Basic Research in Computer Science

Static Analysis for Dynamic XML

Aske Simon Christensen
Anders Møller
Michael I. Schwartzbach

BRICS Report Series **RS-02-24**

ISSN 0909-0878 **May 2002**

See back inner page for a list of recent BRICS Report Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/02/24/`

Static Analysis for Dynamic XML

Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach
BRICS, Department of Computer Science
University of Aarhus, Denmark

Abstract

We describe the *summary graph* lattice for dataflow analysis of programs that dynamically construct XML documents. Summary graphs have successfully been used to provide static guarantees in the JWIG language for programming interactive Web services. In particular, the JWIG compiler is able to check validity of dynamically generated XHTML documents and to type check dynamic form data. In this paper we present summary graphs and indicate their applicability for various scenarios. We also show that summary graphs have exactly the same expressive power as the regular expression types from XDuce, but that the extra structure in summary graphs makes them more suitable for certain program analyses.

1 Introduction

XML documents will often be generated dynamically by programs. A common example is XHTML documents being generated by interactive Web services in response to requests from clients. Typically, there are no static guarantees that the generated documents are valid according to the DTD for XHTML. In fact, a quick study of the outputs from many large commercial Web services shows that most generated documents are in fact invalid. This is not a huge problem, since the browsers interpreting this output are quite forgiving and do a fair job of rendering invalid documents. Increasingly, however, Web services will generate output in other XML languages for less tolerant clients, many of whom will themselves be Web services.

Thus it is certainly an interesting question to statically guarantee validity of dynamically generated XML. Our approach is to perform a dataflow analysis of the program generating XML documents. This is a standard technique that is basically just parameterized by the finite lattice used to abstract the computed values and the transfer functions modeling the statements. The contribution described in this paper is the definition of an appropriate lattice of *summary graphs* that strikes a balance between expressive power and complexity. We show how summary graphs have been used to efficiently analyze realistic Web services with great accuracy. Also, we discuss what kind of operations on XML values that successfully can be captured by such dataflow analysis. Finally,

we show that summary graphs have the same expressive power as the regular expression types of XDuce [5, 7, 6].

2 XML Templates

We have concretely analyzed programs in the JWIG language, which is an extension of Java designed for programming interactive Web services. JWIG is a descendant of the `<bigwig>` language [2]. For the current discussions, we only need to consider how XML documents are built. JWIG is based on the notion of XML *templates*, which are just sequences of XML trees containing named *gaps*. A special *plug* operation is used to construct new templates by inserting existing templates or strings into gaps in other templates. Using XHTML as an example, the main method of a JWIG program manipulating templates could look like:

```
public void main() {
  XML wrapper = [[ <html>
                     <head>
                       <title>JWIG Example</title>
                     </head>
                     <body>
                       <[contents]>
                     </body>
                   </html> ]];

  XML item = [[ <li> <[text]> </li> <[items]> ]];

  XML x = [[ <ul class=[kind]> <[items]> </ul> ]];

  for (int i=0; i<n; i++) {
    x = x<[items = item<[text=i]];
  }
  show wrapper<[contents=x]<[kind="large"];
}
```

Gaps appear either as *template gaps*, such as `contents`, or as *attribute gaps*, such as `kind`. Both strings and templates may be plugged into template gaps, whereas attribute gaps only allow strings. The plug operation, x`<[`g`=`y`]`, returns a copy of x where copies of y have been inserted into all g gap. Template constants are denoted by `[[...]]`.

Note that XML values need not be constructed bottom-up, since gaps can be left in templates as targets for later plug operations. Also, a plug operation will fill in all occurrences of the given gap, even if they originate from different subtemplates. The more common language design of building XML values from constructors is a special case of this mechanism, since e.g. a construction like `l[`X`]` from XDuce corresponds to `[[ <l><[g]></l> ]] <[ g=`X` ]`. The plug operation has proved itself to be flexible and intuitive. Also, it is convenient

to write larger constant fragments in ordinary XML syntax rather than using nested constructor invocations.

