strong references since it may be the only one averting the garbage collector from recycling the data, whereas the temporary storage can settle for weak references since those items should always be kept alive primarily by the client working on them.

```
class Document {
  String schema;
  String id;
  SAXBuffer buffer;
  //...
}
```

Listing 7.3: Bundle of document and associated data

Documents are represented by a plain data structure that *bundles* keys necessary to locate the meta-data together with a compiled form of the events that make out its contents (cf. section 6.3.3 on page 134), as is displayed in listing 7.3. Instances of this structure are records that embodies the documents in the database. It will later be extended with other information that is retrieved from the stream through filters. Use of this class is only internal to the package and it will therefore never be exposed to the outside.

When uploading of a document is initiated, the database creates a new representation in temporary storage including a buffer to which the client can stream the data, before associating it with a handle that is returned. Listing 7.4 outlines the steps of this procedure.

```
// class InMemoryDB implements BackEnd
public ContentHandler startPost( String schema, String id ) {
  XMLReader buf = new SAXBuffer();
  Document doc = new Document( schema, doc, buf );
  sessions.put( buf, doc );
  return buf;
}
```

Listing 7.4: Initializing upload of new data

Note before proceeding that all code for synchronization and error handling has been omitted from these listings so as to simplify them in order to better focus on the algorithm.

For full details, refer to the source code.

Names of the schema and the document are stored along with the buffer in the bundle, so all the framework needs in order to regain those properties, is the handle by which to perform a reverse lookup in the map of active sessions. This is the technique used in finalization of an upload, which can be viewed in listing 7.5.

```
// class InMemoryDB implements BackEnd
public boolean endPost( ContentHandler handler, boolean commit ) {
  if( commit ) {
    Document doc = (InMemoryDB.Document)sessions.get(handler);
    Map documents = getSchema( doc.schema, true );
    documents.put( doc.id, doc );
  }
  sessions.remove( handler );
  return commit;
}
```

Listing 7.5: Finalizing upload of new data

If the client decides to commit the upload, the correct container for the schema is located — creating it if necessary — and the document bundle put therein. The upload is terminated by a call to `endPost` no matter what the outcome is, so the bundle should be removed from temporary storage and the client must not use its handle any more. The actual code contains counter-measures against the possibility that a client mistakenly corrupts data through an terminated handle by orphaning it.

```
// class InMemoryDB implements BackEnd
Map getSchema( String schema, boolean create ) {
  Map documents = (Map) schemata.get( schema );
  if( documents == null && create ) {
    documents = new HashMap();
    schemata.put( schema, documents );
  }
  return documents;
}
```

Listing 7.6: Localizing the container for a given schema

Getting the container for the schema is done by the helper routine in listing 7.6. It first queries the global document area to find if there are already a subcollection for the particular schema in question. If there is no matches and the argument `create` flags that a non-**`null`** result is required, a new map for this and future use is created and returned.

```
// class InMemoryDB implements BackEnd
public void get( String schema, String id, ContentHandler target, boolean forEdit ) {
  Map documents = getSchema( schema, false );
  Document doc = (Document) documents.get( id );
  doc.buffer.replay( target );
}
```

Listing 7.7: Downloading existing data

The same helper is employed in the process of downloading a document, depicted in

listing 7.7 on the page before. In this case, a reference to the document bundle is obtained from the schema collection and subsequently sent to the client handler by replaying all the events from the buffer. (Again the reader is reminded that the error checking performed in code is distilled from the listing).

7.2.2 Binary large objects

Relational database management systems (RDBMS) are ubiquitous and have over the years been demonstrated to possess capabilities to not only be very versatile but at the same time be able to meet performance and scalability demand from larger projects. Being able to employ such a system is sometimes the key to successful deployment in many real-world scenarios, and it is therefore vital that coping with it has been in mind when designing the framework.

The easiest way to leverage a relational database is to store each document bundle as its own record in a huge table, using a binary large object field to hold the entire data buffer. Instead of accumulating events locally in a `SAXBuffer`, uploading is done by connecting an `XMLWriter` that will encode the values using XML to the stream provided by the management system for writing to this BLOB.

An advantage with this scheme is that performance is very good if the entire document is operated upon, i.e. all of the data is uploaded or downloaded at once, and it is more scalable since the temporary storage is handled by the management system's cache controller which probably uses sophisticated and fine-tuned paging routines. The buffer-based implementation presented in the previous section can also be easily be adapted to use BLOBs, as they are conceptually not very different from a file system although the transaction management is better.

The downside of such a solution is that the structure that is inherently present in a regular tree document becomes opaque to the back-end, and hence any advanced searching facilities can not be utilized. Finding documents that matches a criteria based on content either requires that a full scan is performed or that filters are installed that copies the information which should be indexed over to extension properties in the document bundle. The latter technique limits the flexibility that can be provided to the client in locating data and gives additional implementation headaches in that it requires those properties to be incessantly synchronized with the main data.

7.2.3 Object-relational mapping

Creating a full relational schema to hold the data is a more innate approach to an RDBMS back-end. In an object-relational mapping scenario, no XML is stored in the database but is instead only used as an interchange medium for the underlaying objects which is modeled in the database. The internal representation is rather left to the management system's discretion.

