Abstract Interpretation of XSLT

Abstrakte Interpretation von XSLT
Bachelor-Thesis von Kai Patrick Reisert
März 2015

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department of Computer Science
Software Technology Group

Abstract Interpretation of XSLT
Abstrakte Interpretation von XSLT

Vorgelegte Bachelor-Thesis von Kai Patrick Reisert

Examiner: Prof. Dr.-Ing. Mira Mezini
Supervisor: Dr. rer. nat. Sebastian Erdweg

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 16. März 2015

(Kai Patrick Reisert)

Abstract

XSLT is a language for defining transformations between XML documents. Our goal is to statically know the structure of output documents resulting from such transformations. To this end, we develop a static analysis. Following the concept of abstract interpretation, our analysis is based on a concrete interpreter for a large fragment of XSLT 1.0 and XPath 1.0, which we then modify to work with abstractions of the involved data structures. Thus, we approximate the result of a given XSLT transformation without knowing concrete input documents.

Since the abstractions have to be defined in terms of complete lattices in order to apply abstract interpretation, we define and implement abstractions fulfilling this property for XPath values, XML nodes and lists of XML nodes. An evaluation shows that these abstractions are expressive enough to encode some important properties of the structure of XML documents. We allow the use of a user-defined recursion limit to ensure termination of our analysis in the presence of recursive template invocations.

Contents

1 Introduction

The e**X**tensible **M**arkup **L**anguage (XML) is used in a wide variety of contexts for storing, exchanging and processing structured data. Because different applications require different formats, transforming XML documents from one structure into another is important, and XSLT is a language that is specifically designed for that purpose.

As depicted in Figure 1.1, the task of an XSLT processor (or interpreter[1]) is to process an input document according to the rules defined in a given XSLT stylesheet and produce an output document.

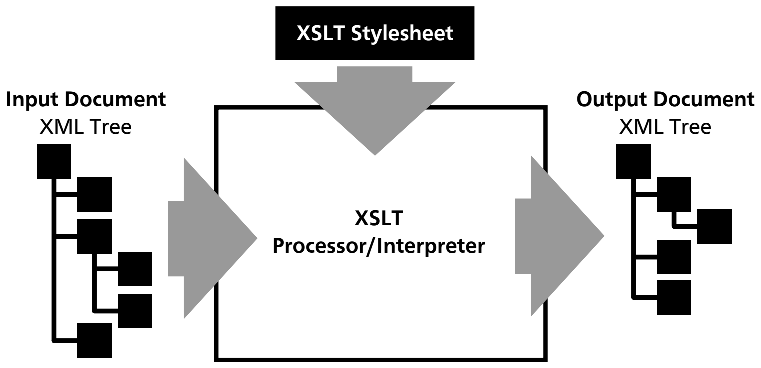

Figure 1.1: Schematic visualization of XSLT processing

Such an output document is usually required to have a specific structure, also called *schema*, which is defined either informally by the application that will deal with the data, or formally by a Document Type Definition (DTD) or an XML Schema Definition (XSD). Since XSLT works in a way that is independent of such schema definitions, results of XSLT transformations do not necessarily adhere to the expected or required schema. Therefore, static analysis of XSLT stylesheets with the aim of making statements about the output schema without knowing concrete input documents is useful.

This thesis explores the feasibility of using abstract interpretation to approach this goal: We first implement an XSLT processor that transforms concrete inputs, and in a second step introduce abstractions of the involved data structures, modifying the interpreter in such a way that it produces an abstract result from an abstract input. Such an abstract input can represent the "set of all possible XML documents", and the result then describes the structure of a subset of documents that can result from a given stylesheet.

As abstract interpretation uses abstractions based on the mathematical concept of complete lattices, we can not use existing XML schema definition formats such as DTDs directly. Instead, a large part of our work deals with the definition and implementation of an abstract domain interface that encodes the ways in which the abstract interpreter may operate on the aforementioned abstractions.

As far as the scope of this thesis is concerned, we aim to support a subset of XSLT 1.0 that is sufficiently rich in features that our approach can be tested at least with some publicly available "real-world" transformations. Although more recent versions of the XSLT standard exist, we decided to restrict our attention to XSLT 1.0 for the following reasons:

- Later versions introduced substantially more complicated features and we wanted to keep the implementation as simple as possible in order to explore the feasibility of abstract interpretation.

- XSLT 1.0 is still actively used because all major web browsers support it natively for client-side processing. Later versions of XSLT, however, are not supported by any browser [1].

[1] We will use the terms *processor* and *interpreter* interchangeably. The former is the usual term in the context of XSLT, the latter becomes more fitting when we talk about *abstract interpretation*.

While the performance of our implementation is generally not among our objectives, we will give details about an optimization that increases the performance of our analysis for some inputs, thus improving its usability.

The rest of this thesis is structured as follows: Chapter 2 introduces XSLT and the closely related XPath language, as well as the concepts of abstract interpration. Chapter 3 presents our implementation of an XSLT processor to be used with concrete input documents. Based on this concrete interpreter, Chapter 4 discusses our implementation of an abstract interpreter. In Chapter 5 we evaluate our implementation by demonstrating analysis results of some real-world transformations in addition to smaller test cases. Finally, Chapter 6 discusses other work related to static analysis of XML transformations.

2 Background

This chapter gives some background information that is necessary to understand abstract interpretation in the context of XSLT. First, we give an introduction into XSLT as well as XPath, which is closely related. After that, the essentials of abstract interpretation are covered, as far as they are mandatory to understand the rest of this thesis.

2.1 XSLT and XPath

XSLT is the abbreviated name for *e**X**tensible **S**tylesheet **L**anguage: **T**ransformations* and it was developed and specified by the World Wide Web Consortium (W3C) as part of a wider set of technologies concerned with styling and transforming XML documents, called XSL [2, 3]. XSLT is designed to transform one XML document into another, with the original intention that the output document is more suitable for presentation (e.g. (X)HTML) than the source document, but this does not have to be the case. In many cases, XSLT is also used to convert between different XML-based formats from the same application domain, e.g. office documents or pieces of music, or even between different versions or variations of the same format (cf. the collection of various stylesheets presented in Section 5.2).

2.1.1 XML

Because XSLT transformations deal with XML data and are even written as XML documents themselves, some understanding of XML is required, which shall be provided in this brief section.[1]

XML documents represent a tree of *nodes*. The root of this tree (*root node*) is not directly visible in the textual representation, but rather describes the document as a whole. The root node has exactly one child, which is an *element node* (actually there can also be a number of processing instructions and comments, but our implementation does not support this). Element nodes have a name as well as an arbitrary number of *children* and *attributes* (elements correspond to *tags* in the textual representation). An element's child can either be another element, a *text node*, a *comment node*, or a *processing instruction node*. For simplicity, we do not support processing instructions at all, neither do we support namespaces – which are prefixes of the names of elements and attributes – with the exception of the `xls` namespace in stylesheets, which is needed to distinguish between XSLT instructions and literal output elements.

As we are mainly interested in the logical tree representation of XML documents, we do not deal with specifics of the textual representation, such as input encodings, CDATA-sections, whitespace handling and special character entities, which means that special characters such as '<' and '>' experience no special treatment in our implementation and could be used within text nodes, for example.

The following listing shows the textual representation of a simple XML document containing elements (`catalog`, `cd`), attributes (`artist`, `year`) and text nodes (everything in between `<cd>...</cd>`):

```xml
<catalog>
  <cd artist="Gary Moore" year="1990">Still got the blues</cd>
  <cd artist="Eros Ramazotti" year="1997">Eros</cd>
  <cd artist="Bee Gees" year="1998">One night only</cd>
</catalog>
```

Listing 2.1: Simple XML document with elements, attributes and text nodes[2]

[1] In this thesis we use the terms as they are used in the XPath 1.0 specification [4, Section 5]. In later versions as well as in the official XML document object model (DOM) some names differ slightly.

[2] This example is inspired by `http://www.w3schools.com/xml/cd_catalog.xml` – accessed 2015-01-07

The XPath specification also defines an order on the nodes in an XML document: The *document order* is the order in which nodes appear in the textual representation of the document, which can be seen as a depth-first ordering of the XML tree.

2.1.2 XSLT Stylesheets

As already mentioned and apparent from Listing 2.2, XSLT stylesheet definitions are written using XML themselves. They consist of a number of *template rules*, also called just *templates*, that are used to transform a single input node into a sequence of output nodes.

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
  <xsl:template match="/">
    <html>
      <body>
        <h2>Available CDs</h2>
        <ul>
          <xsl:apply-templates select="catalog/cd"/>
        </ul>
      </body>
    </html>
  </xsl:template>
  <xsl:template match="cd">
    <li>
      <b><xsl:value-of select="text()"/></b> by <xsl:value-of select="@artist"/>
    </li>
  </xsl:template>
</xsl:stylesheet>
```

Listing 2.2: Simple XSLT stylesheet applicable to the document in Listing 2.1

Template rules are either invoked directly by name or selected by a process called *matching* where for a given input node a template is chosen that fits that node best (details of this process are described in Section 2.1.4). When a template has been selected, it is subsequently *instantiated* for the current input node, which means that the content of the template is evaluated with respect to the input node, which is then part of the *evaluation context*.

The content of a template is a mix of literal output nodes and elements from the XSLT namespace[3] (e.g. `<xsl:value-of>`), which we will be calling *instructions*. While we refer to the XSLT specification [3] for a complete list of possible instructions, those instructions that are supported in our implementation are explained in Section 3.3.

Each instruction can generate a sequence of XML nodes and may contain invocations of other templates (again, either by name using `<xsl:call-template>` or by matching using `<xsl:apply-templates>`), which are then instantiated in turn and the results are inserted at the position of the invocation.

The evaluation of an XSLT stylesheet starts with the root node of the input document: A matching template rule is searched for that node and the resulting template is instantiated with the root node as the *context node*. The output of that template must be a single element node (with arbitrary attributes and children) because it will be made the (only) child of the root node that constitutes the output document of the transformation.

If no user-defined template matches the root node, one of the built-in template rules is used [3, Section 5.8]: It recursively applies matching templates on the input's children if the input is the root node or an element. Other built-in rules are defined to copy text nodes as well as the values of attribute nodes to the output, and to ignore comments (and processing instructions). Because of the built-in rules, it can never happen that no rule matches a given node.

[3] That namespace has the URI `http://www.w3.org/1999/XSL/Transform`. Though any prefix can be associated with this namespace in a stylesheet definition, usually the `xsl` prefix is used for referring to elements within that namespace. We follow this practice.

In the example presented in Listing 2.2, the first template matches the root node and generates an XHTML page as a result. Inside the <ul> element, all <cd> elements from the input are selected and a matching template is instantiated for each of those. The template rule that matches those elements will be the second one, which outputs a list item for each of the <cd> elements, presenting some information about that CD.

```
<html>
  <body>
    <h2>Available CDs</h2>
    <ul>
      <li><b>Still got the blues</b> by Gary Moore</li>
      <li><b>Eros</b> by Eros Ramazotti</li>
      <li><b>One night only</b> by Bee Gees</li>
    </ul>
  </body>
</html>
```

Listing 2.3: XHTML output of the transformation in Listing 2.2

In this case, the output is an XHTML document that can be viewed in a web browser, but the result could also be in any other XML-based format. XSLT even allows the resulting document to be in plain text format [3, Section 16], but for the purpose of our work we only consider XML output.

2.1.3 XPath

XPath [4] is another part of XSL and though its name-giving purpose is to describe paths in XML documents, it is used in various places in XSLT to express conditions and more general calculations and computations involving strings, numbers, boolean values and (sets of) XML nodes.

In contrast to XSLT, the syntax of XPath is not based on XML. Instead, XPath is an expression-oriented language, which is embedded into XSLT stylesheets in the form of attribute values. A simplified grammar of XPath expressions is depicted in Figure 2.1 and we refer to the XPath specification for a precise formal definition [4, Sections 2 and 3].

$$
\begin{array}{lll}
\langle \textit{expr} \rangle & ::= & \langle \textit{expr} \rangle \text{ `+' } \langle \textit{expr} \rangle \mid \langle \textit{expr} \rangle \text{ `-' } \langle \textit{expr} \rangle \mid \text{ `-' } \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{expr} \rangle \text{ `*' } \langle \textit{expr} \rangle \mid \langle \textit{expr} \rangle \text{ `}\texttt{div}\text{' } \langle \textit{expr} \rangle \mid \langle \textit{expr} \rangle \text{ `}\texttt{mod}\text{' } \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{expr} \rangle\ \langle \textit{rel-op} \rangle\ \langle \textit{expr} \rangle \mid \langle \textit{expr} \rangle \text{ `}\texttt{and}\text{' } \langle \textit{expr} \rangle \mid \langle \textit{expr} \rangle \text{ `}\texttt{or}\text{' } \langle \textit{expr} \rangle \mid \langle \textit{expr} \rangle \text{ `|' } \langle \textit{expr} \rangle \\
 & \mid & \langle \textit{string-literal} \rangle \mid \langle \textit{number-literal} \rangle \mid \langle \textit{function-call} \rangle \mid \langle \textit{variable-reference} \rangle \mid \langle \textit{location-path} \rangle
\end{array}
$$

$$\langle \textit{rel-op} \rangle ::= \text{ `=' } \mid \text{ `!=' } \mid \text{ `<' } \mid \text{ `>' } \mid \text{ `<=' } \mid \text{ `>='}$$

$$\langle \textit{function-call} \rangle ::= \langle \textit{identifier} \rangle \text{ `(' } (\ \langle \textit{expr} \rangle \ (\text{`,' } \langle \textit{expr} \rangle)^* \ )? \text{ `)'}$$

$$\langle \textit{variable-reference} \rangle ::= \text{ `\$' } \langle \textit{identifier} \rangle$$

$$\langle \textit{location-path} \rangle ::= \text{ ... explained below ...}$$

Figure 2.1: Simplified XPath expression grammar

XPath has string and number literals (instead of boolean literals, the parameter-less functions `true()` and `false()` are used), operators to combine them, and predefined functions that allow more complex operations, for example data type conversion (`number(...)`, etc.) or string manipulation (`concat(...)`, `substring(...)`, etc). There is no way to define custom functions, but the specification allows implementation-defined extension functions.[4]

Variable references can be used to refer to the value of variables, but the assignment of values to variables is only possible outside of XPath (using the `<xsl:variable>` instruction or template parameters). The definition and usage of variables in XSLT/XPath follows static scoping rules.

The evaluation of XPath expressions is basically side-effect free, but because XPath is not statically typed, there is the possibility of run-time type errors, although most operations just convert their input to the required data type. Apart from run-time errors, every XPath expression evaluates to a result of one of the following data types [2]: *String* (a sequence of unicode characters), *Number* (a double-precision floating point number as defined in IEEE 754), *Boolean* (true or false) or *Node-set* (an ordered set of nodes in the source tree). There is one additional data type called *Result Tree Fragment* that is not defined by XPath itself, but by XSLT. However, a Result Tree Fragment behaves mostly like a node-set containing a single root node[5] , with some restrictions on the permitted operations [3, Section 11.1]. For this reason, in our implementation we can treat Result Tree Fragments as if they were node-sets, making the simplifying assumption that the prohibited operations are never used.

The distinctive syntactical feature of XPath is the ability to use *(location) path expressions* in order to obtain a set of XML nodes from the input document [4, Section 2]. For example, a location path expression might look like the one in Listing 2.4, that, when applied to the document in Listing 2.1, selects the '`year`' attributes of all <cd> elements.