3 Summary Graphs

We want to perform dataflow analysis of programs constructing XML values by plugging together templates. This key ingredient for such an analysis is a finite lattice for summarizing the state of a computation for each program point. Based on earlier experiences [10, 1], we have defined the lattice of *summary graphs*. Such a graph has as nodes the set of template constants occurring in the given program. The edges correspond to possible pluggings of gaps with strings or other templates. Given a concrete program, we let G be the set of gap names that occur and N be a set of *template indices* denoting the instances of XML template constants. A summary graph SG is formally defined as follows:

$$SG = (R, T, S, P)$$

where:

$R \subseteq N$ is a set of *root nodes*,
$T \subseteq N \times G \times N$ is a set of *template edges*,
$S : N \times G \to REG$ is a *string edge* map, and
$P : G \to 2^N \times \Gamma \times \Gamma$ is a *gap presence* map.

Here $\Gamma = 2^{\{\mathsf{OPEN},\mathsf{CLOSED}\}}$ is the *gap presence lattice* whose ordering is set inclusion, and REG is the set of regular languages over the Unicode alphabet.

Intuitively, the language $\mathcal{L}(SG)$ of a summary graph SG is the set of XML documents that can be obtained by unfolding its templates, starting from a root node and plugging templates and strings into gaps according to the edges. The presence of a template edge $(n_1, g, n_2) \in T$ informally means that the template with index n_2 may be plugged into the g gaps in the template with index n_1, and a string edge $S(n, g) = L$ means that every string in the regular language L may be plugged into the g gaps in the template with index n.

The gap presence map, P, specifies for each gap name g which template constants may contain open g gaps reachable from a root and whether g gaps may or must appear somewhere in the unfolding of the graph, either as template gaps or as attribute gaps. The first component of $P(g)$ denotes the set of template constants with open g gaps, and the second and third components describe the presence of template gaps and attribute gaps, respectively. Given such a triple, $P(g)$, we let $nodes(P(g))$ denote the first component. For the other components, the value OPEN means that the gaps may be open, and CLOSED means that they may be closed or never have occurred. At runtime, if a document is shown with open template gaps, these are treated as empty strings. For open attribute gaps, the entire attribute is removed. We need the gap presence information in the summary graphs to 1) determine where edges should be added when modeling plug operations, 2) model the removal of gaps

that remain open when a document is shown, and 3) detect that plug operations may fail because the specified gaps have already been closed.

This unfolding of summary graphs is explained more precisely with the following formalization:

$$\mathit{unfold}(SG) = \{d \mid \exists r \in R : SG, r \vdash t(r) \Rightarrow d \;\text{ where } SG = (R, T, S, P)\}$$

Here, $t(n)$ denotes the template with index n. The *unfolding relation*, $\Rightarrow$, is defined by induction in the structure of the XML template. For the parts that do not involve gaps the definition is a simple recursive traversal:

$$\frac{}{SG, n \vdash \mathit{str} \Rightarrow \mathit{str}}$$

$$\frac{SG, n \vdash \mathit{xml}_1 \Rightarrow \mathit{xml}'_1 \quad SG, n \vdash \mathit{xml}_2 \Rightarrow \mathit{xml}'_2}{SG, n \vdash \mathit{xml}_1\ \mathit{xml}_2 \Rightarrow \mathit{xml}'_1\ \mathit{xml}'_2}$$

$$\frac{SG, n \vdash \mathit{atts} \Rightarrow \mathit{atts}' \quad SG, n \vdash \mathit{xml} \Rightarrow \mathit{xml}'}{SG, n \vdash \texttt{<}\mathit{name}\ \mathit{atts}\texttt{>}\ \mathit{xml}\ \texttt{</}\mathit{name}\texttt{>} \Rightarrow \texttt{<}\mathit{name}\ \mathit{atts}'\texttt{>}\ \mathit{xml}'\ \texttt{</}\mathit{name}\texttt{>}}$$

$$\frac{}{SG, n \vdash \epsilon \Rightarrow \epsilon}$$

$$\frac{}{SG, n \vdash \mathit{name}\texttt{="}\mathit{str}\texttt{"} \Rightarrow \mathit{name}\texttt{="}\mathit{str}\texttt{"}}$$

$$\frac{SG, n \vdash \mathit{atts}_1 \Rightarrow \mathit{atts}'_1 \quad SG, n \vdash \mathit{atts}_2 \Rightarrow \mathit{atts}'_2}{SG, n \vdash \mathit{atts}_1\ \mathit{atts}_2 \Rightarrow \mathit{atts}'_1\ \mathit{atts}'_2}$$

