Lepper, Markus; Trancón y Widemann, Baltasar:

A simple and efficient step towards type-correct XSLT transformations

Original published in: 26th International Conference on Rewriting Techniques and Applications / RTA 26, 2015 Warschau, Poland. - Saarbrücken/Wadern : Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. - (2015), p. 350-364. ISBN 978-3-939897-85-9 (Leibniz International Proceedings in Informatics (LIPIcs) ; 36)

Original published: June 2015

ISSN: 1868-8969

DOI: 10.4230/LIPIcs.RTA.2015.350

[*Visited:* 2020-03-02]

A Simple and Efficient Step Towards Type-Correct XSLT Transformations

Markus Lepper[1] and Baltasar Trancón y Widemann[2]

1 `<semantics/>` GmbH, Berlin, DE
2 Ilmenau University of Technology, Ilmenau, DE

Abstract

XSLT 1.0 is a standardized functional programming language and widely used for defining transformations on XML models and documents, in many areas of industry and publishing. The problem of *XSLT type checking* is to verify that a given transformation, when applied to an input which conforms to a given structure definition, e.g. an XML DTD, will always produce an output which adheres to a second structure definition. This problem is known to be undecidable for the full range of XSLT and document structure definition languages. Either one or both of them must be significantly restricted, or only approximations can be calculated. The algorithm presented here takes a different approach towards type correct XSLT transformations. It does not consider the type of the input document at all. Instead it parses the fragments of the result document contained *verbatim* in the transformation code and verifies that these can potentially appear in the result language, as defined by a given DTD. This is a kind of *abstract interpretation*, which can be executed on the fly and in linear time when parsing the XSLT program. Generated error messages are located accurately to a child subsequence of a single result element node. Apparently the method eliminates a considerable share of XSLT programming errors, on the same order of magnitude as a full fledged global control-flow analysis.

1998 ACM Subject Classification D.1.1 Functional Programming, D.3.2 Functional Language I.7.2 Scripting languages

Keywords and phrases XSLT, type checking, abstract interpretation

Digital Object Identifier 10.4230/LIPIcs.RTA.2015.350

1 Introduction

1.1 XSLT Transformations and Document Types

XSLT, in its different versions, is a standard transformation language for processing XML documents. There are different implementations of XSLT processors, employing various technologies. The contribution in this article is about XSLT 1.0. All versions of the language are Turing complete and fully functional programming languages. "Functional" in the most obvious sense means that there are no variables which can change state, but instead functions which can be applied to constant parameters and thereby yield a certain result. These function calls can be recursive. Functions are called "templates" in the context of the language specification.

An XSLT program is in most cases used to convert an XML document, serving as the *input*, into a second XML document serving as its *result*.[1] The templates which are applied to

[1] The conversion into unstructured, plain text, or into text structured by other means than XML is also possible but not covered by this article.

26th International Conference on Rewriting Techniques and Applications (RTA'15).
Editor: Maribel Fernández; pp. 350–364

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

certain elements of the input document are selected by a simple mechanism of pattern matching — either in a fully automated way by pre-defined standard rules, or semi-automatically after some explicit pre-selection by the author of the program. For writing meaningful programs, it is therefore a prerequisite that all input documents to a given transformation have certain invariant structural properties. This is *not* a technical necessity imposed by the language: the semantics of XSLT are rather "robust" and as long as no errors are raised *explicitly* by the programmer, arbitrary input will be transformed into some output. But in practice, the document type of the input document will be defined in some precise formalism anyhow, e.g. as RELAX NG grammar, W3C Schema, or W3C DTD.

In many cases it is required that the output of an XSLT transformation adheres also to such a precise document type. Of course this can always be checked a posteriori by *validating* the result of every transformation explicitly against this result document type. All cases where the result document violates the intended result document type are caused by programming errors in the transformation.

Obviously, it is highly desirable to find these programming errors earlier, when constructing the XSLT program. This is not only relevant for performance issues, since validating always implies total parsing, but also for increased reliability of services based on transformations: XML and XSLT processing are more and more applied to critical data, like business objects, physical real-time data, medical files, etc.

The general problem is that of *type checking*: an XSLT program is type correct, if and only if every input which adheres to a given input type will produce an output which adheres to a given output type. This problem has been proven to be intractable in the general case [9]. Furthermore, restrictions of the involved two languages (XSLT and document type definition) have been defined for which it is solvable, and the complexity of these problems has been thoroughly analyzed in the last two decades. For surveys see e.g. [11] and [12].

1.2 Fragmented Validation

In contrast to these theoretically advanced studies, this paper presents a totally different approach, a very simple and pragmatic idea which turns out to be rather effective for concrete programming, called *fragmented validation* (FV). It does not consider type of the input document at all, but *only the consecutive sequences of nodes from the result language* which occur in the contents of an element somewhere in the transformation program's XML representation, and which serve as constant data for the transformation process. The transformation result will be produced by combining these fragments;[2] therefore they must match the result document type in at least *some* context. In other words, there must exist at least one rule in the result document type definition which is able to produce contents that contains the fragment.

For example, when producing XHTML output, any XSLT code like

```
<xsl:template ...>
  <xsl:.../><xsl:.../><tr>...</tr><xsl:.../><td>...</td>
</xsl:template>
```

will never produce a valid result: there is no "content model" (i.e. regular expression, see section 2.1) in the type definition of XHTML which allows the elements `<tr>` (table row)

[2] There are other ways of producing output, e.g. by copying nodes of the input document, or explicit element construction, but in many cases their role is marginal.

and `<td>` (table data cell) to appear on the same level of nesting. This violation can be found independently from the contents of the other embedded XSLT commands in this example. These can produce anything from the empty sequence to arbitrary sequences of *complete* elements. So their outcome cannot change the levels of nesting, and cannot heal the violation.[3]

That all appearing constant result fragments match the result document type is neither a necessary nor a sufficient condition for the correctness of the transformation in a strict mathematical sense: the combination of correct fragments can still violate the result type, and incorrect fragments appearing in the source may belong to "dead code" and may never be used. Debugging XSLT transformations is a rather tedious task, in spite of the intended readability of the "graphical" text format, see next section. We found that not only at about thirty percent of programming errors are detected a priori by fragmented validation, but also that the remaining errors of illegal combinations are much easier diagnosed, because the strategy for debugging applied by the programmers change fundamentally and becomes much more focused, as soon it is guaranteed that the constant fragments themselves cannot be the source of typing errors in some generated output.