```
/child::catalog/child::cd/attribute::year
```

Listing 2.4: Location path that selects the '`year`' attributes in the example above

Location paths are built from *location steps*, separated by the '/' character (the example path is made of three such steps). Every step then contains an *axis*-specifier and a *node test* seperated by '`::`', as well as (optionally) a number of *predicates*.

- The *axis* specifies the relationship between the nodes selected by the current step and the ones selected by the previous step, for example whether we are interested in children, descendants, parents or attributes of the nodes selected in the preceding step. There is a total of 13 possible axes.

- The *node test* specifies the type and (optionally) name of the selected nodes. For example, `child::text()` selects all children that are text nodes. It is noteworthy that children and attributes are distinguished by their axis, not by their type, so `child::*` selects all child nodes while `attribute::*` selects all attributes (the '*' character is used as a wildcard to match any name). The node test `node()` is true for all nodes.

- A *predicate*[6], syntactically specified in square brackets at the end of a step, limits the selection to only a subset using an arbitrary boolean XPath expression. If that expression evaluates to true, a given node is included in the result. If it evaluates to false, it is not included.

If a path starts with the '/' character, it is said to be *absolute* and the evaluation of the path starts with the root node of the input document. The evaluation of paths that are not absolute (i.e. *relative*) starts with the current *context node*.

There are a number of syntactic abbreviations for commonly used location steps:

[4] Extensions such as EXSLT (`http://exslt.org`) define some commonly used extra functionality and are implemented by various XSLT processors, but these exceed the scope of this work.

[5] In the case of a Result Tree Fragment, the root node may have arbitrary children instead of only a single element child, because such a root node represents not a complete document, but an arbitrary fragment of an XML document.

[6] Predicates are only mentioned here for the sake of completeness. We decided to skip them in our implementation in order to simplify both XPath evaluation and template pattern matching (for the latter, cf. Section 2.1.4).

- When the axis specifier is left out, the `child` axis is used implicitly.
- The '`@`' prefix can be used as an abbreviation for the `attribute` axis.
- '`//`' is a shorthand for `/descendant-or-self::node()/` and selects the node itself and all of its descendants.
- The step '`.`' is short for `self::node()`.
- The step '`..`' is short for `parent::node()`.
- In the position of a predicate, a numeric literal can be used instead of a boolean expression. In that case, it is equivalent to the expression `position() = <num>`, which checks the current node's position among the selected nodes.

The path from Listing 2.4 can therefore also be written as `/catalog/cd/@year`. We will prefer this short notation in the rest of this thesis.

When a location path is evaluated, the result will be a *node-set* containing all the selected nodes. Two node-sets can be joined with the union operator '`|`', but every source tree node will appear at most once in the result, as it is a set. Additionally, because in various places node-sets need to be traversed in *document order*, node-sets can be regarded as ordered sets, although this is not explicitly prescribed by the XPath specification.

2.1.4 Patterns and Matching

XSLT template rules identify the nodes to which they apply by using *patterns*. Syntactically, patterns form a subset of XPath location paths, which is evidence of how closely related these two technologies are. In patterns, steps may only use the `child` and `attribute` axes. Additionally, the '`//`' operator is allowed (as noted above, this is a shorthand for `/descendant-or-self::node()/`, but other uses of the `descendant-or-self` axis are not allowed in patterns).

According to the specification, "a pattern is defined to match a node if and only if there is possible context such that when the pattern is evaluated as an expression with that context, the node is a member of the resulting node-set" [3, Section 5.2].

This means that the location path in Listing 2.4, which is also a valid pattern, matches any node that would be in the resulting node-set when that path were evaluated (because it is an absolute path, the context node does not matter here), i.e. it matches any node that is an attribute named 'category' and a child of the 'item' element, which must itself be a child of the root node (the only child, to be precise, because the root node can only have one element child).

The intuitive way to understand a pattern and check whether it matches a given node is to evaluate the steps from right to left: The rightmost step must match the given node and the parent of that node (or any ancestor in case of the '`//`' operator) must match the rest of the pattern.

Multiple patterns can also be combined with the union operator '`|`'. A template rule where this is the case is treated as if it was defined multiple times, once for each single pattern. This effectively means that the template matches any of the given patterns.

Furthermore, template rule definitions can specify a *mode*, so that there can be different sets of rule definitions for the same patterns. When templates are applied, one can select one of those modes with the optional `mode` attribute in the `<xsl:apply-templates>` instruction. Whenever this attribute is missing, the default ("empty") mode is used.

When multiple template rules match a given node (which is usually the case because the built-in rules already match any node), an order is defined for choosing the template that will actually be instantiated [3, Section 5.5]:[7]

[7] The specification states this as a matter of eliminating all templates with lower precedence/priority and allows, but does not require, to signal an error when multiple templates remain after that elimination.

- The templates are first ordered by *import precedence*, so templates with higher import precedence are preferred over templates with lower import precedence. We do not support imports, but built-in templates are defined to have lower import precedence than user-defined ones, so this ensures that the latter are preferred.
- Templates with equal import precedence are then ordered by their *priority*. Custom priorities can be assigned, but by default the priority is calculated from the specificity of the pattern, in such a way that more specific patterns are assigned a higher priority.
- If this still leaves templates with equal import precedence and equal priority, the templates are ordered by the position of their definition in the stylesheet, in such a way that the template is preferred that is defined last.

These rules in conjunction with the fact, that the built-in templates already cover all possible nodes (in every possible mode), ensure that exactly one template is chosen whenever a matching template for any given input node is requested.

2.2 Abstract Interpretation

Abstract interpretation is a theory and formal method of static analysis of computer programs. It was first introduced by Patrick Cousot and Radhia Cousot in 1977 [5].

Static analysis in general tries to make statements about the execution behavior (semantics) of a given program without actually executing it. There are various ways to achieve this goal, and abstract interpretation approaches it by approximating the execution using abstractions of the program input, output and intermediate states. This means that instead of dealing with concrete objects, an abstract interpreter uses abstract objects that are modeled in such a way that they represent those properties of the concrete objects that one is interested in.

For example, when analyzing a program that deals with integer computations, we might only be interested in the parity of the values (whether a number is even or odd). Therefore, instead of concrete integers from the domain $\mathbb{Z}$, we are now using abstract objects from the domain $P = \{\textbf{even}, \textbf{odd}\}$. Given a program that takes any integer as an input, multiplies it by 2 and returns the result, a suitable abstract interpreter that uses a domain like P could statically determine that this program will always return an even number. However, if the interpreter does not know about a program whether its result will be even or odd, it can not express such an “unknown” value within P. This reveals that P is not actually sufficient as a domain and we need to extend it to be a *complete lattice*, which is the formal mathematical structure that the concept of abstract interpretation is based on.

A *complete lattice* is a set L with a partial ordering $\leq$, such that all subsets have least upper bounds as well as greatest lower bounds. For any two elements $a, b \in L$, we write $a \sqcup b$ for the least upper bound (also called *supremum* or *join*) of a and b, and likewise $a \sqcap b$ for the greatest lower bound (also called *infimum* or *meet*). The supremum of the entire set $\top = \bigsqcup L$ gives us the greatest element, also called *top* element. The infimum of the entire set $\bot = \bigsqcap L$ is the least element, also called *bottom* element.

Our domain P can be turned into a complete lattice by adding top and bottom elements, e.g. $\top = \textbf{any} = \textbf{even} \sqcup \textbf{odd}$ and $\bot = \textbf{none} = \textbf{even} \sqcap \textbf{odd}$. The top element now represents a number that is even *or* odd, and this is exactly what we were missing to express a number with unknown parity. The bottom element would represent any number that is both even *and* odd, but no such number exists. Hence, $\bot$ as the result of an abstract computation signifies that any corresponding concrete computation would have had no result at all, for instance because of an error or exception.

For such an abstract domain, we can also define a mapping that maps each abstract object to a set of concrete objects. This mapping is commonly known as the *concretization function* γ [6, Section 4.3], and we mention this because it is helpful to think of abstract objects as the sets of concrete objects that they represent. Figure 2.2 visualizes the concretization function of our (extended) domain P.

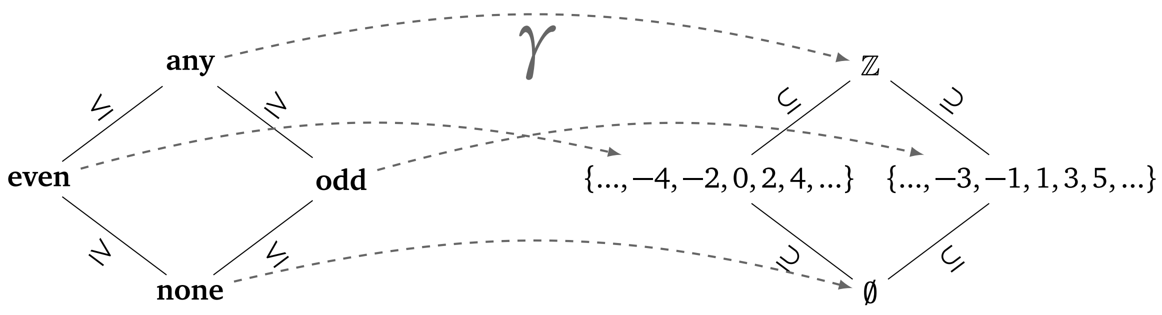

Figure 2.2: Lattice P with concretization function γ

It is noteworthy that the structure on the right also forms a complete lattice with the subset relation $\subseteq$ as partial ordering, and γ upholds the partial ordering in such a way that if $a \leq b$ then $\gamma(a) \subseteq \gamma(b)$ for any $a, b \in P$. The partial ordering also conveys a notion of precision: Greater elements in the lattice are less precise, because they represent a larger set of concrete objects. Consequently, the greatest element $\top$ is the least precise, which is also apparent from the fact that it represents the entire set of possible concrete values.

In order to be useful, the approximation that is computed by an abstract interpreter must be correct, i.e. given some abstract object a as input, the result of every run of the concrete program on any concrete input $c \in \gamma(a)$ must be included in the abstract result. On the other hand, the abstract result will most probably be an over-approximation of the set of concrete results that are actually possible.

The precision of the abstract interpretation depends on the granularity of the abstract domain: If the domain is too coarse, it might not be able to convey the amount of information that one is interested in. However, if the domain is too fine-grained, the analysis might be too expensive or even impossible to compute. For example, one could try to use the power set $\mathscr{P}(\mathbb{Z})$ as an abstract domain for $\mathbb{Z}$, but it contains infinite sets that are not even representable within software that has only a finite amount of memory available.

Multiple abstract domains like P can be combined to form new abstractions [6, Chapter 4.4]: Different abstractions of the same concrete data can be combined to obtain a more useful and precise abstraction of the concrete domain. For example, for $\mathbb{Z}$ one could also use an abstract domain that ignores everything but the *signs* of the values. Then, this domain could be combined with P in order to analyze both the signs and the parities in a single step. Moreover, another kind of combination allows multiple abstractions of individual components of a composite structure to be combined in order to obtain an analysis for the whole structure. These techniques enable us to build complex abstract domains from simple ones in a way that does not greatly complicate reasoning about them.

The last issue we want to consider is that of control flow in the analyzed program. What happens in the presence of conditional branches, or even loops or recursive function calls? The former can be dealt with easily by analyzing every possible branch independently and then *joining* the result. In the case of P, if we do not know which branch will be taken, and the first one yields an even result while the other returns an odd one, the result will be $\top$. Loops and recursion, however, are more difficult because they may introduce cases where our analysis is not guaranteed to terminate any more. For instance, consider the domain P again and a program that contains a function that calls itself with doubled input until the value is greater than 10, as shown in Listing 2.5.

```scala
def f(x: Int): Int = {
  if (x > 10) x
  else f(x * 2)
}
```

Listing 2.5: Example program with recursion (Scala syntax)

Furthermore, assume that this function is called in a context where `x` is known to be odd. The analysis does not know which branch will be taken, because it does not know whether `x > 10` holds, so it has to evaluate both branches. The first one will result in an odd value, because `x` is directly returned. The second one will result in a recursive call where `x` is now known to be even, but it is still impossible for the analysis to find out whether `x` is greater than 10, so both possibilities have to be evaluated again. Using a naive approach, this will lead to infinite recursion in the abstract interpreter.

One solution to this problem is to search for a *fixed point* for the abstract value of `x`, i.e. a value that does not change after repeated application of `f`. In this case, the abstract interpreter should find out that after the first recursive call, the input to `f` will always be even and therefore the output will always be even. So the result of the outermost call to `f` will be even or odd, which is expressed as $\mathbf{even} \sqcup \mathbf{odd} = \mathbf{any} = \top$. As $\top$ is also the *least* fixed point in this case, the result can not be any more precise than this, but it is still better than a non-terminating analysis.

However, when the function call is analyzed in a context where `x` is known to be even, for the reasons stated above the analysis can actually conclude that the result will also be even.

3 Implementation of a Concrete Interpreter

As a first step of our work, we implemented an XSLT processor that transforms concrete XML documents with the intention that we would understand the details of the specification and the requirements that it imposes on an implementation. It was also meant to serve as a basis for the abstract interpreter, if possible (cf. next chapter).

Our implementation was done in Scala and we made heavy use of algebraic data types (e.g. case classes) and pattern matching. We also tried to maintain a functional programming style, preferring recursion over loops and immutable variables and data structures over mutable ones.

First, we will now cover our implementation of the object model of the data that we are dealing with: XML documents, or rather the nodes that they consist of. We will then continue with the parsing and evaluation of XPath expressions and finally we will put it all together and see how XSLT stylesheets are parsed and processed in our implementation.

3.1 Parsing and Representing XML Documents

Because XML documents are used as input and output of our XSLT processor, we need a way to represent and manipulate those documents in Scala. Fortunately, there is the `scala-xml` library[1] that is capable of parsing XML documents and even allows to use XML literals directly inside of Scala source code. However, we did not want to use it as our XML representation directly, as the full XML standard is quite complex and we do not need many of the more arcane features.

Therefore we designed our own XML node classes for representing XML data in a simple way that is fully under our control. We ended up with the following class hierarchy that closely resembles what we presented in Section 2.1.1:

```scala
abstract class XMLNode extends Ordered[XMLNode] { /* ... */ }

case class XMLRoot(children: List[XMLNode]) extends XMLNode { /* ... */ }

case class XMLElement(name: String,
                      attributes: MutableList[XMLAttribute],
                      children: MutableList[XMLNode],
                      var parent: XMLNode) extends XMLNode { /* ... */ }

case class XMLAttribute(name: String,
                        value: String,
                        var parent: XMLNode) extends XMLNode { /* ... */ }

case class XMLTextNode(value: String,
                       var parent: XMLNode) extends XMLNode { /* ... */ }

case class XMLComment(value: String,
                      var parent: XMLNode) extends XMLNode { /* ... */ }
```

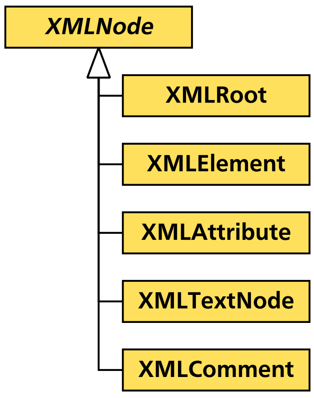

Figure 3.1: Class hierarchy of the XML data representation

We provide the `xml.XMLParser` class to transform a document from `scala-xml`'s representation into our own.

[1] It was originally part of the Scala distribution but has become a separate library as of Scala 2.11 and is now available via `https://github.com/scala/scala-xml` – accessed 2015-01-08.

The following list presents some observations about our implementation, supplementing Figure 3.1:

- We use a mutable data structure because we need to model the recursive relationship between nodes: Element nodes need to know their children and attributes, and every node (except the root) needs to know its parent. This is required because the XML tree needs to be traversed in both directions, upwards and downwards. Therefore, the parent field of a newly created node will be `null` initially. We call such a node that is not a root node and does not have an associated parent node (and therefore no associated root node) a *fragment*. Fragments appear in XSLT during the building up of the output document, because the document is generated bottom-up and the root node is created last.

- We allow root nodes to have arbitrary children instead of just a single element child. This is required to support Result Tree Fragments. (Note that Result Tree Fragments are not *fragments* in the sense of the definition above. They do represent only a part of a document, but they are wrapped in a "synthetic" root node.) For root nodes that represent complete XML documents, the list of children will always consist of just a single element node.

- We implemented the `Ordered` trait for our nodes, in such a way that two nodes are compared regarding their position in the *document order*. Positional comparison of two nodes is only possible if the nodes are not fragments, because fragments have no relationship between each other as they do not belong to the same document – in fact, they do not belong to any document (yet).

- Relatedly, node equality is implemented in two different ways, depending on whether the node is a fragment or not: If both nodes are not fragments and belong to the same document (i.e. have the same root node), they are compared using *reference equality*. Fragment nodes are compared using *structural equality*, ignoring the parent to prevent infinite recursion. It would be wrong to compare non-fragment nodes structurally, because two nodes with the same structure but from different positions in the same document should not be considered equal.

- For every node we provide methods to obtain not only its root, but also the lists of descendants and ancestors, as well as the *string-value*, which is defined by the XPath specification for each node type [4, Section 5].

- Every node has a `copy` method to create a copy of itself (and of all of its attributes and descendants in the case of an element node). It is implemented in such a way that it ignores the node's parent and thus turns the copy into a *fragment*, which can then be inserted into the output document of a transformation.

- Adjacent text nodes are merged into a single text node whenever they are added as children to an element, because this is required by the XSLT specification [3, Section 7.2].

3.2 XPath Evaluation

3.2.1 Syntax

Since XPath is a separate language from XML and defines its own grammar, we needed a way to parse XPath expressions in order to evaluate them.

Similar to the way we treated XML, we found an existing library, Jaxen[2], that is able to parse XPath 1.0 expressions, but transform the result into an AST (abstract syntax tree) model that we defined ourselves. This transformation is implemented in the `XPathParser` class. We did not use the Jaxen AST, because

[2] See `http://jaxen.codehaus.org` – accessed 2015-01-08

Jaxen is a Java library that was built without Scala in mind and we did not want to give up the ability to pattern match on our syntax trees.

The base class of our AST model is called `XPathExpr` and various AST node types extend this class – roughly one for each case of the ⟨*expr*⟩ production rule in Figure 2.1.

Note that, even though we do not support some features in the interpreter, our AST supports the full feature set of XPath 1.0.[3]

3.2.2 Data Representation

Before we can start to look at the semantics of XPath evaluation, we need a way to represent the result of such an evaluation. We defined the type `XPathValue` as an algebraic data type with four possible cases, one for each XPath type. This definition is shown in Figure 3.2.

```
abstract class XPathValue { /* ... */ }

case class StringValue(value: String) extends XPathValue
case class NumberValue(value: Double) extends XPathValue
case class BooleanValue(value: Boolean) extends XPathValue
case class NodeSetValue(nodes: TreeSet[XMLNode]) extends XPathValue
```

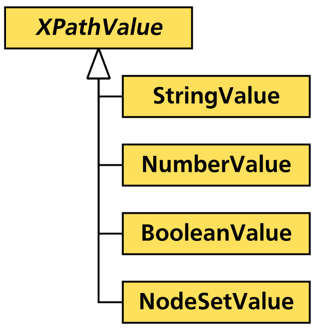

Figure 3.2: Class hierarchy of XPath values

While string, number and boolean values are trivial, node-sets are somewhat more interesting: They are implemented as ordered sets of `XMLNodes`. This way we make sure that no node is contained twice and nodes are automatically kept in document order.

The `XPathValue` class also implements methods for converting between the types. How these conversions work is defined by the specification of the XPath functions `number(...)` (conversion to number), `string(...)` (conversion to string) and `boolean(...)` (conversion to boolean) [4, Section 4]. Conversion to node-sets is not possible.

3.2.3 Semantics

The actual interpreter that implements the semantic of XPath evaluation is defined in `XPathEvaluator`. It exposes one function, `evaluate`, which is outlined in Listing 3.2. It takes an XPath expression and returns a XPath value. Additionally it requires an evaluation context, which is provided by the XSLT processor. The `XPathContext` class is defined in Listing 3.1 and wraps four values, corresponding to the definition found in the XPath specification [4, Section 1]:

- The current *context node*. This designates the node that is currently transformed by the XSLT transformation. It is used as a starting point for evaluating relative location paths.
- The *context position* and *context size*. These refer to the list of nodes that is currently processed by XSLT. For example, when a template is applied to all children of a node, this list contains exactly all child nodes of the given node and the context node is one of those. The context position then is the position of the context node in that list, and the context size is the number of nodes in the list. Those values are needed to evaluate the functions `position()` and `last()`.

[3] This was necessary in order to be able to parse and analyze arbitrary XPath expressions in our evaluation of used features in real XSLT stylesheets, cf. Section 5.2.

- Finally, there is a set of *variable bindings* that are currently in scope and therefore accessible from XPath expressions. They are stored as a mapping from variable names to XPath values.

```scala
case class XPathContext(node: XMLNode, position: Int, size: Int, variables: Map[String, XPathValue])
```

Listing 3.1: Definition of XPath context class

```scala
/** Evaluates a given XPath expression using a specified context and returns the result. */
def evaluate(expr: XPathExpr, ctx: XPathContext): XPathValue = expr match {
  case PlusExpr(lhs, rhs) => NumberValue(evaluate(lhs, ctx).toNumberValue.value +
    evaluate(rhs, ctx).toNumberValue.value)
  /* ... several more arithmetic operators ... */
  case RelationalExpr(lhs, rhs, relOp) =>
    // evaluation is specified in the XPath spec section 3.4
    BooleanValue(evaluate(lhs, ctx).compare(evaluate(rhs, ctx), relOp))
  case StringLiteralExpr(literal) => StringValue(literal)
  case NumLiteralExpr(num) => NumberValue(num)
  case VariableReferenceExpr(name) => ctx.variables.get(name) match {
    case Some(value) => value
    case None => throw new ProcessingError(f"Variable $name is not defined")
  }
  case UnionExpr(lhs, rhs) => (evaluate(lhs, ctx), evaluate(rhs, ctx)) match {
    case (NodeSetValue(left), NodeSetValue(right)) => NodeSetValue(left | right)
    case (left, right) => throw new ProcessingError(f"Wrong types for union expression, must be node-
        sets ($left | $right)")
  }
  case FunctionCallExpr(None, name, params) =>
    evaluateFunctionCall(name, params.map(p => evaluate(p, ctx)), ctx)
  case LocationPath(steps, isAbsolute) =>
    NodeSetValue(evaluateLocationPath(ctx.node, steps, isAbsolute))
  /* ... some more operations ... */
}
```

Listing 3.2: Evaluation function for XPath expressions

The evaluation function is a large match expression with cases for each AST node. While some cases are left out in Listing 3.2, the most important cases are these:

- `PlusExpr` and other arithmetic expressions are evaluated by first evaluating their operands, converting the resulting values to `NumberValues` and then applying the operation (e.g. addition) on the underlying floating-point numbers. Arithmetic operations will never result in a type error, because any operand value can be converted to a number.

- A `RelationalExpr` is evaluated by delegating to the `compare` function defined for `XPathValue`.

- `StringLiteralExpr` and `NumLiteralExpr` just wrap the literal value in an `XPathValue`.

- Variable references are looked up in the variable bindings that are contained in the context. If no variable with the requested name is defined, an error is thrown.

- A `UnionExpr` is evaluated by evaluating both operands and taking the union of the containing ordered sets, if the evaluation resulted in two `NodeSetValues`. If not, an error is thrown.

- A `FunctionCallExpr` is evaluated by delegating to the helper function `evaluateFunctionCall` after the parameters are evaluated. In this helper function, which is not listed here, we implemented some, but not all of the functions defined in the *Core Function Library* [4, Section 4].

- A `LocationPath` is evaluated by delegating to another helper function, `evaluateLocationPath`, and the current context node is passed as a starting node to be used for evaluating the first (i.e. leftmost) step.

The definition of `evaluateLocationPath` is shown in Listing 3.3.