For template gaps we unfold according to the string edges and template edges and check whether the gap may be open:

$$\frac{\mathit{str} \in S(n, g)}{(R, T, S, P), n \vdash \texttt{<[}g\texttt{]>} \Rightarrow \mathit{str}}$$

$$\frac{(n, g, m) \in T \quad (R, T, S, P), m \vdash t(m) \Rightarrow \mathit{xml}}{(R, T, S, P), n \vdash \texttt{<[}g\texttt{]>} \Rightarrow \mathit{xml}}$$

$$\frac{n \in \mathit{nodes}(P(g))}{(R, T, S, P), n \vdash \texttt{<[}g\texttt{]>} \Rightarrow \texttt{<[}g\texttt{]>}}$$

For attribute gaps we unfold according to the string edges, and check whether the gap may be open:

$$\frac{\mathit{str} \in S(n, g)}{(R, T, S, P), n \vdash \mathit{name}\texttt{=[}g\texttt{]} \Rightarrow \mathit{name}\texttt{="}\mathit{str}\texttt{"}}$$

$$\frac{n \in \mathit{nodes}(P(g))}{(R, T, S, P), n \vdash \mathit{name}\texttt{=[}g\texttt{]} \Rightarrow \mathit{name}\texttt{=[}g\texttt{]}}$$

Using a function *close* that removes all remaining gaps in an XML template, we now define the language of a summary graph by:

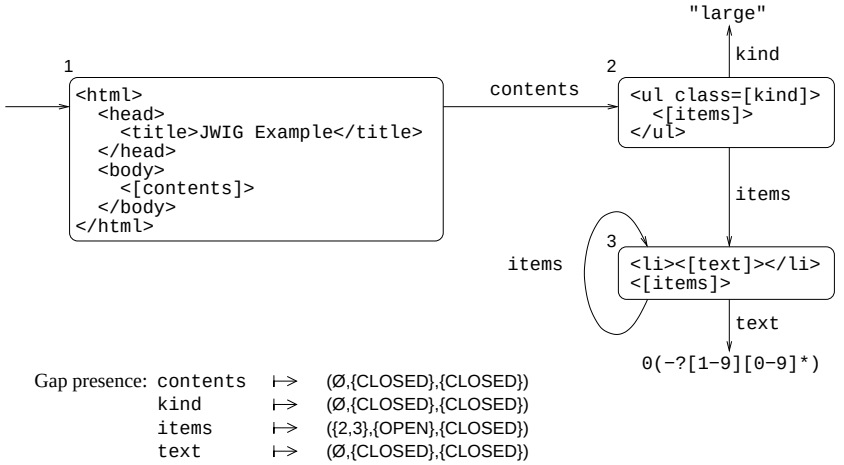
$$\mathcal{L}(SG) = \{\mathit{close}(d) \mid d \in \mathit{unfold}(SG)\}$$

Summary graphs for a given program form a lattice where the ordering is defined as one would expect:

$$(R_1, T_1, S_1, P_1) \sqsubseteq (R_2, T_2, S_2, P_2) \quad \Leftrightarrow$$
$$R_1 \subseteq R_2 \wedge T_1 \subseteq T_2 \ \wedge \ \forall n \in N, g \in G : S_1(n,g) \subseteq S_2(n,g) \wedge P_1(g) \sqsubseteq P_2(g)$$

where the ordering on gap presence maps is defined by componentwise set inclusion. This respects language inclusion: if $SG_1 \sqsubseteq SG_2$, then $\mathcal{L}(SG_1) \subseteq \mathcal{L}(SG_2)$, but the converse implication is false.

Continuing the previous example, the summary graph inferred for the XML value being shown to the client is:



It is relatively simple to define appropriate transfer functions and perform a standard monotone dataflow analysis for the JWIG language [3]. The most interesting example is the template plug operation, z **=** x **<[** g **=** y **]** where y is of type XML. This operation assigns to z a copy of x where y has been plugged into all g gaps. It is modeled by the following transfer function:

