XML Processing and Logic Programming

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. In this paper we describe a constraint solving module which we call CLP(Flex), for dealing with the unification of terms with flexible arity function symbols. This approach results in a flexible and high declarative model for XML processing.

Keywords: XML Processing Languages, Logic Programming, Constraint Logic Programming

1 Introduction

XML is a notation for describing trees with an arbitrary finite number of leaf nodes. Thus a constraint programming language dealing with terms where function symbols have an arbitrary finite arity should lead to an elegant and declarative way of processing XML.

In this paper we describe a constraint logic programming language, CLP(Flex), similar in spirit to mainstream CLP languages but specialized to the domain of XML processing. Its novel features are the use of *flexible arity function symbols* and a corresponding mechanism for a *non-standard unification* in a theory with flexible arity symbols and variables which can be instantiated by an arbitrary finite sequence of terms. We translate XML documents to terms and use the new unification to process parts of these terms.

Unification with flexible arity symbols is no new notion. An unification algorithm for these terms was defined in [18] where it was used as a *Mathematica* package incorporated in the *Theorema* system (see [5]). Here we changed the algorithm presented in [18] to give the solutions incrementally, an essential feature to use it in a non-deterministic backtracking-based programming language such as *Prolog*. The main contributions of this paper are the presentation of the implementation and use of a constraint programming module, based on unification of terms with function symbols of flexible arity, as a highly declarative model for XML processing. Note that the work described in this paper was first presented in a previous paper from the authors ([8]).

This article focuses on language design, shows its adequacy to write applications that handle, transform and query XML documents, and sketches solutions to implementation issues. A distribution of our language can be found in:

http://www.ncc.up.pt/xcentric/

We assume that the reader is familiar with logic programming ([19]) and CLP ([16, 15]), and knows the fundamental features of XML ([25]).

2 Related Work

Mainstream languages for XML processing such as XSLT ([26]), XDuce ([13]), CDuce ([1]) and Xtatic ([27]) 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 the key feature here is pattern matching, not unification. The main motivation of our work was to extend unification for XML processing, such as the previous functional based languages extended pattern matching. Constraints revealed to be the natural solution to our problem.

Languages with flexible arity symbols have been used in various areas: Xcerpt ([3]) is a query and transformation 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* ([4]), used for solving inequations of the form $q \leq t$ where q is a query term and t a term representing XML data. This framework was technically quite different from ours, being more directed to query languages and less to XML processing. The Knowledge Interchange Format KIF ([12]) and the tool Ontolingua [11] extend first order logic with variable arity function symbols and apply it to knowledge management. Feature terms [24] can also be used to denote terms with flexible arity and have been used in logic programming, unification grammars and knowledge representation. Unification for flexible terms has as particular instances previous work on word unification ([14, 23]), equations over lists of atoms with a concatenation operator ([9]) and equations over free semigroups ([20]). Kutsia ([18]) defined a procedure for unification with sequence variables and flexible arity symbols applied to an extension of *Mathematica* for mathematical proving ([5]). From all the previous frameworks we followed the work of Kutsia because it is the one that fits better in our initial goal, which was to define a highly declarative language for XML processing based on an extension of standard unification to denote the same objects denoted by XML: trees with an arbitrary number of leafs. Although our algorithm is based on this previous one it has some differences motivated by its use as a constraint solving method in a CLP package:

- Kutsia algorithm gave the whole set of solutions to an equality problem as output. We changed that point accordingly to the standard *backtracking* model of *Prolog*. We give as output one answer substitution and subsequent calls to the same query will result in different answer substitutions computed by backtracking. When every solution is computed the query fails indicating that there are no more solutions.

– a direct consequence of the previous point is that our implementation deals with infinite sets of solutions (see example 46). It simply gives all solutions by backtracking.
– Kutsia algorithm was a new definition of unification for the case of terms with flexible arity symbols. Our implementation transforms the initial set of constraints into a different (larger) set of equalities solved by standard unification and uses standard *Prolog* unification for propagating substitutions.

Finally we should refer that the use of standard terms (with fixed arity function symbols) to denote XML documents was made before in several systems. For example Pillow ([22]) used a low level representation of XML where the leaf nodes in the XML trees were represented by lists of nodes and Prolog standard unification was used for processing. In [2] and [7] XML was represented directly by terms of fixed arity. A last reference to some query languages for XML (such as XPathLog [21]) where Prolog style variables are used as an extension to XPath in a query language for XML.