```
 1 private def evaluateLocationPath(node: XMLNode, steps: List[XPathStep], isAbsolute: Boolean):
       TreeSet[XMLNode] = {
 2   (steps, isAbsolute) match {
 3     case (_, true) => evaluateLocationPath(node.root, steps, false) // absolute path -> handle as
           relative but start with root node
 4     case (Nil, false) => TreeSet(node) // no steps left -> just return input node
 5     case (first :: rest, false) =>
 6       val nodes: TreeSet[XMLNode] = first.axis match {
 7         // "the child axis contains the children of the context node"
 8         case ChildAxis => node match {
 9           case XMLRoot(children) => children.to[TreeSet]
10           case XMLElement(_, _, children, _) => children.to[TreeSet]
11           case _ => TreeSet()
12         }
13         // "the descendant axis contains the descendants of the context node
14         // a descendant is a child or a child of a child and so on"
15         case DescendantAxis => node.descendants.to[TreeSet]
16         // "the attribute axis contains the attributes of the context node; the axis will be
17         // empty unless the context node is an element"
18         case AttributeAxis => node match {
19           case XMLElement(_, attrs, _, _) => TreeSet[XMLNode]() ++ attrs
20           case _ => TreeSet[XMLNode]()
21         }
22         /* ... some axes not listed ... */
23       }
24       val testedNodes = nodes.filter { n => first.test match {
25         case NameTest(Some(_), _) => throw new NotImplementedError("Prefixed names are not
             implemented.")
26         case NameTest(None, "*") => XPathAxis.isPrincipalNodeType(first.axis, n)
27         case NameTest(None, testName) => XPathAxis.isPrincipalNodeType(first.axis, n) && (n match {
28           case XMLElement(name, _, _, _) => name == testName
29           case XMLAttribute(name, _, _) => name == testName
30           case _ => false
31         })
32         case TextNodeTest => n.isInstanceOf[XMLTextNode]
33         case CommentNodeTest => n.isInstanceOf[XMLComment]
34         case AllNodeTest => true
35       }}
36       if (first.predicates.nonEmpty) throw new NotImplementedError("Predicates are not supported")
37       testedNodes.flatMap { n => evaluateLocationPath(n, rest, false)}
38   }
39 }
```

Listing 3.3: Evaluation function for location paths

This function recursively evaluates the path from left to right. If the path is absolute, the root node of the current context node is retrieved and evaluation continues with that node as the context node as if the path was relative (Line 3). If the list of steps is empty, i.e. no more steps are left to evaluate, a singleton set containing the input node is returned. This is the base case of the recursion.

The usual case – when the list is not empty – is handled as follows (Lines 5 ff.): According to the axis of the first (i.e. leftmost) step of the remaining path, a set of nodes is selected from the current context node, e.g. its children, descendants or attributes. This set is then filtered using the node test of the same step.[4] For every node that passes the test, the rest of the location path is recursively evaluated relative to that node and the resulting sets are accumulated into a single set using `flatMap` (Line 37). So the result

[4] The specification defines the notion of *principal node type*, which is required here: The principal node type of the `attribute` and `namespace` axes is *attribute* and *namespace*, respectively; for all other axes, the principle node type is *element* [4, Section 2.3].

will be a single ordered set that contains all the nodes from the source document that are reachable via the given path, relative to the context node.

3.3 XSLT Processing

As XSLT stylesheets are XML documents, we could have used the object model of `scala-xml` or the one we defined in Section 3.1 to process a stylesheet in one of those representations. For the sake of understandability, we decided to introduce another layer of abstraction in the form of some classes that model stylesheets (`XSLTStylesheet`), templates (`XSLTTemplate`) and instructions (`XSLTInstruction`) and dismiss the underlying XML representation. Again, we provide an `XSLTParser` to transfer a stylesheet from `scala-xml`'s representation into our own.

The `XSLTTemplate` class (Figure 3.4) is very simple. It contains a sequence of instructions that will be executed when the template is instantiated as well as a (possibly empty) set of parameter definitions, stored as a list of pairs, where the first element is the parameter's name and the second is the unevaluated default value of the parameter. A list of pairs is used instead of a `Map`, because the order in which the parameters are evaluated must be the one in which they are defined, and an associative data structure would not maintain that order. Furthermore, we use the type `Either[XPathExpr, Seq[XSLTInstruction]]` wherever we are dealing with unevaluated variable values, because those can either be declared as an XPath expression directly, or as instructions to generate a Result Tree Fragment [3, Section 11]. Each of the template's parameters can be overwritten when the template is instantiated and its value must then be evaluated with respect to the current context.

```
case class XSLTTemplate(
  content: Seq[XSLTInstruction],
  defaultParams: List[(String, Either[XPathExpr, Seq[XSLTInstruction]])] = Nil
)
```

Listing 3.4: Definition of `XSLTTemplate` class

When template rules are defined, they also have a name and/or a `match`-clause (with optional mode). At least one of those is required and we refer to those with a name as *named templates* and those with a `match`-clause as *matchable templates*. Neither of these appear in the `XSLTTemplate` class, because we delegate the management of templates, including the division into named and matchable templates, to the `XSLTStylesheet` class, which is outlined in Listing 3.5.

```
class XSLTStylesheet(
  templates: List[(XSLTTemplate, Option[String], Option[XPathExpr], Option[String])],
  val globalVariables: List[(String, Either[XPathExpr, Seq[XSLTInstruction]])],
  disableBuiltinTemplates: Boolean
) {
  val namedTemplates: Map[String, XSLTTemplate] = Map() ++ templates.collect {
    case (tmpl, Some(name), _) => (name, tmpl)
  }

  val matchableTemplates: Map[Option[String], List[(LocationPath, XSLTTemplate)]] =
    /* ... extract matchable templates for each mode, add built-in templates, and sort them ... */
}
```

Listing 3.5: Definition of `XSLTStylesheet` class

The constructor of the class receives a list of all template definitions in the stylesheet as tuples comprising the `XSLTTemplate` itself, the (optional) name, the XPath expression found in the (optional) `match`-clause and the (optional) match mode. The match clause must be a valid pattern, which means that it is either a `LocationPath`, following the restrictions laid out in Section 2.1.4, or a `UnionExpr` consisting of multiple patterns that are subsequently split into single `LocationPath`s.

All named templates are then collected into the field `namedTemplates`. The matchable templates are similarly extracted and – together with the built-in templates[5] – put into a map that maps each mode occurring in the stylesheet (at least the default mode, represented here as `None`) to a list that contains pairs of `LocationPath`s and their associated templates. This list is sorted in descending order by precedence and priority according to the rules presented in Section 2.1.4. User-defined priorities are not supported in our implementation, hence each template is assigned its default priority.

Additionally, the `XSLTStylesheet` contains a list of global variable definitions that correspond to the top-level <`xsl:variable`> and <`xsl:param`> definitions found in the original stylesheet [3, Section 11.4].

The parsed stylesheet object can now be used to transform an XML document. This is implemented in `XSLTProcessor`, which constitutes the actual interpreter and is outlined in Listing 3.6.

```scala
def transform(sheet: XSLTStylesheet, source: XMLRoot): XMLRoot = {
  val globalVariables = evaluateVariables(sheet, sheet.globalVariables,
    XSLTContext(source, List(source), 1, Map(), Map()))
  applyTemplates(sheet, List(source), None, globalVariables, Map(), Map()) match {
    case List(elem: XMLElement) => XMLRoot(List(elem))
    case x => throw new ProcessingError("Transformation result must be a single XMLElement")
  }
}

def applyTemplates(sheet: XSLTStylesheet, sources: List[XMLNode], mode: Option[String],
     globalVariables: Map[String, XPathValue], localVariables: Map[String, XPathValue], XPathValue],
     params: Map[String, XPathValue]): List[XMLNode] = {
  sources.zipWithIndex.flatMap { case (n, i) =>
    val tmpl = chooseTemplate(sheet, n, mode)
    val context = XSLTContext(n, sources, i + 1, globalVariables, localVariables)
    instantiateTemplate(sheet, tmpl, context, params)
  }
}

def chooseTemplate(sheet: XSLTStylesheet, node: XMLNode, mode: Option[String]): XSLTTemplate = {
  sheet.matchableTemplates(mode).find { case (path, _) => XSLTPatternMatcher.matches(node, path) }
      match {
    case Some((_, tmpl)) => tmpl
    case None => throw new ProcessingError(f"Found no matching template for input node ‘${XMLNode.
        formatPath(node)}‘ [NOTE: this can only happen when builtin templates are disabled]")
  }
}

def instantiateTemplate(sheet: XSLTStylesheet, tmpl: XSLTTemplate, context: XSLTContext, params: Map
     [String, XPathValue]): List[XMLNode] = {
  val evaluatedParams = tmpl.defaultParams.foldLeft(Map[String, XPathValue]()) {
    case (current, (name, _)) if params.contains(name) => current + (name -> params(name))
    case (current, (name, value)) => current + (name -> evaluateVariable(sheet, value, context.
        addLocalVariables(current)))
  }
  processAll(sheet, tmpl.content, context.replaceLocalVariables(evaluatedParams))
}

def processAll(sheet: XSLTStylesheet, instructions: Seq[XSLTInstruction], context: XSLTContext):
     List[XMLNode] = /* ... process each instruction in turn and collect variable definitions, so
     they are available in subsequent instructions (in the same scope) ... */

def process(sheet: XSLTStylesheet, instruction: XSLTInstruction, context: XSLTContext):
  Either[List[XMLNode], (String, XPathValue)] = /* ... match on instruction type ... */
```

Listing 3.6: Functions within XSLTProcessor

[5] We incorporated the `disableBuiltinTemplates` constructor parameter here as an option to not include the built-in templates. This can be used to simplify the results and is especially useful when we proceed to abstract interpretation.

The entry point of the interpreter is the `transform` function, that takes a stylesheet definition and a source document in the form of its root node. This root node is put in a list and – after all global variables have been evaluated – is passed on to the `applyTemplates` function (Line 4). The purpose of this function is to apply matching templates to each node in the given list, which entails selecting a template for each node, instantiating it, and finally concatenating all results (which is implicitly done here by `flatMap`).

The `chooseTemplate` function chooses, out of the stylesheet's matchable templates with the correct mode, the first one that matches the given node. The task of determining whether the location path of a template rule matches a node is delegated to the `XSLTPatternMatcher.matches` function, which will be described later.

When a template has been selected, it is instantiated using the `instantiateTemplate` function. The current node as well as the position of that node in the list are passed as part of the `XSLTContext`, which is very similar to the `XPathContext` that was already introduced.[6] Template instantiation can be parameterized, but parameters are accepted only if the template contained a default definition (formal parameter) with the same name already. The actual parameter values are added to the context after they are computed using the `evaluateVariable` auxiliary function, which either resorts to the `XPathEvaluator` for XPath expressions or uses the `processAll` function for Result Tree Fragments.

The last and most important step of instantiating a template is the evaluation of the instructions in the template (Line 30). The function `processAll` is used to process a sequence of instructions in a given context by processing each instruction in turn and making sure that new variable definitions are visible by subsequent instructions. The immutability of the context class[7] ensures that variable definitions never escape their scope.

Finally, `process` is responsible for the processing of a single instruction, of which the result will either be a list of XML nodes or a new variable binding. Our implementation distinguishes the following instruction types, which are subclasses of `XSLTInstruction`:

- **`CreateElementInstruction`** creates a new element in the result tree. This is used whenever a literal result element (an element that is not from the XSLT namespace) or the `<xsl:element>` instruction is encountered in the original stylesheet. It contains a name, which is an attribute value template (i.e. it can be a mix of literal strings and XPath expressions that will be evaluated and concatenated), and a sequence of more instructions that, when evaluated, form the attributes and children of the element [3, Section 7.1.1 and 7.1.2].

- **`SetAttributeInstruction`** is used to add attributes to result elements and is therefore only allowed in the list of child instructions in a `CreateElementInstruction` (or `CopyInstruction`). This is used for attributes of literal result elements as well as for the `<xsl:attribute>` instruction. It contains the name of the attribute (as an attribute value template) and a sequence of instructions that generate the value of the attribute. Conforming to the specification, attributes that are added to an element after children have been added already are not allowed and are ignored. If an attribute with the same name has already been added, it is replaced with the new one [3, Section 7.1.3].

- **`CreateTextInstruction`** creates a new text node in the result tree. This is used whenever a literal text node or the `<xsl:text>` instruction is encountered in the original stylesheet. The text is stored as a simple string (when a text node with dynamic content should be created, the `CopyOfInstruction` is used). Note that we do not handle output escaping or whitespace stripping [3, Section 7.2].

[6] An XSLT context contains the node list and not just its size, cf. [3, Section 1]. Using two separate classes also enables us to to distinguish global and local variables in the XSLT context. We provide an auxiliary method to construct an XPath context from an XSLT context (the other direction is not needed).

[7] We provide auxiliary methods such as `addLocalVariable` to "modify" contexts, but these always create a new modified context.

- **`CreateCommentInstruction`** creates a new comment node in the result tree. This is used whenever the `<xsl:comment>` instruction is encountered. It contains a sequence of other instructions that, when evaluated, generate the content of the comment in the form of text nodes [3, Section 7.4].

- **`ApplyTemplatesInstruction`** corresponds to `<xsl:apply-templates>` and is used to select some nodes from the source tree and apply matching templates to them, as implemented in the `applyTemplates` function shown above. It contains an XPath expression to select the nodes (which defaults to all children of the current template node), a mode identifier, and a set of parameters that are passed to the template [3, Section 5.4].

- **`CallTemplateInstruction`** corresponds to `<xsl:call-template name="...">` and is used to instantiate one of the named templates directly [3, Section 6].

- **`CopyOfInstruction`** inserts the result of the evaluation of some XPath expression into the result tree. If the evaluation results in a node-set, the nodes in those set are copied. Any other result is converted to a string and inserted as a text node. Therefore, we use this instruction type not only for `<xsl:copy-of select="...">`, but also for `<xsl:value-of select="...">` as the latter is equivalent to `<xsl:copy-of select="string(...)">`, i.e. the expression result is first converted to a string, even if it is a node-set [3, Sections 7.6.1 and 11.3].

- **`CopyInstruction`** corresponds to `<xsl:copy>` and just copies the current context node, but without its attributes or children. The attributes and children of the resulting node are generated by another sequence of instructions that form the content of this instruction [3, Section 7.5].

- **`ChooseInstruction`** is a conditional branching instruction that contains a list of branches, each with an XPath expression that determines the condition of that branch. The instructions of the first branch where the condition evaluates to `true` are executed. An additional `otherwise` branch is executed when all conditions evaluate to `false`. This corresponds to the `<xsl:choose>` instruction, but it is also used for the more restricted `<xsl:if>` instruction which only allows for a single conditional branch and no – or rather an empty – `otherwise` branch [3, Section 9].

- **`ForEachInstruction`** corresponds to `<xsl:for-each>` and iterates over a set of nodes that is selected by an XPath expression. For each node in this set, the contained sequence of instructions is executed with that node as the context node [3, Section 8].

- **`VariableDefinitionInstruction`** corresponds to `<xsl:variable>` and adds a new local variable binding with a given name to the current context. The value of the variable is, as seen before, either the evaluation result of an XPath expression or a Result Tree Fragment generated by a sequence of instructions [3, Section 11].

We will now look at the actual implementation of only two of the more interesting instructions in detail: The cases for `CreateElementInstruction` and `CopyOfInstruction` are shown in Listing 3.7 and 3.8 respectively, and explained below, mainly to get an impression of what kind of operations are needed to implement the intended semantics.

In order to create an element, the contained instructions (`content`, of type `Seq[XSLTInstruction]`) are processed first, as these generate the attributes and children of the new element node. From the resulting list of nodes we then extract separate lists of attributes and children (Lines 4–5). Since the specification defines that "adding an attribute to an element after children have been added to it" [3, Section 7.1.3] is an error and the attribute should be ignored in that case, we only accept attributes until some other node is seen.

The name of the created element is specified as an attribute value template (`name`, of type `List[Either[String, XPathExpr]]`), which is a list of segments that are each either a literal string or an XPathExpression. These expressions are evaluated and all segments are concatenated (Lines 6–9)

```
case CreateElementInstruction(name, content) =>
  val innerNodes = processAll(sheet, content, context)
  // attributes must come before all other result nodes, afterwards they are ignored
  val attributes = innerNodes.takeWhile(n => n.isInstanceOf[XMLAttribute]).map(n => n.asInstanceOf[XMLAttribute])
  val children = innerNodes.filter(n => !n.isInstanceOf[XMLAttribute])
  val evaluatedName = name.map {
    case Left(str) => str
    case Right(expr) => XPathEvaluator.evaluate(expr, context.toXPathContext).toStringValue.value
  }.mkString // concatenate literal and expression parts to get the element name
  Left(List(XMLElement(evaluatedName, attributes, children)))
```

Listing 3.7: Case for `CreateElementInstruction` within `process`

before the resulting `XMLElement` can be constructed and put in a list that is then returned as the result of the `process` function. Note that the constructor of `XMLElement` merges adjacent text node children into a single text node, so we do not need to do this here explicitly.

