Type-based XML Processing in Logic Programming

Jorge Coelho[1] and Mário Florido[2]

[1] Instituto Superior de Engenharia do Porto
Porto, Portugal
jcoelho@dei.isep.ipp.pt
[2] University of Porto, DCC-FC & LIACC
Porto, Portugal
amf@ncc.up.pt

Abstract. In this paper we propose a type-based framework for using logic programming for XML processing. We transform XML documents into terms and DTDs into regular types. We implemented a standard type inference algorithm for logic programs and use the types corresponding to the DTDs as additional type declarations for logic programs for XML processing. Due to the correctness of the type inference this makes it possible to use logic programs as an implicitly typed processing language for XML with static type (in this case DTDs) validation. As far as we know this is the first work adding type validation at compile time to the use of logic programming for XML processing.

1 Introduction

In this paper, we present the design and implementation of a statically typed logic programming language specialized for XML processing [3]. One of the nice features of XML is the static typing of documents provided by DTDs or XML Schema. Some programming languages go beyond types for documents and provide static type checking for programs for XML processing. One example is the functional language XDuce [9].

In this paper we present an *implicitly typed* version of *pure Prolog* and apply it to the domain of XML processing. The approach used is the following:

- we translate XML elements and documents to Prolog terms whose syntax is specified by a DTD;
- we translate DTDs to *regular types* [6]. Regular types are the most used type language for typing logic programs [13, 24, 23, 7, 6, 8, 11];
- we use implicitly typed *pure Prolog* clauses as rules for expressing element and data transformations;

[3] The framework described in this paper is available on the WWW at http://www.dei.isep.ipp.pt/∼jcoelho/x-prolog/

– we implemented a type inference module which, given a predicate which relates two or more terms representing XML documents checks at compile time if the predicate is *well-typed* (i.e. if the transformation defined by the predicate respects the DTDs of the XML documents involved)

The novel feature of our framework is the use of regular type inference for logic programming to guarantee type safety at compile time. This feature points out the similarities between DTDs and *regular types* for logic programming. We now give a simple example of the kind of translations involved in our work.

Given the XML document

```
<teachers>
   <name>Jorge Coelho</name>
   <office>403</office>
   <email>jcoelho@isep.ipp.pt</email>
   <name>Mario Florido</name>
   <office>202</office>
</teachers>
```

and the DTD

```
<!ELEMENT teachers (name,office,email?)*>
<!ELEMENT name #PCDATA>
<!ELEMENT office #PCDATA>
<!ELEMENT email #PCDATA>
```

our program builds the pure Prolog term

```
teachers([(name(‘‘Jorge Coelho’’),
           office(‘‘403’’),
           email(‘‘jcoelho@isep.ipp.pt’’)),
           (name(‘‘Mario Florido’’),
           office(‘‘202’’))])
```

and the regular type

$$\tau_1 \rightarrow \{\mathit{teachers}(\tau_2)\}$$
$$\tau_2 \rightarrow \{\mathit{nil}, .(\tau_3, \tau_2)\}$$
$$\tau_3 \rightarrow \{\tau_4, \tau_5\}$$
$$\tau_4 \rightarrow \{(\mathit{name(string)}, \mathit{office(string)}, \mathit{email(string)})\}$$
$$\tau_5 \rightarrow \{(\mathit{name(string)}, \mathit{office(string)})\}$$

We follow Dart and Zobel ([6]) in the syntax of regular types. A *regular type* is defined by a set of type rules of the form $\tau \rightarrow T$ where τ is a type symbol and T a set of type terms. The rule $\tau \rightarrow T$ should be read as τ is defined by any member of T. Note that T represents a union of types. In the previous example τ_2 is the usual type for a list of elements of type τ_3 (as usual in logic programming, we use

the functor . for the list constructor). Note that τ_3 is τ_4 or τ_5. This disjunction comes from the optional operator ? in the corresponding DTD.

In this paper, we assume that the reader is familiar with XML ([22]) and logic programming [10]. We start in Section 2 with a brief overview of XML. Then, in Section 3, we present the translation from XML to Prolog terms. In Section 4 we show the relation between DTDs and regular types. In Section 5 we present an example of the use of pure Prolog for XML transformation. We then give an overview of the implementation and then we present the related work. Finally, we conclude and outline the future work.