The algorithm presented here performs simple and comprehensible validation of *all* result fragments contained in an XSLT program. This is done by a kind of *abstract interpretation*, which operates on sets of states, and pre-figures a possible later parsing process of the result document. It does so in linear time, when the XSLT program is parsed and its data model is constructed, "on the fly", by tracking the corresponding SAX events.

1.3 XSLT Program as a Two-Coloured Tree

The front-end representation chosen by the designers of XSLT integrates the XSLT language constructs and the constants of the result language seamlessly: both are represented as well-formed XML structures, which can be interspered in a rather free fashion. The intention is to make the formatted program easily readable from two different viewpoints, in the style of a visual ambiguity: the XSLT language constructs can contain fragments of the result language, as they are to appear in the output, as their operands, and these fragments in turn can contain XSLT elements which will be replaced by their evaluation result when executing the transformation.

Therefore an XSLT program comes as a *two-coloured* XML tree, in which elements from XSLT and elements from the result language are mixed. The rules for either embedding are defined atop a categorization of the XSLT elements:

- The top element is always a `stylesheet` element from XSLT.[4]
- This element contains elements from the **T** or "top" category of XSLT elements.
- Some of those and of their children belong to the **R** or "result element-containing" category, which can contain result-language elements directly in their contents.
- Vice versa, there is the **I** or "instruction" category[5], which can be inserted anywhere

[3] This principle is the core of the type safety of XSLT compared to string manipulation languages like PHP, which produce "tag soup", and can indeed be a severe obstacle for programmers used to these, as the long lasting discussion in `https://bugzilla.mozilla.org/show_bug.cgi?id=98168` about "`disable-output-escaping`" shows.

[4] This element can also be called `transform`, and all definitions must be doubled accordingly. The following text ignores this synonym for better readability. The nomenclature in the XSLT standard [14] is a little bit peculiar anyhow, e.g. functions are called "templates", etc.

[5] This strange wording is again taken from the standard; the one character abbreviations of the categories are ours.

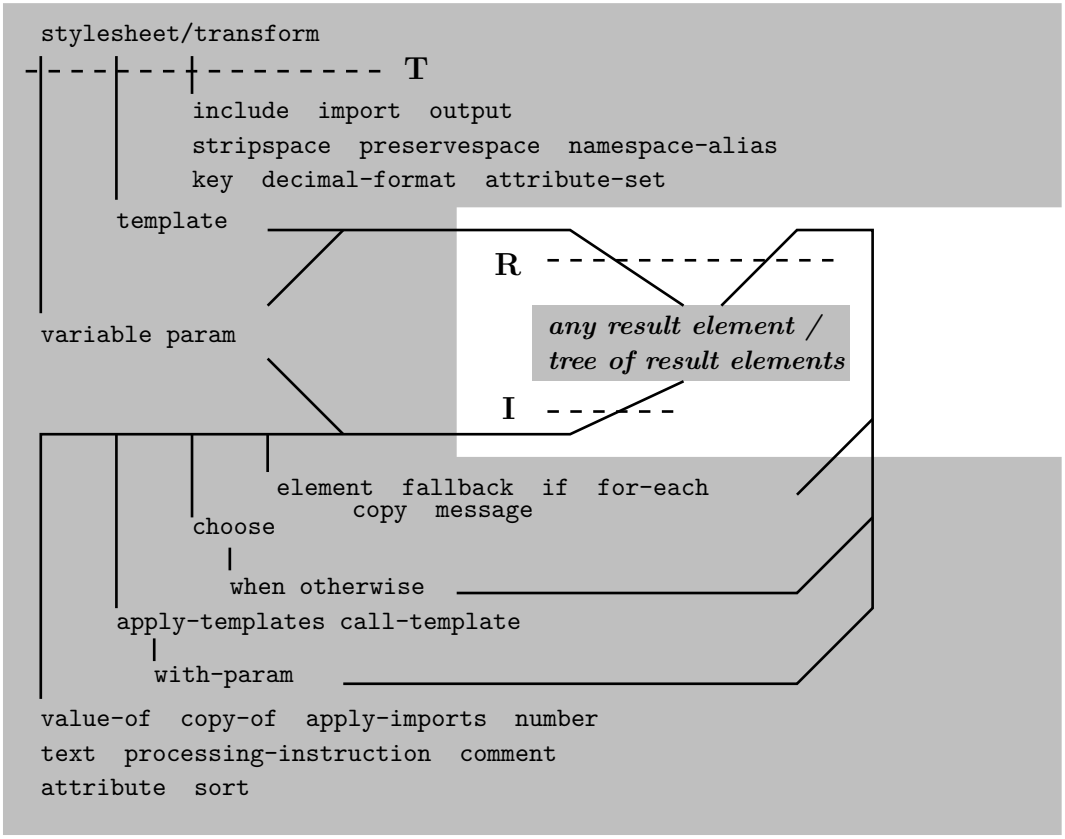

Figure 1 XSLT and result language nesting, with categories.

in such contents, or arbitrarily deeply nested in a result language structure. Later, when the whole program will be executed, these XSLT elements will evaluate to some result language elements ("produce" them), which will be inserted into the surrounding constant result structure.

Figure 1 shows the resulting "sandwich" structure and the possible transitions between both element sets: each arrow indicates a possible parent-child relationship in an XSLT tree. Figure 2 shows a typical XSLT source, a bi-coloured tree with pure and mixed sub-trees. The positions where result language elements may occur in an XSLT structure are limited, namely the complete contents of the elements of category **R**; there arbitrary sequences of result language elements may appear, mixed again with XSLT elements from category **I**.

By defining an artificial XSLT element e_{R}, which wraps these sequences as its contents, and which appears (like a normal XSLT element) in the content models of the **R** elements as a placeholder for this of embedding, we can integrate those mixed sequences smoothly into the parsing process of XSLT.