```
 1 case CopyOfInstruction(select) =>
 2   XPathEvaluator.evaluate(select, context.toXPathContext) match {
 3     case NodeSetValue(nodes) => Left(nodes.toList.flatMap {
 4       case XMLRoot(children) => children.map(_.copy)
 5       case n => List(n.copy)
 6     })
 7     case other =>
 8       val textValue = other.toStringValue.value
 9       if (textValue != "")
10         Left(List(XMLTextNode(other.toStringValue.value)))
11       else
12         Left(Nil) // text nodes with empty content are not allowed
13   }
```

Listing 3.8: Case for `CopyOfInstruction` within `process`

The `CopyOfInstruction` is used to copy the result of an XPath expression (here called `select`) into the result document. To achieve this, we first evaluate the expression in the current context.

If the result is a node-set (which could also be a Result Tree Fragment), we return a list of copies of all nodes in that set (Lines 3–6). According to the specification, "a root node is copied by copying its children" [3, Section 11.3], so a special case is needed to handle this (Line 4). Copying a node-set is also one of the cases where it is important that it is an ordered set, because the nodes must be copied in document order.

If the result is not a node-set, it is converted to a string, and a text node is created that has this value as its content. Because text nodes with empty content are not allowed, we return an empty list whenever the converted value is the empty string.

3.3.1 XSLT Pattern Matching

The last remaining piece is the pattern matching function `XSLTPatternMatcher.matches`, that we already referred to.

As can be seen in Listing 3.9, the publicly visible `matches` function that is supposed to check whether the given XML node matches the given location path (which must be a pattern, cf. Section 2.1.4) delegates the work to a private overload that takes the steps of the path in reversed order (Line 2). This

```scala
 1 def matches(node: XMLNode, path: LocationPath): Boolean =
 2   matches(node, path.steps.reverse, path.isAbsolute)
 3 
 4 private def matches(node: XMLNode, reversedPathSteps: List[XPathStep], pathIsAbsolute: Boolean):
       Boolean = {
 5   // match recursively from right to left (path steps are reversed!)
 6   reversedPathSteps match {
 7     case nextStep :: restPath =>
 8       if (matchesSingleStep(node, nextStep)) {
 9         restPath match {
10           case XPathStep(DescendantOrSelfAxis, AllNodeTest, Nil) :: Nil => true
11           case XPathStep(DescendantOrSelfAxis, AllNodeTest, Nil) :: restTail =>
12             // the next step is '//' and must be handled separately:
13             // does any ancestor match the rest of the path?
14             node.ancestors.exists(a => matches(a, restTail, pathIsAbsolute))
15           case _ =>
16             // does the parent match the rest of the path?
17             matches(node.parent, restPath, pathIsAbsolute)
18         }
19       } else { false }
20     case Nil =>
21       if (pathIsAbsolute) node.isInstanceOf[XMLRoot] else true
22   }
23 }
```

Listing 3.9: Function that checks if a node matches a location path pattern

is necessary because the location path steps are stored in such a way that the leftmost step is the first in the list, but the matching algorithm needs to traverse the steps from right to left.

It does this in a recursive manner, checking whether the current node matches the next step (i.e. the rightmost one) and then continuing with the parent node and the rest of the path. If the next step of the remaining path is `/descendant-or-self::node()/`, which is the expanded form of '//', the algorithm looks for any *ancestor* that matches the remaining path (skipping the `descendant-or-self` step itself), instead of checking only the *parent* (Lines 11–12). If the '//' step is the last (i.e. leftmost) one in the list, we can return `true` directly, because every node is a descendant of the root node and thus matches such a pattern (Line 10). For example, the pattern `//body/p` is equivalent to the pattern `body/p`. We call the step a *trivial* descendant step in this case.

The matching of a single step is delegated to the `matchesSingleStep` function that return `true` if and only if the given node matches the combination of axis specifier and node test specified in the step. The `child`-axis is used to match elements as well as text and comment nodes, while the `attribute`-axis only matches attributes (recall that other axes besides these are not allowed in patterns). The node test further specifies what type (and optionally what name) the node should have in order to match the step.

In this way, the matching algorithm walks up the XML tree until the match fails on some level or the list of steps is empty. The latter means that no steps remain to be checked and we know that the node matches the path, except when the path is an absolute path and the current node is not a root node (Lines 16–17).

This concludes the chapter about our implementation of the concrete interpreter, and based on this implementation we will now turn towards implementing an abstract interpreter.

4 Towards Abstract Interpretation

The main objective of our work was the implementation and evaluation of using *abstract interpretation* to analyze XSLT stylesheets. We used the concrete interpreter from the previous chapter as a basis and made two fundamental changes according to the principles introduced in Section 2.2: Firstly, we replaced the concrete data structures that are used in the interpreter with abstracted versions of the same structures. Secondly, we modified the interpreter to use operations such as *join* ($\sqcup$) to handle control flow and investigated the need to take additional measures in order to ensure termination.

4.1 Abstract Domain Interface

In our explanation of the concrete interpreter, we first looked at the data structures that were needed to represent the necessary data, i.e. XML documents and XPath values. Likewise, we will now look at the abstract objects that correspond to those concrete objects in our interpreter. Since we wanted to experiment with different abstract domains, we defined an interface for them that the abstract interpreter can use knowing neither the representations of the abstract objects, nor the implementations of the operations dealing with those objects.

The interface for abstract domains consists of the traits `XMLDomain` for operations on XML trees and `XPathDomain` for operations on XPath values. While these traits define the operations, the actual data representations of the abstract objects are defined by three type parameters that are used by the domain traits and throughout the abstract interpreter:

- The `N` type is used for the abstract objects representing *XML* ***n****odes*. This resembles the `XMLNode` class in the concrete interpreter.
- The `V` type is used for the abstract objects representing *XPath* ***v****alues*. In the concrete interpreter this corresponds to the `XPathValue` class.
- The `L` type is used for the abstract structure of ***l****ists of XML nodes*. An extra type is required for those, because there are domain operations that operate on such lists and using e.g. `List[N]` is not sufficient as it is not a complete lattice. The `L` type is also used for (ordered) sets of nodes, because they are very similar and we did not want to introduce yet another type parameter for simplicity.

The concrete types for `N` and `L` are fixed by implementations of the `XMLDomain` trait, whereas the concrete type for `V` is fixed by implementations of the `XPathDomain` trait. A third trait, `Domain`, joins together implementations of `XMLDomain` and `XPathDomain` and thus fixes the entire abstract domain for an analysis. The reason we chose to split the domain interface into the subdomains for XML data and XPath data is that we could combine different implementations of those subdomains independently, though some operations need to know about the internal representations of *both* subdomains and therefore had to be implemented individually for each combination.

Furthermore, it might be noted that in our abstract domains there is no way to distinguish between different types of XML nodes (e.g. element and text node) or different types of XPath values (e.g. string and number value) within the Scala type system (which is not surprising given that a single abstract object can represent concrete objects of several types at once). Instead, in those cases where it is required by the interpreter, we provide domain operations to distinguish them.

On the following pages we present a detailed list of all domain operations that we defined in the traits for both subdomains. Note that every implementation of one of the three types `N`, `L` and `V` must be a complete lattice, therefore operations to compare two objects of such a type w.r.t. the partial ordering, to obtain the top and bottom elements as well as the join of two elements are required for each of those types. The remaining operations were added as needed during the implementation of the abstract interpreter, whenever an operation could not be expressed using the already existing functions.

4.1.1 Operations in `XPathDomain`

`def top: V`

This function returns the *top* ($\top$) element in the complete lattice of abstract XPath values. Recall that $\top$ represents a value that the interpreter knows nothing about (except that it is an XPath value).

`def bottom: V`

This function returns the *bottom* ($\bot$) element in the complete lattice of abstract XPath values. Recall that $\bot$ represents an "impossible" value and is used whenever an operation does not have any valid output, especially whenever the input for an operation was already $\bot$.

`def join(v1: V, v2: V): V`

This operation calculates the least upper bound $\texttt{v1} \sqcup \texttt{v2}$. A *meet* operation for `V` was not required by the abstract interpreter.

`def lessThanOrEqual(v1: V, v2: V): Boolean`

This operation defines the partial ordering of the lattice and returns `true` whenever $\texttt{v1} \leq \texttt{v2}$.

`def topNumber: V` / `def topString: V`

These functions are similar to `top`, but the result is an element of the lattice that is the least upper bound of all numbers (resp. strings), i.e. an element of which we know only that it is a number (resp. string), but nothing more. In a domain where this is not representable, the result might be equal to $\top$.

`def liftString(lit: String): V` / `def liftNumber(num: Double): V` / `def liftBoolean(bool: Boolean): V`

These operations lift a single literal string (resp. number/boolean) value into the lattice `V`. The result will be the least element $v \in \texttt{V}$ such that the concretization $\gamma(v)$ contains the given string (resp. number/boolean) value (recall that $\gamma(v)$ is a set of concrete XPath values).

`def add(v1: V, v2: V): V`

This function defines the addition operation for two abstract XPath values. Its operands are automatically converted to numbers if they aren't already. The result will always be a number ($\leq$ `topNumber`). If one of the operands is $\bot$, the result will be $\bot$.

Similar functions exist for subtraction, multiplication, division and the modulo operation as well as the unary negation operation, but they are skipped in this list.

`def compareRelational(v1: V, v2: V, relOp: RelationalOperator): V`

This function defines the comparison of two XPath values using a relational operator (one of =, !=, <=, <, >= or >). The expected semantics (for the concrete case) is defined in the XPath specification [4, Section 3.4]. Note that this comparison is not to be confused with the partial ordering in the lattice.

`def concatStrings(v1: V, v2: V): V`

This operation is used to concatenate two string values. In contrast to the arithmetic operations, this operation does not convert its operands to strings first. This means that whenever $\gamma(\texttt{v1})$ or $\gamma(\texttt{v2})$ do not contain any values of type string, the result will be $\bot$.

`def nodeSetUnion(v1: V, v2: V): V`

This computes the union of two node-set values. Like `concatStrings`, it does not perform any type conversions.

`def toStringValue(v: V): V` / `def toBooleanValue(v: V): V` / `def toNumberValue(v: V): V`

Functions for type conversions as specified by the XPath functions `string(...)`, `boolean(...)` and `number(...)` [4, Section 4]. Conversion to node-sets is not possible.

`def createNodeSet(list: L): V`

This operation and the following one use the `L` type in addition to `V`, because they provide a bridge between the XML domain and the XPath domain: `createNodeSet` creates an abstract node-set XPath value from an abstract list of XML nodes. While the input list is not necessarily an ordered set, the implementation has to ensure that there are no duplicates in the resulting node-set and the nodes are ordered in document order.

`def matchNodeSet(v: V): (L, V)`

This operation is similar to a match statement distinguishing between node-sets and other XPath values. The result is a pair of two values, of which the first component is an abstract list of all the nodes in the set (in document order) and the second component is the “remaining part” of the input value, such that its concretization does not contain any node-set values.

To further illustrate the last two functions, the following Scala expression should be true in any domain, provided that the input `list` already contains no duplicates and is ordered in document order: `matchNodeSet(createNodeSet(list)) == (list, bottom)`.

4.1.2 Operations in `XMLDomain`

`def top: N` / `def bottom: N` / `def topList: L` / `def bottomList: L`

These functions return the *top* ($\top$) and *bottom* ($\bot$) elements of the lattice of abstract nodes, `N`, and the lattice of abstract node lists, `L`, respectively.

`def join(n1: N, n2: N): N` / `def meet(n1: N, n2: N): N`

These functions compute the *join* ($\sqcup$) and *meet* ($\sqcap$) of two abstract nodes.

`def joinLists(l1: L, l2: L): L`

This function computes the join $l1 \sqcup l2$ of two abstract node lists. A *meet* operation for node lists was not required by the abstract interpreter.

`def lessThanOrEqual(n1: N, n2: N): Boolean` / `def lessThanOrEqualLists(l1: L, l2: L): Boolean`

These operations define the partial orderings of the lattices `N` and `L`, respectively.

`def createElement(name: V, attributes: L, children: L): N`

This function creates a new element node with the given name, attributes and children. The name is from the subdomain `V` and the lists of attributes and children are of type `L`, so the implementation of this operation depends on the representation of all three types `N`, `L` and `V`. Furthermore, this operation is responsible for merging consecutive text node children before the element is actually created.

`def createAttribute(name: V, value: V): N`

This function creates a new attribute node with the given name and value. The parameters are given as abstract XPath values and only string values ($\leq$ `topString`) are allowed (other values evaluate to $\bot$).

`def createTextNode(value: V): N` / `def createComment(value: V): N`

These functions create new text and comment nodes with the given content, which must be string values. Furthermore, the empty string is not allowed as content for text nodes, so the expression `createTextNode(liftString("")) == bottom` should be true in any implementation of this interface.

`def createRoot(children: L, isResultTreeFragment: Boolean): N`

This function creates a new root node with the given children that either represents a complete document, or a Result Tree Fragment (if `isResultTreeFragment == true`). In the latter case the list of children is unrestricted, but in the former case only a single element child is allowed.

`def createEmptyList(): L` / `def createSingletonList(node: N): L`

These operations create an empty node list, or a list with a single element, respectively.

`def concatLists(list1: L, list2: L): L`

This operation concatenates two lists, i.e. it returns the result of appending the second list to the first.

`def getFirst(list: L): N`

This function extracts the first element of a list of nodes. In the case of an empty list, $\perp$ is returned.

`def getNodeListSize(list: L): V`

This operation obtains the size of a node list as a numeric XPath value.

`def getChildren(node: N): L` / `def getAttributes(node: N): L`

These two functions are used to get a node's children and attributes, hence they allow to navigate downwards in the XML document tree. For nodes that cannot have any children (resp. attributes), an empty list is returned (not $\perp$).

`def getParent(node: N): N` / `def getRoot(node: N): N`

These two functions are used to navigate upwards in an XML tree. Since root nodes do not have a parent, `getParent` returns $\perp$ for those.

`def getDescendants(node: N): L`

This function is used to get a node's descendants. A default implementation based on `getChildren` is provided, but a domain may override this and provide a custom implementation, which is needed in some cases because the default implementation does not enforce termination in the general case.

`def getNodeName(node: N): V`

This function returns the name of a node. For nodes that cannot have a name (i.e. nodes that are not elements or attributes), the empty string is returned (not $\perp$).

`def copyToOutput(list: L): L`

Copies a list of nodes by creating a new list that contains copies of each node in the list. A root nodes is copied by copying its children, so the resulting list will not contain any root nodes.

`def getStringValue(node: N): V` / `def getConcatenatedTextNodeValues(list: L): V`

The first of these functions returns the string value of a node, as defined by the XPath specification [4, Section 5]. The second one obtains the string value for each text node in the given list and returns the concatenation of those, which is required to compute the contents of attribute and comment nodes.

`def isElement(node: N): (N, N)`

This function, as well as the following ones, are *predicate functions* that can be used to check whether an abstract node has a certain property. Predicates always return two results, a positive and a negative one. In the case of `isElement`, the first result is an abstract node that is known to be an element node, and the second result is a node that is not an element node. If we call the two results n_{pos} and n_{neg} for an input node n, the following must hold for all predicate functions: $n_{pos} \leq n$, $n_{neg} \leq n$ and $n_{pos} \sqcup n_{neg} = n$. It is not required that $\gamma(n_{pos})$ and $\gamma(n_{neg})$ are disjoint.

Similar predicate functions for determining whether a given node is a root node, an attribute node, a text node, or a comment node are skipped in this list.

`def hasName(node: N, name: String): (N, N)` / `def isContainedIn(node: N, list: L): (N, N)`

These are further predicate functions that check whether the given node has the specified name, or whether it is contained in a given list, respectively.

`def filter(list: L, predicate: N =>(N, N)): L` / `def takeWhile(list: L, predicate: N =>(N, N)): L`

These operations filter a given abstract node list using a predicate function. The latter stops as soon as the predicate does not match the value and thus always returns a prefix of the original list.

`def flatMapWithIndex(list: L, f: (N, V)=>L): L`

This operation applies a given function `f` to each abstract node in the input list. Additionally, the function is provided the index of the current node in the list as a numeric abstract XPath value. For each node, the function returns a new list of nodes and the resulting lists are concatenated to form the final result. This corresponds to `list.zipWithIndex.flatMap(f)` for concrete objects.

4.2 The Powerset Domain

After having presented the domain interface, we will now look at our first implementation of this interface and also give examples for some of the operations listed above. We called the first domain that we implemented the `PowersetDomain`, because abstract objects in this domain are modeled as sets of concrete values, i.e. each abstract object is an element of the powerset of concrete values. However, in Section 2.2 we have already argued that this is actually impossible for infinite sets, therefore only *finite* sets are explicitly modeled as sets, whereas *infinite* sets are uniformly modeled as a single abstract value that just means "this is an infinite set".

The definition of the three types `N`, `L` and `V` is as follows:

```
type N = Option[Set[XMLNode]] // defined in PowersetXMLDomain
type L = Option[Set[List[XMLNode]]] // defined in PowersetXMLDomain
type V = Option[Set[XPathValue]] // defined in PowersetXPathDomain
```

Listing 4.1: Definition of type parameters `N`, `L`, `V` within `PowersetDomain`

`Option` can be `None` or `Some(...)`, so we use `None` as the special value that designates infinite sets, and `Some(Set(...))` for finite sets of concrete values. It follows that $\top =$ `None` and $\bot =$ `Some(Set())` (i.e. the empty set). The *join* operation returns `None` if any of its two operands is `None`. Otherwise, it corresponds to the union of the two finite sets. The *meet* operation returns the right operand if the left one is `None`, the left operand if the right one is `None`, and the intersection of its operands if neither is `None` (i.e. both are finite sets).

As an example, the lattice for `V` is visualized in Figure 4.1. Only a small part of the lattice is explicitly depicted, because there are infinitely many more finite sets (in fact, there are already infinitely many singleton sets). The dashed lines below `None` shall indicate that there are more elements (finite sets of cardinality 4, 5, 6, ...) in between.

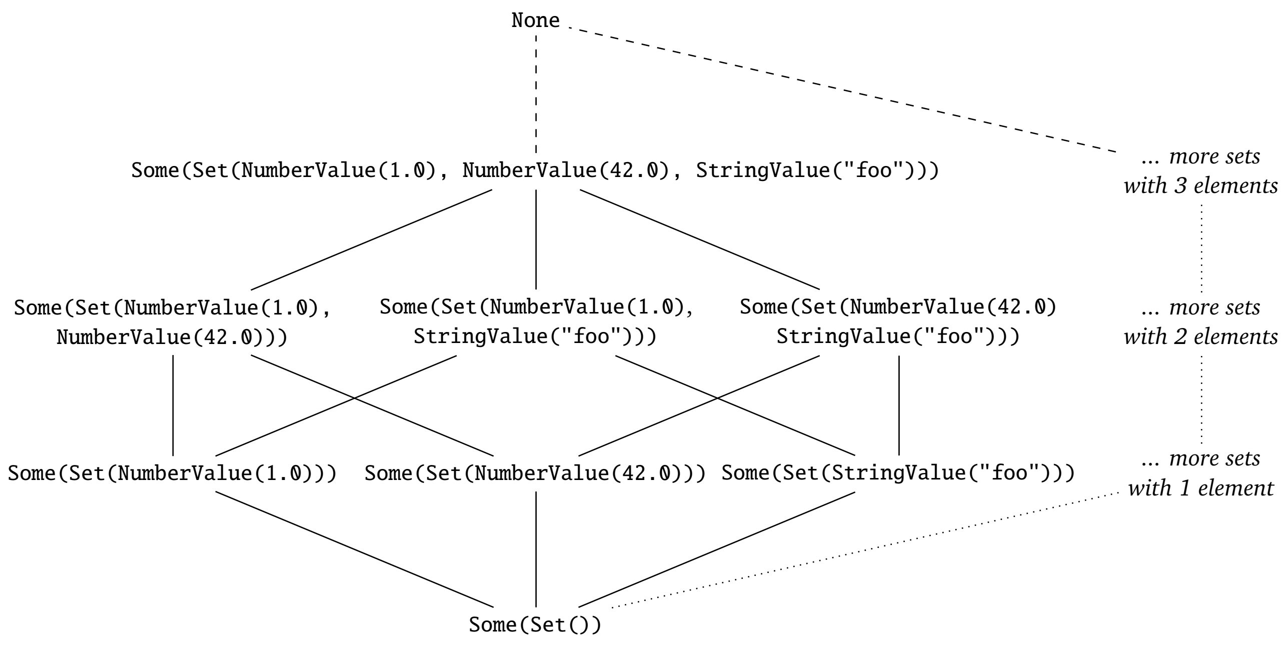
None
Some(Set(NumberValue(1.0), NumberValue(42.0), StringValue("foo")))
... more sets with 3 elements
Some(Set(NumberValue(1.0), NumberValue(42.0)))
Some(Set(NumberValue(1.0), StringValue("foo")))
Some(Set(NumberValue(42.0) StringValue("foo")))
... more sets with 2 elements
Some(Set(NumberValue(1.0)))
Some(Set(NumberValue(42.0)))
Some(Set(StringValue("foo")))
... more sets with 1 element
Some(Set())

Figure 4.1: Lattice `V` in `PowersetXPathDomain`

The advantage of this domain is that most operations can be implemented easily in the following way, making use of the concrete operations that are already implemented for `XMLNode` and `XPathValue`:

- For operations with a single abstract input operand (e.g. `getChildren`), we return `None` when the input is `None`, and operate on each element of the set independently otherwise. Note that this corresponds to two nested calls of `map` in Scala, one for the `Option` and one for the `Set`.

- The result of binary operations (e.g. arithmetic operations in `V`) is computed as follows: Whenever one of the operands is $\perp$ (i.e. the empty set), the result is $\perp$; when no operand is $\perp$, but at least one operand is $\top$ (i.e. `None`), the result is $\top$; otherwise we compute the Cartesian product of the input sets and apply the operation to each element of that product. For example, the addition operation on the (finite) sets of number values $\{0, 1\}$ and $\{1, 2\}$ (where we write x for `NumberValue(`x`)`) results in the set $\{0+1, 0+2, 1+1, 1+2\} = \{1, 2, 3\}$.

- To implement predicate functions, we return `None` for both the positive and the negative result if the input is `None`. If the input is a finite set, we use the Scala function `partition` to obtain two sets such that all elements in the first set fulfill the predicate (this is the positive result) and all elements in the other do not fulfill the predicate (this is the negative result).

The clear disadvantage of this domain is that as soon as a *single property* of an object, e.g. the name of an attribute or the content of a text node, is unknown (in the sense that there are infinitely many possibilities), the whole object can only be represented as `None`. Therefore we can not expect useful results when performing abstract interpretation on the input $\top$ (i.e. `None`), which is what we will use in order to gain information about the possible outputs of XSLT transformations without knowing anything at all about the input document.

A better domain implementation is presented in Section 4.4, but the main differences there are in the XML subdomain. The XPath subdomain that we use there (`TypedPowersetXPathDomain`, see Listing 4.2), however, is similar to the `PowersetXPathDomain` presented above, but it allows better composition with other XML subdomain implementations, whereas `PowersetXPathDomain` is tied to the `XMLNode` class, because node-sets inside of XPath values are always sets of `XMLNode` instances.

