XCentric: Constraint based XML Processing

Jorge Coelho[1] and Mário Florido[2]

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

Abstract. Here we present the logic-programming language XCentric, discuss design issues, and show its adequacy for XML processing. Distinctive features of XCentric are a powerful unification algorithm for terms with functors of arbitrary arity (which correspond closely to XML documents) and a rich type language that uses operators such as repetition (*), alternation, etc, as types allowing a compact representation of terms with functors with an arbitrary number of arguments (closely related to standard type languages for XML). This new form of unification together with an appropriate use of types yields a substantial degree of flexibility in programming.

1 Introduction

XML is a powerful format for tree-structured data. A need for programming language support for XML processing led to the definition of XML programming languages, such as XSLT [28], XDuce [14], CDuce [1], Xtatic [29] and Xcerpt [2].

In this paper we present XCentric, a logic programming (LP) language based on the *unification of terms with flexible arity function symbols* extended with a new *type system* for dealing with sequences and new features for searching sequences inside trees at arbitrary depth.

The main features of XCentric rely on the use of:

Regular expression types: regular expression types give us a compact representation of sequences of arguments of functors with flexible arity. They also add extra expressiveness to the unification process. Let us present an illustrating example.

The following declaration introduces regular expression types describing terms in a simple bibliographic database:

```
:-type bib ---> bib(book+).
:-type book ---> book(author+, name).
:-type author ---> author(string).
:-type name ---> name(string).
```

Type expressions of the form $f(...)$ classify tree nodes with the label f (XML structures of the form $< f > ... < /f >$). Type expressions of the form $t*$ denote

a sequence of arbitrary many ts, and $t+$ denotes a sequence of arbitrary many ts with at least one t. Thus terms with type *bib* have *bib* as functor and their arguments are a sequence of one or more books. Terms with type *book* have *book* as functor and their arguments are a sequence consisting of one or more authors followed by the name of the book.

The next type describes arbitrary sequences of authors with at least two authors:

```
:-type type_a ---> (author(string),author(string)+).
```

A new form of unification: in the previous example, to get the names of all the books with two or more authors in XCentric, one just needs the following query ($= * =$ stands for unification of terms with flexible arity functors and $t :: \tau$ means that term t has type τ):

```
?-bib(_,book(X::type_a,name(N)),_)=*=BibDoc::bib.
```

This unifies two terms typed by *bib*. The type declaration in the first argument is not needed because it can be easily reconstructed from the term. In this case we bind the variable N to the content of the name element of the first book element with at least two authors. Note how the type *type_a* in the first argument of the unification is used to jump over an arbitrary number of arguments and extract the name of the first book with at least two authors. All the results can then be obtained, one by one, by backtracking. This goes far beyond standard Prolog unification.

Sequence variables and unification of terms with functors of flexible arity gives XCentric the power of partially specifying terms in breadth (i.e. within the subterms of the same term). In XCentric one can also partially specify a term in depth using the *deep* predicate. A query term of the form $deep(t_1, t_2)$ matches all subterms of t_2 that match the term t_1. Consider the following example (in XCentric we can explicitly refer to sequences of terms $t_1, ..., t_n$ as $< t_1, ..., t_n >$): suppose we want to find sequences of two authors in a document to which the variable *XML* is bound. We can use the query:

```
?-deep(<author(A1),author(A2)>,XML).
```

The names of the two authors will bind variables A_1 and A_2, and all solutions can be found by backtracking.

The main contributions of XCentric to the logic programming paradigm, are to give programmers a tool that makes it much easier and more declarative to process XML, and to show the impact of a new form of unification (*typed unification of terms with functors of flexible arity*) in programming. Note that subjects such as databases, data-mining and Web programming, are quite relevant application areas of logic programming, and in these areas XML is becoming more and more a standard data format for information exchange.