The other way round, starting with an element of the result language, this easy procedure is not applicable, because the **I** category elements from XSLT can appear *ubiquitously* in the result language structure. Therefore the theoretically possible approach of factoring out all possible interleavings and treat them by constructing the joint deterministic automaton is not feasible in practice, due to combinatorial explosion. Instead, we construct two independent conventional deterministic automata for both language definitions in the form of a *transition relation* and some auxiliary mappings. Simple algebraic operations on this

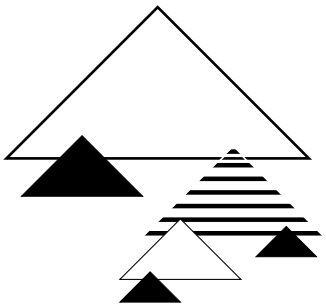

Figure 2 XSLT (white), result (black) and unparsed, two-coloured subtrees of an XSLT program.

Table 1 Content models and calculation of transition relation.

$$C ::= (E, S) \mid (C\texttt{,}\ \ldots\ \texttt{,}C) \mid (C\texttt{|}\ \ldots\ \texttt{|}C) \mid C\texttt{?} \mid C\texttt{*} \mid C\texttt{+}$$

$$\mathsf{startState} : (E \setminus \{e_\mathrm{C}, e_\mathrm{R}\}) \to S$$

$$\mathsf{refTo} : S \twoheadrightarrow E$$

$$\mathsf{collect} : C \times \mathbb{P}S \times (S \leftrightarrow S) \quad \to \quad \mathbb{P}S \times (S \leftrightarrow S)$$

$$\frac{\mathsf{collect}(c_1, T, U) = (V, W) \qquad \mathsf{collect}(c_2, T, U) = (X, Y)}{\mathsf{collect}\big((c_1\texttt{|}c_2), T, U\big) = (V \cup X, W \cup Y)}$$

$$\frac{\mathsf{collect}(c_1, T, U) = (V, W) \qquad \mathsf{collect}(c_2, V, W) = (X, Y)}{\mathsf{collect}\big((c_1\texttt{,}c_2), T, U\big) = (X, W \cup Y)}$$

$$\frac{\mathsf{collect}(c, T, U) = (V, W)}{\mathsf{collect}(c\texttt{?}, T, U) = (T \cup V, W)}$$

$$\mathsf{collect}(c\texttt{+}, T, U) = \mathsf{collect}\big((c\texttt{,}c\texttt{?}), T, U\big)$$

$$\mathsf{collect}(c\texttt{*}, T, U) = \mathsf{collect}\big((c\texttt{+})\texttt{?}, T, U\big)$$

$$\mathsf{collect}\big((e, s), T, U\big) = \big(\{s\}, U \cup (T \times \{s\})\big)$$

$$S_\mathrm{acc} = \bigcup_{e \in E} \pi_1\big(\mathsf{collect}(c_e, \{\mathsf{startState}(e)\}, \{\})\big)$$

$$\mathsf{goesTo} = \bigcup_{e \in E} \pi_2\big(\mathsf{collect}(c_e, \{\mathsf{startState}(e)\}, \{\})\big)$$

relation allow us to switch between deterministic and non-deterministic modes of operation, whenever the special parsing situations arise.

By these means the algorithm constitutes a transition relation, which can be evaluated on the fly when parsing an XSLT source. As soon as for the given input no further transition is possible, either a validation of genuine XSLT syntax is found, or a fragment of the result language which is not part of any valid result document.

2 Parsing XSLT Programs With Fragmented Validation

2.1 Validation of the XSLT Program

The algorithm presented here is based on DTDs as the means for defining the structure of the involved documents. This formalism is specified in the core XML standard [3]. A DTD

defines (1) a set of string values as *element tags*, and for every such element (2) a list of *attributes* with names, types and default values[6], and for every element (3) a *content model*, which is an *extended regular expression* over the set of element names. As ususal, a regular expression defines an accepted language. In this case this is a set of sequences of element names, which defines the legal contents of the element under definition.

While the XSLT language is not specified in terms of a DTD, this can easily be constructed, and is called D_{X} in the following.[7] The corresponding sets of element tags is E_{X}. Beside the element tags defined in the standard, E_{X} contains an element e_{D} which represents the top-most level of the document tree (in XML this called "document" and is an additional level one above the top-most element), and an additional element e_{R} which wraps all embedded sequences of result elements, as described above.

The structure of the result language must be given by a second DTD, called D_{R} and a set of element names E_{R}. References to character data in the content models (`#PCDATA`) are realized as a reference to some additional, reserved element e_{C}. This is the only element contained in both sets E_{X} and E_{R}. Their union is E.

Each DTD contains a mapping from its subset of E to content models C, see the definitions in Table 1. Let $c_e \in C$ be the content model for a given $e \in E$. A content model is an extended regular expression: the atoms are references to elements; the unary constructors are option, star and plus; the n-ary constructors are sequence and alternative, which can be realized by associative binary operators.

In our approach the references to elements in a content model are realized by a pair of the element's name e and a state number $s \in S$. These states are *unique* over all content models of both DTDs and identify the symbolic state of the accepting automaton *after* a complete element with the name e has been consumed in that position. The content model of the document level is additionally defined to $c_{e_{\mathrm{D}}} = (\texttt{stylesheet}, s_{\mathrm{D}})$.

That a state appears in a pair (e, s) in a content model is reflected by (s, e) appearing in the mapping $\mathsf{refTo} : S \twoheadrightarrow E$. Additionally, there is an initial state for every content model, given by $\mathsf{startState} : E \twoheadrightarrow S$, having consumed nothing and thus not in the domain of refTo.

All automata of all content models of both DTDs are realized by the one global relation $\mathsf{goesTo} : S \leftrightarrow S$, together with the set of accepting states $S_{\mathrm{acc}} \subset S$. These are constructed by the function $\mathsf{collect}(c, T, U)$, which performs an abstract parsing process of the content model c, with T being the set of final states of all preceding parsing steps, i.e. the "incoming states", and U being the accumulated transition relation so far.

For each element, this function is started with its whole content model, its start state, and an emtpy "goesTo" relation, see the last two lines in Table 1. The rules for the different kinds of content models are written as logical inference rules and can be executed deterministically by function evaluation. The rule for alternatives $c_1 \,|\, c_2$ simply unifies both sets; the rule for sequences $c_1 \,|\, c_2$ starts parsing of c_2 with the results of c_1 and unifies the transition relation; the rule for $c?$ simply adds the incoming states to the set of final states, thus reflecting the epsilon case of its parsing. The other combinators are defined by equivalence; finally the parsing of a reference adds transitions from all incoming state to its own state, which also becomes the single final state.