$$\begin{array}{l}(R_z, T_z, S_z, P_z) = \\ \quad (R_x, \\ \quad T_x \cup T_y \cup \{(n,g,m) \mid n \in \mathit{nodes}(P_x(g)) \ \wedge \ m \in R_y)\}, \\ \quad \lambda(m,h).S_x(m,h) \cup S_y(m,h), \\ \quad \lambda h.\mathsf{if}\ h{=}g\ \mathsf{then}\ P_y(h) \\ \qquad\qquad \mathsf{else}\ (p_x \cup p_y, \mathit{merge}(t_x,t_y), \mathit{merge}(a_x,a_y)) \\ \qquad\qquad\quad \mathsf{where}\ P_x(h) = (p_x,t_x,a_x)\ \mathsf{and}\ P_y(h) = (p_y,t_y,a_y))\end{array}$$

where

$$\mathit{merge}(\gamma_1,\gamma_2) = \mathsf{if}\ \gamma_1{=}\{\mathsf{OPEN}\} \vee \gamma_2{=}\{\mathsf{OPEN}\}\ \mathsf{then}\ \{\mathsf{OPEN}\}\ \mathsf{else}\ \gamma_1 \cup \gamma_2.$$

The tuples (R_x, T_x, S_x, P_x) and (R_y, T_y, S_y, P_y) denote the summary graphs that are associated to x and y at the entry point, and (R_z, T_z, S_z, P_z) is the summary

graph for z at the exit point. The roots in the resulting graph are those of the x graph since it represents the outermost template. The template edges become the union of those in the two given graphs plus a new edge from each node that may have open gaps of the given name to each root in the second graph. The string edge sets are simply joined without adding new information. For the gaps that are plugged into, we take the gap presence information from the second graph. For the other gaps we merge the information appropriately.

The set of regular expressions describing computed string values are determined through a separate dataflow analysis. Thus we obtain summary graphs that conservatively describe all computed XML values at each program point; for more details see [3]. The worst-case complexity for this algorithm is $O(n^6)$, where n is the size of the program.

4 Static Guarantees in JWIG

Based on the inferred summary graphs, various static guarantees can be issued.

First we must deal with a self-inflicted problem stemming from the liberal gap-and-plug mechanism. We need to know that whenever a gap is being plugged, it is actually present in the XML value. However, this information is directly available in the gap presence map component, and a trivial inspection suffices. Note that in the special case of constructors, corresponding to `[[ <l><[g]></l> ]] <[ g=`X` ]`, this property trivially holds.

Second, we need to validate the XML values being generated. This is dependent of the XML language in question, which must first be specified. We could use ordinary DTDs for this purpose, but have instead chosen the XML schema language DSD2 [9], which is a further development of DSD [8]. We have a general algorithm that given a summary graph SG and a DSD2 schema can verify that every document in $\mathcal{L}(SG)$ validates according to the schema. The DSD2 schema for XHTML is more comprehensive than most others, since it specifies correct formats for attribute values that are URIs and includes several context-sensitive requirements that are only stated as comments in the official DTD.

The final analysis is specific to XHTML and verifies that the form data expected by the server is actually present in the last document being shown to the client.

These analyses are fully specified in [3]. They have rather high worst-case complexities, but are in practice able to handle realistic program. The following table shows statistics for some small to largish benchmarks.

Name	Lines	Templates	Largest Graph	Total Time
`Chat`	80	4	(2,6)	5.37
`Guess`	94	8	(2,4)	7.15
`Calendar`	133	6	(5,14)	7.03
`Memory`	167	9	(7,13)	9.72
`TempMan`	238	13	(11,22)	7.72
`WebBoard`	766	32	(9,22)	9.77
`Bachelor`	1,078	88	(47,107)	115.64
`Jaoo`	3,923	198	(33,93)	36.00

To indicate the scale of each benchmark, we give the number of lines of code and the number of template constants. We also show the size of the largest summary graph computed for each benchmark by indicating the number of nodes (reachable from the roots) and the number of edges. The time, measured in seconds, is the total for inferring summary graphs and performing the three subsequent analyses.

5 Analyzing Deconstruction

The present version of the JWIG language does not contain any mechanism for deconstruction of XML values. However, the summary graph analysis can—with simple modifications—easily handle this.

We extend the JWIG language with a notion of deconstruction based on XPath [4] that generalizes most other proposals. Since we are working on XML values containing gaps, we get two variations.

The *select* expression looks like x`>[`*path*`]`, where x is an XML value and *path* is a location path. The result is an array of XML values corresponding to those subtrees that are rooted by the elements of the computed node set. The XML value x is first closed, that is, all gaps are removed, as was the case before `show` operations. Other deconstruction mechanisms can clearly be obtained as special cases. For example, the pattern matching of XDuce corresponds to writing a selector path for each case and trying them out in turn.