In this paper we show the previous claim about XCentric, showing its contribution with respect to:

Prior non-LP XML processing work: mainstream languages for XML processing such as XSLT ([28]), XDuce ([14]), CDuce ([1]) and Xtatic ([29]) rely on the notion of trees with an arbitrary number of leaf nodes to abstract XML documents. However these languages are based on functional programming and thus their key feature is pattern matching, not unification. Regular expression patterns are often ambiguous and thus functional XML processing languages presume a fixed matching strategy for making the matching deterministic. In contrast, XCentric just leaves ambiguous matching non-deterministic and examines every possible matching case by backtracking. This makes it possible to write complicated XML processing tasks in a quite declarative way. Although pattern matching is desirable in many applications of XML processing, there are situations where the use of unification is a gain. It is known that unification may even improve efficiency in some cases (by careful use of the logical variable) and it is on the basis of a truly relational programming, improving declarativeness and modularity in many cases. In this paper we show examples where these situations also arise in XML processing. This, and the use of unification of terms with functors of flexible arity, show that there are aspects of XML processing in XCentric that do not have counterparts in other approaches to XML processing.

Prior LP work: some Prolog systems have libraries [23, 22] to deal with XML. These libraries translate XML documents to a list of Prolog terms and use standard Prolog for processing. XCentric uses recent results of unification theory (*unification of terms with functors of flexible arity* [17, 7]) and novel type languages based on regular types [8, 6], as the theoretical basis of a logic programming language for XML processing where sequences of terms are first class objects of the language. This leads to a much more declarative way of processing XML when compared to the standard Prolog libraries for XML processing. Xcerpt ([2]) is a logic programming query language for XML which also used terms with flexible arity function symbols as an abstraction of XML documents. It used a special notion of term (called *query terms*) as patterns specifying selection of XML terms much like Prolog goal atoms. The underlying mechanism of the query process was *simulation unification* ([3]), used for solving inequations of the form $q \leq t$ where q is a query term and t a term representing XML data. Concepts behind Xcerpt are more directed to query languages and technically quite different from the typed unification of terms with functors of flexible arity used in XCentric.

Regular expression matching was also used in [18, 19] to extend context sequence matching with context and sequence variables. This work dealt with matching, not unification, and it was not integrated in a programming language. Unification of terms with functors of flexible arity generalizes previous work on word unification ([15]), equations over lists of atoms with a concatenation operator ([10]) and equations over free semigroups ([21]), by enabling the use of as many flexible arity symbols as we wish and of arbitrarily nested terms. Recently, in [16], a specific language was defined to denote relations between XML documents. This

language used its own new syntax, based on mainstream functional languages for XML processing, and its definition stresses the usefulness of an approach based on the truly relational programming paradigm: logic programming.

Monadic Datalog has also been successfully applied to XML querying in the absence of data values [13].

Note that the work described in this paper was first presented in a previous paper from the authors ([5]).

In the rest of the paper we assume that the reader is familiar with logic programming [20] and XML [26]. We start in section 2 by presenting some examples of the language. In section 3 we present sequence variables and flexible arity functions and in section 4 we present sequence types. In section 5 we explain the role of types in the unification process, and finally in section 6 we conclude and outline the future work.

2 XCentric by example

Here we present several simple examples to familiarize the reader with the language before presenting the details.

In XCentric, an XML document is translated to a term with flexible arity function symbol. This term has a main functor (the root tag) and zero or more arguments. Although our actual implementation translates attributes to a list of pairs, since attributes do not play a relevant role in this work we will omit them in the examples, for the sake of simplicity.

Example 1. Consider the simple XML file:

```
<addressbook>
  <record>
   <name>John</name>
   <address>New York</address>
   <email>john.ny@mail.com</email>
  </record>
  ...
</addressbook>
```

Its corresponding term is:

```
addressbook(record(name('John'),address('New_York'),
                                email('john.ny@mail.com')),...)
```

Suppose that the previous XML file is valid with respect to the following DTD:

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

This DTD can be described in XCentric by the type rule:

```
:-type type_addr ---> addressbook(record(name(string),address(string),
                                         phone(string)?,email(string))*)
```

From now on, whenever a variable is presented without any type information it is implicitly associated with the universal type *any* (which types any term). Through the following examples we will use the built-in predicates *xml2pro* and

pro2xml which respectively convert XML files into terms and vice-versa. We will also use the predicate *newdoc(Root,Args, Doc)* where *Doc* is a term with functor *Root* and arguments *Args*.