So far, goesTo, S_{acc} and refTo are nothing more than a decomposed representation of a standard labeled transition graph. Since all epsilon transitions are eliminated on the

[6] Attributes are not treated in this article.

[7] The DTD in the appendix of [14] is non-normative. We took it as a starting point, but defined slightly different abstractions.

fly, and since XML is basically LL(1), the resulting relation corresponds to a *deterministic* finite automaton.[8] This decomposition makes parsing look more complicated in the simple, deterministic case. But it allows to switch into non-deterministic mode by two simple set-based operations, which are the heart of our algorithm, see the boxed expressions in Table 2.[9]

This table shows the operational semantics of the complete parsing process, omitting tactics for error recovery, which can easily be integrated according to [6]. An XML source can be seen as a stream of elements from J, which are open tags, close tags and character data. The nesting of the tags constitute the borders of the encoded element contents. Parsing means to translate such a stream into a tree-like structure N, which is an algebraic data type, built from an element tag and the sequence of the element's children. This is realized by the function translate from Table 2, which starts the process with e_{D}, and succeeds if both the accepting state s_D of this content model and the end of the input are reached.

A stack frame is a pair of the node currently under construction, and the set of active states of the accepting (deterministic or non-deterministic) automaton. The parsing function $\longmapsto$ operates on a pair of such a stack and an input stream and is written as $\kappa \triangleleft f \;/\; b \triangleright \beta \longmapsto \kappa' \triangleleft f' \;/\; b' \triangleright \beta'$. There $\kappa \triangleleft f$ stands for a stack or list with last (top) element f with the predecessors κ, whereas $b \triangleright \beta$ stands for a stream with the first (head) element b and tail β.

Corresponding to the tree nature of XML, the overall parsing is realized by a recursive call of parsers for content models, and the first three rules of table 2 are very similar to those known from standard tree parsers.

The parsing process can take the following steps:

(open) — Whenever an open tag is found, and at least one of the current states goes to a state which consumes this element, then a new stack frame is opened and the parsing of this element is started. This new frame contains the start state of this element as its only member, so the parsing process begins in a traditional deterministic way. This rule is the same for XSLT and result elements and can only be taken if the current and the new element are from the same set; otherwise there is no transition in goesTo.[10]

(chars) — Character data in the input stream is simply appended to the contents of the currently parsed element. If it is not totally made of white space characters (indicated by *WS* in the formula) then the set of states is modified and reflects the consumption of the pseudo element e_{C}. This rule is the same for XSLT and result elements.

(charsWs) — Character data which is totally made of white space characters does not change the set of states. All states which represent "mixed" contents do accept this input without effect on the further transitions, and states from the other content models will treat

[8] XML allows non-deterministic parsing only of empty input sequences, e.g. content models like "`(a*|b*)`", as long as "`(a+|b+)`" would be LL(1).

[9] By employing a library for the manipulation of relations, this algorithm directly describes a practical implementation.

[10] Technical detail: the set of states *after* this step is needed already here to check the legality of the input open tag (the set must be non-empty). For optimization, the calculated result is stored in the stack frame, which from now on reflects the *future* situation after the successful acceptance of the whole new element.

Table 2 Unified parsing process, deterministic and non-deterministic.

$$J ::= \mathsf{open}(E \setminus \{e_{\mathrm{C}}, e_{\mathrm{R}}, e_{\mathrm{D}}\}) \mid \mathsf{close}(E \setminus \{e_{\mathrm{C}}, e_{\mathrm{R}}, e_{\mathrm{D}}\}) \mid \mathit{Chars}$$

$$N ::= \mathsf{node}(E \setminus \{e_{\mathrm{C}}\} \times \mathsf{SEQ}(N \cup \mathit{Chars}))$$

$$\mathsf{frame} = N \times \mathbb{P}S$$

$$\_ \,/\, \_ \longmapsto \_ \,/\, \_ : (\mathsf{SEQ}\ \mathsf{frame}) \times (\mathsf{SEQ}\ J) \to (\mathsf{SEQ}\ \mathsf{frame}) \times (\mathsf{SEQ}\ J)$$

$$\mathsf{startFrame}(b) = (\{\mathsf{startState}(b)\}, \mathsf{node}(b, \langle\rangle))$$

$$\frac{\mathsf{refTo}^{-1}(b) \cap \mathsf{goesTo}(|s|) = s' \neq \{\}}{\kappa \triangleleft (s, n) \,/\, \mathsf{open}(b) \triangleright \beta \longmapsto \kappa \triangleleft (s', n) \triangleleft \mathsf{startFrame}(b) \,/\, \beta} \textbf{(open)}$$

$$\frac{\mathsf{refTo}^{-1}(e_{\mathrm{C}}) \cap \mathsf{goesTo}(|s|) = s' \neq \{\} \qquad \mathit{chars} \in \mathit{Chars} \setminus \mathit{WS}}{\kappa \triangleleft (s, \mathsf{node}(a, \alpha)) \,/\, \mathit{chars} \triangleright \beta \longmapsto \kappa \triangleleft (s', \mathsf{node}(a, \alpha \triangleleft \mathit{chars})) \,/\, \beta} \textbf{(chars)}$$

$$\frac{\mathit{chars} \in \mathit{WS}}{\kappa \triangleleft (s, \mathsf{node}(a, \alpha)) \,/\, \mathit{chars} \triangleright \beta \longmapsto \kappa \triangleleft (s, \mathsf{node}(a, \alpha \triangleleft \mathit{chars})) \,/\, \beta} \textbf{(charsWs)}$$

$$\frac{S_{\mathrm{acc}} \cap s \neq \{\}}{\kappa \triangleleft (s', \mathsf{node}(a, \alpha)) \triangleleft (s, \mathsf{node}(b, \beta)) \,/\, \mathsf{close}(b) \triangleright \gamma \longmapsto \kappa \triangleleft (s', \mathsf{node}(a, \alpha \triangleleft \mathsf{node}(b, \beta))) \,/\, \gamma} \textbf{(close)}$$