```
type V = (Set[Boolean], Option[Set[Double]], Option[Set[String]], L)
```

Listing 4.2: Definition of type parameter `V` within `TypedPowersetXPathDomain`

In contrast to the previously shown type definition, the structure is now a 4-tuple, and each component of the tuple represents one of the XPath types independently from the others. We do not use the `XPathValue` class from the concrete interpreter, but work directly on booleans, doubles and strings. The fourth component represents node-sets and is generic in the type `L`, as is the `TypedPowersetXPathDomain` as a whole. This means that it can reuse any representation for abstract node lists that is defined by some implementation of `XMLDomain`, which makes it a lot more flexible than the previous version.

Each of the four components can be regarded as a lattice on its own. For numbers and strings we still use the `Option[Set[...]]` approach, but for booleans this is not necessary because the set of all possible boolean values (i.e. $\{\texttt{true}, \texttt{false}\}$) is finite.[1]

It might be noted that, while the `PowersetXPathDomain` has to return `None` (i.e. $\top$) not only for the `top` operation, but also for `topNumber` and `topString`, the typed version can actually distinguish these. For example, `topNumber` is represented as the 4-tuple `(Set(), None, Some(Set()), bottomList)`, where `bottomList` depends on the XML domain.

In order to avoid repetition in the definition of the basic lattice operations, we defined the type class `Lattice[A]` (see Listing 4.3) and provide a generic implementation of `Lattice[Option[Set[T]]]` for every `T`.

```
trait Lattice[A] {
  def top: A
  def bottom: A
  def join(left: A, right: A): A
  def meet(left: A, right: A): A
  def lessThanOrEqual(left: A, right: A): Boolean
}
```

Listing 4.3: Definition of the lattice type class `Lattice[A]`

4.3 Abstract Interpreter

Now that we have presented the domain interface and outlined a possible implementation thereof, we will turn towards the abstract interpreter itself, and how it uses the previously introduced domain types and operations. Since it is based on the concrete interpreter from the previous chapter, it is not surprising that the overall structure of the implementation is very similar.

4.3.1 XPath

We start with the class modeling the abstract XPath context, shown in Listing 4.4 (cf. Listing 3.1 for the concrete case). It contains the same data as the concrete context, but the context node is now an abstract node, and the position, size, as well as all variable values are abstract XPath values.

```
case class AbstractXPathContext[N, L, V](node: N, position: V, size: V, variables: Map[String, V])
```

Listing 4.4: Definition of abstract XPath context class

The abstract version of `XPathEvaluator` is `XPathAnalyzer`, which is also generic in `V`, `N` and `L`, and its evaluation function is similar to the concrete one shown in Listing 3.2, with the following modifications:

- The evaluation of literal number and string expressions involves calling the corresponding `lift` operations defined in the domain interface.

[1] Technically this is also true for doubles, because they have exactly 64 bits of precision, but representing an unknown number as a set of 2^{64} values is inconvenient, to say the least.

- The evaluation of arithmetic operations, relational comparisons and node-set unions is implemented using direct calls to the corresponding domain interface methods.
- There are no designated domain operations for boolean conjunction and disjunction, because the evaluation of 'and' and 'or' expressions (after the operands are converted using `toBooleanValue`) only requires comparisons of the operands with the (lifted) values `true` and `false` (w.r.t. the order in the lattice). However, the evaluation of these operations is slightly complicated by the fact that they are specified to use *short-circuit semantics*. For example, in the case of the 'or' operator this means that "the right operand is not evaluated if the left operand evaluates to true" [4, Section 3.4], which implies that our abstract result needs to contain `true` if the left operand is greater than `true`, even if the the right operand evaluates to $\bot$.

Furthermore, the evaluation of location paths in the abstract interpreter is presented in Listing 4.5. Note that the values of `xmlDom` and `xpathDom`, which are specified in the constructor of `XPathAnalyzer`, are implementations of the domain interface `XMLDomain` and `XPathDomain`, respectively.

```
 1 private def evaluateLocationPath(node: N, steps: List[XPathStep], isAbsolute: Boolean): L = {
 2   (steps, isAbsolute) match {
 3     case (_, true) => evaluateLocationPath(xmlDom.getRoot(node), steps, false)
 4     case (Nil, false) => xmlDom.createSingletonList(node)
 5     case (first :: rest, false) =>
 6       val nodes: L = first.axis match {
 7         case ChildAxis => xmlDom.getChildren(node)
 8         case DescendantAxis => xmlDom.getDescendants(node)
 9         case ParentAxis =>
10           val (root, nonRoot) = xmlDom.isRoot(node)
11           val result = xmlDom.createSingletonList(xmlDom.getParent(nonRoot))
12           if (!xmlDom.lessThanOrEqual(root, xmlDom.bottom))
13             xmlDom.joinLists(xmlDom.createEmptyList(), result)
14           else
15             result
16         case AttributeAxis => xmlDom.getAttributes(node)
17         case SelfAxis => xmlDom.createSingletonList(node)
18         case DescendantOrSelfAxis => xmlDom.concatLists(xmlDom.createSingletonList(node), xmlDom.
              getDescendants(node))
19       }
20       val testedNodes = xmlDom.filter(nodes, n =>
21         /* ... filter according to node test in current step, using predicate functions ... */)
22       if (first.predicates.nonEmpty) throw new NotImplementedError("Predicates are not supported")
23       val (testedNodeSet, _) = xpathDom.matchNodeSet(xpathDom.createNodeSet(testedNodes))
24       xmlDom.flatMapWithIndex(testedNodeSet, {
25         case (n, _) => evaluateLocationPath(n, rest, false)
26       })
27   }
28 }
```

Listing 4.5: Abstract evaluation function for location paths

Again, the basic structure of this code is the same as that of the corresponding method in the concrete interpreter (cf. Listing 3.3), but the current node is now an abstract node (`N`) and the result is an abstract node list (`L`). To aggregate the results of the recursive evaluation (Lines 24–26), the `flatMapWithIndex` operation is used (the index is not required here and therefore ignored, but it is required in the `applyTemplates` method, which is why the operation was included in the domain just as it is).

To retrieve the nodes of the specified axis in each step, the corresponding domain operation is called (Lines 7–18). However, the parent axis needs to be treated specially, because we have to correctly handle the case where the current node is the root node, which does not have a parent. In this case, `getParent` returns $\bot$, and if we put that into a list (using `createSingletonList`), the resulting list would also be

$\perp$. The correct result, however, would be an empty list. For this reason we check whether the current node might be a root node by comparing the positive result of the `isRoot` predicate function with $\perp$: If it is greater than $\perp$ (i.e. $\gamma(node)$ contains at least one root node), we include the empty list in the result (Lines 12–13). The parent is retrieved only for the negative result of `isRoot` (Line 11).

Note that in the abstract interpreter we did not implemented other axes apart from the six shown in Listing 4.5, because these are the most commonly used (cf. the evaluation in Section 5.2), and the other ones do not give useful results with the Zipper Domain (cf. Section 4.4) anyway, since we do not have information about sibling nodes there.

Finally, the list of nodes specified by the axis in the current step is filtered (using the `filter` domain operation) according to the node test of the same step. This involves the predicate functions `isElement`, `isAttribute`, `isTextNode`, `isComment` and `hasName`.

We make sure that the list of nodes contains no duplicates and is sorted in document order (i.e. it represents an ordered set) in every step, by first creating a node-set XPath value of it and then extracting a value of type `L` again (Line 23). This is not strictly necessary, but it removes duplicates early and therefore the remaining path is not evaluated multiple times for the same node. One could argue that we should not do this, because in some domains sorting the set can involve a loss of precision (for example, in the Zipper Domain, which will be explained in detail in Section 4.4, it can be the case that it is impossible to determine whether two abstract nodes a and b refer to the same node, or a comes before b, or b before a in document order). Sorting the list is required, however, at least once when the location path is fully evaluated (this is not obvious in the excerpt above, but when `evaluateLocationPath` is called from `evaluate`, its result is always put into a node-set value using `createNodeSet`). Hence the loss of precision is unavoidable in general.

4.3.2 XSLT

The abstract version of `XSLTProcessor` (cf. Listing 3.6) is `XSLTAnalyzer`, outlined in Listing 4.6. The differences between this abstract version and the concrete one are as follows:

- In `transform`, which is the entry point for the analysis of a transformation, we first enforce that the source node is a root node (and because root nodes are also used to model Result Tree Fragments, we have to disallow this explicitly). We continue only with the positive result of the `isRoot` predicate function (Line 2).

- In `applyTemplates` we apply matching template rules to each node in the abstract node list `sources`, using `flatMapWithIndex` (Line 12). As the context contains the current position in the list, the index is actually required here (Line 16–17). In contrast to the concrete interpreter, the abstract interpreter can not determine a unique template that matches the node, since an abstract node might represent various concrete nodes that all match different templates. For this reason `chooseTemplates` returns a list of templates, together with refinements of the abstract nodes that match them. The results of the instantiation of these templates are then joined into a single result (Lines 15–19).

- `joinAllLists` (code omitted) basically folds the input sequence using the `joinLists` domain operation, but returns early if an intermediate result is already equal to $\top$ (since $\top \sqcup x = \top$ for any x). This is a performance optimization: In the result of `chooseTemplates`, the least specific template (e.g. the built-in template that matches any element) comes last. This list is reversed and mapped *lazily* (using `view`), so that the least specific template, which is also most likely to return $\top$, is instantiated first and the remaining templates are not instantiated any more if it is actually the case that the instantiation of a template (or a combination thereof) results in $\top$.

- `chooseTemplates` iterates through all templates of the given mode and calls the `matches` method of an instance of `AbstractPatternMatcher` for the currently used domain. This method has the

```scala
def transform(sheet: XSLTStylesheet, source: N, recursionLimit: Option[Int] = None): N = {
  val (rootSource, _) = xmlDom.isRoot(source, allowResultTreeFragments = false)
  val globalVariables = /* ... evaluate global variables in stylesheet ... */
  val result = applyTemplates(sheet, xmlDom.createSingletonList(rootSource), None, globalVariables,
    Map(), Map(), recursionLimit)
  xmlDom.createRoot(result, isResultTreeFragment = false) // wrap result in a root node
}

def applyTemplates(sheet: XSLTStylesheet, sources: L, mode: Option[String],
    globalVariables: Map[String, V], localVariables: Map[String, V], params: Map[String, V],
    recursionLimit: Option[Int]): L = {
  xmlDom.flatMapWithIndex(sources, (node, index) => {
    val templates = chooseTemplates(sheet, node, mode)

    joinAllLists(templates.reverse.view.map { case (tmpl, specificNode) =>
      val context = AbstractXSLTContext[N, L, V](specificNode, sources,
        xpathDom.add(index, xpathDom.liftNumber(1)), globalVariables, localVariables)
      instantiateTemplate(sheet, tmpl, context, params, recursionLimit)
    })
  })
}

def chooseTemplates(sheet: XSLTStylesheet, node: N, mode: Option[String]):
    List[(XSLTTemplate, N)] = {
  val result = MutList[(XSLTTemplate, N)]()
  val matchable = sheet.matchableTemplates.getOrElse(mode, Map())
  var currentNode = node
  breakable {
    matchable.foreach { case (path, tpl) =>
      val (matches, notMatches) = patternMatcher.matches(currentNode, path)
      if (!xmlDom.lessThanOrEqual(matches, xmlDom.bottom))
        result += ((tpl, matches)) // template might match, so add it to possible results
      if (xmlDom.lessThanOrEqual(notMatches, xmlDom.bottom) || xmlDom.lessThanOrEqual(currentNode, matches))
        break() // the node matched completely, so we can stop the process
      currentNode = notMatches // continue with that "part" of the node that did not match
    }
  }
  result.toList
}

def instantiateTemplate(sheet: XSLTStylesheet, tmpl: XSLTTemplate, context: AbstractXSLTContext[N, L, V], params: Map[String, V], recursionLimit: Option[Int]): L = {
  if (recursionLimit == Some(0)) return xmlDom.topList // return TOP if recursion limit is reached
  val evaluatedParams = /* ... evaluate parameters in current context ... */
  processAll(sheet, tmpl.content, context.replaceLocalVariables(evaluatedParams),
    recursionLimit.map(_ - 1))
}

def processAll(sheet: XSLTStylesheet, instructions: Seq[XSLTInstruction], context: AbstractXSLTContext[N, L, V], recursionLimit: Option[Int]): L = /* ... process each instruction in turn and collect variable definitions, so they are available in subsequent instructions (in the same scope) ... */

def process(sheet: XSLTStylesheet, instruction: XSLTInstruction, context: AbstractXSLTContext[N, L, V], recursionLimit: Option[Int]): Either[L, (String, V)] = /* ... match on instruction type ... */
```

Listing 4.6: Functions within XSLTAnalyzer

same semantics as a predicate function, i.e. it returns a positive and a negative result, such that, given an input node n, e.g. the positive result n_{pos} matches the template and $n_{pos} \leq n$. We add a template (and its matching node) to the list of possible matching templates whenever $n_{pos} > \perp$ (Lines 31–32). When the node matches completely (i.e. $n_{neg} = \perp$), we can stop the process because of the way the templates are ordered (Lines 33–34). If it did not match completely, we continue with the negative result (Line 35), because only for that we have not yet found a template.

- Every function (except chooseTemplates) has an additional parameter recursionLimit, which can be used to limit the recursion depth of template instantiation. It can be set to Some(x) in the beginning and is decremented in instantiateTemplate. When it has reached 0, $\top$ is returned. Setting the limit to None allows unlimited recursion. The reason why such a limitation is needed will be the subject of Section 4.5.

As examples for the processing of individual XSLT instructions in process, we now present and explain the cases for CreateElementInstruction (Listing 4.7) and CopyOfInstruction (Listing 4.8) as we have for the concrete interpreter (cf. Listing 3.7 and 3.8).

```
1  case CreateElementInstruction(name, content) =>
2     val innerNodes = processAll(sheet, content, context, recursionLimit)
3     val attributes = xmlDom.takeWhile(innerNodes, n => xmlDom.isAttribute(n))
4     val children = xmlDom.filter(innerNodes, n => {
5        val (isAttr, isNotAttr) = xmlDom.isAttribute(n)
6        (isNotAttr, isAttr) // swap positive and negative results of isAttribute
7     })
8     val evaluatedName = xpathDom.concatAllStrings(name.map {
9        case Left(str) => xpathDom.liftString(str)
10       case Right(expr) => xpathDom.toStringValue(xpathAnalyzer.evaluate(expr, xsltToXPathContext(
            context)))
11    })
12    val result = xmlDom.createElement(evaluatedName, attributes, children)
13    Left(xmlDom.createSingletonList(result))
```

Listing 4.7: Case for CreateElementInstruction within process (abstract interpreter)

The CreateElementInstruction is processed by first processing the sequence of instructions that form the content (Line 2). The resulting abstract node list contains attributes as well as other child nodes, which have to be separated. This is done using the takeWhile and filter domain operations, together with the isAttribute predicate function (Lines 3–7). To find all child nodes that are *not* attributes, the positive and negative results of isAttribute can simply be swapped.

Then, the attribute value template for the element's name is evaluated (Lines 8–11). This involves the evaluation of XPath expressions, using xpathAnalyzer, an instance of the XPathAnalyzer class for the currently used domain. Finally, the abstract element node is created and returned as the only element of an abstract node list (Lines 12–13).

The CopyOfInstruction (recall that this is used for <xsl:copy-of> as well as for <xsl:value-of>) is processed by first evaluating the XPath expression that selects the nodes that are to be copied (Line 2). The result is an abstract XPath value that might be a mix of node-set values and other values. Because these must be treated in different ways, we split them using the matchNodeSet domain operation (Line 3). Values that are node-sets are copied directly using the copyToOutput operation. Values that are not node-sets are converted to strings and then wrapped in a text node (Lines 5–6). However, if the empty string is part of the result, we include the empty list in the final result (Lines 7–9), since text nodes with empty content are not allowed (recall that createTextNode returns $\perp$ for the empty string, and without this special treatment, the resulting list would then also be $\perp$ instead of an empty list).

Finally, the results for node-set values on the one hand and other values on the other hand, which have been computed independently, are joined into a single result list (Line 10).

```
case CopyOfInstruction(select) =>
   val result = xpathAnalyzer.evaluate(select, xsltToXPathContext(context))
   val (nodeSets, rest) = xpathDom.matchNodeSet(result)
   val nodeSetsOutput = xmlDom.copyToOutput(nodeSets)
   val restAsString = xpathDom.toStringValue(rest)
   var restOutput = xmlDom.createSingletonList(xmlDom.createTextNode(restAsString))
   if (xpathDom.lessThanOrEqual(xpathDom.liftString(""), restAsString)) {
      restOutput = xmlDom.joinLists(restOutput, xmlDom.createEmptyList())
   }
   Left(xmlDom.joinLists(nodeSetsOutput, restOutput))
```

Listing 4.8: Case for `CopyOfInstruction` within `process` (abstract interpreter)

Pattern Matching

In the `chooseTemplate` function, which is responsible for selecting templates that potentially match an abstract node, we have assumed the existence of an `AbstractPatternMatcher` class that provides the functionality corresponding to `XSLTPatternMatcher` in the concrete case (cf. Section 3.3.1). This last section dealing with the implementation of the abstract interpreter is dedicated to the implementation of the `matches` function within that class.