Generally, there are two ways of mapping objects to a relational schema: Using a separately customized schema for each object type [ML02], or using a universal schema

that is capable of storing anything for all of them [FK99].

The former method gives good search capabilities but carry a large overhead in operations that transfer data as it require a layer that can marshal the data from the hierarchical structure to the appropriate (relational) schema. It may also require that the framework has special administrative privileges in the database in order to run the commands to install the data definitions, and that is not always practical.

Use of a relational schema created specially for the type will normally be desirable as it yields higher data fidelity, but that is a minor point in this case since all documents will be validated prior to entering the repository and it is assumed that the client cannot otherwise achieve access.

The latter method is however also typically penalized by low performance when composing and decomposing documents, due to its inability to take advantage of any structural clustering in the object graph. Children are not referenced with foreign key fields that is put inline the record but always through intersection relations [Kro95], resulting in a greater number of table joins to produce the final data set.

7.2.4 Native XML databases

Database management systems that regard marked-up documents as their *native* data model stores information accordingly in a hierarchical manner rather than relational, and provide facilities for selection and projection of trees instead of tuples [Bou03]. Such a back-end would be a more natural fit for a repository containing regular tree languages as its paradigmatic view is not discrepant from that of the front-end, which should ensure that the construction of such an adapter within the current design is not a mission that in the end would turn out to be too laborious.

At the time of this writing, a wide range of database management systems that has support for XML is available. Some of these have their roots in relational or object-oriented data models, but has been extended to also deal with regular tree grammars [Gen03]. Others are designed around this concept from the ground up [FHK+02], and operate with no alternative notions.

Using a database specialized for handling XML can be advantageous in that it will be designed to make the appropriate trade-offs between features and performance for both searching and manipulation, in contrast to solutions that are bolted on top of other engines, which tends pay attention to only some of these areas. Furthermore, ad-hoc report generators can choose to access the database through the framework or directly.

However, transparency of data structure can be a threat to the repository's integrity if clients can revise the database without the framework's knowledge or authorization, rendering some documents incompatible. Also, the database may not be prepared to handle the situation where a schema for existing data change, or it may perform superfluous checks in addition to those already done by the subtyping algorithm, leading to loss of efficiency.

Lack of interface standardization is still an obstacle to overcome when trying to write portable code that works with multiple products, and the actual offerings on the market

has yet to prove their worth as vendors have not accumulated critical mass of real-world experience.

Despite these immediate short-comings, it is the opinion of this thesis that native XML databases seems to be the option that has the greatest potential to eventually provide the best back-end for the repository.

7.3 Integration

The thesis will now shift its focus from the internals of the back-end adapter and devote the rest of this chapter onto the task of integrating the repository with the algorithms for validation and compatibility, in order to devise the missing piece of a completely functional content management framework.

7.3.1 Façade

Additional services are layered on top of the database back-end, filtering not only the documents that passes through the interface but the commands themselves, too. Hence, the front-end is designed after the *Russian doll* principle, where an outer component fully encapsulate a similar inner. Figure 7.8 visualizes this setup.

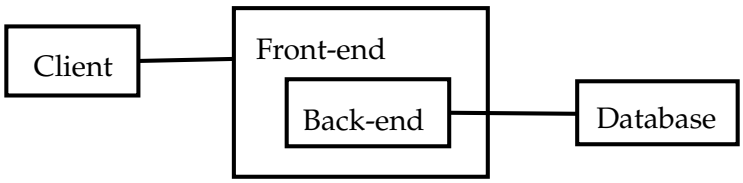


Figure 7.8: Encapsulation of back-end

Components only communicate bilaterally with its adjacent neighbors, to reduce coupling and encourage modularization. Clients only speak to the front-end, never directly with the back-end, and only the back-end is permitted to talk to the database. The architecture of the framework is extensible in that it allows supplementary filters to be added to the handler that is returned when uploading is started, letting several components cooperate to produce and determine the outcome of the final result.

The tasks of ensuring repository integrity will be placed in a component called *Manager*, which is as much a natural candidate to be a singleton as the database. It has the same interface as the back-end, of which it will aggregate an instance to do the grunt work after its business logic is done processing the request for its purposes.

Listing 7.9 shows how the aggregated back-end is stored in an instance field, to be available for the code in the class. After the back-end is aggregated, the manager will "bootstrap" the meta-schema into the repository to ready it for receiving further content. The meta-schema is needed to validate other schemata, but it must be added unconditionally itself to circumvent the circular dependency problem.

```
// class Manager implements BackEnd
BackEnd backend;

public Manager( BackEnd backend ) {
  this.backend = backend;
  //...initialization of repository...
}
```

Listing 7.9: Front-end aggregates back-end

The manager receives the back-end as a constructor argument as opposed to directly initializing it or attempting to look up the reference in some directory. Both of these alternatives require the framework to make assumptions about the environment in which it is running, which is best avoided. Passing the back-end explicitly makes it possible to introduce extra layers into the system without concerning this component. Procuring a factory that selects the appropriate database and creates an adapter for it is the only concern the client will have in this matter.

Information associated with a document upload is stored in session objects, which play the same role in the manager as the bundles do in the in-memory database back-end (cf. section 7.2.1) and can hence be implemented using the same pattern. Each session has a list of *checks* which are to be performed. They are installed as filters in the event handler chain, monitoring the content as it passes through.