$$\frac{\mathsf{refTo}^{-1}(e_{\mathrm{R}}) \cap \mathsf{goesTo}(|s|) = s' \neq \{\} \qquad b \in E_{\mathrm{R}}}{\kappa \triangleleft (s, n) \,/\, \mathsf{open}(b) \triangleright \beta \longmapsto \kappa \triangleleft (s', n) \triangleleft (\boxed{\mathsf{refTo}^{-1}(b)}, \mathsf{node}(e_{\mathrm{R}}, \langle\rangle)) \triangleleft \mathsf{startFrame}(b) \,/\, \beta} \textbf{(x2r)}$$

$$\frac{}{\kappa \triangleleft (s, \mathsf{node}(x, \alpha)) \triangleleft (s', \mathsf{node}(e_{\mathrm{R}}, \alpha')) \,/\, \mathsf{close}(x) \triangleright \beta \longmapsto \kappa \triangleleft (s, \mathsf{node}(x, \alpha \frown \alpha')) \,/\, \mathsf{close}(x) \triangleright \beta} \textbf{(x2r')}$$

$$\frac{a \in (E_{\mathrm{R}} \cup \{e_{\mathrm{R}}\}) \qquad b \in E_{\mathbf{XI}}}{\kappa \triangleleft (s, \mathsf{node}(a, \alpha)) \,/\, \mathsf{open}(b) \triangleright \beta \longmapsto \kappa \triangleleft (\boxed{\mathsf{goesTo}^{*}(|s|)} \,/\, \mathsf{node}(a, \alpha)) \triangleleft \mathsf{startFrame}(b), \beta} \textbf{(r2x)}$$

$$\mathsf{translate} : \mathsf{SEQ}\ J \twoheadrightarrow N$$

$$\mathsf{translate}(j) = n \iff \mathsf{startFrame}(e_{\mathrm{D}}), j \longmapsto^{*} \langle(\{s_{\mathrm{D}}\}, n)\rangle, \langle\rangle$$

the input as "ignorable whitespace" and are not affected either.[11]

(close) — Whenever the expected close tag is found, and at least one of the active states is accepting, then the top-most stack frame is dropped, the currently parsed element is

[11] Note that the rule "A validating XML processor MUST also inform the application which of these characters constitute white space appearing in element content" from the XML specification [3, sect. 2.10], is somehow ill-defined in the context of XSLT sources. Let "_" symbolize some white space in the input text, like in
`<xsl:with-param> ___<b/>__<c/></xsl:with-param>`
Then, together with the result language DTD definitions
`<!ELEMENT x (a,b,c)>`
`<!ELEMENT y (#PCDATA|a|b|c)*>`
this whitespace will be ignorable or not, depending on the *dynamic* context of the later expansion.

considered complete and it is appended to the contents of the element parsed one level above. This rule is the same for any combination of XSLT and result elements on both positions.

So far the rules are only a complicated implementation of well-known standard parsing. But the following rules are specific for fragmented validation and represent the topic and main contribution of this article. They manage the transition between the two sets of D_{X} and D_{R}, and introduce non-determinism:

(x2r) — Whenever an open tag of the result language element appears while an XSLT element is parsed, *two* stack frames are added: the upper one has the artificial element e_{R} as its growing node, and represents the embedding of a sequence of result elements in the XSLT elements' contents. This integration is in a smooth way: no further ad-hoc action or adjustment are necessary, as the comparison between **(open)** and the upper and first part of **(x2r)** shows.

The second frame represents the result element b and its further contents, in the same way as in rule **(open)**. The framed part of the formula makes the difference: since we do not know statically in which context the result elements will be inserted later, when executing the XSLT program, *all states* which refer to the result element (i.e. which are reached by consuming it) are put into the state set of the e_{R}-frame. Here a *first source of non-determinism* comes into play.

Then parsing continues normally, according to **(close)** and **(open)** (and possibly (r2x), see below): the contents of b will be completed, and after its close tag all those sequences of result elements may follow, which may follow in *any* content model from C_R after *any* occurence of b. This is achieved because *all* corresponding states have been entered into the frame by refTo^{-1}.

(x2r') — This process continues until the close tag of x is parsed. Now the accumulated contents of e_{R} are appended to the contents of x, and the stack frame for e_{R} is discarded.[12] After this, the rule **(close)** will apply normally.

(r2x) — Let $E_{\mathrm{XI}} \subset E_{\mathrm{X}}$ be the XSLT elements from the **I** category, which can appear ubiquitously in result elements. Whenever a corresponding open tag appears in a result element's contents, this is *always* legal and starts the XSLT parsing process. This is very similar to the simple rule **(open)**, as the comparison of the formulas shows.

Additionally, the state set of the result element is upgraded by applying the reflexive-transitive closure of the transition function; see again the framed expression. This reflects the fact that we do not know *how many* of the still required/allowed children from the result content model will later be delivered by the expansion of the XSLT term. This may be anything from the empty list, up to all the missing rest.

Both kinds of non-determinism combine nicely. The underlying relational operations can be implemented efficiently as table lookups; the relations depend only on the two DTDs and their setup time and space complexity grow polynomially with $|S|$ only, hence they can feasibly be precomputed and cached ahead of time. Time consumption of FV is linear with

[12] In this version of XSLT, the upper stack frame created by rule (x2r) could be spared, since the information s' is not required after completion of e_R: it stands always at a terminal and accepting position of an XSLT content model. But this does not hold for the general case in which there could be more than one appearance of e_R in a content model.

the size of the XSLT source, and it can be executed in parallel with parsing. As mentioned above, as soon as no transition is possible, an error has been detected. Nevertheless, parsing and FV can be resumed with the next top-level XSLT `template` element. FV is incremental, it can be applied to incomplete programs in each phase of programming, and to general purpose libraries, since it is independent of the input format.

2.2 Generating Diagnostic Information

Whenever the set of active states becomes empty due to the value of the head of the input stream, either a violation of the XSLT syntax or an invalid fragment of the result language is detected. In the latter case error information like

```
Error in xslt file (file id / line number):
The sequence of elements
      [tr][xsl:if][td]
cannot produce valid content w.r.t. "xhtml-1-0-strict.dtd"
```

can easily be derived. A concrete tool can give further hints to the programmer, e.g. print out all content models which ever contributed to the state set in this stack frame.