Example 2. Suppose we have an XML document with a catalog of books like the following one:

```
<catalog>
  ...
   <book number="500">
     <author>Simon Thompson</author>
     <name>
     Haskell: The Craft of Functional Programming (2nd Edition)
     </name>
     <price>41</price>
     <year>1999</year>
   </book>
    ...
</catalog>
```

To get all the books with two or more authors using SWI-Prolog [23] (which has a quite good library for processing XML in Prolog) we need the following code:

```
pbib([element(_,_,L)]):-
                    pbib2(L).
pbib2([]).

pbib2([element('book',_,Cont)|Books]):-
                    authors(Cont),!,pbib2(Books).
pbib2([_|Books]):-
                    pbib2(Books).

authors([element('author',_,_),element('author',_,_)|R]):-
                    write_name(R).

write_name([element('name',_,[N])]):-
                    write(N),nl.
write_name([_|R]):-
                    write_name(R).
```

To do the same in XCentric, assuming that the XML file is translated to a term binding variable *BibDoc*, the following query is enough:

```
catalog(_,book(name(N),author(_),author(_),_),_) =*= BibDoc.
```

All the solutions can then be obtained, one by one, by backtracking. The simplicity and declarativeness of the second solution speaks by itself when compared to the first one.

If we want to verify the document consistency with respect to a given DTD, we just have to replace in the previous query, the variable *BibDoc*, by the typed variable $BibDoc :: tc$, where type *tc* is the translation of the DTD to its corresponding XCentric type.

2.1 Incomplete terms in depth

XCentric also provides predicates that allow the programmer to find a sequence of elements at arbitrary depth, to search for the *nth* occurrence of a sequence of elements and to count the number of occurrences of a sequence. The predicates are *deep/2*, *deepp/3* and *deepc/3*, respectively.

Example 3. This example is based on a medical report using the HL7 Patient Record Architecture and inspired by the XQuery use cases available at [27]. Given *report1.xml* (figure 1), find what happened between the first *incision* and the second *incision* and write the result in a file named critical.xml:

```
<report>
  <section>
    <section_title>Procedure</section_title>
     <section_content>
      The patient was taken to the operating room where she was placed...
      <anesthesia>induced under general anesthesia.</anesthesia>
     <prep>
       <action>A Foley catheter was placed to decompress </action>...
     </prep>
     <incision>
       A curvilinear incision was made <geography> in the midline
       immediately infraumbilical </geography> and the subcutaneous
       tissue was divided <instrument> using electrocautery.
       </instrument>
     </incision>
     The fascia was identified and <action> #2 0 Maxon stay sutures were
     placed on each side of the... </action>
     <incision>
       The fascia was divided using <instrument> electrocautery
       </instrument> and the peritoneum was entered. </incision>
     <observation> The small bowel was identified. </observation>...
    </section_content>
  </section>
</report>
```

Fig. 1. report1.xml

```
translate:-
     xml2pro('report1.xml',Rep),
     deep(<incision(_),Critical,incision(_)>,Rep),
     newdoc(critical_sequence,Critical,FL),
     pro2xml(FL,'critical.xml').
```

The result is:

```
<critical_sequence>
 The fascia was identified and <action> #2 0 Maxon stay
 sutures were placed on each side of the...</action>
</critical_sequence>
```

2.2 Unification and XML Processing

Mainstream XML processing languages rely on pattern matching. In our approach we also use unification for XML processing. In this section we present some examples where unification has advantages over pattern matching.

Example 4. In functional based languages for XML processing transformations are unidirectional, enabling the use of a description of the structure of an XML document and the use of pattern matching to extract some of its subparts.

