Towards static type checking of Web query language

Sacha Berger, François Bry
Ludwig-Maximilians Universität, Institut für Informatik
Oettingenstr. 67, D-80538 München
sacha.berger@ifi.lmu.de bry@ifi.lmu.de

Zusammenfassung

This article reports on a research project investigating the following two complementary issues: (1) improving how the structure of XML and HTML can be specified, (2) using structure specification (of XML and HTML documents) for static type checking of Web (and Semantic Web) query programs. The first step towards this goal is to provide a schema language like DTD, XML Schema or Relax-NG with better support of graph structured data.

1 Introduction

The schema languages DTD, XML Schema[Con01] and Relax-NG[CM01] are used to specify the type and structure of XML and WWW documents. Although documents and data on the Web often represent graph structures based on references (expressed using eg. ID and IDREF attributes, RDF triples or Hyperlinks), the above mentioned schema languages can only specify tree structures. This article first introduces a novel schema language for specifying XML documents and semi structured data called Regular Rooted Graph Grammars – also referred to as R^2G^2. R^2G^2 gives rise to specify graph shaped XML and semistructured related data types possibly with (directed or undirected) cycles. R^2G^2 is an extension of regular tree grammars (that underly many Web schema languages).

The second issue addressed is using a regular rooted graph grammar as a type language for static type checking of web query or transformation programs. Static type checking consists of (1) inferring data types for all program constructs and (2) detecting type inconsistencies, both at compile time, ie. before query evaluation. The main advantage of static type checking is, as its proponent Robin Milner said, that "well-typed programs do not go wrong"[Mil74]. The Web query language Xcerpt is used as test bed for using Regular Rooted Graph for static type checking.

2 R^2G^2introduced on an example

XML documents are serialisations of tree, or graph, structured data i.e. representations in a linear, or textual, form using parenthesis, the opening and closing tags. Non-tree XML documents play an important role in practice. An example of such an XML document is given below, in which children elements are used to describe (possibly symmetric) friendship relationships. Recall that DTD, XML Schema and Relax-NG cannot express the structure of graph structured XML documents. R^2G^2grammars have been designed to model graph structured documents (possibly containing directed or undirected cycles).

The following figure is (part of) an XML document expressed in Xcerpt syntax. used is actually the Xcerpt data term[SB04] syntax:

```
addressbook{ id1@card{ name{"Snoopy"}, friends[^id1,^id2], phone{"9310"} },
             id2@card{ first{"Charly"}, last{"Brown"}, phone{"9316"} }
           }
```

According to [SB04], such an element is called in the following a *dataterm.* The example represents two addresses (called `card`) of an `addressbook`. The `card` elements are defining occurrences [1] of elements that may be referenced using the unique identifiers at the left of the @ signs (as seen in the friends element of the first card). The use of square brackets indicate that the order of the sub elements is relevant (as always in XML), the curly braces indicate the irrelevance of the sub elements order (as common in databases, an extension of XML introduced by Xcerpt).

The following example is a R^{2}G^2 specifying a structure, or schema, of addressbooks like the previous one. Due to lack of space, R^{2}G^{2}will informally be introduced on this example:

```
elmtype AddressBook = addressbook{ @Card* };
elmtype Card        = card{{ Name,
                             friends[ ^Card+ ]?, Contact* }}
elmtype Contact     = phone{ String } | /[Ee]?mail/{ String };
elmtype Name        = name{ String };
```

The grammar consists of 6 rules similar to rules of regular tree languages. The first rule, given on the first line, defines an element type named `AddressBook` with label `addressbook`. In contrast to DTD and similar to Relax-NG and XML-Schema, the element label and the type name may not be the same or related to each other, so that the same element label can be used with different structure in different places in a document.