```
private def matches(node: N, reversedPathSteps: List[XPathStep], pathIsAbsolute: Boolean):
    (N, N) = {
  // match recursively from right to left (path steps are reversed!)
  reversedPathSteps match {
    case nextStep :: restPath =>
      val (nextStepMatches, notNextStepMatches) = matchesSingleStep(node, nextStep)
      if (!xmlDom.lessThanOrEqual(nextStepMatches, xmlDom.bottom)) restPath match {
        case XPathStep(DescendantOrSelfAxis, AllNodeTest, Nil) :: Nil =>
          (nextStepMatches, notNextStepMatches)
        case XPathStep(DescendantOrSelfAxis, AllNodeTest, Nil) :: restTail =>
          /* ... find nodes where some ancestor matches the rest of the path ... */
        case _ =>
          // find nodes where the parent matches the rest of the path
          val parent = xmlDom.getParent(nextStepMatches)
          val (parentMatchesRest, _) = matches(parent, restPath, pathIsAbsolute)
          val (parentMatches, notParentMatches) = hasParent(nextStepMatches, parentMatchesRest)
          (parentMatches, xmlDom.join(notParentMatches, notNextStepMatches))
      } else (xmlDom.bottom, notNextStepMatches)
    case Nil => if (pathIsAbsolute) xmlDom.isRoot(node) else (node, xmlDom.bottom)
  }
}

private def hasParent(node: N, parent: N): (N, N) = {
  val (isChild, isNotChild) = xmlDom.isContainedIn(node, xmlDom.getChildren(parent))
  val (isAttribute, isNeither) = xmlDom.isContainedIn(isNotChild, xmlDom.getAttributes(parent))
  (xmlDom.join(isChild, isAttribute), isNeither)
}
```

Listing 4.9: Matching of location path patterns in the abstract interpreter

Listing 4.9 outlines the relevant functions (the publicly exposed overload of `matches`, that is responsible for reversing the path steps, is not shown). The obvious difference to the concrete case is the type of the return value, which has already been mentioned in passing: Pattern Matching no longer returns a `Boolean`, but two abstract XML nodes that constitute the positive and the negative result of the matching process.

The matching of a single step (delegated to the `matchesSingleStep` function, not shown here) combines various predicate functions to obtain refinements of `node`, depending on whether the combination of axis and node test is fulfilled by the node. If the step definitely does not match the node, `nextStepMatches` will be $\perp$ and no recursive call is made (cf. the check in Line 7).

If that is not the case and the next step is not a descendant step, the matching must continue recursively. Note that the recursive call to check whether the rest of the path matches the parent is no longer a tail call, because now the result of the recursive call does not simply tell us whether the parent matches, but instead gives us a refined node (`parentMatchesRest`, Line 15). We then have have to refine the matching result of the current step (`nextStepMatches`) to obtain a node of which we know that its parent matches the rest of the path (`parentMatches`, Line 16).

The example shown in Figure 4.2 shall illustrate this.

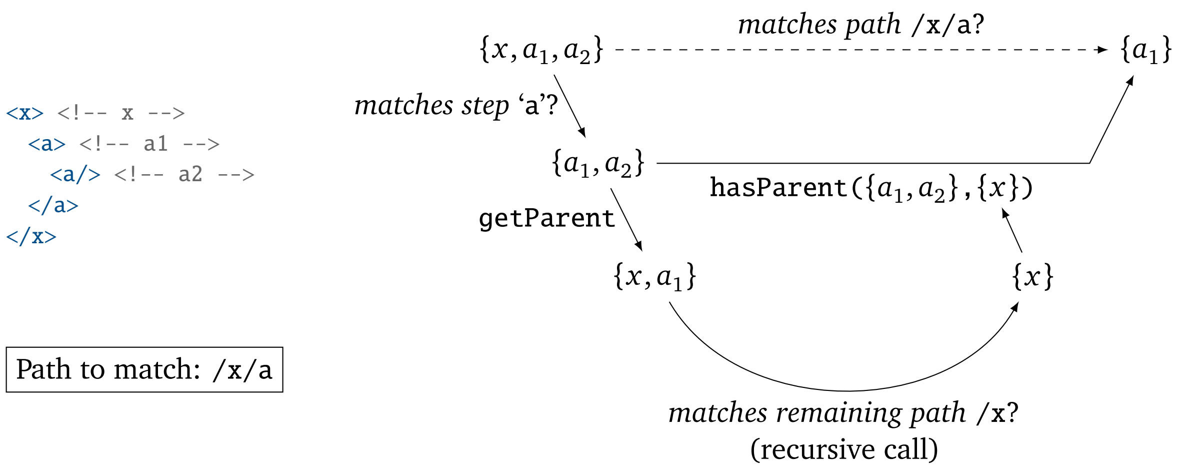

Figure 4.2: Illustration of abstract location path pattern matching

Here, we use x, a_1 and a_2 to refer to the element nodes in the depicted XML document, and sets of those nodes to refer to abstract nodes in the sense of the Powerset Domain, e.g. $\{x, a_1\}$ represents the abstract node that is either x or a_1.

We start from the abstract node $\{x, a_1, a_2\}$, which represents any element in the document, and want to apply the matching algorithm to the path `/x/a`, restricting our attention to the positive result. To do this, we first test against the leftmost step 'a' (unsurprisingly, this uses `isElement` and `hasName` internally) and obtain the abstract node $\{a_1, a_2\}$, because a_1 and a_2 match this step, but x does not. We then call `getParent`, which – in the powerset domain – just returns the set of all parents of the original nodes, i.e. $\{x, a_1\}$. The `matches` algorithm is applied recursively to that abstract node and the remaining path `/x`, so the result is $\{x\}$ because a_1 does not match `/x`.

This is then used to refine the original node $\{x, a_1, a_2\}$ in such a way that only those concrete nodes remain whose parent is x. Therefore, the final result is $\{a_1\}$, which is indeed the only node from the document that matches the path `/x/a`.

In the case of a (non-trivial) descendant step (i.e. we are interested in whether any *ancestor* matches the remaining path), the calls to `getParent` and `hasParent` are iteratively repeated until a root node is reached. However, our implementation is not guaranteed to terminate when `getParent` returns possibly non-root nodes infinitely (this can occur in the Zipper Domain, but not in the Powerset Domain). Since in our evaluation of real-world stylesheets (cf. Section 5.2) none of the analyzed transformations used non-trivial descendant steps, we did not spend more time improving this situation.

4.4 The Zipper Domain

We have seen that the Powerset Domain as introduced in Section 4.2 is not very useful when running the abstract interpreter on the unknown input $\top$. The `ZipperXMLDomain` is another implementation of the `XMLDomain` interface, that together with the `TypedPowersetXPathDomain` forms the `ZipperDomain`. It allows an abstract representation of XML trees and nodes that are only partially known.

The Zipper Domain received its name from the *Zipper* data structure as described by Gérard Huet [7]. The Zipper is an implementation of a (concrete) tree where every node is modeled as a combination of a subtree and a path. The subtree contains the part of the tree from the current node downwards to the leaves, while the path contains the rest of the tree, i.e. everything in between the current node and the root as well as the siblings of the current node. Thus, every node contains a representation of the complete tree, which can be traversed in all directions (this is important because in XSLT processing the input tree is usually traversed top-down, but patterns are matched bottom-up). In contrast to our previous implementation of XML nodes, however, the Zipper avoids cyclic references between child and parent nodes. This is important because cyclic references would significantly complicate the implementation of basic lattice operations. On the other hand, with an acyclic object graph, these operations can be implemented in a straightforward way using a divide-and-conquer principle (this can be observed in Listing 4.15).

Following this principle, in order to model an abstract node we created a lattice `S` for abstract subtrees and a lattice `P` for abstract paths. Each abstract node then is a combination of an abstract subtree and an abstract path (recall that multiple lattices can be combined to form new lattices). It might also be noted that the representations of subtrees and paths are independent, so e.g. `P` could be replaced by some other definition without changing the general design of the Zipper Domain (however, the implementation of some operations depend on internals of both types).

```
type S = Subtree // type of abstract subtrees
type P = Set[Path] // type of abstract paths
type N = (S, P) // a node is a subtree and a path
type L = ZList[N] // type of abstract node lists
type V = (Set[Boolean], Option[Set[Double]], Option[Set[String]], L)
```

Listing 4.10: Overview of the type definitions in `ZipperDomain`

The type definitions for the lattices within the Zipper Domain are outlined in Listing 4.10, and Figure 4.3 visualizes how an abstract node in this domain is composed of a path and a subtree. The rest of this subsection elaborates on the definitions of `Subtree`, `Path` and `ZList`, the latter being a representation of abstract lists that we use not only for lists of nodes, but also for the list of children and attributes within each subtree.

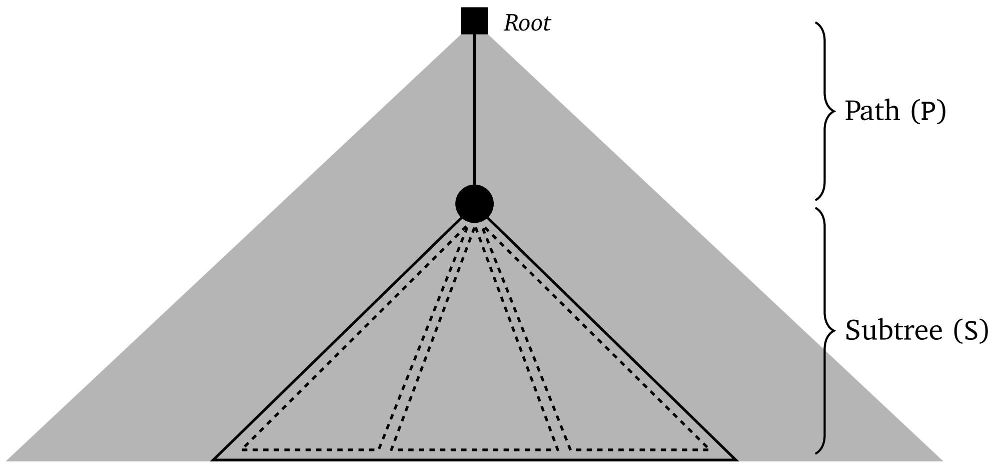

Figure 4.3: Schematic visualization of an abstract node in the Zipper Domain

4.4.1 Abstract Paths

Paths in the original *Zipper* structure are defined as being either the root of the tree, or a combination of left siblings (a list of subtrees), a parent path, and right siblings (again a list of subtrees). Our design of abstract paths, shown in Listing 4.11, deviates from that structure in some ways:

```
abstract class PathStepDescriptor
case object AnyElementStep extends PathStepDescriptor
case class NamedElementStep(name: String) extends PathStepDescriptor
case object AnyAttributeStep extends PathStepDescriptor
case class NamedAttributeStep(name: String) extends PathStepDescriptor
case object AnyTextNodeStep extends PathStepDescriptor
case object AnyCommentNodeStep extends PathStepDescriptor

abstract class Path
case class RootPath(isFragment: Boolean) extends Path
case class ChildStep(descriptor: PathStepDescriptor, parent: Path) extends Path
case class DescendantStep(descriptor: PathStepDescriptor, ancestor: Path) extends Path

type P = Set[Path]
```

Listing 4.11: Type definitions for abstract paths

- The original *Zipper* models unlabeled trees. Our trees, however, are labeled, so we added an additional `PathStepDescriptor` to each step. These form a partial order (but not a lattice): `AnyElementStep` is greater than `NamedElementStep(name)` for any name and the same holds for `AnyAttributeStep` and `NamedAttributeStep`, respectively. Any other two descriptors are incomparable.
- A path is built from steps, where each step references a parent path until the root is reached. The original *Zipper* only knows the child relationship, but we additionally allow a descendant relationship, which is less precise, because the steps in between are unknown.
- We do not store any information about siblings. While this leads to a loss of precision when navigating down a tree, then up, then down another path (i.e. to a sibling of a previously visited node), it simplifies the implementation of operations. Furthermore, the loss of precision only occurs when the `parent` axis is used in XPath selectors, since that is the only case where the tree is navigated upwards with the possibility of going down again (other axes that navigate upwards in the tree are not supported in our implementation). During pattern matching the tree is also navigated upwards, but without predicates the matching algorithm can only go further up and never down again.
- The `RootPath` has an additional field that is used to distinguish true document roots from Result Tree Fragments. We will not go into details about this and continue as if this flag did not exist.

This way, every instance of `Path` is essentially equivalent to some XPath location path pattern. For example, the path `DescendantStep(AnyAttributeStep, ChildStep(NamedElementStep("foo", RootPath)))` is equivalent to the path */foo//@**, meaning that the current node is an attribute with any name, which is a descendant of an element named “foo”, which in turn is a child of the root node. This correspondence between instances of `Path` and location path patterns is intended and shows that `Paths` contain exactly the kind of information that is required for pattern matching. Furthermore, it allows us to use the shorter pattern notation in the following paragraphs.[2]

[2] For the same reason we implemented `toString` for `Paths` in such a way that it prints the corresponding XPath pattern. We also implemented a method to create a `Path` from a location path pattern, to be used in unit tests.

Instances of `Path` also form a partial order, in the sense that two paths can be compared w.r.t. their precision (note that we regard $p_1 < p_2$ if p_1 is *more* precise than p_2, as we did for lattices, where the greatest element $\top$ is the *least* precise element). For example, a node that has the path `/foo/bar/@attr` also has the (less precise) path `/foo//@*`, therefore `/foo/bar/@attr` $<$ `/foo//@*`. To obtain a complete lattice from those, we look at *sets* of instances of `Path` and make the following observations:

- Every node is either the root node, or an element, attribute, comment or text node that is a descendant of the root node. So the path of every node is contained in the set {`/`, `//*`, `//@*`, `//text()`, `//comment()`} (recall that `text()` is the node test for text nodes, hence in this context it refers to the `AnyTextNodeStep` descriptor). Consequently that set is the top element in the lattice of abstract paths. The bottom element is the empty set $\bot = \emptyset$.

- A set of `Path`s might contain elements that are more precise than other elements in the same set. We do not allow such sets and implemented a method `normalize` that removes any such elements. For example, `normalize`({`/foo/bar/@attr`, `/foo//@*`}) = {`/foo//@*`}.

- The join operation can be defined by taking the union of two sets and normalizing the result: $a \sqcup b =$ `normalize`$(a \cup b)$. For example, {`/a`} $\sqcup$ {`/a/b`} = {`/a`, `/a/b`} and {`/a`} $\sqcup$ {`/*`} = {`/*`}.

- The meet operation can be defined by looking at the Cartesian product of two sets and for each two elements calculating a set of paths representing all paths that are less than both of them. When the paths do not contain descendant steps, this is the singleton set containing the lesser of the two if they can be compared, or the empty set if they are incomparable. For example, {`/a`, `/a/b`} $\sqcap$ {`/*`} = {`/a`}, because `/a` $<$ `/*`, but `/a/b` and `/*` are incomparable. For paths with descendant steps, the result corresponds to all possible interleavings of path steps, for example {`//a//b`} $\sqcap$ {`//c//b`} = {`//a//c//b`, `//c//a//b`}. Here, the abstract path describes a node that is an element named 'b' and has an ancestor named 'a' and also an ancestor named 'c'.

Therefore, `P = Set[Path]` is the final type that we use for abstract paths, and we implemented an instance of the `Lattice` type class for `P` that provides implementations for the basic lattice operations as described above.

4.4.2 Abstract Lists

Before we explain abstract subtrees, which constitute the other part of abstract nodes, we now introduce the data structure that we use for abstract lists, called `ZList`, since they are also used within subtrees.

A `ZList` can be viewed as a kind of linked list with more constructors in addition to the usual *Cons* and *Nil* (cf. lists in Haskell or the Scala `List` type). We implemented `ZList` in such a way that it is generic in the type of its elements, but an (implicit) instance of the lattice type class is required for this element type for all operations in `ZList`. For a given element type `T`, instances of `ZList[T]` form a lattice themselves.

```
abstract class ZList[T] { /* ... */ }
case class ZBottom[T]() extends ZList[T]
abstract class ZListElement[T] extends ZList[T]
case class ZTop[T]() extends ZListElement[T]
case class ZUnknownLength[T](elems: T) extends ZListElement[T]
case class ZCons[T](first: T, rest: ZListElement[T]) extends ZListElement[T]
case class ZMaybeNil[T](first: T, rest: ZListElement[T]) extends ZListElement[T]
case class ZNil[T]() extends ZListElement[T]
```

Listing 4.12: Type definitions for `ZList`

Listing 4.12 shows the type definition for all ZList constructors: `ZCons`, `ZNil`, `ZBottom`, `ZTop`, `ZUnknownLength` and `ZMaybeNil` (the purpose of the intermediate abstract class `ZListElement`, that distinguishes `ZBottom` from the other constructors, will become clear later). The meaning of the individual constructors is as follows:

- `ZNil` describes the empty list. We also write the empty list as $[]$.
- `ZCons` describes a list that has one element, followed by another list of elements, which might be empty. We also write e.g. $[a, b, c]$ for the list `ZList(`a`,ZList(`b`,ZList(`c`,ZNil())))`. Recall that a, b and c are of type `T` and therefore lattice elements themselves.
- `ZMaybeNil` is either the empty list or a *Cons*. It means that the list might end here, but might as well be followed by another list. We also write e.g. $[a, \textsc{Nil} + b]$ for the list `ZList(`a`,ZMaybeNil(`b`,ZNil()))`, where the '+' sign means "either Nil (end of list) or what follows". `ZMaybeNil` can also be regarded as the *join* of `ZNil` and `ZCons`.
- `ZUnknownLength` describes a list whose content is known, but whose length is not. It contains no further following list segment, meaning that the list either ends at this point or contains an unknown number of repetitions of the contained lattice element. We also write e.g. $[a, \textsc{Nil} + b \ldots]$ for the list `ZList(`a`,ZUnknownLength(`b`))`, where '$\textsc{Nil} + b \ldots$' means "b repeated zero or more times".
- `ZTop` describes any list. This could actually also be represented as `ZUnknownLength(`$\top$`)`, where $\top$ is the top element from the lattice of list elements (i.e. any element, repeated any number of times), but an additional constructor is required to model recursive data structures where the element lattice contains a `ZList` of itself, as is the case with abstract subtrees. Without `ZTop`, the top element of such a structure would not be finitely representable. We also write e.g. $[a, \textsc{Top}]$ for the list `ZList(`a`,ZTop())`
- `ZBottom` describes no list and constitutes the bottom element of the `ZList` lattice. This is not strictly required, since e.g. `ZCons(`$\bot$`)` also describes no list, but we normalize the representation in such a way that `ZBottom` is used whenever a single list element would be $\bot$. Using the intermediate type `ZListElement` for everything but `ZBottom`, we statically ensure that only the whole list can be `ZBottom`, i.e. `ZBottom` is not used as a tail in some other list. We also write $[\textsc{Bottom}]$ for `ZBottom()`.

In the next paragraphs we give some examples of the set of concrete lists that correspond to abstract lists represented using `ZList`, and show results for some lattice and list operations. For explanatory purposes, we will use the lattice of integer sets $\mathscr{P}(\mathbb{Z})$ for the element type `T`.

Consider the abstract list $l_1 = [\{0\}, \{1\}, \{2\}]$. Since the list has exactly three elements and each element is a singleton set, the only concrete list that is represented by l_1 is $[0, 1, 2]$, so $\gamma(l_1) = \{[0, 1, 2]\}$. Let $l_2 = [\{3\}]$ be the list that only contains the element 3 (so $\gamma(l_2) = \{[3]\}$). Joining l_1 and l_2 gives us a new abstract list: $l_3 = l_1 \sqcup l_2 = [\{0, 3\}, \textsc{Nil} + \{1\}, \{2\}]$. The first element of l_3 is either 0 or 3 and the list might end after the first element. However, $\gamma(l_3) = \{[0], [3], [0, 1, 2], [3, 1, 2]\} \supsetneq \gamma(l_1) \cup \gamma(l_2)$ shows that this indeed is an over-approximation.

To better understand `ZUnknownLength`, consider $l_4 = [\textsc{Nil} + \{0, 1\} \ldots]$, which describes all lists that contain any number of 0's and 1's in any order (note that the concretization $\gamma(l_4) = \{[], [0], [1], [0, 0], [0, 1], [1, 0], ...\}$ is now actually infinite). On the other hand, the list $l_5 = [\{0\}, \textsc{Top}]$ describes any list that starts with a 0. We can now use the meet operation on l_4 and l_5 to obtain a representation of any list that only contains the elements 0 and 1 *and* starts with a 0, as follows: $l_4 \sqcap l_5 = [\{0\}, \textsc{Nil} + \{0, 1\} \ldots]$.

Furthermore, it might be noted that $l_4 \sqcap l_1 = [\textsc{Bottom}]$, since l_1 contains the element $\{2\}$, which is not allowed by l_4 (since $\{2\} \sqcap \{0, 1\} = \emptyset$).

We do not elaborate on the details of how the lattice operations (i.e. meet, join and comparison) are implemented, besides the remark that they recursively traverse the list and make use of the corresponding operations of the underlying element type T. Apart from these basic lattice operations, we provide implementations for other list operations such as concatenation (the ++ operator), map and filter. Listing 4.13 shows the implementation of the concatenation operation for ZList, which is also used by the flatMapWithIndex domain operation.