2.3 Example

Table 3 illustrates the operation of the algorithm. The top shows content models which are indeed a small fragment of a typical situation in practice. The informal notation shows the initial states from $\mathsf{startState}(e)$ and the states from (e, s), as defined in Table 1, as exponents. Below the transition system generated by the algorithm from that table.

The operation of the transition function $\longmapsto$ shows on the left side only the element names and state sets of the stack frames, ie. omits the nodes' contents, and on the right only the heads of the input stream. All possibly intervening states and inputs for parsing of the contents of the nodes are also omitted.

The trace ends with the detection of an error: There is no content model which allows more than one `title` element on the same level of nesting.

2.4 Tests

Table 4 shows the results of applying our local analysis FV in comparison to the global *control-flow based* analysis from Møller et al. [10] to their test data. Their approach is referred to as "XSLV" in the following, for a description see the following section on related work. We used version 0.9 of their tool. The rather tedious mining and preparation of this real-world test data is described in detail in [10]. We applied the XSLV tool and our implementation of FV against the XSLT sources of ten of their test cases and the XHTML 1.0 DTD.[13]

The figures in Table 4 are in no way meant competitive. They only can give an impression of the possible impact of both tools on programming and debugging practice.

The first numeric column gives the lines of code of the xslt source. The second column gives the number of errors signalled by XSLV, and the third, labeled "(cf)", the subset of those which really can only found by control flow analysis. These are the errors which can

[13] Four of the fifteen cases have been excluded because they do not use HTML as their result language; one test case was not re-producable.

Table 3 Example run of the algorithm.

$$
\begin{array}{l}
\texttt{html} = {}^{1}\ \texttt{head}^{2}\ \texttt{,}\ \texttt{body}^{3} \\
\texttt{head} = {}^{4}\ \texttt{(}\ \texttt{script}^{5}\ \texttt{|}\ \texttt{style}^{6}\ \texttt{)*,} \\
\quad \texttt{(}\ \texttt{title}^{7}\ \texttt{,}\ \texttt{(}\ \texttt{script}^{8}\ \texttt{|}\ \texttt{style}^{9}\ \texttt{)*,}\ \texttt{(}\ \texttt{base}^{10}\ \texttt{,}\ \texttt{(}\ \texttt{script}^{11}\ \texttt{|}\ \texttt{style}^{12}\ \texttt{)*)?)} \\
\quad \texttt{|}\ \texttt{base}^{13}\ \texttt{(}\ \texttt{script}^{14}\ \texttt{|}\ \texttt{style}^{15}\ \texttt{)*,}\ \texttt{title}^{16}\ \texttt{,}\ \texttt{(}\ \texttt{script}^{17}\ \texttt{|}\ \texttt{style}^{18}\ \texttt{)*} \\
\quad \texttt{)} \\
\ldots \\
\texttt{h1} = {}^{71}\ \texttt{(}\ \texttt{\#chars}^{72}\ \texttt{|}\ \texttt{a}^{73}\ \texttt{|}\ \texttt{b}^{74}\ \texttt{|}\ \texttt{script}^{75}\ \texttt{)*,}
\end{array}
$$

$$
\begin{array}{rcl}
\mathsf{goesTo} & = \{ & 1 \mapsto 2, 2 \mapsto 3, \\
 & & 4 \mapsto 5, 4 \mapsto 6, 4 \mapsto 7, 4 \mapsto 13 \\
 & & 5 \mapsto 5, 5 \mapsto 6, 5 \mapsto 7, 5 \mapsto 13, \\
 & & 6 \mapsto 5, 6 \mapsto 6, 6 \mapsto 7, 6 \mapsto 13, \\
 & & 7 \mapsto 8, 7 \mapsto 9, 7 \mapsto 10, \\
 & & 8 \mapsto 8, 8 \mapsto 9, 8 \mapsto 10, \\
 & & 9 \mapsto 8, 9 \mapsto 9, 9 \mapsto 10, \\
 & & 10 \mapsto 11, 10 \mapsto 12, 11 \mapsto 11, 11 \mapsto 12, 12 \mapsto 11, 12 \mapsto 12, \\
 & & 13 \mapsto 14, 13 \mapsto 15, 13 \mapsto 16, \\
 & & 14 \mapsto 14, 14 \mapsto 15, 14 \mapsto 16, 15 \mapsto 14, 15 \mapsto 15, 15 \mapsto 16, \\
 & & 16 \mapsto 17, 16 \mapsto 18, 17 \mapsto 17, 17 \mapsto 18, 18 \mapsto 17, 18 \mapsto 18, \\
 & & \ldots \\
 & & 71 \mapsto 72, 71 \mapsto 73, 71 \mapsto 74, 71 \mapsto 75, \\
 & & 72 \mapsto 72, 72 \mapsto 73, 72 \mapsto 74, 72 \mapsto 75, \\
 & & 73 \mapsto 72, 73 \mapsto 73, 73 \mapsto 74, 73 \mapsto 75, \\
 & & 74 \mapsto 72, 74 \mapsto 73, 74 \mapsto 74, 74 \mapsto 75, \\
 & & 75 \mapsto 72, 75 \mapsto 73, 75 \mapsto 74, 75 \mapsto 75 \\
 & \} & \\
S_{\mathrm{acc}} & = & \{3, 7, 8, 9, 10, 11, 12, 16, 17, 18, 71, 72, 73, 74, 75\}
\end{array}
$$

$$
\begin{array}{rcl}
 & \kappa_1 = \langle \ldots, (\texttt{xsl:template}\langle\rangle\{\ldots\}) \rangle & / \ \mathsf{open}(\texttt{script}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathsf{e_R}\langle\rangle\{5, 8, 11, 14, 17, 75\}) \triangleleft (\texttt{script}\langle\rangle\{\ldots\}) & / \ \mathsf{close}(\texttt{script}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathsf{e_R}\langle\rangle\{5, 8, 11, 14, 17, 75\}) & / \ \mathsf{open}(\texttt{style}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathsf{e_R}\langle\rangle\{6, 9, 12, 15, 18\}) \triangleleft (\texttt{style}\langle\rangle\{\ldots\}) & / \ \mathsf{close}(\texttt{style}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathsf{e_R}\langle\rangle\{6, 9, 12, 15, 18\}) & / \ \mathsf{open}(\texttt{title}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathsf{e_R}\langle\rangle\{7, 16\}) \triangleleft (\texttt{title}\langle\rangle\{\ldots\}) & / \ \mathsf{close}(\texttt{title}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathsf{e_R}\langle\rangle\{7, 16\}) & / \ \mathsf{open}(\texttt{xsl:if}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathsf{e_R}\langle\rangle\{7, 8, 9, 10, 11, 12, 16, 17, 18\}) \triangleleft (\texttt{xsl:if}\langle\rangle\{\ldots\}) & / \ \mathsf{close}(\texttt{xsl:if}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathsf{e_R}\langle\rangle\{7, 8, 9, 10, 11, 12, 16, 17, 18\}) & / \ \mathsf{open}(\texttt{title}) \triangleright \ldots \\
\longmapsto & \kappa_1 \triangleleft (\mathsf{e_R}\langle\rangle\{\}) &
\end{array}
$$