Being a relational language, XCentric can easily describe the structures of two documents and relate their subparts. For example, we can write the following simple predicate in XCentric for converting between fragments of two kinds of address books.

```
translate(<person(name(N),C1)>,<card(person-name(N),C2)>):-
                                        address_content(C1,C2).

address_content(<>,<>).

address_content(<A1,address(A),A2>,<L1,location(A),L2>):-
                                address_content(<A1,A2>,<L1,L2>).
```

This relates a *person* element and a *card* element, where the pattern of the first argument of *translate* requires the *person* to contain a *name* element followed by

a sequence of *address* elements, and the second argument describes the structure of the *card* containing a *person-name* element follower by a sequence o *location* elements. Variable N, which appears in both arguments, specifies that its corresponding subparts, the contents of *name* and *person-name*, are the same. Predicate *address_content* relates *address* in a person element with *location* in the corresponding *card* element. This is trivially expressed in an unification-based relational language such as XCentric, but impossible to express in a functional (thus unidirectional) language based on pattern matching. Note that variables occurring in sequences, are interpreted in the domain of sequences, thus, in this program, unification is not Prolog unification, but the non-standard unification of XCentric. Also note the gain in modularity: this predicate can be used in three different ways. 1) to transform an XML document with the format specified by the first argument of *translate* into the format specified by its second argument, 2) to do the opposite transformation, or 3) to guarantee that two different documents in the two different formats are related in the way specified by the predicate (corresponding, respectively, to call it with the first, second or both arguments ground). In a functional-based language these three different behaviors have to be implemented by three different functions.

Example 5. Suppose we have an XML document that represents an article entry:

```
<text>
Mainstream languages for <b>XML</b> processing such as XSLT <ref>W3C
</ref>, XDuce <ref>Hosoya </ref>, CDuce <ref>Frish Casagna and Benzaken
</ref> and Xtatic <ref>Pierce</ref> rely on the notion of trees with an
arbitrary number of leaf nodes to abstract <b>XML</b> documents.
</text>
```

This document has references like **<ref>W3C</ref>** which appear in a simple bibliography database, where each **ref** element has a corresponding **author**:

```
<bibliography>
 <bib>
   <author>Coelho and Florido</author>
   <name>Type-based XML Processing in Logic Programming</name>
 </bib>
 <bib>
   <author>Hosoya</author>
   <name>XDuce: A Typed XML processing language</name>
 </bib>
...
</bibliography>
```

The idea is the following:

1. Create a new bibliography document only with references occurring in the article but ordered by author name.
2. Create a new article were each reference is replaced by a number corresponding to the author order in the bibliography.

As result we want the following:

```
<text>
Mainstream languages for <b>XML</b> processing such as XSLT <i>4</i>,
XDuce <i>2</i>, CDuce <i>1</i> and Xtatic <i>3</i> rely on the notion
of trees with an arbitrary number of leaf nodes to abstract <b>XML</b>
 documents.
</text>
```

and the bibliography file as:

```
<bibliography>
 <bib>
  <index> 1 </index>
  <author>Frish Casagna and Benzaken</author>
  <name>CDuce an XML-centric general-purpose language</name>
 </bib>
...
</bibliography>
```

To achieve this result using a similar method but based in pattern matching approach, note that, as we only know all the references after processing the entire document, we must process the document, retrieve all the references found, order the references and then process the document a second time to replace the references with the corresponding indexes. Using unification it is possible to process the document only once: the references are replaced by free variables which are associated with the corresponding references (by means of an association list) creating an intermediate document which is not a ground term. We can then order the association list by author and bind the corresponding variables with the correct index. The document now becomes a ground term (by the use of unification) which is the desired output. Note that we only processed the document once (the complete implementation can be found at http://www.ncc.up.pt/xcentric/).

3 Sequence Variables and Flexible Arity Functions

Here we briefly review the notions of sequence, sequence variable and functor with flexible arity. A detailed description of this subject and of (untyped) unification of flexible arity terms can be found in [7]. We extend the domain of discourse of Prolog (trees over uninterpreted functors) with finite sequences of trees.