```
def ++(other: ZList[T])(implicit lat: Lattice[T]): ZList[T] = (this, other) match {
  case (ZBottom(), _) | (_, ZBottom()) => ZBottom()
  case (ZTop(), _) => ZTop()
  case (ZUnknownLength(elem), _) => ZUnknownLength(lat.join(elem, other.joinInner))
  case (ZCons(first, rest), _) => ZCons(first, (rest ++ other).asInstanceOf[ZListElement[T]])
  case (ZMaybeNil(first, rest), _) => other | (ZCons(first, rest) ++ other)
  case (ZNil(), _) => other
}

def joinInner(implicit lat: Lattice[T]): T = this match {
  case ZTop() => lat.top
  case ZUnknownLength(elems) => elems
  case ZBottom() => lat.bottom
  case ZCons(first, rest) => lat.join(first, rest.joinInner)
  case ZMaybeNil(first, rest) => lat.join(first, rest.joinInner)
  case ZNil() => lat.bottom
}
```

Listing 4.13: Concatenation of ZLists

There one can see that the result is ZBottom when at least one of the operands is ZBottom. When the left operand is ZTop, the result is ZTop as well, since ZTop says that the remaining list content could be anything, which is still true after appending some other list. Similarly, when the left operand is ZUnknownLength, the result is ZUnknownLength, but in this case there is more information about what elements are in that list: It is the join of all elements in the left and all elements in the right operand. Since the right operand can be any subtype of ZList, the joinInner auxiliary method is defined to extract the join of all elements *within* the list. This auxiliary method is also utilized by other ZList operations.

Appending to ZCons or ZNil is straightforward, and appending to ZMaybeNil essentially calculates both of these results (i.e. appending once as if the remaining list was empty and once as if the remaining list was not empty) and joins them (the join operation in ZList is implemented as the '|' operator).

Another thing that is apparent from Listing 4.13 is the usage of the Lattice[T] type class.

4.4.3 Abstract Nodes

Recall that abstract nodes are modeled as a combination of abstract paths and abstract subtrees. Subtrees, as defined in Listing 4.14, are in turn a combination of

- an abstract label, which we represent as a set of NodeDescriptors,
- an abstract list of attributes, which we represent as an abstract list of attribute labels, and
- an abstract list of children, each represented as an abstract subtree, thus making the Subtree structure recursive.

The representation of abstract labels is similar to that used in abstract paths (cf. PathStepDescriptor in Listing 4.11). As for PathStepDescriptor, we associate a partial ordering with node descriptors, in such a way that e.g. AnyAttribute $>$ NamedAttribute(name) $>$ Attribute(name, value) for any name and value. We then obtain a lattice by looking at *sets* of descriptors, which must be *normalized* in

```
case class Subtree(desc: Set[NodeDescriptor],
  attributes: ZList[Set[NodeDescriptor]],
  children: ZList[Subtree]) { /* ... */ }

abstract class NodeDescriptor
case object Root extends NodeDescriptor
case class Element(name: String) extends NodeDescriptor
case object AnyElement extends NodeDescriptor
case class Attribute(name: String, value: String) extends NodeDescriptor
case class NamedAttribute(name: String) extends NodeDescriptor
case object AnyAttribute extends NodeDescriptor
case class Text(value: String) extends NodeDescriptor
case object AnyText extends NodeDescriptor
case class Comment(value: String) extends NodeDescriptor
case object AnyComment extends NodeDescriptor
```

Listing 4.14: Types for representing abstract subtrees

the sense that no descriptor in the set is less than another descriptor in the same set. For example, the set {`Root`, `Element("a")`, `AnyText`}, when used as an abstract label, describes any node that is either a root node, an element named ‘a’, or a text node (with arbitrary content).

The handling of attributes is complicated by the fact that the list of attributes must not contain multiple attributes with the same name. To bypass the complications of implementing this explicitly (which is especially problematic when the name of an attribute is unknown, i.e. `AnyAttribute`), we use the `ZUnknownLength` constructor instead whenever we are dealing with attribute lists, since concatenating two such lists automatically joins the attribute descriptors. This is again an over-approximation because we can only express that an attribute *might* be contained in the list, not that an attribute *must* be contained in the list (since γ(`ZUnknownLength(...)`) always contains the empty list).

Listing 4.15 (where `latD` is an implementation of `Lattice[Set[NodeDescriptor]]`) shows how the implementation of the `Lattice` type class for subtrees is composed of its constituents: The join and meet operations delegate to the corresponding operations for each component, and an instance of `Subtree` is less than or equal to another instance if that relationship holds for each component individually.

```
implicit object SubtreeLattice extends Lattice[Subtree] {
  override def top = Subtree(latD.top, ZUnknownLength(Set(AnyAttribute)), ZTop())
  override def bottom = Subtree(latD.bottom, ZBottom(), ZBottom())
  override def join(left: Subtree, right: Subtree): Subtree =
    Subtree(latD.join(left.desc, right.desc),
      left.attributes | right.attributes,
      left.children | right.children)
  override def meet(left: Subtree, right: Subtree): Subtree =
    Subtree(latD.meet(left.desc, right.desc),
      left.attributes & right.attributes,
      left.children & right.children)
  override def lessThanOrEqual(left: Subtree, right: Subtree): Boolean =
    latD.lessThanOrEqual(left.desc, right.desc) &&
      left.attributes <= right.attributes &&
      left.children <= right.children
}
```

Listing 4.15: Definition of `SubtreeLattice` (instance of the `Lattice` type class)

In a similar way, the lattices for subtrees and for paths are eventually combined to form the lattice for abstract nodes, `N`. Since we can use this `N` as the element type in `ZList`, the type for abstract node lists `L` is just `ZList[N]`.

We will not go into the details of how individual operations on abstract nodes are implemented in the Zipper Domain, but two particularities are worth mentioning:

Firstly, we try to ensure that abstract nodes are always *normalized*. In this context, we say that a node is *normalized* if its components do not contradict each other with regard to what nodes they describe. For example, the subtree component of a node could be $\bot$ while the path component is not. As soon as one component is $\bot$, however, there is no concrete node that is represented by such an abstract node, and therefore the whole structure is effectively $\bot$. When the node is normalized, both components will be $\bot$ in this case. Furthermore, a node could be represented by a subtree that only has an element label, and a path that only has an attribute node label. Such a node does not exist either, so the whole structure should be $\bot$ in this case as well. By normalizing nodes, i.e. removing incompatible combinations from each component, we allow operations to only look at individual components (i.e. either the subtree or the path) and still work correctly.

Given a function `normalize` that performs this normalization, we can also create an abstract node from only one component by using $\top$ for the other component: $n =$ `normalize`$((\top,$`/foo/bar`$))$, for example, is the abstract node that represents any element named '`bar`' that is a child of '`foo`', which is in turn a child of the root node. Normalization now ensures that the label within the subtree component of n only contains the `Element("bar")` descriptor.[3]

Secondly, we did not implement all operations with the maximum possible precision that our representation would allow. This is mostly due to complex interactions between the `ZList` type and text node contents. In particular, the merging of text nodes within arbitrary instances of `ZList[N]` returns `ZUnknownLength` in our implementation in some cases where it could be more precise. Since the concept of abstract interpretation is an over-approximation already, this is not too much of an issue as long as correctness is preserved (i.e. it would not be an option to not merge text nodes at all). The results of our evaluation presented in the next chapter will confirm this.

4.5 Recursive Templates and Termination

The attentive reader will have noticed that we discussed the necessity to ensure termination in the introductory section on abstract interpretation, but did not yet deal with this issue in detail as far as our abstract interpreter is concerned.

While unbounded loops, which pose one possible source of non-termination of an analysis, are not expressible in XSLT (the `<xsl:for-each>` instruction is not a loop in this sense), there are two ways in which template instantiations can be recursive and possibly non-terminating:

- Named templates can be called recursively, meaning that a template calls itself (or another template that later calls the original one) using `<xsl:call-template>`, usually with different parameters than before and only if a certain condition is fulfilled. This is analogous to recursive functions in other programming languages.

- Matchable templates might be applied recursively when they use `<xsl:apply-templates>` with a `select`-clause that selects nodes which are in turn matched by the same template. This is unique to XSLT (and possibly other transformation languages that adhere to the same rule-matching semantics).

An example of the latter case is the following built-in template rule, which is implicitly added to every stylesheet and matches the root node as well as any element node, if not overridden by other user-defined rules:

[3] The redundancy between subtrees and paths, as far as the label is concerned, is an effect of the recursive nature of how subtrees and paths are represented, since both representations start with the node in question. The same issue exists with the original *Zipper* structure when extended to labeled trees.

```
<xsl:template match="*|/">
  <xsl:apply-templates/>
</xsl:template>
```

Since the template application instruction misses a `select`-clause, the default selector is used which selects all children. If one or more of those child nodes are not matched by another template rule, the same built-in template is instantiated again, leading to recursion whose depth depends on the input tree.

The issue is not limited to built-in templates as the following example illustrates:

```
<xsl:template match="a">
  <b><xsl:apply-templates select="a"/></b>
</xsl:template>
```

This template rule transforms a tree of <a> elements into a tree of <b> elements. Assuming that this is the only template in a stylesheet (apart from the built-in rule above, which provides the entry point for the transformation by matching the root node), the result will be a tree consisting solely of <b> elements. This, however, can not be expressed in any of our domain implementations, since they do not support recursive structures (i.e. defining the children of an element in terms of the element itself). If they did, the least fixed point would be a tree of <b> elements of unknown depth, but in our domains this has to be approximated at some point using $\top$.

In order to do this, the abstract interpreter has to recognize recursive template instantiations. This could be done by inspecting the stack of previous context nodes for multiple instantiations of the same template, and comparing the nodes in this stack w.r.t. their precision: As long as further recursive calls lead to a gain of precision in the result, the abstract interpreter should continue. However, due to descending one level further in the tree leads to incomparable abstract paths in the Zipper Domain (consider the paths `/a/a` and `/a/a/a`, for example), this approach does not work without employing more advanced techniques such as widening [8], which exceed the scope of our work.

For these reasons, we decided to introduce the recursion limit already shown in Section 4.3, which decrements a user-defined counter value whenever a template is instantiated. This approach works for both named and matchable templates. The question of how different values for the recursion limit influence the result and performance of the analysis will be discussed in the evaluation presented in the next chapter.

5 Evaluation

This chapter presents an evaluation of our implementations of both the concrete and the abstract interpreter. We first show how the results of our interpreters compare to the ones of the XSLT processor implemented in the Java standard library. We then evaluate the extent to which our abstract interpreter is applicable to a collection of real-world XSLT transformations from various sources and show some results. The source code of our implementation as well as the collection of stylesheets that were used in this evaluation can be found on GitHub at `https://github.com/Boddlnagg/analyzexslt`.

5.1 Reference Tests

Using ScalaTest, we wrote a test suite that tests whether the results of our interpreters match the results of the XSLT processor from the `javax.xml.transform` package in the Java standard library, which is an implementation of the XSLT 1.0 specification. Each test consists of an XSLT stylesheet definition and an input XML document. The Java XSLT processor is used to generate a reference result, which is then compared to the result of our implementation.

The suite consists of 52 test cases and covers every supported XSLT instruction as well as various corner cases of the specification, e.g. empty text nodes. Some tests also test for invalid inputs, meaning that the reference processor will throw an exception. In these cases, our implementation must also throw an exception or return $\perp$. For example, the transformation in one test case produces multiple result elements on the top level of the output document, which is not allowed.

The complete reference test suite is run once for each of our interpreter implementations, and each implementation passes all tests:

- It is run once for the **concrete interpreter**, and the results must match exactly in order for the tests to pass. In order to be able to compare the results, each reference result is parsed into our own XML representation (cf. Section 3.1).

- It is also run once for the **abstract interpreter using the Powerset Domain** (using the typed version of the XPath subdomain, `TypedPowersetXPathDomain`). The input is lifted into the domain by wrapping it in a singleton set. The abstract interpreter runs on that input, and the result must be another singleton set whose only element must match the reference result in order to pass a test. The implementation would still be correct if the abstract result contained more than one concrete document, but the Powerset Domain allows for maximal precision given a concrete input.

- Finally it is run for the **abstract interpreter using the Zipper Domain**. The input is lifted into the domain using the `lift...` and `create...` domain operations. The reference result is lifted into the domain as well and compared to the actual result w.r.t. the order in the lattice. We can not guarantee maximal precision in this domain, therefore in order for a test to pass, the actual result (r_{actual}) must be a correct over-approximation of the (lifted) reference result (r_{ref}), i.e. $r_{actual} \geq r_{ref}$.

Besides the reference tests, there are other test suites covering individual areas of our implementation, e.g. the operations in `ZList`, the parsing and evaluation of XPath expressions, and the abstract location path pattern matching algorithm.

Another test suite runs the abstract interpreter on some simple stylesheets using the Zipper Domain and $\top$ as the input. One of those is the CD catalog stylesheet we presented in the introduction (Listing 2.2). The result of this is shown in Figure 5.1 next to the concrete result that was already shown in the introduction.

```
<html>
  <body>
    <h2>
      <![TEXT[Available CDs]]>
    </h2>
    <ul>
      NIL + <li>
        <b>
          NIL + <![ANY-TEXT]>
        </b>
        <![ANY-TEXT]>
      </li>
      ...
    </ul>
  </body>
</html>
```

(a) Abstract result, from input ⊤

```
<html>
  <body>
    <h2>Available CDs</h2>
    <ul>
      <li>
        <b>Still got the blues</b> by Gary Moore
      </li>
      <li>
        <b>Eros</b> by Eros Ramazotti
      </li>
      <li>
        <b>One night only</b> by Bee Gees
      </li>
    </ul>
  </body>
</html>
```

(b) Concrete result, same as Listing 2.3

Figure 5.1: Comparison of abstract and concrete result of the transformation in Listing 2.2

A few words must be given to the notation used here, since it is also used in the next section:[1] It is a mix of the XML format and the ZList notation introduced in Section 4.4.2. Hence, `NIL` means that the list of child nodes might end at that position, and '`...`' means that the previous node is repeated zero or more times.

In order to distinguish text nodes from the ZList markers (such as `NIL`) and from the whitespace used for indentation (which would actually be part of the document tree in real XML documents), we use the `<![TEXT[...]]>` syntax, which is borrowed from the `CDATA` syntax found in XML. A similar syntax is used for comment nodes.

Further examples in the next section will also show that node labels (such as `html`) are sets. We could have also written `<{html}>`, but the curly braces are omitted for singleton sets in order to resemble the XML format more closely.

The abstract result shows that our implementation statically (i.e. without concrete input) determines the structure of the result document. One can see that the outer elements (`html`, `body`, `h2`, `ul`) do not depend on the input, and that the `<ul>` element contains an unknown number of `<li>` elements, each containing a `<b>` element and some text. The content of the `<b>` element might be empty, because the source text where this content is taken from might be empty, but `<![ANY-TEXT]>` means that there is *some* text node, which must not be empty.

5.2 Real-World XSLT Transformations

The previous section showed that our implementation achieves reasonable results for simple examples. In order to find out how it works for more complex and realistic XSLT stylesheets, we gathered a collection of 16 XSLT stylesheets from various sources on the internet.[2]

In a first step, we then analyzed the features used in those stylesheets to evaluate the implications of our decision to not implement the full feature set of XSLT. To do this, we used the XSLT and XPath parsers that we had implemented and added the `XSLTFeatureAnalyzer` class that traverses the respective ab-

[1] We have implemented a pretty printing function that generates this output from abstract nodes in the Zipper Domain. By default, however, abstract nodes are printed using another representation that corresponds directly to the *internal* representation, with added indentation of ZList elements for readability.

[2] These files as well as information about their sources and abstract interpretation results can be found in the `xslt-collection` directory within the Git repository.

		00-musicxml	01-libreoffice-spreadsheetml	02-svg-example	03-nunitreport	04-sick-of-beige-svg	05-pocket2keepass	06-docbook-html5	07-ead2002-to-wordml	08-epub-nav	09-zefania2html	10-dc2mods	11-sphinxxml2review	12-umbraco-mediahelper	13-brainfuck	14-iati-activities-rdf	15-github2html
Output Method		xml	xml	html	html	xml	xml	default[a]	xml	default[a]	html	xml	text	xml	xml	xml	xml
Namespaces		–	○	○	–	○	–	–	○	–	–	○	–	–	–	○	–
`<xsl:import>/<xsl:include>`		–/–	○/–	○/–	–/–	–/–	–/–	–/○	–/–	–/–	–/–	–/○	–/–	–/–	–/–	–/–	–/–
Top-Level Variables/Parameters		–/–	●/●	–/●	–/–	●/●	–/–	–/●	–/–	–/–	●/–	●/–	–/–	–/–	●/–	–/–	–/–
Other Top-Level Elements (`<xsl:...>`), *unsupported*		–	key	–	–	–	–	–	strip-space	–	–	–	–	–	–	–	–
Named Templates (`<xsl:call-template>`)		–	●	●	–	●	–	●	●	–	–	●	–	–	●	–	–
Template Modes		–	●	●	●	–	–	●	–	–	●	–	–	●	–	–	–
Variables/Template Parameters		●/–	●/●	●/●	–/–	●/●	–/–	●/●	–/–	–/–	●/–	●/●	–/–	●/●	●/●	–/–	–/–
Result Tree Fragments		●	●	●	–	–	–	●	–	–	●	●	–	●	●	–	–
Literal Text/`<xsl:text>`		–/–	●/●	●/–	●/–	●/–	–/–	●/●	●/●	●/–	●/●	–/●	●/●	–/–	–/–	–/–	●/–
Literal Elements/`<xsl:element>`		–/●	●/●	●/–	●/–	●/–	–/●	●/●	●/–	●/–	●/–	●/–	–/–	●/–	–/–	●/–	●/●
`<xsl:attribute>`		●	●	–	–	●	–	●	–	–	–	●	–	●	–	–	●
Attribute Value Templates		–	●	●	●	●	●	●	–	–	●	–	–	●	–	●	●
`<xsl:processing-instruction>`		–	–	○	–	–	–	–	○	–	–	–	–	–	–	–	–
Conditionals (`<xsl:if>/<xsl:choose>`)		●/–	●/●	●/–	–/–	●/–	–/–	●/●	●/●	●/●	–/–	●/●	–/●	●/●	–/●	●/–	–/–
`<xsl:for-each>`		●	●	●	–	–	–	●	●	●	–	●	●	–	–	●	●
`<xsl:value-of>/<xsl:copy-of>/<xsl:copy>`		●/●/●	●/●/–	●/–/–	●/–/–	●/–/●	●/–/–	●/●/–	●/–/–	●/–/–	●/●/–	●/–/–	●/–/–	●/–/–	●/●/●	●/–/–	●/●/–
Other Instructions (`<xsl:...>`), *unsupported*		–	message	–	–	–	–	message	sort	number	–	–	–	–	–	–	–
Union Operator in Selectors/Patterns		●/●	●/●	–/–	–/–	●/●	●/●	●/●	●/●	–/–	–/–	●/–	–/–	–/–	–/–	–/–	●/●
Max. Number of Steps in Patterns		1	2	1	1	1	1	3	5	1	1	1	5	1	1	0[b]	2
Predicates in Patterns	Literal position	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
	Attribute literal value	–	–	–	○	–	–	–	–	–	–	–	–	–	–	–	○
	Arbitrary expression	–	–	–	–	–	–	–	–	–	–	○	–	○	–	–	○
Patterns with ‘//’ Step (trivial/non-trivial)		–/–	–/–	–/–	–/–	–/–	–/–	●/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	–/–	●/–
Predicates in Selectors	Literal position	○	○	–	–	–	–	–	○	–	–	–	–	–	–	–	–
	Attribute literal value	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	○
	Arbitrary expression	○	○	–	–	–	–	–	–	○	–	○	○	○	–	–	–
Selectors with Absolute Path		–	●	–	–	–	–	●	–	●	●	–	–	–	–	●	●
Axes used in Selectors	`child/attribute/parent`	●/●/●	●/●/●	●/●/–	●/●/–	●/●/–	●/●/–	●/●/●	●/●/●	●/–/–	●/●/●	●/–/●	●/●/●	●/●/–	●/●/–	●/●/●	●/●/–
	`self/descendant/ descendant-or-self`	●/–/–	●/●/●	●/–/–	–/–/–	–/–/–	–/–/–	●/–/●	●/–/–	–/–/–	●/–/–	●/–/–	●/–/–	●/●/–	–/–/–	●/–/●	●/–/–
	Others *(unsupported)*	–	5 more	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Arithmetic/Boolean XPath Operators		–/–	●/●	●/–	–/–	●/–	–/–	–/●	–/●	–/●	–/–	–/●	–/–	–/●	–/●	–/–	–/●
Simple XPath Functions	Conversions	–	string number	number	–	string number	–	–	string	–	–	string	–	–	–	–	–
	Booleans	–	true false not	–	–	–	–	–	–	–	–	not	not	not	not	–	–
	Others	–	name count position concat	position	–	concat	–	local-name count position last	position last	–	–	concat	name count	concat	–	concat	local-name concat
Other XPath Functions *(unsupported)*	Strings/Math/Others	–/–/–	8/3/4	–/–/–	–/–/–	1/–/–	1/–/–	1/–/2	–/–/–	–/–/–	–/–/–	5/–/–	1/–/–	2/–/2	3/–/–	2/–/–	1/–/–
	Extension Functions	–	○	–	–	–	–	–	–	–	–	–	–	○	–	–	–

[a] Either ‘xml’ or ‘html’ will be selected depending on the result of the transformation (cf. [3, Section 16]).

[b] The only pattern used here is the root pattern ‘/’, which has no steps.

●	Feature used and supported in our implementation
○	Feature used but not supported
–	Feature not used

Table 5.1: Features used in a collection of real-world XSLT stylesheets

stract syntax trees and extracts a set of used features. To work around the fact that we do not support importing stylesheets from external files, we copied the contents of imported stylesheets into the main file.

We have compiled the results of this feature analysis in Table 5.1 in a way that additionally shows which of the features are supported by our implementation. Several items in this table are noteworthy:

- Different stylesheets use fairly different sets of features. We were unabled to indentify a common “core” feature set.
- Several features, some of which are not listed in the table, were not used in any of the analyzed transformations. Among those are named attribute sets [3, Section 7.1.4], custom template priorities, the `<xsl:comment>` instruction, and, most notably, non-trivial descendant steps in patterns. Only one of the transformations, namely ‘01-libreoffice-spreadsheetml’, which is an import filter for Excel files in LibreOffice, uses more than the six XPath axes that we support.
- Most stylesheets that use predicates use them for arbitrary expressions. This means that even if we did implement support for only a subset of predicate expressions, such as literal position predicates (specifying that a node must be the n-th element in the current node list) or literal attribute value predicates (specifying that the value of an attribute must match a given string literal), these stylesheets would still remain unsupported. Furthermore, many stylesheets do not use predicates at all.
- Most of the XPath functions that we do not support are string functions (e.g. `substring`, `contains`, etc.), each of which would require an additional domain operation. For some of those operations our abstract interpreter provides stubs, so that the function can be evaluated but the result will be `topString`, i.e. an unknown string value.

In the next step, we analyzed the collected real-world stylesheets using our abstract interpreter with the Zipper Domain. In order to also analyze some of the stylesheets that use features which are not supported by our implementation, we reduced them by removing usages of unsupported features while (mostly) preserving the original semantics. For example, the usage of namespaces could be removed in most cases by simply using unprefixed names for output elements. However, we did not reduce *all* stylesheets, because some were simply too large (e.g. ‘01-libreoffice-spreadsheetml’ is 434 KB) or used unsupported XPath functions to such an extent that any reduction would alter the semantics completely (e.g. ‘13-brainfuck’ is a Brainfuck interpreter implemented in XSLT that makes heavy use of various string operations; ‘12-umbraco-mediahelper’ uses custom extension functions that are only available in conjunction with the Umbraco CMS).

For lack of concrete input documents, we were not able to compare the abstract results to concrete results, but the following excerpts show that our abstract interpreter produces sensible results nonetheless.

Generally, the best results could be achieved for stylesheets which can be categorized as “Fill-in-the-blanks stylesheets” according to M. Kay [2, p. 533]. These stylesheets usually consist of only one template rule for the root node, which establishes the overall structure of the result document and has “blanks” at certain places, which are filled according to data from the input document. From our collection, stylesheets 04 and 14 fall into this category and therefore give good results even without a recursion limit.

In ‘07-ead2002-to-wordml’, the header of the resulting WordML document is constant, i.e. predefined in the stylesheet itself using literal result elements, but the body is dynamic, resulting in a “hole” in the analysis result, as shown in Listing 5.1.

The content of this hole is essentially $\top$ (‘*’ designates any element node in our notation) because the stylesheet uses `<xsl:apply-templates/>` to generate the contents of the `<body>` element, which can trigger the built-in templates. Therefore, a recursion limit was required to terminate the analysis.