never be found by the purely local method of FV. But if the test data is fairly representative, as claimed in the discussion in [10], then these low figures support our approach.

The last column shows the number of errors detected by our tool, most of them by FV, enhanced by checks for missing and wrongly used *attribute values*, based on a similar, but simpler local strategy. Not considering the (cf) errors, both tools always found the same errors, and one tool some additionals. The cases when FV "won" are due to implementation flaws: Theoretically XSLV finds all errors FV can find.

2.5 Execution of the XSLT program

In our implementation fragmented validation is performed when constructing an internal data model of an XSLT program. In the context of our metatools framework [16], XML models are driven by DTDs, and rely on the fundamental property that DTD content models

Table 4 Test results: control flow-based XSLV vs. local-only FV.

Nr	Testcase Nickname	loc	XSLV tool	(cf)	FV tool
1	poem	36	3		3
2	AffordableSupplies	42	8		8
3	agenda	43	2		1
6	adressebog	76	2	1	1
8	slideshow	119	2		0
9	psicode-links	128	8	2	12
11	proc-def	258	7	1	10
12	email_list	243	2		3
13	tip	265	7	2	3
14	window	701	4		1

are specific for element names and do not depend on the context. In this framework, the tdom tool is a program which translates a DTD into a collection of JAVA sources which can realize exclusively all well-typed text corpora w.r.t. this DTD: classes are generated for every element declaration, and for every sub-expression of a nested content model. On all levels, all constructor methods ensure and all setter methods preserve validity.

In the context of XSLT, this technology is applied throughout, except for the "two-coloured" lists which combine elements from both sets. They are realized by lists of the supertype of both tdom models. Since no tdom element instance can be constructed with such a sequence as contents, the two-coloured nature propagates up the document tree, until it is absorbed by the synthetic element e_{R}, which combines a two-coloured downside with a well-typed upside (see Fig. 2).

With this data model the evaluation of an XSLT program becomes a very simple transformation: all subtrees from XSLT must be replaced by their evaluation results; the two-coloured lists turn into homogeneous ones of result type, and only these must finally be parsed incrementally into a tdom subtree. All other contents have already been checked statically when creating the program's model, by fragmented validation.

3 Outlook

We have presented an enhancement of a standard tree parser algorithm applied to XSLT sources. Simple algebraic operations on the transition relation reflect the non-determinism which is induced by XSLT elements serving as parents or siblings of result elements. This abstract interpretation allows to detect a substantial share of typing errors in sources of XSLT transformations by an easy to implement "on-the-fly" algorithm.

3.1 Future Work

This kind of abstract interpretation, which operates on sequences of sets of states, seems promising for further research and possible natural extensions.

First, the treatment of XSLT instructions embedded into output can be differentiated further, according to their known meaning: for instance, a `<comment>` element can not affect the parsing state, and an `<element name=...>` instruction can be treated as a *verbatim* result element, if the name attribute's value can be determined statically.

Also, in a general sense, extensions are possible: assume a sequence mixed from result and XSLT elements r_1, x, r_2 is parsed, and the set of states after parsing r_1 is s_1. Then when parsing x the rule **(r2x)** widens the set of states to the set s_2 by applying the reflexive-transitive closure of the transition function, which represents the uncertainty of the future execution, and for r_2 the rule **(open)** narrows the set of states again to the set s_3. From these sets immediately follows a certain *type* of the XSLT function x, which in any case must produce a sequence of elements which lie on a path from some state in s_1 to some state in s_3.

In many cases this type information can be exploited. E.g. an XSLT `<if>` expression with constant contents can only expand to that contents or to the empty sequence. Similarly, a `<choose>` represents the disjunction of its contents. When x is the call of a "named template", then all types demanded by all places of such calls can be intersected for inference.

It seems worth exploring how far such a notion of "type" will lead. Possibly not very far w.r.t. absolutely preventing typing errors, simply due to the proven un decidability of this problem in general. E.g. as soon as dynamically selected processing is initiated by an `<apply-templates>` statement, our approach reaches its limits. But in any case a regular expression can be synthesized and delivered to the programmer as a *hint* which sequence of elements the code of such a "framed" sub-expression, like x in the example above, must deliver.

In the field of XSLT processing some benchmarks and standard conformance test suites had been developed, but most of them already have disappeared again. A recent benchmark framework and test case collection has been released by Saxonica company as open source project [5]. How far this can be adopted to our "DTD aware" approach is currently under research. The same holds for the only still available conformance test suite by OASIS [1]. The results of applying them both would be very valuable, but since for our tool the result document DTD must always be provided, e.g. re-constructed, we expect this to become rather expensive soon, esp. in the second case with its nearly four thousand test cases.

3.2 Related Work

The problem of type checking XML transformations in a general sense has been studied thoroughly during the last decade in dozens of papers in the context of data base queries and of dedicated functional XML transformation languages, but seldom w.r.t. XSLT. For a survey see [12] and [11].

Very early proposals and influential suggestions can be found in [2]. This paper is purely theoretical. It translates a very simple subset of XSLT into a collection of formal constraints, but excludes the implications of XPath navigation and the `<apply-templates>` matching mechanism completely.

Tozawa [13] also restricts the analyzed transformation language to a non-Turing-complete subset of XSLT, excluding XPath horizontal and upwards navigation, and again the implications in control flow induced by pattern matching. The chosen technique is *backward type inference*, which infers the type of all input documents w.r.t. a given result type. It seems that this interesting approach could not be extended to full-scale XSLT.