By exploiting the gap mechanism, we can also introduce a complementary operation that replaces parts of an XML values by gaps. The *gapify* expression looks like x`>[`*path*`=`g`]`, where x is an XML value, *path* is a location path, and g is an identifier. The result is a copy of x where the subtrees rooted by the computed node set are replaced by gaps named g. Again, `x` is first closed. If `x` is the XML value:

```
<html>
  <head><title>JWIG example</title></head>
  <body>
    <ul class="large">
      <li>0</li> <li>1</li> <li>2</li> <li>3</li>
    </ul>
  </body>
</html>
```

then the result of `x>[//li[text()>'0']` is the following XML array with three entries:

```
{ [[ <li>1</li> ]], [[ <li>2</li> ]], [[ <li>3</li> ]] }
```

and the result of `x>[//li[text()>'0' = g]` is the "negative image" in form of the XML value with gaps named `g` in place of the selected subtemplates:

```
<html>
  <head><title>JWIG example</title></head>
```

```
  <body>
    <ul class="large">
      <li>0</li> <[g]> <[g]> <[g]>
    </ul>
  </body>
</html>
```

The summary graph analysis can be extended with transfer functions for select and gapify. In [3], the close operation only occurs in connection with `show` operations. However, because of our extensions with the select and gapify we now need to model close operations separately. We must remove all gaps that might be open according to the gap presence map. To model that a template gap is removed, one simply adds a template edge to a node with an empty template. For attribute gaps, we need a small modification of the string edge component of the summary graph structure:

$$S : N \times G \to REG \times \{\div\}$$

The new element $\div$ represents the possibility that the designated attribute might be removed. The definition of the unfolding relation is extended with a rule describing this meaning:

$$\frac{\div \in S(n, g)}{(R, T, S, P), n \vdash name\texttt{=[}g\texttt{]} \Rightarrow \epsilon}$$

To model that an attribute gap is removed in a close operation, we just add $\div$ to the appropriate string edge. The gap presence map of the result of a close operation maps all gaps to $(\emptyset, \{\mathsf{CLOSED}\}, \{\mathsf{CLOSED}\})$.

The core observation when modeling select and gapify operations is in both cases that an XPath selector path can be evaluated symbolically on a summary graph. The resulting node set is represented abstractly by assigning a status to each element in all templates assigned to nodes in the summary graph. The possible status values are:

- *all*: every occurrence of this element belongs to the node set in every unfolding of the summary graph;
- *some*: at least one occurrence of this element belongs to the node set in every unfolding of the summary graph;
- *definite*: the conditions for both *all* and *some* are satisfied;
- *none*: no occurrences of this element belong to the node set in any unfolding of the summary graph;
- *don't know*: none of the above can be determined.

This forms a 5-valued logic reminiscent of the logic used when analyzing validity with respect to DSD2 schemas [3]. Based on these status values, it is straightforward conservatively to compute summary graphs for the results of select and gapify.

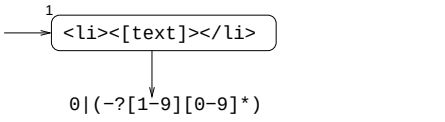
For a select expression, all subtemplates whose root elements do not have status *none* are added to the summary graph, inherit all relevant edges, and are made the only root nodes.

For a gapify expression, all subtemplates whose root elements do not have status *none* are replaced by a gap named g. If the status is *some* or *don't know*, the new gap will have a template edge to a copy of the old subtemplate. The gap presence map of the new summary graph will be

$$\lambda h.\mathsf{if}\ h{=}g\ \mathsf{then}\ (\mathit{hits}, \mathit{any}, \{\mathsf{CLOSED}\})\ \mathsf{else}\ (\emptyset, \{\mathsf{CLOSED}\}, \{\mathsf{CLOSED}\})$$