Sessions are kept in a map indexed by the handler returned to the client. Listing 7.10 displays the establishment of a session. First, a new session object is instantiated. Notice that as `Session` is an inner class, it will have access to the member fields of the manager. The constructor will invoke the `startPost` method of the back-end, which provides the end sink for the events. Next, the appropriate checks are added. All documents must be at least validated and are subject to statistics collection. If and only if the document is a schema, a compatibility check must be performed as well. These checks will be explained in the coming sections. Finally, after the session is prepared, the `register` method is invoked to retrieve the start of the handler chain, put the session in the map under this key and then return the handler to the client.

```
// class Manager implements BackEnd
Map/*<ContentHandler, Session>*/ sessions = new HashMap();

public ContentHandler startPost( String schema, String id ) {
  Session session = new this.Session( schema, id );

  session.add( session.new ValidityCheck( schema ) );
  session.add( session.new StatisticalGatherer() );
  if( schema.equals( MetaSchema.NAMESPACE ) )
    session.add( session.new CompatibilityCheck( schema, id ) );

  return session.register();
}
```

Listing 7.10: Checks to be performed are added upon start of uploading

When the client has completed the upload and call `endPost` of the manager, the

session object is found through a reverse lookup on the handler and all the checks subsequently enumerated to poll their status, as can be witnessed in listing 7.11. Any check has the power to veto the decision to commit the document successfully, and so has the client, thus is the flag indicating success initialized to the parameter `commit` which tells if the client regards the upload as completed.

```
// class Manager implements BackEnd
public boolean endPost( ContentHandler handler, boolean commit ) {
  Session session = extract( handler );

  boolean valid = commit;
  for( Iterator i = session.checks.iterator(); i.hasNext(); )
    valid = valid && ((Session.Check) i.next() ).check();

  return session.close( valid );
}
```

Listing 7.11: Only documents that passes all checks are committed

Only if the document is condoned by all of them is the back-end notified to proceed with the update; otherwise it will be canceled. The method `close` will call `endPost` in the back-end after removing the session from the map, performing any tear-down activities that is needed.

7.3.2 Validation and upgrade

Checks are components that test the document's adherence to certain aspects, gathering information necessary to form an opinion by filtering the event stream. In this implementation they are inner classes of the session, so that they can access the meta-data of the document such as its name, in addition to being able to query the associated back-end for other data needed for this purpose.

The `add` method of the session will inspect the `getFilter` property of the check to find the callback object that will be installed in the handler chain. Observe from listing 7.12 that a builder can be returned to for instance construct a value carrier while the document is read.

```
// class Manager.Session
private class ValidityCheck extends Check {
  ValueBuilder vb = new ValueBuilder();

  XMLFilterImpl getFilter() { return vb; }

  boolean check() {
    TypeBuilder tb = new TypeBuilder();
    get( MetaSchema.NAMESPACE, schema, tb, false );
    LangRel lang = new LangRel();
    return lang.isIn( tb.get(), vb.get() );
  }
}
```

Listing 7.12: Language relation checks document updates

After the client ceases writing events and completes the upload, the manager will survey the component through the `check` method to determine its stance on the issue of whether the document should be allowed to pass or not. `ValidityCheck` will assess if the value carrier is in compliance with the associated schema by loading the corresponding type carrier from the database and use the language relation to ascertain.

Although the document is only processed once, this particular realization of the technique will still keep the value carrier for the entire document in memory, since confirming validity can not be done without the aid of a buffer (to backtrack from branches in a union). However, this drawback is not inherit in the design of the checking process, as a stream-based validator could also be fitted in if available.

To vindicate the replacement of a schema, its accordance to the meta-schema will not suffice alone but it must be complemented by a proof that it is compatible with the current content of the repository. This extra scrutinization will be performed by `CompatibilityCheck`, displayed in listing 7.13.

```
// class Manager.Session
private class CompatibilityCheck extends Check {
  TypeBuilder replacement = new TypeBuilder();

  XMLFilterImpl getFilter() { return replacement; }

  boolean check() {
    TypeBuilder original = new TypeBuilder();
    get( schema, id, original, false );
    SubRel sub = new SubRel();
    sub.setContexts( getContextsForSchema( id ) );              //(†)
    return sub.isIn( original.get(), replacement.get() );
  }
}
```

Listing 7.13: Subtyping relation checks schema updates

It is virtually identical to `ValidityCheck` except that a type carrier is built instead of a value carrier, the subtype relation is used instead of the language relation, and that the subtype relation is connected to the path usage statistics found in the repository for the document at the line marked with (†), to supply the algorithm with the counters stored in the database. Note that the namespace of the schema that is being uploaded is held by the variable `id` in this setting.

7.3.3 Statistics

Counters that reflect to which extent the schema is currently being used by the documents must be obtainable from the repository, and the back-end must thus be endowed with not only a mean to accumulate totals for individual parts of the type but also a map that indicates which documents that contributed to the measure. Hence, the database is storage for not only documents but statistics about them, too.

Policies regarding how the counters are actually stored are left for the back-end to decide. The in-memory database will for each document simply store a table of the paths associated with their respective counters, summing the present documents for a schema

dynamically to create a total view. For details regarding the implementation of these methods, please refer to the source code.