Definition 1. *A* sequence $\tilde{t}$*, is defined as follows:* ϵ *is the empty sequence and* $t_1, \tilde{t}$ *is a sequence if* t_1 *is a term and* $\tilde{t}$ *is a sequence.*

We now proceed with the syntactic formalization, by extending the standard notion of Prolog term with flexible arity function symbols and sequence variables.

Consider an alphabet consisting of the following sets: the set of standard variables, the set of sequence variables, the set of constants, the set of fixed arity function symbols and the set of flexible arity function symbols.

Definition 2. *The set of terms over the previous alphabet is the smallest set that satisfies the following conditions:*

1. *Constants, standard variables and sequence variables are terms.*
2. *If* f *is a flexible arity function symbol and* $t_1, \ldots, t_n$ *(*$n \geq 0$*) are terms, then* $f(t_1, \ldots, t_n)$ *is a term.*
3. *If* f *is a fixed arity function symbol with arity* n*,* $n \geq 0$ *and* $t_1, \ldots, t_n$ *are terms such that for all* $1 \leq i \leq n$*,* t_i *does not contain sequence variables as subterms, then* $f(t_1, \ldots, t_n)$ *is a term.*

Remark 1. To avoid further formality, we assume that the domain of interpretation of variables is predetermined by the context where they occur. Variables occurring in a constraint of the form $t_1 = * = t_2$ are interpreted in the domain of sequences of trees, otherwise they are standard Prolog variables.

3.1 Sequences

We use a special kind of terms, here called *sequence terms*, for implementing sequences.

Definition 3. *A* sequence term*, $\bar{t}$ is defined as follows:*

- ϵ *is a* sequence term *that represents the empty sequence.*
- $seq(t, \bar{s})$ *is a* sequence term *if t is a term and $\bar{s}$ is a* sequence term*.*

Definition 4. *A sequence term in* normal form *is defined as:*

- ϵ *is in normal form*
- $seq(t_1, t_2)$ *is in normal form if t_1 is not of the form $seq(t_3, t_4)$ and t_2 is in normal form.*

Sequence terms in normal form are the internal representation of sequences. For example, $seq(a, seq(b, \epsilon))$ represents sequence a,b. Note that for simplification purposes we drop the *seq* operators for sequences of just one element.

4 Types

In this section we present the type language starting with a description of *Regular Types* [11] and then their extension to type sequences of terms.

4.1 Regular Types

Definition 5. *Assuming an infinite set of* type symbols*, a* type term *is defined as follows:*

1. *A constant symbol (we use $a, b, c, etc.$) is a type term.*
2. *A type symbol (we use $\alpha, \beta, etc.$) is a type term.*
3. *If f is a flexible arity function symbol and each τ_i is a type term, $f(\tau_1, ..., \tau_n)$ is a type term.*

Definition 6. *A* type rule *is an expression of the form $\alpha \rightarrow \Upsilon$ where α is a type symbol and Υ is a finite set of type terms.*

Sets of type rules correspond to *regular term grammars* [25].

Example 6. Let α and β be type symbols, $\alpha \rightarrow \{a, b\}$ and $\beta \rightarrow \{nil, tree(\beta, \alpha, \beta)\}$ are type rules.

Definition 7. *A type symbol α is defined by a set of type rules T if there exists a type rule $\alpha \rightarrow \Upsilon \in T$.*

Regular types are the class of types that can be defined by finite sets of type rules. In XCentric a type rule $\alpha \rightarrow \{\tau_1, \ldots, \tau_n\}$ is represented by the declaration:

$$\text{:-type } \alpha \text{ - - -> } \tau_1; \ldots; \tau_n.$$

It is well known that regular types can be associated with unary logic programs (see [31, 30]). For every type symbol α, there is a predicate definition α, such that $\alpha(t)$ is true if and only if t is a term with type α (note that we are using the type symbol as the predicate symbol).