2 XML

XML is a meta-language useful to describe domain specific languages for structured documents. Besides its use in the publishing industry XML is now the standard interchange format for data produced by different applications. An XML document is basically a tree structure. There are two basic types of content in a document: elements and plain text. An element consists of a start tag and an end tag which may enclose any sequence of other content. Elements can be nested to any depth and an XML document consists of a single top-level element (the root of the tree) containing other nested elements. For example, the next XML document could be used by a specific address book application:

```
<addressbook>
   <name>John</name>
   <address>London</address>
   <phone>
      <home>12345678</home>
      <mobile>87654321</mobile>
   </phone>
   <email>john@mailserver.uk</email>
   <name>Rita</name>
   <address>Copacabana</address>
   <address>Rio de Janeiro</address>
   <phone>
      <home>12457834</home>
   </phone>
</addressbook>
```

2.1 Document Type Definition (DTD)

A powerful feature of XML is the possibility to create a Document Type Definition, that is, a way to define rules that determine the validity of a document. The example bellow shows a DTD for the address book:

```
<!ELEMENT addressbook (name,address+,phone?,email?)*>
<!ELEMENT name, (#PCDATA)>
```

```
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (home,mobile*)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT home (#PCDATA)>
<!ELEMENT mobile (#PCDATA)>
```

This DTD tells that this kind of address book has the root element < addressbook > and four sub elements <name> <address> <phone> and <email>. It also tells that the <name></name> tags must exist, the <address></address> tags must appear at least once and may appear several times (in a sequence of address lines). The <phone></phone> and <email></email> tags are optional and each phone element may have zero or more <mobile></mobile> tags.

Although it is not mandatory to have a DTD when we conceive a XML document, it is a good idea. We can use validating parsers or other mechanisms that use the DTD to check for document consistency.

3 XML in Prolog

This paper is about processing XML using *pure Prolog*. By *pure Prolog* we mean *Prolog* [16] restricted to definite Horn clauses (see [10] for standard definitions in the area of logic programming). *Pure Prolog* inherits from logic programming the notion of *term*, which is especially suitable to deal with tree-structured data such as XML.

3.1 Translating XML to Prolog Terms

A XML document can be seen as a variable free (ground) term. This ground term is composed by a main functor (the root tag) and zero or more arguments. For example, consider the following simple XML file, describing an address book:

```
<addressbook>
   <name>François</name>
   <address>Paris</address>
   <phone>135680864</phone>
   <name>Frank</name>
   <address>New York</address>
   <email>frank@mailserver.com</email>
</addressbook>
```

One equivalent Prolog term is:

```
addressbook(
   name('François'),
   address('Paris'),
   phone('135680864'),
   name('Frank'),
   address('New York'),
   email('frank@mailserver.com'))
```

In our implementation, the structure of the term is determined by the DTD associated with the XML document. Note that the same document, when it is validated by different DTDs, can give rise to different terms.

In our framework, we can include the attributes in the term as a list. If the previous address book had some kind of attribute, such as:

```
...
   <address>Paris</address>
   <phone type='office'>135680864</phone>
...
```

The corresponding term is:

```
addressbook([],
   name([],'François'),
   address([],'Paris'),
   phone([attribute('type','office')],'135680864'),
   name([],'Frank'),
   address([],'New York'),
   email([],'frank@mailserver.com'))
```

Since attributes do not play a relevant role in our work, we ignore them in future examples.

4 DTD as Regular Types

In this section, we describe the relationship between Document Type Definition (DTD) and Regular Types. We use the Dart-Zobel definition for Regular Types [6].

4.1 Regular Types

The next definitions and examples introduce briefly the notion of Regular Types along the lines presented in [6].

Definition 41 *Assuming an infinite set of* type symbols*, a* type term *is defined as follows:*

1. *A constant symbol is a type term* $(a, b, c, ...)$.
2. *A variable is a type term* $(x, y, ...)$.
3. *A type symbol is a type term* $(\alpha, \beta, ...)$
4. *If* f *is an n-ary function symbol and each* τ_i *is a type term,* $f(\tau_1, ..., \tau_n)$ *is a type term.*

Example 41 *Let a be a constant symbol,* α *a type symbol and* x *a variable. The expressions* a*,* α*,* x *and* $f(a, \alpha, x)$ *are* type terms*. If the expression is variable free, we call it a* pure type term*. The expressions* a*,* α *and* $f(\alpha, g(\beta), c)$ *are* pure type terms*.*

Definition 42 *A* type rule *is an expression of the form* $\alpha \rightarrow \Upsilon$ *where* α *is a type symbol (represents types) and* Υ *is a set of pure type terms. We will use* T *to represent a set of type rules.*

Sets of type rules are *regular term grammars* [18].