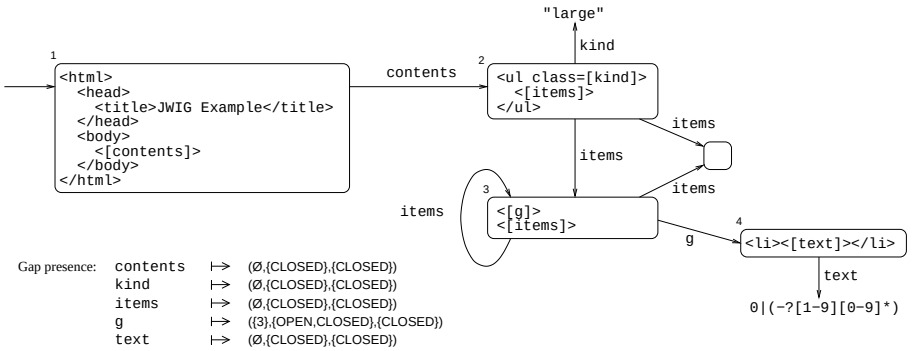
where *hits* is the set of all template nodes containing an element with status different from *none*, and *any* is {OPEN} if there is an element with status *definite* or *some*, {CLOSED} if all elements have status *none*, and {OPEN, CLOSED} otherwise.

Continuing the example from Section 4, the result of the select expression is described by



Gap presence: `text` ↦ (Ø,{CLOSED},{CLOSED})

and the result of the gapify expression by



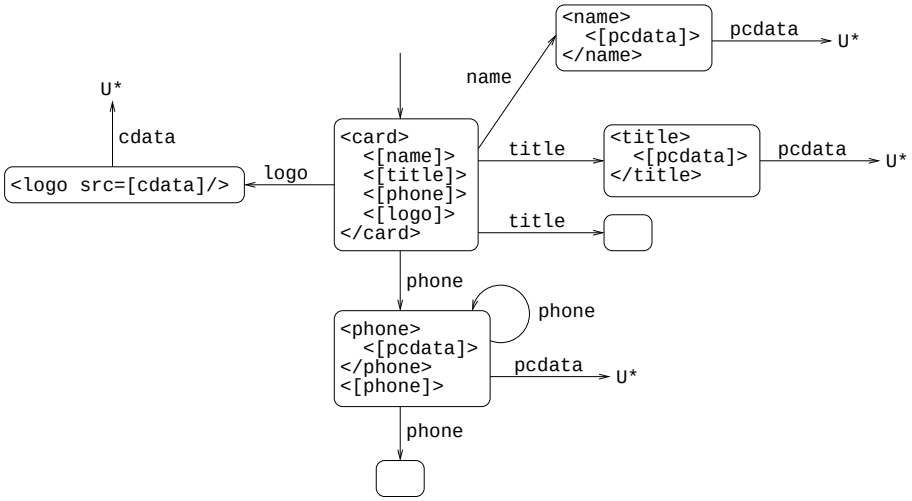
We are currently implementing the select and gapify operations and the associated extensions of the summary graph analysis in the JWIG system to test the analysis precision and performance in practice.

Deconstruction is mainly relevant for XML values that are imported from external sources. To obtain non-trivial analyses, we need to obtain summary graph descriptions of such values. In practice this will by done by performing automatic translations from DTDs or DSD2 schemas. We believe that such translations can be made sufficiently precise. As an example, consider the following DTD:

```
<!ELEMENT card (name,title?,phone+,logo)>
```

```
<!ELEMENT name (#PCDATA)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT logo EMPTY>
<!ATTLIST logo src CDATA #REQUIRED>
```

It is exactly captured by the following summary graph:



where all gaps are closed and `U` is the set of all Unicode characters. For a richer schema language as DSD2 the translation will of course become more complex, and it will in some cases be necessary to perform conservative approximations.

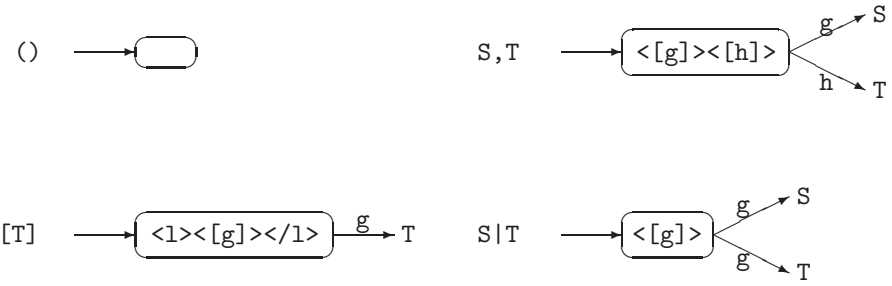
6 Regular Expression Types

Summary graphs turn out to have the same expressive power as the regular expression types of XDuce [5]. To be exact, this comparison only holds for a restricted version of summary graphs. Since XDuce does not support attributes, those must be left out. Also, summary graphs allows for restrictions on character data appearing in element contents, which is also not supported by XDuce.