For a back-end based on a relational database management system, a natural approach would be to have an intersection table between the documents and the schemata, with the path being a field supplying additional knowledge about the relation itself. Performance may be enhanced by caching the total count of paths found for a schema, adding further complexity to ensure coherence.

The subtyping algorithm accesses a view to these counters through an interface called `ContextInput` depicted in listing 7.14, which reads from the underlaying set of contexts whether a particular one exists. This interface is mediated by the code in listing 7.13 on the preceding page to the compatibility relation where it is made use of as described in section 5.4.3 on page 117. If the back-end cannot be queried directly for statistics, the conversion into a form that can be extracted and deposited must be done by the adapter.

```
public interface ContextInput {
  boolean contains( Context context );
}
```

Listing 7.14: Reader for statistics

Statistics are generated by the semantic evaluator `ContextBuilder`, which tracks the paths of the value carriers that passes before it in the same fashion as the technique presented in section 4.3.3 on page 99 albeit reacting to events instead of recursing over pre-built objects, following the principles outlined in section 6.4.1 on page 135. No semantic value is ever returned from this builder as it is only inserted in the handler chain for the sake of the side-effects it produces. Similarly, the check called `StatisticalGatherer` always returns **`true`**.

Notification of path occurrence is signaled to the back-end through the `ContextOutput` interface found in listing 7.15, which is counterpart to `ContextInput`, enabling statistics to be written into the repository. The framework is ignorant of the counter's actual value and care only for the change that should be made, allowing more efficient parallel processing as the requests can be reordered internally to better fit the desired scheduling.

```
public interface ContextOutput {
  void add( Context context );
  void sub( Context context );
}
```

Listing 7.15: Writer for statistics

As the counter is incremented during insertion of a document into the repository, it must correspondingly be decremented when that document is removed. A modification is handled as removal followed by addition.

7.3.4 Auxiliaries

The back-end interface is designed with regard to receiving and delivering data in a pipe to other components in a server system. This kind of organization is directed towards high performance and scalability, but at the expense of perhaps not providing simplicity to the degree desirable for the client.

To oblige clients of lighter weight, the framework bestows a module for reading and writing documents from an ordinary character string. This code also make out examples of the patterns in which methods from the back-end is normally used. Listing 7.16 contains for instance the logic to perform an upload and automatically commit a document to the repository. (Error handling is excluded from the snippet).

```
// class Helper
public boolean add( String schema, String id, String document ) {
  boolean commit = false;
  boolean result = false;
  ContentHandler handler = database.startPost( schema, id );
  try {
    reader.setContentHandler( handler );
    reader.parse( new InputSource( new StringReader( document ) ) );
    commit = true;
  }
  finally {
    result = database.endPost( handler, commit );
  }
  return result;
}
```

Listing 7.16: Adding strings the easy way

A `Helper` instance retain a reference to the back-end (here called `database`) selected so as to not requiring this to be passed as a parameter for every call since most calls will only be to one database throughout the program. It also aggregates a common XML `reader` component for all operations to not incur the cost of loading a new one every time. Although this increases throughput, it also effectively restricts the helper object to a single thread, which should be acceptable for most utilities. Both of these fields are initialized upon construction.

Closing the transfer with `endPost` is done inside a **`try...finally`** block in order to make this happen even if an exceptional condition arises, but it is only called if the invocation of `startPost` outside to scope of the block succeeds. Commitment of the data is first flagged **`true`** when there is no longer any chance of an exception being thrown from the reading process. Any abruption of the flow will hence cause `endPost` to be notified to of a rollback instead. This arrangement ensures that only fully completed data is committed, and that the manager is always given a chance to clean up any outstanding resources.

Notice finally that the result cannot be returned inside the **`finally`** clause as that would nullify any pending exceptions, and indeed is there no notion of a result if an exception has occurred; the back-end should never return **`true`** if a rollback was requested. Therefore, the result is rather stored in a local variable that is only relevant if normal

termination of the routine is reached.

Getting data *out* of the repository on the other hand is, as listing 7.17 will testify, a much less convoluted procedure due to most of the work being done by the `XMLWriter` class (cf. section 6.3.3 on page 134) and the only task carried out by the helper is providing the buffer into which the events will be serialized.

```
// class Helper
public String retrieve( String schema, String id ) {
  StringWriter output = new StringWriter();
  database.get( schema, id, new XMLWriter( output ), false );
  return output.toString();
}
```

Listing 7.17: Fetching a document into a character string

7.4 Summary

This chapter starts out with presenting contemplations about the organization of a repository. All documents are associated with a name that identifies them and a type. Schemata are also considered to be documents, belonging to the meta-schema.

An interface to a back-end general enough to accommodate writing adapters for a range of databases is defined, and it supports streaming of data so that intermediate buffering of the document will not have to take place. A simpler wrapper can be used for situations where this flexibility is not needed and only adds unneeded complexity.

Various realizations of the back-end such as those based on relational database management systems and native XML databases are then discussed, and an implementation of a database that is entirely based on consuming memory resources given, directed towards experimental purposes.

Finally, the repository is integrated with the algorithms for determining whether a document is in the language relation or a schema is in the subtyping relation, by installing checks as filters in the handler chain for data uploading. The same approach is used to gather statistics regarding schema usage from the documents that are installed.