Example 42 *Let* α *and* β *be type symbols,* $\alpha \rightarrow \{a, b\}$ *and* $\beta \rightarrow \{nil, tree(\beta, \alpha, \beta)\}$ *are type rules.*

Definition 43 *A type symbol* α *is defined by a set of type rules* T *if there exists a type rule* $\alpha \rightarrow \Upsilon \in T$.

We make some assumptions:

1. The constant symbols are partitioned in non-empty subsets, called *base types*. Some examples are, *string*, *int*, and *number*.
2. The existence of μ, the universal type, and ϕ representing the empty type.
3. Each type symbol occurring in a set of type rules T is either μ, ϕ, a base type symbol, or a type symbol defined in T, and each type symbol defined in T has exactly one defining rule in T.

Definition 44 Regular types *are defined as the class of types that can be specified by sets of type rules (or alternatively by regular term grammars).*

Example 43 *Let* α_i *bet the type of the* i^{th} *argument of append. The predicate append is defined as follows:*

append(nil,L,L).
append(.(X,RX),Y,.(X,RZ)):- append(RX,Y,RZ).

Regular types for α_1,α_2,α_3 *are*

$$\alpha_1 \rightarrow \{nil, .(\mu, \alpha_1)\}$$
$$\alpha_2 \rightarrow \{\mu\}$$
$$\alpha_3 \rightarrow \{\alpha_2, .(\mu, \alpha_3)\}$$

4.2 DTDs as Regular Types

So far, we have considered translation from XML to *Prolog terms*. In our framework *regular types* statically type the logic programming language considered. Thus DTDs can be viewed as type declarations for the terms corresponding to the XML documents in the general logic program for XML processing. We now give some examples of the translation from DTDs to the type language considered. Figure 1 shows an example DTD for recipes. Figure 2 shows the corresponding regular types.

```
<!ELEMENT recipe (title, author,description?, ingredients, instructions?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (name,email?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT ingredients (item*)>
<!ELEMENT item (qtd,ingredient)>
<!ELEMENT qtd (#PCDATA)>
<!ELEMENT ingredient (#PCDATA)>
<!ELEMENT instructions (step+)>
<!ELEMENT step (#PCDATA)>
```

Fig. 1. A DTD for recipes

$$
\begin{aligned}
\tau_1 &\rightarrow \{\text{recipe}(\tau_2,\ \tau_3,\ \tau_6,\ \tau_7,\ \tau_{12}),\\
&\quad\ \text{recipe}(\tau_2,\ \tau_3,\ \tau_7,\ \tau_{12}),\\
&\quad\ \text{recipe}(\tau_2,\ \tau_3,\ \tau_7),\\
&\quad\ \text{recipe}(\tau_2,\ \tau_3,\ \tau_6,\ \tau_7)\}\\
\tau_2 &\rightarrow \{\text{title(string)}\}\\
\tau_3 &\rightarrow \{\text{author}(\tau_4,\tau_5),\ \text{author}(\tau_4)\}\\
\tau_4 &\rightarrow \{\text{name(string)}\}\\
\tau_5 &\rightarrow \{\text{email(string)}\}\\
\tau_6 &\rightarrow \{\text{description(string)}\}\\
\tau_7 &\rightarrow \{\text{ingredients}(\tau_8)\}\\
\tau_8 &\rightarrow \{\text{nil},\ .(\tau_9, \tau_8)\}\\
\tau_9 &\rightarrow \{\text{item}(\tau_{10},\tau_{11})\}\\
\tau_{10} &\rightarrow \{\text{qtd(string)}\}\\
\tau_{11} &\rightarrow \{\text{ingredient(string)}\}\\
\tau_{12} &\rightarrow \{\text{instructions}(\tau_{13})\}\\
\tau_{13} &\rightarrow \{.(\tau_{14}, \mathit{nil}),\ .(\tau_{14}, \tau_{13})\}\\
\tau_{14} &\rightarrow \{\text{step(string)}\}
\end{aligned}
$$

Fig. 2. Regular types for the receipts DTD

The type rule for a tag is defined by a type symbol and a set of disjoint types for the tag body. A simple element, without any subelements, is a ground term. When a tag has one or more optional subelements, the rule must include a set of ground terms that represent all the possible subelements combinations. When a tag must include at least one element, we represent its rule by a list of at least one element (for instance τ_{13} represents step+). When a tag must have zero or more elements, we represent it by a list of elements of its type (for instance τ_8 represents item*). When a tag has optional elements, they are all listed in the type rule.