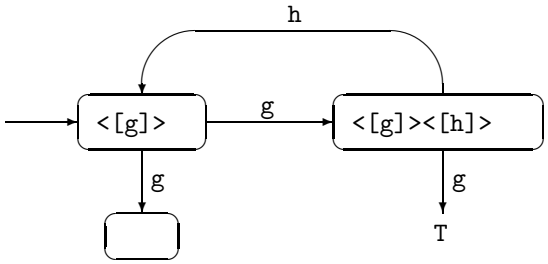
Regular expression types are essentially solutions to recursive equations using the operators `()` (the empty value), `l[T]` (singleton element), `S|T` (union), and `S,T` (sequencing). For example, the derived operator `T*` is defined by the equation:

```
X = T,X | ()
```

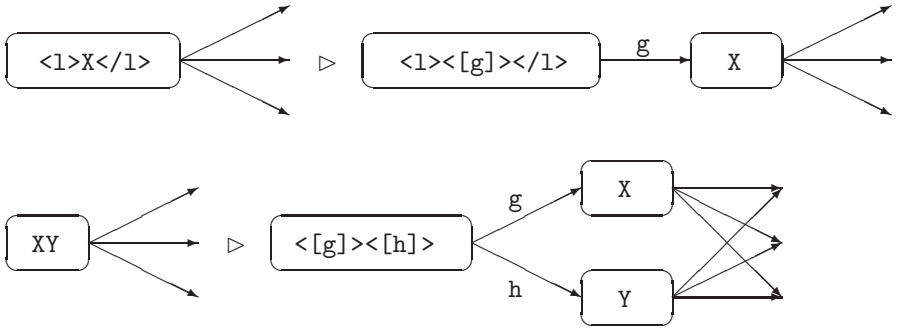
A regular expression type defines a set of XML values corresponding to all finite unfoldings. It is now a simple matter to build inductively a summary graph that defines the same set of XML values. The four operators are modeled by summary graphs as follows:



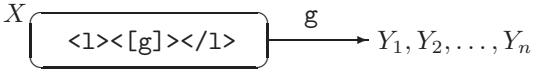
All gaps are closed in these summary graphs. An edge to a variable is modeled by an edge to the root node of the summary graph corresponding to its right-hand side. For example, the derived summary graph for `T*` is:



Note that the sequencing operator is associative as required in XDuce. The inverse translation is equally straightforward, but requires that the summary graph is first *normalized*. First, all open gaps are translated into closed ones by adding a template edge to an empty template. Second, all non-empty template constants are decomposed into one of the forms `<l><[g]></l>` or `<[g]><[h]>`. This is done by repeatedly applying the rewritings sketched by:



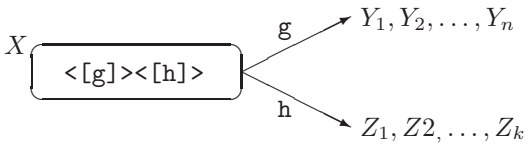
Given a normalized summary graph, we first assign a type variable to each node. Then for each node of the form:



we define the type equations:

X = `l[`Y`]`
$Y = Y_1$ `|` $\ldots$ `|` Y_n

for each node of the form:



we define the type equations:

$X = Y$ `,` Z
$Y = Y_1$ `|` $\ldots$ `|` Y_n
$Z = Z_1$ `|` $\ldots$ `|` Z_k

and for empty nodes:

we define the type equation:

X = `()`

Finally, for the root nodes R_1, $\ldots$, R_n we define the type equation:

$R = R_1$ `|` $\ldots$ `|` R_n

and the type R is the final result of the translation. These two translations demonstrate the close relationship between our approach and that of XDuce.

Our analysis is thus also able to infer regular expression types for programs that dynamically construct XML values. This result does not directly apply to XDuce, since we infer different types for variables at each program point. It is, however, possible to subsequently verify a subtype relationship between each declared and inferred type. The resulting type reconstruction algorithm is of course not complete with respect to the type rules of XDuce [5]; in fact, their abilities to accept programs are most likely incomparable.