Chapter 8

Conclusion

Finally, this last chapter will summarize the thesis by presenting visions for how the framework can be further augmented, provide an overview of the work that has been done, and review how well the goals it initially set out to solve have been accomplished.

8.1 Future work

Several directions may be taken to extend and enhance the library; more functionality can be added to the repository, or improvements can be made to the existing algorithms.

8.1.1 Repository

Version control. Multiple versions of a document may be allowed in the repository, each using a possible different schema as mentioned in section 1.2.1 on page 19. The purpose would not be to contain and insulate older versions from schema changes but to provide configuration management in form of an option to roll back changes. Schema differences can perhaps be used when determining how to resolve merging conflicts for check-ins whose origin have become stale.

Combining the name of a schema or document with a version number or date as an identifier seems to be the most straight-forward way to come up with an implementation for this.

Efficient search mechanism. The retrieval of data from the repository to be used in the stylesheets can be studied. Instead of merely extracting a resource known by its identity, a request can be formulated; the correct data is found and then piped through applicable stylesheets.

Acceleration of queries can be done by storing indices containing the data of commonly sought properties together with the paths in the documents matching these criteria. Maintaining an index for each and every element in a schema would probably lead to insertion and removal operations costing more than what is gained through the increased efficiency of retrieval.

Hand-tuning could therefore be brought into the equation by manually adding a special attribute on the applicable elements in the schema:

```
<element name="blum" hyp-ed:index="true"/>
```

Indeed could any kind of plug-in be associated with custom schema attributes. Encountering this attribute in the schema records the fact that the attribute should be indexed and when documents are entered in the repository, filters are installed according to these records to collect the necessary statistics.

Upgrading the schema will potentially invalidate the type with which an index is associated, and may also add new indices on top of the existing ones, triggering a full rescan of all the documents unless the system is designed so that rebuilds can be done incrementally as the use of "dirty" indices are detected.

Linking. Special support for XLink [DMO01] can be added so that embedding of other documents in the database is performed automatically when a presentation view of a document is requested. Edit views remains unchanged. Correspondingly, the repository can be enhanced to check that link targets are of the correct schema when a document containing pointers is uploaded, and to keep track of dependencies between documents.

Dependencies between documents and schemata and between documents themselves could for instance be viewed as RDF[1] triples, and be used to locate documents that fits certain criteria.

Repository of stylesheets. The model can be extended to also include the checking of schemata against existing stylesheets, and the checking of new stylesheets against the schemata for the documents they process. However, this requires that one is able to formulate the semantics of stylesheets as operations upon schema instances instead of document information sets. The gains of this would be large, as the content of the repositories would always be internally consistent. Such an improved system is illustrated in figure 8.1.

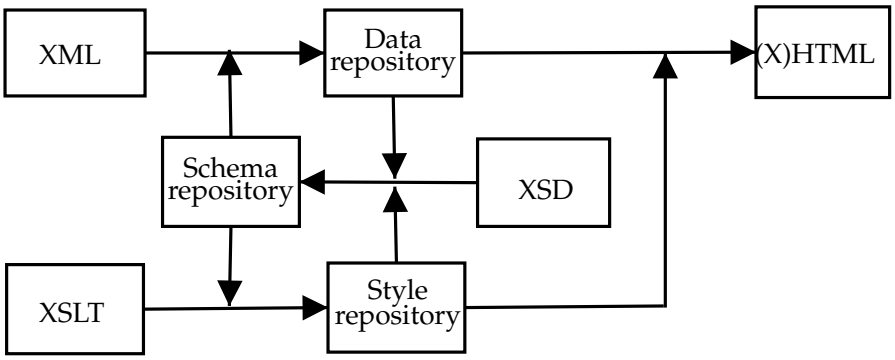


Figure 8.1: System where stylesheets participate in the repository

[1]RDF is an abbreviation of Resource Description Framework

8.1.2 Algorithm

The framework opens for experimentation with various algorithms for subtyping to compare their ability and to address the problem at hand. Particularly interesting would be to determine whether the procedure described in [KPS95] could be adapted for types with anonymous (a.k.a. untagged) unions, since this algorithm only has a time complexity of $O(n^2)$. In order to do this however, a mapping between λ-calculus and regular tree grammars must be devised and applied.

8.2 Results

The aim of this thesis is to obtain a way to control concordance between schema upgrades and existing documents, and its mandate is to create a component which in that respect ensure integrity in a content management system.

8.2.1 Overview

A library for checking validity and compatibility, directed towards integration in such systems is the result of the work presented, and this library provide the means to detect a conflict between a new schema version and the current content in the repository in order to protect the data from corruption. The framework is founded in a representation of the underlaying algebra, which again builds primarily on work done on regular tree grammars by Lee, Mani and Murata, and on subtyping by Hosoya, Vouillon and Pierce.

Contributions from this thesis to this field are: an adaption of the algebra to a Java library, modifications of the algorithms which restrict them to only take into account the current data, and integration of the system with a repository database. The subtyping algorithm is enhanced to branch over portions of the schema that is not marked as being used by any document.