3 Terms with Flexible Arity Symbols and Sequence Variables

3.1 Constraint Logic Programming

Constraint Logic Programming (CLP) [16] is the name given to a class of languages based on the paradigm of rule-based constraint programming. Each different language is obtained by specifying the domain of discourse and the functions and relations on the particular domain. This framework extends the logic programming framework because it extends the Herbrand universe, the notion of *unification* and the notion of equation, accordingly to the new computational domains. There are many examples of CLP languages, such as, Prolog III [10] which employs equations and disequations over rational trees and a boolean algebra, CLP(R), [17] which has linear arithmetic constraints over the real numbers and ECLiPSe, [6], that computes over several domains: a Boolean algebra, linear arithmetic over the rational numbers, constraints over finite domains and finite sets. Prolog itself can be viewed as a CLP language where constraints are equations over an algebra of finite trees. A complete description of the major trends of the fundamental concepts about CLP can be found in [16].

3.2 CLP(Flex)

The idea behind CLP(Flex) is to extend Prolog with terms with flexible arity symbols and sequence variables. We now describe the syntax of CLP(Flex) programs and their intuitive semantics.

In CLP(Flex) we extend the domain of discourse of Prolog (trees over uninterpreted functors) with finite sequences of trees.

Definition 31 *A* sequence $\tilde{t}$*, is defined as follows:*

– *ε is the empty sequence.*
– *$t_1, \tilde{t}$ is a sequence if t_1 is a term and $\tilde{t}$ is a sequence*

Example 31 *Given the terms $f(a)$, b and X, then $\tilde{t} = f(a), b, X$ is a sequence.*

Equality is the only relation between trees. Equality between trees is defined in the standard way: two trees are equal if and only if their root functor are the same and their corresponding subtrees, if any, are equal.

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

We consider an alphabet consisting of the following sets: the set of standard variables, the set of sequence variables (variables are denoted by upper case letters), the set of constants (denoted by lower case letters), the set of fixed arity function symbols and the set of flexible arity function symbols.

Definition 32 *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.*

Terms of the form $f(t_1, \ldots, t_n)$ where f is a function symbol and $t_1, \ldots, t_n$ are terms are called *compound terms*.

Definition 33 *If t_1 and t_2 are terms then $t_1 = t_2$ (standard Prolog unification) and $t_1 = * = t_2$ (unification of terms with flexible arity symbols) are constraints.*

A constraint $t_1 = * = t_2$ or $t_1 = t_2$ is solvable if and only if there is an assignment of sequences or ground terms, respectively, to variables therein such that the constraint evaluates to *true*, i.e. such that after that assignment the terms become equal.

Remark 31 *In what follows, to avoid further formality, we shall 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. In CLP(Flex) programs, therefore, each predicate symbol, functor and variable is used in a consistent way with respect to its domain of interpretation.*

CLP(Flex) programs have a syntax similar to Prolog extended with the new constraint $= * =$. The operational model of CLP(Flex) is the same of Prolog.

3.3 Constraint Solving

Constraints of the form $t_1 = * = t_2$ are solved by a non-standard unification that calculates the corresponding minimal complete set of unifiers. This non-standard unification is based on Kutsia algorithm [18]. As motivation we present some examples of unification:

Example 32 *Given the terms $f(X,b,Y)$ and $f(a,b,b,b)$ where X and Y are sequence variables, $f(X,b,Y) = * = f(a,b,b,b)$ gives three results:*

1. *$X = a$ and $Y = b,b$*
2. *$X = a,b$ and $Y = b$*
3. *$X = a,b,b$ and $Y = \varepsilon$*

Example 33 *Given the terms $f(b,X)$ and $f(Y,d)$ where X and Y are sequence variables, $f(b,X) = * = f(Y,d)$ gives two possible solutions:*

1. *$X = d$ and $Y = b$*
2. *$X = N,d$ and $Y = b,N$ where N is a new sequence variable.*

Note that this non-standard unification is conservative with respect to standard unification: in the last example the first solution corresponds to the use of standard unification.

4 XML Processing in CLP(Flex)

In CLP(Flex) there are some auxiliary predicates for XML processing. Through the following examples we will use the builtin 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* (this predicate is similar to =.. in Prolog).

4.1 XML as Terms with Flexible Arity Symbols

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. Consider the simple XML file presented bellow:

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

The equivalent term is:

```
addressbook(record(
               name('John'),
                address('New York'),
                email('john.ny@mailserver.com')),
                ...)
```

4.2 Using Constraints in CLP(Flex)

One application of CLP(Flex) constraint solving is XML processing. With non-standard unification it is easy to handle parts of XML files. In this particular case, parts of terms representing XML documents.