The right hand side of a rule may be an element type definition term (as seen in the 1st, 2nd and 4th rule) or a disjunction of type definition terms (as seen in the 3rd rule). The content of an element type definition term is a regular expression of element type names or element type definition terms. Elements of type `Card` e.g., contain an element of type `Name`, an optional element labelled `friends` and an arbitrary number of elements of type `Contact` as children. The element type definition term of this rule is enclosed by double braces, indicating that the content definition is incomplete, that therefore elements conforming to this definition may contain further arbitrary elements. This is a convenience feature not used in current XML schema languages, but useful for modelling partly arbitrary content models[2].

Content models enclosed by square brackets (used in the 2nd rule) are ordered content models: only ordered element instance's content can conform to such a content model[3]. Content models enclosed by curly braces are content models with irrelevant order, instances of those types may either be ordered or unordered.

It is possible to model elements with the same type name and content model, but with different labels: this is accomplished by using a string regular expressions (enclosed by "/ßlashes) at the position of the label in the type definition term (see the 3rd rule).

As in the 1st rule's type definition term the `Card` is prefixed with an `@`, child elements of type `Card` must have an an unique identifier to be referrable. The modelling of elements with unique identifier is a feature also available in XML Schema, DTD and Relax-NG: they provide the ability to model ID attributes, which is of comparable expressiveness as Xcerpt's defining occurrence mechanism. In the second rule, a reference type (`^Card`) is part of the definition of the friends element's content model: elements conforming to this type definition term may only contain references to elements of type Card. In contrast to the typed reference mechanism in R^{2}G^2, current schema languages only provide the ability to model references to arbitrary ID attributes. Hence R^{2}G^{2}provides a better way to model graph shaped XML documents.

Further Features of R^{2}G^2 not mentioned above (for space reason) are (1) *content model rules*, used to provide names for parts of content models, (2) *alias rules*, used to provide alternative names to the same type, (3) *scalar types*, as `String` in the former example, (4) *label types*, to provide names to sets of alternative labels (or label regular expressions) and (5) *local definition and modules.*

3 Validation of Data terms with R^{2}G^2

Validation, commonly referred to as acceptance test in automata theory, means to test if a "word" is accepted by an automaton, ie. if the word is part of a language associated to the

[1]The term *defining occurrence* is introduced in [SB04]
[2]The expressiveness of R^{2}G^{2}is not enhanced by this extension, cf. section 3.
[3]A precise definition of the matching is given in section 3.

automaton or its corresponding grammar. Here an XML document is the word to test for acceptance under the grammar (or its corresponding automaton) represented by the schema, with the root indicating the non terminal to use as start symbol.

The approach to validation presented here, is an extension of tree automaton-based validation [MLM01], more precisely, a non deterministic bottom up approach of regular tree grammar based validation [MLM01]. Validation is done using a recursive function returning a set of non terminal symbols for validated elements. Each non terminal represents one possible derivation for the element under the given grammar, an empty set indicates an invalid element.

$$\begin{aligned} \mathit{validate}(\mathbf{l}[\mathbf{t_1},\cdots,\mathbf{t_n}],G) &= \{X | N_1 \cdots N_n \in L(r_c) \wedge \texttt{elmtype X =}\mathbf{l}[r_c] \in G\} \\ &\mathit{where}\quad N_i \in \mathit{validate}(\mathbf{t_i},G) \end{aligned}$$

This approach needs some extension to cope with the following R^2G^2features: (1) incomplete or double brace content models, (2) label regular expressions, (3) unordered or curly brace content models and (4) handling of typed references.

$$\begin{aligned} \mathit{validate}(\mathbf{l}\{\mathbf{t_1},\cdots,\mathbf{t_n}\},G) &= \{X | W \in L(r_c) \\ &\quad \wedge W \in \mathit{permutation}(N_1 \cdots N_n) \\ &\quad \wedge \mathbf{l} \in L(r_l) \\ &\quad \wedge \texttt{elmtype X =} r_l\{r_c\} \in G'_{+ALL} \\ &\mathit{where}\quad N_i \in \mathit{validate}(\mathbf{t_i},G) \end{aligned}$$