4.2 Regular Expression Types

We now define regular expression types, which describe sequences of values: a* (sequence of zero or more a's), a+ (sequence of one or more a's), a? (zero or one a), a|b (a or b) and a,b (a followed by b). We translate regular expression types to our internal sequence notation:

$$\begin{array}{l} a* \Rightarrow \alpha_* \rightarrow \{\epsilon, seq(a, \alpha_*)\} \\ a+ \Rightarrow \alpha_+ \rightarrow \{a, seq(a, \alpha_+)\} \\ a? \Rightarrow \alpha_? \rightarrow \{\epsilon,\ seq(a, \epsilon)\} \\ a|b \Rightarrow \alpha_| \rightarrow \{a, b\} \\ a, b \Rightarrow \alpha_{seq} \rightarrow \{seq(a, seq(b, \epsilon))\} \end{array}$$

Note that DTDs (Document Type Definition) [26] can be trivially translated to regular expression types. XCentric also has some *XML Schema* [24] support, basic types like string, integer, boolean and float, bounding the occurrences of sequences and orderless sequences are supported.

5 Types in the unification process

In this section we explain the role of types in the unification process.

Definition 8. *A* type declaration *for a term t with respect to a set of type rules T is a pair $t :: \alpha$ where α is a type symbol defined in T.*

Example 7. Consider the equation $a(X, b, Y) :: \alpha_a = * = a(a, b, b, b) :: \mu$, where α_a is defined by the type rules:

$$\begin{array}{l} \alpha_a \longrightarrow \{a(\alpha_x, b, \alpha_y)\} \\ \alpha_x \longrightarrow \{\mu\} \\ \alpha_y \longrightarrow \{b, (b, \alpha_y)\} \end{array}$$

then this unification gives only two results:

1. $X = a$ and $Y = b, b$
2. $X = a, b$ and $Y = b$

Note that without the types the solution $X = a, b, b$ and $Y = \epsilon$ would also be valid.

Implementation: An equation of the form $t_1 :: \alpha_1 = * = t_2 :: \alpha_2$ is translated to the following query:

$$? \text{ - } t_1 = * = t_2, \alpha_1(t_1), \alpha_2(t_2).$$

and the respective predicate definitions for α_1 and α_2 (as described in section 4). Correctness of $t_1 :: \alpha_1 = * = t_2 :: \alpha_2$ comes for free from the correctness of the untyped version of $= * =$ (presented in [7]) and noticing that if $? - t_1 = * = t_2, \alpha_1(t_1), \alpha_2(t_2)$ succeeds then $t_1\theta \in [\alpha_1]_T \cap [\alpha_2]_T$, where θ is the substitution resulting from $t_1 = * = t_2$.

6 Conclusions and Future Work

XCentric is an extension of Prolog with a richer form of unification and regular types, designed specifically for XML processing in logic programming. It enables a highly declarative style of XML-processing and it is based on a sound foundation of a very small core of well studied key features, such as *unification of terms with flexible arity* [7, 17] and *regular types for logic programming* [31, 12]. Also note that XCentric is now being used successfully in practice in some areas, such as website auditing and verification [9, 4]. Ongoing work is being done to improve efficiency. We have benchmarks indicating that, compared with pattern matching, XCentric is rather inefficient when applied to large files (more than 15 KB). This is, somehow, expected, since pattern matching itself is more efficient than unification, and pattern-matching based languages, such as XDuce or CDuce, are compiled, highly optimized languages. Future and ongoing work on this matter, includes the use of tabling in the unification algorithm and extending the WAM with new instructions for dealing directly with sequences. We are also interested on applying XCentric to other application areas besides XML. Bioinformatics, relying intensively on the notion of sequence, is a natural candidate for new applications.