Example 41 ***Address Book translation.*** *In this example we use the address book document of the previous example. In this address book we have sometimes records with a phone tag. We want to build a new XML document without this tag. Thus, we need to get all the records and ignore their phone tag (if they have one). This can be done by the following program (this example is similar to one presented in XDuce [13]):*

```
translate:-
  xml2pro('addressbook.xml','addressbook.dtd',Xml),
  process(Xml,NewXml),
  pro2xml(NewXml,'addressbook2.xml').

process(A,NewA):-
  findall(Record,records_without_phone(A,Record),LRecords),
  newdoc(addressbook,LRecords,NewA).

records_without_phone(A1,A2):-
  A1 =*= addressbook(_,record(name(N),address(A),_,email(E)),_),
  A2 = record(name(N),address(A),email(E)).
```

Predicate translate/0 first translates the file “addressbook.xml” into a CLP(Flex) term, which is processed by process/2, giving rise to a new CLP(Flex) term and then to the new document “addressbook2.xml”. This last file contains the address records without the phone tag.

Example 42 ***Book Stores.*** *In this example we have two XML documents with a catalogue of books in each (“bookstore1.xml” and “bookstore2.xml”). These catalogues refer to two different book stores. Both “bookstore1.xml” and “bookstore2.xml” have the same DTD and may have similar books. A sample of one of this XML documents can be:*

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
 <book number="1">
   <name>Art of Computer Programming</name>
   <author>Donald Knuth</author>
```

```
  <price>140</price>
  <year>1998</year>
 </book>
 ...
 <book number="500">
  <name>Haskell:The Craft of Functional Programming (2nd Edition)</name>
  <author>Simon Thompson</author>
  <price>41</price>
  <year>1999</year>
 </book>
</catalog>
```

1. *To check which books are cheaper at bookstore 1 we have the following program:*

   ```
   best_prices(B):-
      xml2pro('bookstore1.xml','bookstore1.dtd',T1),
      xml2pro('bookstore2.xml','bookstore2.dtd',T2),
      process(T1,T2,B).

   process(Books1,Books2,[N,A]):-
      Books1 =*= catalog(_,book(name(N),author(A),price(P1),year(Y)),_),
      Books2 =*= catalog(_,book(name(N),author(A),price(P2),year(Y)),_),
      atom2number(P1,P1f),
      atom2number(P2,P2f),
      P1f < P2f.
   ```

   The predicate best_prices/1 *returns the cheaper books at "bookstore1.xml", one by one, by backtracking.*
2. *To get all the books from one author, the author of a book or all the pairs author/book, we have the following code:*

   ```
   books_from(Author,Book):-
     xml2pro('bookstore1.xml','bookstore1.dtd',Xml),
     process2(Xml,Author,Book).