Two solutions are established: One which runs the risk of experiencing spurious hits, and one which involves storing large amounts of statistics. The time complexity of the algorithm is $O(2^n)$, i.e. of exponential order, but where n is referring to the size of the schema and not the number of documents. In practice, blow-up due to problem size only occurs in some pathological cases which involves a lot of backtracking over non-deterministic symbols. Termination is always ensured, but without a performance guarantee. However, the author expect others to find that behavior for normal problems lays within acceptable limits.

8.2.2 Structure

The framework is targeted exclusively as a middle-tier to be included in other systems. Integration does not require intimate knowledge of the theory behind the algorithms, as documents can be read and processed from standard formats. Couplings to other parts are specified to an extent that still leaves flexibility for other adaptions and are then

demonstrated as a proof-of-concept for the purpose of testing and experimentation. Yet is development of front-ends and back-ends suitable for production use mainly a task that is outside the scope of this thesis and has a considerable potential for improvement. Natural candidates for such extensions are for instance a WebDAV front-end and a SQL database back-end.

Not all areas within the core are as copious as could be due to limitations in time and to keep the thesis' circumference at a manageable level. In particular does this apply to the meta-schema subset which could consist of more constructions from XML Schema such as namespaces and other simple datatypes than strings, and the repository manager could be extended to include an automatic revalidation of all potentially spurious hits, which could improve the attractiveness of the context-based approach to compatibility testing. Albeit a complete system is not delivered, the framework appears to be mature enough for inclusion in other projects.

Hence, the accompanying code is not directly applicable in production. While it is conceivable that a deployable system could be created from it with minimal effort, the best results will of course probably be obtained by annexing it to a larger, more encompassing one, as the framework is not intended as a turn-key solution in itself but rather as a component.

Four different "pillars" can categorize adequately the parts of which the system consist:

(1) Datatypes for regular tree grammars and hierarchical documents

(2) Algorithms for validation and subtyping

(3) Conversion to and from a standard format of exchange

(4) Interface to the storage layer

and the code is structured correspondingly. Other, related projects such as for instance XDuce, often take a two-layer approach where both an external and an internal representation is present, whereas this thesis attempts to use the same structures for all purposes. It remains yet to be seen if this strategy results in any specific advantages or disadvantages.

8.2.3 Implementation

The program consists of 74 classes in 4 different packages (grammars, markup, repository and miscellaneous auxiliaries) with more than 3700 non-commenting source statements and just about as much comments. Program testing is done by running a total of over 150 unit tests which complete in less than 5 seconds of total time. A set of options can be used to toggle between various approaches where more than one solution have been mentioned. This thesis together with the comments in the source code and the JavaDoc generated from them comprises the documentation for the framework. The unit tests also contains examples of its usage.

8.3 Lessons learned

In the process of constructing a program to go along with the thesis, the author has as a guideline tried to create source code that is comprehensible, straightforward and undemanding to use, and that has a functional flavor to make it easier to reason about its workings. There has however been no effort to concretize measures for these items, but the experience does suggest that they seem to be achievable even in Java, which is not normally coined as a language especially suitable for formal theory.

8.3.1 Design

An object-functional design like those selected for most algebras here, appears to be highly appropriate although it needs to be "helped out" in some places — like circle detection — by some imperative techniques. The implementation is otherwise done more or less as a continuation of the design. Any parts of the code that are not described have been left out because they do not show anything extra beyond what is already there.

It seems reasonable to conclude that code that is based in a theoretically strong foundation can be written in a more compact and more robust manner, while the lack of such conversely induces more compromises and workarounds. Before committing to a design, the model should first be experimented with to gain experience and understanding of the forces at work and then a new version devised from this afterwards.

8.3.2 Improvements

Lessons learned more concretely in that respect are that construction of carriers could perhaps be scrutinized more carefully to determine if automatically arriving at canonical elements would be feasible, list insertion could be more efficient, and more time should be devoted to the groundwork of finding homomorphisms between types.

8.3.3 Testing

Due to the absence of a larger system in which to perform real-world assessment, evaluation of the code has been done with unit testing and integration testing in between the modules themselves. The unit tests are directed at all parts of the library, in addition to some larger tests to probe handling of a larger load. The framework can successfully validate its own meta-schema, and is reckoned to be able to handle other common schemata too. Observation of problem solving methodology employed when the user is faced with schema incompatibility would undoubtedly be an interesting study in order to explore what further contributions that could be made from the framework, but this is considered to be outside the realm of the thesis.

8.4 Summary

The goal of contriving a method to check compatibility without performing a full scan through all documents is nonetheless considered to be reached.

Bibliography

[ABC+98] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, and Lauren Wood. Document object model (DOM) level 1 specification, 1998. `http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/`. (Cited on p. 96).

[ADL91] Rakesh Agrawal, Linda G. DeMichiel, and Bruce G. Lindsay. Static type checking of multi-methods. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 113–128, 1991. `http://www.almaden.ibm.com/cs/people/ragrawal/papers/oopsla91.ps`. (Cited on p. 38).

[AM91] Alexander Aiken and Brian R. Murphy. Implementing regular tree expressions. In *Lecture Notes in Computer Science 523*, pages 427–447. Springer–Verlag, 1991. `http://www.cs.berkeley.edu/~aiken/publications/papers/fpca91.ps`. (Cited on pp. 55, 55, 71).