```
<wordDocument embeddedObjPresent="no"? ocxPresent="no"? macrosPresent="no"?>
  <fonts>
    <defaultFonts fareast="Times New Roman"? h-ansi="Times New Roman"? cs="Times New Roman"? ascii="
         Times New Roman"? />
  </fonts>
  <!-- ... rest of the constant header omitted in this listing ... -->
  <body any-attributes>
    NIL + <{*,![ANY-TEXT],![ANY-COMMENT]} any-attributes>TOP</*>
    ...
  </body>
</wordDocument>
```

Listing 5.1: Result of analyzing '07-ead2002-to-wordml'

Interestingly, disabling built-in templates (recall that we added an option to do this, cf. Section 3.3) in this case yielded the same results, indicating that the stylesheet contains the possibility of unbounded recursive template application even without built-in templates.

The influence of built-in templates was more directly noticeable in other stylesheets. For example, analyzing '02-svg-example' results in $\top$ when built-in templates are enabled, but gives meaningful results without them. However, we can not be sure if the stylesheet actually relies on built-in templates, so disabling them might lead to incorrect results.

The analysis of some stylesheets, such as 03, 05, 08 and 09 from our collection, resulted in $\perp$ when built-in templates were disabled. In these cases, we *can* be sure that built-in templates are required for correctness. For 08 and 09, slightly modifying the stylesheets by adding a new template rule for the root node eliminated the need for the recursive built-in template rule, in such a way that good results could be achieved even with built-in templates enabled and without a recursion limit. The result of the latter is shown in Listing 5.2, which nicely illustrates what can be modeled in the Zipper Domain.

```
<html DIR="LTR"?>
  <head>
    <title>
      NIL + <![ANY-TEXT]>
    </title>
    <link rel="SHORTCUT ICON"? href="../images/cs_icon.ico"? />
    <script src="../js/zefania.js"? type="text/javascript"? />
    <link rel="stylesheet"? type="text/css"? href="../css/zefania-ltr.css"? />
  </head>
  <body onLoad="javascript:toggledivchap()"?>
    <!-- ... some <div> elements omitted in this listing ... -->
    NIL + <div class={"chapterbody1","searchresults","chapterbody2","floatright","floatleft"}? id=*?>
      <{ul,h1,p} class="treeview"? id={"menus_book1","menus_book2"}?>
        NIL + <{li,![ANY-TEXT]}>
          NIL + <a href="javascript:;"? onclick=*?>
            NIL + <![ANY-TEXT]>
          </a>
        </li>
        ...
      </{ul,h1,p}>
      NIL + <{script,p} type="text/javascript"?>
        NIL + <{sup,![ANY-TEXT]}>
          NIL + <![TEXT[ ]]>
          <font color="#cccccc"?>
            NIL + <![ANY-TEXT]>
          </font>
          <![TEXT[ ]]>
        </sup>
        ...
      </{script,p}>
    </div>
    ...
  </body>
</html>
```

Listing 5.2: Result of analyzing '09-zefania2html'

For example, the node in Line 13 is either an `<ul>`, an `<h1>` or a `<p>` element, which might have the attribute `class="treeview"` and also might have an `id` attribute with one of the two values `"menus_book1"` or `"menus_book2"`. The content of this node is a (possibly empty) combination of `<li>` elements and text nodes.

It might be noted that apart from the addition of a template rule for handling the root node, we did not have to modify or reduce this stylesheet in order to obtain the result above, since it does not use unsupported features.

Finally, we have measured the performance of our analysis for three of the stylesheets in our collection with varying recursion limits. Additionally, we compared the results to a version of our implementation without the performance optimization described in Section 4.3, showing that this is indeed an optimization.

The benchmarks were run using Scala 2.11.4 (Java HotSpot 64-Bit Server VM, Java 1.8.0_25) on an Intel Core i7-2630QM CPU (2.00 GHz) and 8 GB RAM. Each analysis was run 7 times for each recursion limit up to a maximum of 10, as long as it took less than 120 seconds. The first 2 runs were then discarded to account for JIT warmup, and the final result is the arithmetic mean of the remaining 5 runs.[3] Furthermore, built-in templates were disabled for the stylesheet '02-svg-example', as explained above. The diagrams in Figure 5.2 show the results.

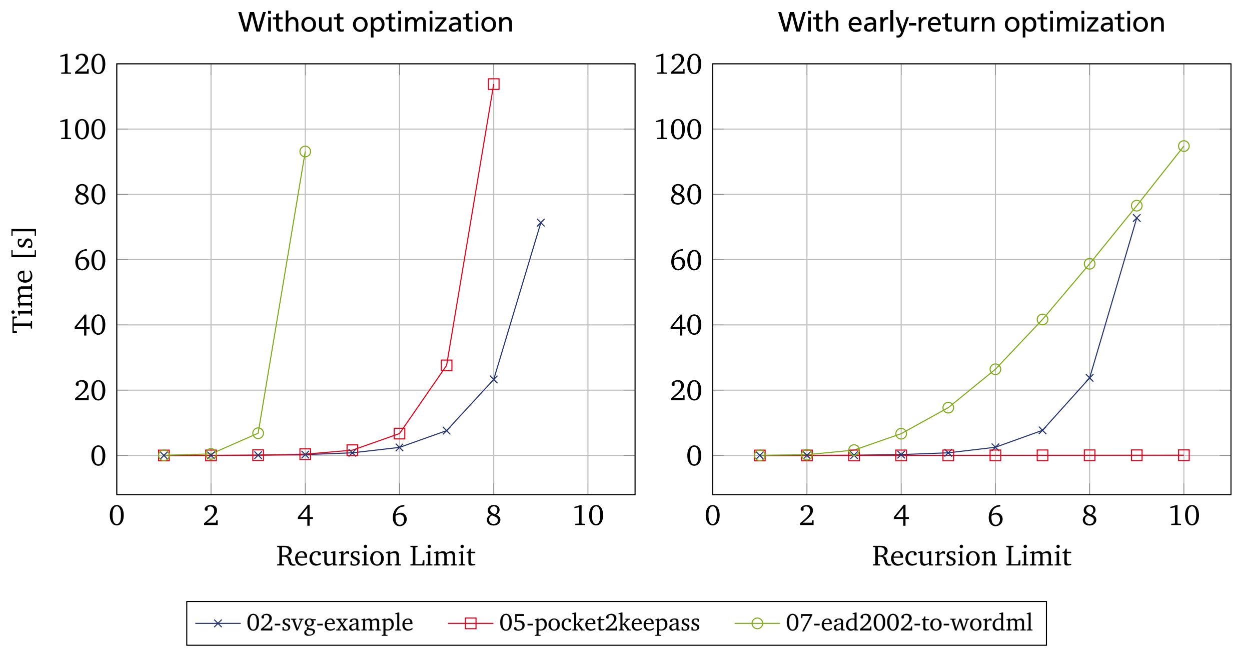

Figure 5.2: Performance of analysis with varying recursion limits

One can see that, without the performance optimization, the time required for an analysis increases exponentially with increasing recursion limit for all three stylesheets. This can be explained by the fact that abstract template matching usually results in a number of possibly matching templates, and each of those may in turn contain recursive template applications with multiple possibilities. Each of those

[3] The code used for these benchmarks as well as their exact outcomes are available in the branches 'bench' (with optimization) and 'bench-noopt' (without optimization) on GitHub.

possibilities has to be analyzed until the recursion limit is reached, leading to the exponential explosion visible in the diagram.

The early-return optimization improves the situation for stylesheets 05 and 07 and has no impact on stylesheet 02. This, as well as the vast performance increase for stylesheet 05, is likely due to the influence of the built-in template rules, which are disabled in stylesheet 02, but are apparently triggered immediately in stylesheet 05 (recall that the optimization prefers template rules with low priority, therefore built-in templates – if applicable at all – are evaluated first, and no further templates are evaluated as soon as $\top$ is returned). This assumption is supported by the fact that the result of analyzing stylesheet 05 is $\top$ (the final analysis results are, of course, independent of the optimization).

Furthermore, it is noteworthy that the results of stylesheet 02 become more precise as the recursion limit is increased, whereas the results of stylesheet 07 do not differ at all for the tested limits up to 10 and always contain the static header and the "hole" as shown in Listing 5.1.

For comparison, we also measured the time required to analyze stylesheet 09, which did not need any recursion limit: The analysis, of which the result has already been shown in Listing 5.2, took 0.022 seconds, regardless of the early-return optimization.

In conclusion, the preceding paragraphs have shown that both the quality (i.e. precision) of the results as well as the performance of the analysis vary greatly depending on the given stylesheet. It could also be observed that the Zipper Domain is able to express important properties such as restrictions of element names and attribute values to a given set of possibilities, optional nodes, and repetitions. However, our approach suffers from an inability to cope well with structural recursion in XML documents.

6 Related Work

We are not aware of any prior work using abstract interpretation for the analysis of XSLT transformations, but there is a lot of work in the area of static analysis of XSLT and XML transformations in general. We do not claim that the following list is complete, rather we want to give an overview over other approaches and related technologies.

Static Validation of XSL Transformations

A. Møller et al. [9] present a static analysis that is based on DTDs and handles the complete XSLT 1.0 language. Their analysis is able to statically verify whether the result of a transformation, given any input document which is valid according to a specified input DTD, is again a valid document relative to a specified output DTD. Since DTDs can not express certain criteria, such as the content of text nodes, they apply an excessive preprocessing step which simplifies the given stylesheet by introducing various approximations. Then, they present a fixed point algorithm to compute the flow of template invocations and construct an XML graph which is eventually validated relative to the output DTD.

Both the flow analysis and the XML graph are able to deal with structural recursion in a way that our naive, generic abstract interpretation approach is not.

They also provide statistical data on `select` and `match` expressions used in real-world XSLT stylesheets, which is more exhaustive than what we were able to collect in this area, but basically confirms that a vast majority of stylesheets use no descendant steps within patterns and no XPath predicates. Since they note that test cases of stylesheets together with input and output DTDs are "remarkably difficult to obtain" [9], it can be regarded as an advantage of our approach that it does not need to know about any externally provided schema definitions.

XSLT 2.0/3.0

We have already mentioned that more recent versions of the XSLT standard exist [10, 11]. The most interesting addition compared to XSLT 1.0 in the context of this work is the option for XSLT processors to be *schema-aware*. This means that a processor can make use of type information in the form of an XML Schema Definition (XSD). When such a processor evaluates a stylesheet, it may use schema definitions to validate either the input document or the output document, or both, on the fly. The specification, however, does not prescribe any static guarantees, rather "it is implementation-defined whether type errors are signaled statically" [10, Section 2.9].

A master's thesis by S. Kuula [12] extends the aforementioned approach by A. Møller et al. [9] to work with XSLT 2.0 and XSDs, which are much more expressive than DTDs. Kuula claims that his approach is fast enough for real-time analysis even for large stylesheets and schemas, but admits that his over-approximations still lead to spurious errors, i.e. cases where his analysis can not prove that the output is always valid relative to the target schema even though it is.

The main additions of XSLT 3.0, which is still in Working Draft status at the time of this writing, are higher-order functions and better support for streaming. We are not aware of any research in the area of statically analyzing XSLT 3.0 stylesheets.

A System for the Static Analysis of XPath

P. Genevès et al. [13] present a static analysis of XPath based on the propositional μ-calculus, which is a propositional modal logic extended with least and greatest fixpoint operators. They are able to analyze a fragment of XPath – which is restricted to location paths and only works with element nodes, but allows all relevant axes (upward and downward) as well as predicates, union, intersection and negation – by translating given XPath expressions into formulae in this logic. Then, they present an implementation of a solver which can automatically check these formulae for satisfiability, in the sense that a formula is satisfiable if the expression might return a nonempty set of nodes. Morever, they show how other XML decision problems, such as XPath containment (i.e. whether the result of one XPath expression is a subset of the result of another) and equivalence (i.e. whether two XPath expressions always have the same result) can be reduced to satisfiability.

By also presenting a possibility to lift DTDs into the μ-calculus, their approach allows to check for satisfiability in restriction to a given DTD. Since XPath location paths are a fundamental component of XSLT transformations, the static analysis of such transformations is mentioned as a use case for their work, though they do not expand on this topic.

ℂDuce: An XML-Centric General-Purpose Language

Benzaken et al. introduce ℂDuce [14], a functional programming language designed for working with XML documents. It can be used to write XML transformations similar to what is possible with XSLT, but ℂDuce's rich static type system allows these transformations to be statically analyzed in a much more direct and comprehensive way. ℂDuce features ML-style pattern matching, which, in combination with function overloading, can be used in a way similar to XSLT template rule matching, but is more versatile, since patterns may be recursive and can contain regular expressions over sequences of XML nodes and text characters. Since types can also be recursive, structural recursion in XML documents can be encoded in the type system in a natural way.

As we faced the problem of merging adjacent segments of text in XML documents, it is noteworthy that ℂDuce addresses this issue by always modeling strings as sequences of single characters instead of using a distinct string or text node type. This way, for example the sequences [‘a’, ‘b’, ‘c’], [‘ab’, ‘c’] and [‘abc’] are all implicitly equivalent. Furthermore, by using regular expressions over such character sequences, values of text nodes can be specified in a much more fine-grained way than what is possible in DTDs or in our implementation.

Benzaken et al. also present an implementation of a run-time system that executes ℂDuce programs, which is based on deterministic tree automata and uses the available static type information to optimize execution. Moreover, they mention the possibility of importing XML Schema definitions into the ℂDuce type system, so that ℂDuce programs can be validated relative to existing schemas.

List of Figures

List of Listings

Bibliography

[1] O'Neil Delpratt. XML on the Web: Is it still relevant? In *XML London 2013 – Conference Proceedings*, 2013.

[2] Michael Kay. *XSLT – Programmer's Reference*. Wrox Press, 2000.

[3] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 1.0 – W3C Recommendation. http://www.w3.org/TR/xslt.html, November 1999. accessed 2015-01-02.

[4] World Wide Web Consortium (W3C). XML Path Language (XPath) Version 1.0 – W3C Recommendation. http://www.w3.org/TR/xpath/, November 1999. accessed 2015-01-02.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[6] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[7] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, September 1997.

[8] Agostino Cortesi. Widening operators for abstract interpretation. In *Sixth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2008, Cape Town, South Africa, 10-14 November 2008*, pages 31–40, 2008.

[9] Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations. *ACM Transactions on Programming Languages and Systems*, 29(4), July 2007.

[10] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 2.0 – W3C Recommendation. http://www.w3.org/TR/xslt20/, January 2007. accessed 2015-03-11.

[11] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 3.0 – W3C Last Call Working Draft. http://www.w3.org/TR/xslt-30/, October 2014. accessed 2015-03-11.

[12] Søren Kuula. Practical type-safe XSLT 2.0 stylesheet authoring. Master's thesis, Department of Computer Science, University of Aarhus, 2006.

[13] Pierre Genevès and Nabil Layaïda. A system for the static analysis of XPath. *ACM TOIS*, 24(4):475–502, 2006.

[14] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 51–63, New York, NY, USA, 2003. ACM.