$$\begin{aligned} \mathit{validate}(\mathbf{l}[\mathbf{t_1},\cdots,\mathbf{t_n}],G) &= \{X | W \in L(r_c) \\ &\quad \wedge(\ (W \in \mathit{permutation}(N_1 \cdots N_n) \\ &\qquad\quad \wedge \mathbf{l} \in L(r_l) \\ &\qquad\quad \wedge \texttt{elmtype X =} r_l\{r_c\} \in G'_{+ALL}) \\ &\qquad \vee(W = N_1 \cdots N_n \\ &\qquad\quad \wedge \mathbf{l} \in L(r_l) \\ &\qquad\quad \wedge \texttt{elmtype X =} r_l[r_c] \in G'_{+ALL}))\} \\ &\mathit{where}\quad N_i \in \mathit{validate}(\mathbf{t_i},G) \end{aligned}$$

$$\mathit{validate}(\mathbf{id}@\mathbf{l}\alpha\mathbf{t_1},\cdots,\mathbf{t_n}\beta,G) = \{@X | X \in \mathit{validate}(\mathbf{l}\alpha\mathbf{t_1},\cdots,\mathbf{t_n}\beta,G)$$

$$\mathit{validate}(\hat{}\,\mathbf{id},G) = \{@X | X \in \mathit{validate}(\mathit{lookup}(\mathbf{id}),G)$$

The modified grammar G'_{+ALL} expresses the same as G, but does not use double braces, ie. incomplete content models. Rules with incomplete content model are replaced by rules with complete content model, by (1) changing the double braces into single braces, while maintaining the order type of the braces, (2) adding `ALL*` as first atom of the regular expression and (3) modifying the atoms of the content regular expression as follows:

1. replacing all `a*` atoms by `(a,ALL*)*`
2. replacing all `a+` atoms by `(a,ALL*)+`
3. replacing all `a?` atoms by `(a,ALL)?`
4. replacing all other atoms r_a by r_a`,ALL*`

The rule for the `ALL` type is

```
elmtype ALL = /.+/{ALL*} | /.+/[ALL*] | String
```

For example, the 2nd grammar rule of section 2 can be rewritten to

```
elmtype CARD = card{ ALL*, NAME,
                     ALL*, (friends[^CARD], ALL)?,
                     (Contact, ALL)* }
```

The variables α and β (in the 3rd rule) has been used to capture square and curly braces, as they are treated the same way. The rules with incomplete content models have been rewritten using the ANY type and the ANY type is added to the set of rules. The ANY type captures arbitrary content[4].

[4]The possibility to rewrite any grammar G to a corresponding schema G'_{+ALL} indicates, that double braces do not extend the expressiveness of R^2G^2.

Label regular expressions are processed by checking the containment of the data term's label against the language defined by the label regular expression or the type definition term's label interpreted as regular expression.

Unordered and ordered data terms, ie. terms like `name{"Snoopy"}` (unordered) and `friends[^id1, ^id2]` (ordered), are handled in the first, respectively in the second function definition case. Note that in essence the right hand side of the first case is contained in the second case, reflecting that ordered data terms may also match with unordered content model specifications. The containment test for unordered terms is achieved by checking the containment of at least one permutation of the unordered content with respect to the regular expression of a given content model.

Validation of defining occurrences of identifiers and of references is handled by the 3rd, respectively the 4th function definition case. For this purpose a *lookup* function, characteristic to each document validated, is used. An implementation could rely on a hashtable, initialised by an additional pass on the document.

Note that a naive implementation of the algorithm explained above, may not terminate when validating circular structures. Termination can easily be ensured by using lazy evaluation or by labelling validated nodes with their matching types, this means by relying on the set of types used as label in preference to calculating this set again.