A variant called "exact type checking" by its authors restricts types and transformations until type checking becomes decidable. (It is of course highly desirable that analyses of this kind would be carried out *before* the corresponding industrial standardization decisions are met.) For most advanced results and a survey on this line, see Maneth, Perst and Seidel [8].

The opposite approach is to look at the full functional range of XSLT and execute analysis as far as possible. This approach is more related to programming practice and thus to our

FV. The current standard in this field is the work of Møller, Olesen and Schwartzbach [10]. Their approach covers two very different problems: first from the collection of all statements of the form `<apply-templates select="`α`">` and `<template match="`β`">` in the program, an upper limit of the control flow graph is derived. (I.e. flow of execution from template to template, enriched with the change of the "current input focus" from one element tag to a set of possible element tags.)

All templates are translated into data graphs, which each can produce a certain language of result trees, and then these graphs are plugged together according to the flow graph from step one. In a second step it is checked that the resulting overall data graph produces only output which is in the language of the required result type. For most violations detected, detailed diagnostic information can be derived. Their solution has been implemented, is freely available, and has been tested with considerable amounts of real-world test data.

This valuable work plays of course in a different league than our approach. Tellingly, their paper takes nearly thirty pages to describe the algorithm, excluding the second step which is cited from an earlier work. Nevertheless, what they can calculate is (naturally) still an approximation, albeit a very good one. A comparison between of practical test results with both tools is given in section 2.4 above.

Currently, the most widely used XSLT processors (Xalan, Saxon, XT, libxslt) do not include any type checking, not even automated validation. The 2.0 version of the XSLT standard [15] defines a feature called "schema awareness" for XSLT processors. This includes the ability to read and define schema information in the sense of the W3C Schema language [4]. The standard foresees this information for *explicit validation* of subtrees of the generated output, controlled selectively by the programmer. Currently only the latest versions of some XSLT engines support this still-evolving standard. While it is arguable from the compiler construction point of view whether this is the right direction of development (these explicit excursions to the meta level are called "pragmas" in general-purpose programming languages, and generally deprecated for portability), the same information could be used in future for feeding type checking algorithms.

Acknowledgments. A two page extended abstract of this work has been published in the ICMT 2013 [7]. Many thanks to Anders Møller for giving us access to the XSLV tool and test cases. And to the anonymous reviewers for valuable hints.

References

1 *OASIS XSLT Conformance TC Public Documents.* OASIS, 2005. `http://www.oasis-open.org/commitees/tc_home.php/?wg_abbrev=xslt`.

2 Philippe Audebaud and Kristoffer Rose. Stylesheet validation. Technical report, Laboratoire de l'Informatique du Parallélisme, 2000.

3 Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, Fran cois Yergeau, and John Cowan. *Extensible Markup Language (XML) 1.1 (Second Edition).* W3C, `http://www.w3.org/TR/2006/REC-xml11-20060816/`, 2006.

4 David C. Fallside and Priscilla Walmsley. *XML Schema Part 0: Primer Second Edition.* W3C, `http://www.w3.org/TR/2004/REC-xmlschema-0-20041028/`, 2004.

5 Michael Kay and Debbie Lockett. Benchmarking xslt performance. In *XML London 2014 – Conference Proceedings*, volume 10, pages 23–38. XML London, 2014.

6 Markus Lepper and Baltasar Trancón y Widemann. d2d — a robust front-end for prototyping, authoring and maintaining XML encoded documents by domain experts. In Joaquim Filipe and J.G.Dietz, editors, *Proceedings of the International Conference on Knowledge*

Engineering and Ontology Delelopgment, KEOD 2011, pages 449–456, Lisboa, 2011. SciTePress.

7 Markus Lepper and Baltasar Trancón y Widemann. Fragmented validation — a simple and efficient contribution to xslt checking (extended abstract). In *Proc. ICMT 2013, Int. Conference on Theory and Practice of Model Transformations*, volume 7909 of *LNCS*. Springer, 2013.

8 Sebastian Maneth, Thomas Perst, and Helmut Seidl. Exact xml type checking in polynomial time. In *In ICDT*, pages 254–268, 2007.

9 Wim Martens and Frank Neven. On the complexity of typechecking top-down XML transformations. *Theoretical Computer Science*, 336:153–180, 2005.

10 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.

11 Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages, 2004.

12 Dan Suciu. The XML typechecking problem. *SIGMOD Rec.*, 31(1):89–96, March 2002.

13 Akihiko Tozawa. Towards static type checking for XSLT. In *Proceedings of the 2001 ACM Symposium on Document engineering*, DocEng '01, pages 18–27, New York, NY, USA, 2001. ACM.

14 W3C, `http://www.w3.org/TR/1999/REC-xslt-19991116`. *XSL Transformations (XSLT) Version 1.0*, 1999.

15 W3C, `http://www.w3.org/TR/2007/REC-xslt20-20070123/`. *XSL Transformations (XSLT) Version 2.0*, 2007.

16 Baltasar Trancon y Widemann, Markus Lepper, and Jacob Wieland. Automatic construction of XML-based tools seen as meta-programming. *Automated Software Engineering*, 10(1):23–38, 2003.

A Mathematical Notation

The employed mathematical notation borrows from the Z formalism. For convenience, the following table shows the details which are beyond basic set theory.

$A \to B$	The type of *total* functions from A to B
$A \nrightarrow B$	The type of *partial* functions from A to B
$A \leftrightarrow B$	The type of relations from A to B
$f\ (\mid s \mid)$	The image of set s under function or relation f
$f^{-1}(y)$	The preimage of value y under function f
r^*	The reflexive-transitive closure of relation r
$A \times B$	The product type of two sets A and B, i.e. all pairs $\{c = (a, b) \mid a \in A \wedge b \in B\}$.
π_n	The nth component of a tuple.
$\mathbb{P}(A)$	Power set, the type of all subsets of the set A.
$\mathsf{SEQ}\ A$	The type of finite sequences from elements of A
$\langle\rangle$	The empty sequence
$\alpha \frown \alpha'$	Concatenation of sequences α and α'
$a \triangleright \alpha$	A stream(/list/sequence) with element a followed by rest α
$\alpha \triangleleft a$	A list(/stack) with element a preceded by rest α