In figure 3 we define a function $\mathcal{T}$, which translates DTDs to regular types:

$$
\begin{array}{ll}
\mathcal{T}(\text{<!ELEMENT } e \text{ (\#PCDATA) >}) & = \tau_e \rightarrow \{e(string)\} \\
\mathcal{T}(\text{<!ELEMENT } e \text{ EMPTY >}) & = \tau_e \rightarrow \{e\} \\
\mathcal{T}(\text{<!ELEMENT } e \text{ ANY >}) & = \tau_e \rightarrow \{e(\mu)\} \\
\mathcal{T}(\text{<!ELEMENT } e\ (e_1, \ldots, e_n) >) & = \tau_e \rightarrow \{e(\tau_{e_1}, \ldots, \tau_{e_n})\}, \text{ where} \\
 & \quad \mathcal{T}(\text{<!ELEMENT } e_i >) = \tau_{e_i} \rightarrow \Upsilon_{e_i}, \\
 & \quad \text{for } 1 \leq i \leq n \\
\mathcal{T}(\text{<!ELEMENT } e\ e_1* >) & = \tau_e \rightarrow \{nil, .(\tau_{e_1}, \tau_e)\}, \text{where} \\
 & \quad \mathcal{T}(\text{<!ELEMENT } e_1 >) = \tau_{e_1} \rightarrow \Upsilon_{e_1} \\
\mathcal{T}(\text{<!ELEMENT } e\ e_1+ >) & = \tau_e \rightarrow \{.(\tau_{e_1}, nil), .(\tau_{e_1}, \tau_e)\}, \text{where} \\
 & \quad \mathcal{T}(\text{<!ELEMENT } e_1 >) = \tau_{e_1} \rightarrow \Upsilon_{e_1} \\
\mathcal{T}(\text{<!ELEMENT } e\ (e_1|\ldots|e_n) >) & = \tau_e \rightarrow \{\tau_{e_1}, \ldots, \tau_{e_n}\}, \text{where} \\
 & \quad \mathcal{T}(\text{<!ELEMENT } e_i >) = \tau_{e_i} \rightarrow \Upsilon_{e_i}, \\
 & \quad \text{for } 1 \leq i \leq n \\
\mathcal{T}(\text{<!ELEMENT } e\ (e_1, \ldots, e_i?, \ldots, e_n)) & = \tau_e \rightarrow \{e(\tau_{e_1}, \ldots, \tau_{e_{i-1}}, \tau_{e_{i+1}}, \ldots, \tau_{e_n}), \\
 & \quad e(\tau_{e_1}, \ldots, \tau_{e_{i-1}}, \tau_{e_i}, \tau_{e_{i+1}}, \ldots, \tau_{e_n})\}, \\
 & \quad \text{where} \\
 & \quad \mathcal{T}(\text{<!ELEMENT } e_i >) = \tau_{e_i} \rightarrow \Upsilon_{e_i}, \\
 & \quad \text{for } 1 \leq i \leq n
\end{array}
$$

Fig. 3. DTD translation rules

5 Processing XML in Pure Prolog

With terms representing XML documents, it is now easy to make pure Prolog programs for processing those terms. For example, given the input DTD:

```
<!ELEMENT addressbook1 (name,country,email?)*>
<!ELEMENT name (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

And given the output DTD:

```
<!ELEMENT addressbook2 (name,email)*>
<!ELEMENT name (#PCDATA)>
<!ELEMENT email (#PCDATA)>
```

We can easily make a program to translate a document valid for the input DTD to a new document valid for the output DTD. The next example translates an address book with triples (name, country, email) to a second address book of pairs (name, email). We omit the XML to term and term to XML translations for the sake of clarity.