[App89] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software — Practice and Experience*, 19(2):171–183, February 1989. `http://www.cs.princeton.edu/~appel/papers/143.ps`. (Cited on p. 49).

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison–Wesley, 1986. ISBN 0-201-10088-6. (Cited on pp. 26, 27, 83, 135, 135).

[BCF97] Alejandro Bia, Rafael C. Carrasco, and Mikel L. Forcada. Identifying a reduced DTD from marked up documents, 1997. (Cited on p. 109).

[BLFM98] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. Uniform resource identifiers (URI): Generic syntax, 1998. `http://www.ietf.org/rfc/rfc2396.txt`. (Cited on p. 122).

[BM01] Paul V. Bison and Ashok Malhotra. XML schema part 2: Datatypes, 2001. `http://www.w3.org/TR/xmlschema-2`. (Cited on p. 53).

[BN97] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. Morgan Kaufmann, 1997. ISBN 1-55860-415-4. (Cited on pp. 56, 57, 152, 152).

[Boe95] Hans-J. Boehm. Mark-sweep vs. copying collection and asymptotic complexity, 1995. `ftp://parcftp.xerox.com/pub/gc/complexity.html`. (Cited on p. 42).

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison–Wesley, 2nd edition, 1994. ISBN 0-805-35340-2. (Cited on p. 32).

[Bou03] Ronald Bourret. XML and databases, 2003. `http://www.rpbourret.com/xml/XMLAndDatabases.htm`. (Cited on p. 157).

[BPSMM00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. Extensible markup language (XML) 1.0, 2000. `http://www.w3.org/TR/REC-xml`. (Cited on pp. 27, 110, 115, 119, 121).

[Bro01] David Brownell. About SAX, 2001. `http://www.saxproject.org`. (Cited on p. 96).

[CD99] James Clark and Steve DeRose. XML path language (xpath), 1999. `http://www.w3.org/TR/1999/REC-xpath-19991116`. (Cited on p. 97).

[CDG+02] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2002. `http://www.grappa.univ-lille3.fr/tata/tata.pdf`. (Cited on pp. 29, 71, 142).

[CM01] James Clark and Makoto Murata. RELAX NG tutorial, 2001. `http://www.oasis-open.org/committees/relax-ng/tutorial-20011203.html`. (Cited on pp. 110, 121, 142).

[DMO01] Steve DeRose, Eve Maler, and David Orchard. XML linking language (XLink) version 1.0, 2001. `http://www.w3.org/TR/xlink/`. (Cited on p. 166).

[EMRS97] Sofoklis G. Efremidis, Khalid A. Mughal, John H. Reppy, and Lars Søraas. AML: Attribute grammars in ML. *Nordic Journal of Computing*, 4:37–65, 1997. `ftp://ftp.ii.uib.no/pub/aml/njc-aml-paper.ps.Z`. (Cited on pp. 61, 88, 135).

[FHK+02] Torsten Fiebig, Sven Helmer, Carl-Christian Kanne, Julia Mildenberger, Guido Moerkotte, Robert Schiele, and Till Westmann. Anatomy of a native XML base management system, 2002. `http://lsirpeople.epfl.ch/aberer/citations/TR-02-001.pdf`. (Cited on p. 157).

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*, pages 76–106. University of California, Irvine, 2000. `http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm`. (Cited on p. 151).

[FK99] Daniela Florescu and Donald Kossmann. A performance evaluation of alternative mapping schemes for storing xml data in a relational database. Technical Report No. 3680, INRIA, 1999. `http://www-caravel.inria.fr/dataFiles/FK99.ps`. (Cited on p. 157).

[Gen03] Jonathan Gennick. Make XML native and relative, 2003. `http://otn.oracle.com/oramag/oracle/03-jan/o13xml.html`. (Cited on p. 157).

[GGR+00] Minos Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. XTRACT: a system for extracting document type descriptors from XML documents. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 165–176, 2000. `http://www.bell-labs.com/project/serendip/Papers/sigmod00-cam.ps.gz`. (Cited on p. 90).

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison–Wesley, 1995. ISBN 0-201-63361-2. (Cited on pp. 38, 116).

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison–Wesley, 2nd edition, 2001. ISBN 0-201-44124-1. (Cited on pp. 25, 28, 123, 142).

[Hol99] G. Ken Holman. When to use attributes as opposed to elements, 1999. `http://xml.coverpages.org/holmanElementsAttrs.html`. (Cited on p. 89).

[Hol02] Nils M. Holm. Where does the name *car* come from?, 2002. `http://www.not-compatible.org/LISP/QA/carcdr.html`. (Cited on p. 43).

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, 2000. `http://www.kurims.kyoto-u.ac.jp/~hahosoya/papers/regsub.ps`. (Cited on pp. 51, 55, 71, 71, 72, 76, 123).

[KPS95] Dexter Kozen, Jens Palsberg, and Michael I. Schwartzbach. Efficient recursive subtyping. *Mathematical Structures in Computer Science*, 5(1):113–125, 1995. `http://www.cs.purdue.edu/homes/palsberg/paper/mscs95-kps.ps.gz`. (Cited on pp. 56, 72, 167).

[Kro95] David M. Kroenke. *Database processing: Fundamentals, Design and Implementation*. Prentice Hall, 5th edition, 1995. ISBN: 0-13-320128-7. (Cited on p. 157).