4 $\mathrm{R}^2\mathrm{G}^2$**for static type checking of web query languages**

Static type checking means detecting type inconsistencies at compile time, ie. before query or program evaluation. Essential properties of type checking are decidable and efficiently computability. For these purposes, and in contrast to model checking, type checking works on approximations of the values a program may use or produce, more precisely on superset approximations of the actual set of possible values a specific program construct may compute[Mil74, Car97, Pie02]. All constructs of a language that are to be type checked, must therefore be typeable, this means, such a superset approximation must be assignable to those constructs. The art of finding a good type system is, to have the type approximation as precise as possible, and therefore to find as many errors as possible, while being as general as necessary to still ensure termination and mostly polynomial complexity for the type checking process.

Traditionally all typable constructs of a program had to be explicitly type annotated manually, but modern type systems provide the ability to automatically deduce type annotation based on type inference[Pie02]. This brings the freedom of omitting or providing the type annotation as wished by the programmer. The main advantage of static type checking is, as its proponent Robin Milner said, that "well-typed programs do not go wrong"[Mil74]. General principles of a type system for Web query languages are presented in the following.

The idea of type checking applied to Web query languages means, to detect prior to execution, that query programs (1) do not query the desired data, and that they (2) produce undesired output. Common to many query languages are constructs to (1) query given data or Web sites, and (2) construct results based on the queried information. Language components used for querying usually restrict structure and values of expected results by filtering and pruning elements of the queried data. The schema of elements in the result set can be expressed using $\mathrm{R}^2\mathrm{G}^2$. Language components for the construction of answers to a query program usually provide a template-like mechanism to rearrange and integrate the data collected using query constructs. The schema of elements constructed according to the template and the type of the queried data can also be expressed using $\mathrm{R}^2\mathrm{G}^2$. According to the specialities of a concrete web query language, further constructs may be of relevance for typing.

Based on the two language component classes that may be typed, two classes of errors exist:

- Query terms may be ill-typed, because the type of the queried data or web site has an empty intersection with the type associated to the query result set (this means, that a query may never return anything when applied to a valid data base)
- Construct terms may be ill-typed, because the type of their result set is not contained in the type of the expected result (this means, the constructed results may have invalid

structure or content with respect to a result type annotation)

Note, that therefore it is necessary to (1) calculate the intersection of two data types, (2) that the inclusion test of a type in another type is still decidable, and that (3) the test of emptiness of a type is decidable. All three properties should be achievable with R^2G^2(not proved by now).

Based on principles previously presented, a type system for the Web query language Xcerpt based on $R^2G^2$is currently being developed as test bed. As there exists already a regular tree grammar based approach of typing Xcerpt [AW03], this will be used and extended to provide type checking for Web queries querying and constructing tree and graph structured data.

5 Conclusion

This article presented R^2G^2, a grammar formalism for specifying schemas of graph structured XML documents and semi structured data. A validation algorithm based on non deterministic tree automaton techniques [MLM01] is presented. Further work needs to be done, to prove some set theoretical properties of R^2G^2's needed for static type checking. Based on R^2G^2 a type system for Xcerpt will be conceived.

Literatur

[AW03] W. Drabent A. Wilk. On Types for XML Query Language Xcerpt. In François Bry, Nicola Henze, and Jan Maluszynski, editors, *PPSWR*, volume 2901 of *Lecture Notes in Computer Science*, pages 128–145. Springer, 2003.

[Car97] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.

[CM01] James Clark and Makoto Murata. *RELAX NG Specification.* http://relaxng.org/spec-20011203.html, 2001. ISO/IEC 19757-2:2003.

[Con01] W3 Consortium. XML Schema Part 0: Primer. W3C Recommendation, 2001. http://www.w3.org/TR/xmlschema-0/.

[Mil74] Robin Milner. Fully Abstract Models of Typed lambda-Calculi. *Theoretical Computer Science*, 4:1–22, 1974.

[MLM01] M. Murata, D. Lee, and M. Mani. “Taxonomy of XML Schema Languages using Formal Language Theory”. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[SB04] Sebastian Schaffert and François Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Proceedings of Extreme Markup Languages 2004, Montreal, Quebec, Canada (2nd–6th August 2004)*, 2004.