References

1. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN Int. Conference on Functional Programming*, Uppsala, Sweden, 2003.
2. F. Bry and S. Schaffert. The XML Query Language Xcerpt: Design Principles, Examples, and Semantics. In *2nd Annual International Workshop Web and Databases*, volume 2593 of *LNCS*, 2002.
3. F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *International Conference on Logic Programming (ICLP)*, volume 2401 of *LNCS*, 2002.
4. Jorge Coelho and Mário Florido. Type-based static and dynamic website verification (to appear). In *ICIW'07*. IEEE Computer Society, 2007.
5. Jorge Coelho and Mario Florido. Xcentric: Logic programming for xml processing. In *9th ACM International Workshop on Web Information and Data Management*. ACM Press, 2007.
6. Jorge Coelho and Mário Florido. Type-based XML Processing in Logic Programming. In *Practical Aspects of Declarative Languages*, volume 2562 of *LNCS*, 2003.
7. Jorge Coelho and Mário Florido. CLP(Flex): Constraint Logic Programming Applied to XML Processing. In *Ontologies, Databases and Applications of SEmantics (ODBASE)*, volume 3291 of *LNCS*. Springer Verlag, 2004.
8. Jorge Coelho and Mário Florido. Unification with flexible arity symbols: a typed approach. In *Informal proceedings of the 20th International Workshop on Unification (UNIF'06)*, Seattle, USA, 2006.
9. Jorge Coelho and Mário Florido. VeriFLog: Constraint Logic Programming Applied to Verification of Website Content. In *Int. Workshop XML Research and Applications (XRA'06)*, volume 3842 of *LNCS*. Springer-Verlag, 2006.

10. A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.
11. P. Dart and J. Zobel. A regular type language for logic programs. In Frank Pfenning, editor, *Types in Logic Programming*. The MIT Press, 1992.
12. Mário Florido and Luís Damas. Types as theories. In *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming*, 1992.
13. Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. *J. ACM*, 51(1):74–113, 2004.
14. Haruo Hosoya and Benjamin Pierce. XDuce: A typed XML processing language. In *Third Int. Workshop on the Web and Databases*, volume 1997 of *LNCS*, 2000.
15. J. Jaffar. Minimal and complete word unification. *Journal of the ACM*, 37(1):47–85, 1990.
16. Shinya Kawanaka and Haruo Hosoya. biXid: a bidirectional transformation language for XML. In John H. Reppy and Julia L. Lawall, editors, *ICFP*, pages 201–214. ACM, 2006.
17. Temur Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In *Joint AISC'2002 - Calculemus'2002 conference*, LNAI, 2002.
18. Temur Kutsia. Context sequence matching for xml. In *Proceedings of the 1th Int. Workshop on Automated Specification and Verification of Web Sites*, 2005.
19. Temur Kutsia and Mircea Marin. Can context sequence matching be used for querying xml? In Laurent Vigneron, editor, *Proceedings of the 19th Int. Workshop on Unification (UNIF'05)*, pages 77–92, Nara, Japan, 22 April 2005.
20. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.
21. G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik USSR*, 103:147–236, 1977.
22. Pillow: Programming in (Constraint) Logic Languages on the Web. http://clip.dia.fi.upm.es/Software/pillow/pillow.html, 2001.
23. SWI Prolog. http://www.swi-prolog.org/.
24. XML Schema. http://www.w3.org/XML/Schema/, 2000.
25. J.W. Thatcher. *Tree automata: An informal survey*. Prentice-Hall, 1973.
26. Extensible Markup Language (XML). http://www.w3.org/XML/, 2003.
27. XQuery Use Cases. http://www.w3.org/TR/xquery-use-cases/, 2005.
28. XSL Transformations (XSLT). http://www.w3.org/TR/xslt/, 1999.
29. Xtatic. http://www.cis.upenn.edu/~bcpierce/xtatic/, 2004.
30. E. Yardeni and E. Shapiro. A type system for logic programs. In *The Journal of Logic Programming*, 1990.
31. Justin Zobel. Derivation of polymorphic types for prolog programs. In *Proc. of the 1987 International Conference on Logic Programming*. MIT Press, 1987.