```
process1(addressbook1(X),addressbook2(Y)):-
        process2(X,Y).

process2([],[]).

process2([name(X),country(Y),email(Z)|R1],[name(X),email(Z)|R2]):-
        process2(R1,R2).

process2([name(X),country(Y)|R1],R2):-
        process2(R1,R2).
```

If the program does not produce an output valid with respect to the output DTD, for example:

```
...
process2([],[]).

process2([name(X),country(Y),email(Z)|R1],[name(X),phone(Z)|R2]):-
        process2(R1,R2).

process2([name(X),country(Y)|R1],R2):-
        process2(R1,R2).
```

the type checking module outputs a type error.

6 Implementation

Our implementation has four main parts:

1. Translating XML documents to terms;
2. Translating the DTDs to regular types;
3. Type checking of the *pure Prolog* program used to process XML;
4. Translating the resulting Prolog term to an XML document.

This implementation relies on a toolkit of basic components for processing XML in Prolog (for instance a parser). These supporting components are implemented using existing libraries for SWI Prolog [14]. Type inference for Prolog programs is based on Zobel algorithm ([25], [11]).

7 Discussion and Related Work

7.1 XML and Logic Programming

XML processing in logic programming was the topic of a recent paper of Bry and Shaffert [4] where it was defined an untyped rule-based transformation language based on logic programming. Previous work on the representation of XML using a term language inspired in logic programming was presented in [5], [3] and [2].

The kind of terms used to represent XML in our work follows the representation presented in [2]. In [3] it was presented a term language with *flexible terms*, where the depth of an element may not be fixed. These terms are suitable to express the complex subterm dependencies used for processing XML data. Adapting our work to use *flexible terms* would mean defining new type inference algorithms and we think that this can be a promising line of future work.

All these previous work did not deal with type inference for the transformation languages. As far as we know, our work is the first one dealing with statically type validation in the context of logic programming for XML processing.

Several approaches, referring to *semantic web* [21], adapt techniques from logic programming to XML. Even though connections between our work and *semantic web* are not immediately perceived, it would be a useful investigation whether results from the former can be transfered to the later.

7.2 XML and Functional Programming

Our work was inspired by previous work on the use of functional programming languages for processing XML such as XDuce [9] and HaXml [20]. In XDuce types were denoted by *regular expression types*, a natural generalization of DTDs. Transformation of DTDs to types of Haskell was done in [20], where type checking was done by the Haskell type system itself.

The differences from our approach are in the programming paradigm used (functional *versus* logic) and in the type inference systems. In fact, type inference for logic programming is substantially different from type inference for functional languages due to specific characteristics of the logic paradigm such as built-in unification and the use of the logic variable.

Another difference from XDuce is the representation of XML. We use terms (finite trees) to represent XML documents. XDuce terms are sequences. This difference in the view of XML data leads to significant differences in the processing stage. For example, consider the following DTDs:

```
<!ELEMENT a ((b,b?)*)>
<!ELEMENT b EMPTY>
```

and

```
<!ELEMENT a (b*)>
<!ELEMENT b EMPTY>
```

In our approach the two DTDs don't validate the same set of terms representing XML documents. For example the document:

```
<a>
   <b></b>
   <b></b>
</a>
```

is translated to $a([(b, b)])$ when one uses the first DTD and to $a([b, b])$ when one uses the second DTD. The regular types corresponding to the DTDs are different and in fact they are not equivalent because they type different kinds of terms. In XDuce the elements of every type are sequences, thus the document in the example corresponds to the same value $a[b, b]$. Then XDuce type system can correctly prove that the types corresponding to the DTDs are equivalent.

We could have reached the same conclusion without changing our type inference system if our initial choice to represent XML was lists of elements. In this case both documents would have been translated to the same Prolog term $a([b, b])$ and the regular types corresponding to the DTDs would be:

$$\begin{aligned}\tau_a &\rightarrow \{a(\tau_1)\}\\ \tau_1 &\rightarrow \{nil, .(\tau_b, .(\tau_b, \tau_1)), .(\tau_b, \tau_1)\}\\ \tau_b &\rightarrow \{b\}\end{aligned}$$

and:

$$\begin{aligned}\tau_a &\rightarrow \{a(\tau_1)\}\\ \tau_1 &\rightarrow \{nil, .(\tau_b, \tau_1)\}\\ \tau_b &\rightarrow \{b\}\end{aligned}$$

Then our type system would correctly prove that the types are equivalent (using Zobel *subset* algorithm [25] in the two directions).

This difference is due to a design decision: to translate XML to lists (which correspond to the original view of XML documents) or to finite trees (arbitrary Prolog terms) which is more natural in the logic programming paradigm. We chose the second option because terms are the basic Prolog data-structure. It would be interesting to implement both representations and to compare the final results.

7.3 Type Checking of XML

Type checking for XML was the subject of several previous papers. Milo, Suciu and Vianu studied the type checking problem using *tree transducers* in [12]. This work was extended to deal with data values in [1]. A good presentation on type checking for XML can be found in [17]. Standard languages used to denote types in the previous context were based on extensions of *regular tree languages*. The types we use in our framework correspond to *regular tree languages*.

8 Conclusion and Future Work

In this paper we use standard type inference for logic programming to get static validation of logic programs for XML transformation given their DTDs. There is plenty of scope for further work, in several directions:

1. *Efficiency*: our type inference implementation is based directly on the algorithm described in Zobel thesis [25]. This algorithm is suitable for a first implementation of *regular type* inference for logic programming because it relies on basic operations on *regular types* such as *intersection*, *comparison* and *unification*, which clarify several important concepts related to the type language. It can also be implemented in an efficient way using tabulation. We are now improving the efficiency of our implementation using efficient tabulation techniques for logic programming.
2. *Expressiveness*: More expressive languages for the validation of XML are now proposed (for instance [15]). The study of type languages which enable the use of those more expressive frameworks, deserves a careful research.
3. *Explicitly typed languages*: our framework relies on an implicitly typed logic programming language. The features proposed by our framework can be easily adapted for explicitly typed logic programming languages (such as *Mercury* [19]) by translating DTDs to types in the host language.

References

1. Noga Alon, Tova Milo, Frank Neven, Dan Suciu, and Victor Vianu. XML with data values: Typechecking revisited. In *PODS*, 2001.
2. H. Boley. Relationships between logic programming and XML. In *Proc. 14th Workshop Logische Programmierung*, 2000.
3. François Bry and Norbert Eisinger. Data modeling with markup languages: A logic programming perspective. In *15th Workshop on Logic Programming and Constraint Systems, WLP 2000, Berlin*, 2000.
4. François Bry and Sebastian Schaffert. Towards a declarative query and transformation language for XML and semistructured data: simulation unification. In *Proc. of the 2002 International Conference on Logic Programming*, 2002.
5. D. Cabeza and M. Hermenegildo. Distributed WWW Programming using Ciao-Prolog and the PiLLoW Library. *Theory and Practice of Logic Programming*, 1(3):251–282, May 2001.
6. P. Dart and J. Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. The MIT Press, 1992.
7. M. Florido and L. Damas. Types as theories. In *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming*, 1992.
8. J.P. Gallagher and D.A. de Waal. Fast and precise regular approximation of logic programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, 1993.

9. Haruo Hosoya and Benjamin Pierce. Xduce: A typed XML processing language. In *Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997 of *Lecture Notes in Computer Science*, 2000.
10. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.
11. Lunjin Lu. On Dart-Zobel algorithm for testing regular type inclusion. In *SIGPLAN Notices Volume 36*, 2001.
12. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. In *ACM Symposium on Principles of Database Systems*, 2000.
13. P. Mishra. Towards a theory of types in Prolog. In *Proc. of the 1984 International Symposium on Logic Programming*, 1984.
14. SWI Prolog. http://www.swi-prolog.org/.
15. XML Schema. http://www.w3.org/XML/Schema/, 2000.
16. Leon Sterling and Ehud Shapiro. *The Art of Prolog, 2nd edition*. The MIT Press, 1994.
17. Dan Suciu. Typechecking for semistructured data. In *International Workshop on Database Programming*, 2001.
18. J.W. Thatcher. *Tree automata: An informal survey*. Prentice-Hall, 1973.
19. The Mercury Programming Language. http://www.cs.mu.oz.au/research/mercury/.
20. Malcom Wallace and Colin Runciman. Haskell and XML: Generic combinators or type-based translation? In *International Conference on Functional Programming*, 1999.
21. Semantic Web. http://www.w3.org/2001/sw/, 1999.
22. Extensible Markup Language (XML). http://www.w3.org/XML/.
23. E. Yardeni and E. Shapiro. A type system for logic programs. In *The Journal of Logic Programming*, 1990.
24. Justin Zobel. Derivation of polymorphic types for prolog programs. In *Proc. of the 1987 International Conference on Logic Programming*, pages 817–838, 1987.
25. Justin Zobel. *Analysis of Logic Programs*. PhD thesis, Department of Computer Science, University of Melbourne, 1990.