Note that the summary graph lattice contains extra structure in the form of the root sets and the gap presence maps. Also, the lattice order is more fine-grained than the language inclusion used as subtyping in XDuce. All this constitutes a scaffolding that is required during analysis but is not needed to express the final results. Note also that the analysis uses a kind of constructor polyvariance, since constructors in the summary graph are represented once for each invocation site. We believe that similar ideas would need to be developed for XDuce, if flow-sensitive type reconstruction were to be attempted directly.

7 Conclusion

We have presented the lattice of summary graphs as a convenient means for abstracting sets of XML values during dataflow analyses of programs that dynamically construct XML documents. Summary graphs have been used in the fully implemented JWIG language, and we have indicated the applicability for other scenarios.

References

[1] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE '01*, June 2001.

[2] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. The `<bigwig>` project. *ACM Transactions on Internet Technology*, 2(1), 2002.

[3] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. Technical Report RS-02-11, BRICS, March 2002. Submitted for journal publication.

[4] James Clark and Steve DeRose. XML path language, November 1999. W3C Recommendation. `http://www.w3.org/TR/xpath`.

[5] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proc. 3rd International Workshop on the World Wide Web and Databases, WebDB '00*, volume 1997 of *LNCS*. Springer-Verlag, May 2000.

[6] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *Proc. 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '01*, January 2001.

[7] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming, ICFP '00*, September 2000.

[8] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. The DSD schema language. *Automated Software Engineering*, 2002. Kluwer. Preliminary version in *Proc. 3rd ACM SIGPLAN-SIGSOFT Workshop on Formal Methods in Software Practice, FMSP '00*.

[9] Anders Møller. Document Structure Description 2.0. In preparation, 2002.

[10] Anders Sandholm and Michael I. Schwartzbach. A type system for dynamic Web documents. In *Proc. 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '00*, January 2000.

Recent BRICS Report Series Publications

RS-02-24 **Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach.** ***Static Analysis for Dynamic XML*****. May 2002. 13 pp.**

RS-02-23 **Antonio Di Nola and Laurenţiu Leuştean.** ***Compact Representations of BL-Algebras*****. May 2002. 25 pp.**

RS-02-22 **Mogens Nielsen, Catuscia Palamidessi, and Frank D. Valencia.** ***On the Expressive Power of Concurrent Constraint Programming Languages*****. May 2002. 34 pp.**

RS-02-21 **Zoltán Ésik and Werner Kuich.** ***Formal Tree Series*****. April 2002. 66 pp.**

RS-02-20 **Zoltán Ésik and Kim G. Larsen.** ***Regular Languages Definable by Lindström Quantifiers (Preliminary Version)*****. April 2002. 56 pp.**

RS-02-19 **Stephen L. Bloom and Zoltán Ésik.** ***An Extension Theorem with an Application to Formal Tree Series*****. April 2002. 51 pp. To appear in Blute, editor,** ***Category Theory and Computer Science: 9th International Conference*****, CTCS '02 Proceedings, ENTCS, 2002 under the title** ***Unique Guarded Fixed Points in an Additive Setting*****.**

RS-02-18 **Gerth Stølting Brodal and Rolf Fagerberg.** ***Cache Oblivious Distribution Sweeping*****. April 2002. To appear in** ***29th International Colloquium on Automata, Languages, and Programming*****, ICALP '02 Proceedings, LNCS, 2002.**

RS-02-17 **Bolette Ammitzbøll Madsen, Jesper Makholm Nielsen, and Bjarke Skjernaa.** ***On the Number of Maximal Bipartite Subgraphs of a Graph*****. April 2002. 7 pp.**

RS-02-16 **Jiří Srba.** ***Strong Bisimilarity of Simple Process Algebras: Complexity Lower Bounds*****. April 2002. 33 pp. To appear in** ***29th International Colloquium on Automata, Languages, and Programming*****, ICALP '02 Proceedings, LNCS, 2002.**

RS-02-15 **Jesper Makholm Nielsen.** ***On the Number of Maximal Independent Sets in a Graph*****. April 2002. 10 pp.**

RS-02-14 **Ulrich Berger and Paulo B. Oliva.** ***Modified Bar Recursion*****. April 2002. 23 pp.**