[KT65] Charles H. Kepner and Benjamin B. Tregoe. *The rational manager: a systematic approach to problem solving and decision making*. McGraw–Hill, 1965. ISBN 0-07-034175-3. (Cited on p. 15).

[Lea97] Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison–Wesley, 1997. ISBN 0-201-69581-2. (Cited on p. 152).

[LEW96] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of Abstract Data Types*. John Wiley & Sons, 1996. ISBN 0-471-95067-X. (Cited on pp. 32, 63).

[LG00] Barbara Liskov and John V. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison–Wesley, 2000. ISBN 0-201-65768-6. (Cited on p. 70).

[LMM00a] Dongwon Lee, Murali Mani, and Makoto Murata. On the expressive power and closure properties of XML schema languages. `http://www.cs.ucla.edu/~mani/xml/papers/conferences/WebDB2001/td-main2.ps`, 2000. (Cited on pp. 32, 32).

[LMM00b] Dongwon Lee, Murali Mani, and Makoto Murata. Reasoning about XML schema languages using formal language theory. RJ#10197 Log#95071, IBM Almaden Research Center, 2000. `http://www.cobase.cs.ucla.edu/tech-docs/dongwon/ibm-tr-2000.pdf`. (Cited on pp. 29, 30, 31, 31, 32, 123).

[LS98] Paul J. Leach and Rich Salz. UUIDs and GUIDs, 1998. `http://www.opengroup.org/dce/info/draft-leach-uuids-guids-01.txt`. (Cited on pp. 111, 148).

[Mar96] Robert C. Martin. Acyclic visitor (v1.0), 1996. `http://www.objectmentor.com/resources/articles/acv.pdf`. (Cited on p. 38).

[McG01] Sean McGrath. Attributes versus elements: The never-ending choice, 2001. `http://www.itworld.com/nl/xml_prac/12132001/pf_index.html`. (Cited on p. 88).

[MDB01] Rajiv Mordani, James Duncan Davidson, and Scott Boag. Java API for XML processing, 2001. `http://java.sun.com/xml/jaxp/dist/1.1/jaxp-1_1-spec.pdf`. (Cited on p. 131).

[Mit01] Joseph Mitchell. PostgreSQL's multi-version concurrency control, 2001. `http://www.onlamp.com/lpt/a/872`. (Cited on p. 152).

[ML02] Murali Mani and Dongwon Lee. XML to relational conversion using theory of regular tree grammars. In *VLDB Workshop on Efficiency and Effectiveness of XML Tools, and Techniques (EEXTT)*, Hong Kong, China, 2002. `http://nike.psu.edu/publications/eextt02.pdf`. (Cited on p. 156).

[MLS02] Michael Mealling, Paul Leach, and Rich Salz. A UUID URN namespace, 2002. `ftp://ftp.ietf.org/internet-drafts/draft-mealling-uuid-urn-00.txt`. (Cited on p. 148).

[MS03] Erik Meijer and Wolfram Shulte. Unifying tables, objects and documents, 2003. `http://research.microsoft.com/~emeijer/Papers/oopsla.pdf`. (Cited on p. 22).

[Mur99] Makoto Murata. Hedge automata: a formal model for XML schemata, 1999. `http://www.geocities.com/ResearchTriangle/Lab/6259/hedge_nice.pdf`. (Cited on pp. 30, 130).

[Pau91] Lawrence C. Paulson. *ML for the working programmer*. Cambridge University Press, 2nd edition, 1991. ISBN 0-521-56543-X. (Cited on p. 38).

[Pie02] Benjamin C. Pierce. Xtatic overview, 2002. `http://www.cis.upenn.edu/~bcpierce/xtatic/outline.html`. (Cited on p. 22).

[Pre98] Paul Prescod. Formalizing SGML and XML instances and schemata with forest automata theory, 1998. `http://www.prescod.net/forest/shorttut/`. (Cited on p. 29).

[Sei90] Helmut Seidl. Deciding equivalence of finite tree automata. *SIAM Journal on Computing*, 19(3):424–437, 1990. (Cited on p. 71).

[SG98] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison–Wesley, 5th edition, 1998. ISBN 0-201-59113-8. (Cited on p. 153).

[SR01] Ian Stokes-Rees. fixed and default, 2001. `http://lists.w3.org/Archives/Public/xmlschema-dev/2001Apr/0063.html`. (Cited on p. 126).

[TBMM01] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML schema part 1: Structures, 2001. `http://www.w3.org/TR/xmlschema-1`. (Cited on pp. 27, 47, 110, 115, 121, 148).

[vdV01] Eric van der Vlist. Using W3C XML schema, 2001. `http://www.xml.com/lpt/a/2000/11/29/schemas/part1.html`. (Cited on p. 121).

[Wil01] Sam Wilmott. Content model algebra, 2001. `http://home.chello.no/~mgrsby/sgmlintr/sgmlcont.htm`. (Cited on p. 33).

[Zuk01] John Zukowski. Magic with Merlin: Long-term persistence. Serialize JavaBean component state to XML, 2001. `http://www.ibm.com/developerworks/java/library/j-mer0731/index.html`. (Cited on p. 136).

Index