   process2(Xml,Author,Book):-
     Xml =*= catalog(_,book((name(Book),author(Author),_)),_).
   ```

   Here books_from/2 retrieves, by backtracking, every Author/Book names from file "bookstore1.xml".

The previous programs are rather simple. This stresses the highly declarative nature of CLP(Flex) when used for XML processing.

4.3 The Unification Algorithm

The unification algorithm, as presented in [18], consists of two main steps, *Projection* and *Transformation*. The first step, *Projection* is where some variables are erased from the sequence. This is needed to obtain solutions where those variables are instantiated by the empty sequence. The second step, *Transformation* is defined by a set of rules where the non-standard unification is translated to standard Prolog unification.

Definition 41 *Given terms T_1 and T_2, let V be the set of variables of T_1 and T_2 and A be a subset of V.* Projection *eliminates all variables of A in T_1 and T_2.*

Example 43 *Let $T_1 = f(b, Y, f(X))$ and $T_2 = f(X, f(b, Y))$. In the projection step we obtain the following cases (corresponding to $A = \{\}$, $A = \{X\}$, $A = \{Y\}$ and $A = \{X, Y\}$):*

- $T_1 = f(b, Y, f(X)), T_2 = f(X, f(b, Y))$
- $T_1 = f(b, Y, f), T_2 = f(f(b, Y))$
- $T_1 = f(b, f(X)), T_2 = f(X, f(b))$
- $T_1 = f(b, f), T_2 = f(f(b))$

Our version of Kutsia algorithm uses a special kind of terms, here called, *sequence terms* for representing sequences of arguments.

Definition 42 *A* sequence term*, $\bar{t}$ is defined as follows:*

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

Definition 43 *A sequence term in* normal form *is defined as:*

- *empty 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.*

Example 44 *Given the function symbol f, the variable X and the constants a and b:*

$$seq(f(seq(a, empty)), seq(b, seq(X, empty)))$$

is a sequence term *in normal form.*

Note that sequence terms are lists and sequence terms in normal form are flat lists. We introduced this different notation because sequence terms are going to play a key role in our implementation of the algorithm and it is important to distinguish them from standard Prolog lists. Sequence terms in normal form correspond trivially to the definition of sequence presented in definition 31. In fact sequence terms in normal form are an implementation of this definition. Thus, in our implementation, a term $f(t_1, t_2, \ldots, t_n)$, where f has flexible arity, is internally represented as $f(seq(t_1,seq(t_2, \ldots ,seq(t_n, empty) \ldots)))$, that is, arguments of functions of flexible arity are always represented as elements of a sequence term.

We now define a normalization function to reduce sequence terms to their normal form.

Definition 44 *Given the sequence terms $\bar{t}_1$ and $\bar{t}_2$, we define sequence term* concatenation *as $\bar{t}_1 + +\bar{t}_2$, where the ++ operator is defined as follows:*

$$\begin{aligned} \textit{empty} \;\; ++\; \bar{t} &= \bar{t} \\ \textit{seq}(t_1,\bar{t_2}) \;++\; \bar{t_3} &= \textit{seq}(t_1,\bar{t_2}++\bar{t_3}) \end{aligned}$$

Definition 45 *Given a sequence term, we define sequence term* normalization *as:*

$$\begin{aligned} \textit{normalize}(\textit{empty}) &= \textit{empty} \\ \textit{normalize}(t) &= \textit{seq}(t,\textit{empty}),\ \textit{if } t \textit{ is a constant or variable.} \\ \textit{normalize}(t) &= \textit{seq}(f(\textit{normalize}(t_1)),\textit{empty}),\ \textit{if } t = f(t_1). \\ \textit{normalize}(\textit{seq}(t_1,\bar{t})) &= \textit{normalize}(t_1)\ ++\ \textit{normalize}(\bar{t}) \end{aligned}$$

Proposition 41 *The normalization procedure always terminates yielding a sequence in normal form.*

Transformation rules are defined by the rewrite system presented in figure 1. We consider that upper case letters (X,Y,...) stand for sequence variables, lower case letters (s,t,...) for terms and overlined lower case letters ($\bar{t}, \bar{s}$) for *sequence terms*. These rules implement Kutsia algorithm applied to sequence terms by using standard Prolog unification. Note that rules 6, 7, 8 and 9 are non-deterministic: for example rule 6 states that in order to solve $seq(X, \bar{t}) = * = seq(s_1, \bar{s})$ we can solve $\bar{t} = * = \bar{s}$ with $X = s_1$ or we can solve $normalize(seq(X_1, \bar{t})) = * = normalize(\bar{s})$ with $X = seq(s_1, seq(X_1, empty))$. At the end the solutions given by the algorithm are normalized by the *normalize* function. When none of the rules is applicable the algorithm fails. Kutsia showed in [18] that this algorithm terminated if it had a cycle check, (i.e. it stopped with failure if a unification problem gave rise to a similar unification problem) and if each sequence variable does not occur more than twice in a given unification problem.

For the sake of simplicity, the following examples are presented in sequence notation, alternatively to the *sequence term* notation.

Example 45 *Given $t = f(X, b, Y)$ and $s = f(c, c, b, b, b, b)$ the projection step leads to the following transformation cases:*

- $f(X, b, Y) = * = f(c, c, b, b, b, b)$
- $f(b, Y) = * = f(c, c, b, b, b, b)$
- $f(X, b) = * = f(c, c, b, b, b, b)$
- $f(b) = * = f(c, c, b, b, b, b)$

Using the transformation rules we can see that only the first and third unifications succeed. For $f(X, b, Y) = * = f(c, c, b, b, b, b)$ we have the following answer substitutions:

- $X = c, c$ *and* $Y = b, b, b$
- $X = c, c, b$ *and* $Y = b, b$

Success

$$\begin{array}{llcll}
(1) & t & =*= & s & \Longrightarrow \text{True, if } t == s\ ^{1} \\
(2) & X & =*= & t & \Longrightarrow X = t \text{ if } X \text{ does not occur in } t. \\
(3) & t & =*= & X & \Longrightarrow X = t \text{ if } X \text{ does not occur in } t.
\end{array}$$

Eliminate

$$\begin{array}{llcll}
(4) & f(\bar{t}) & =*= & f(\bar{s}) & \Longrightarrow \bar{t} =*= \bar{s} \\
(5) & seq(t_1, \bar{t}_n) & =*= & seq(s_1, \bar{s}_m) & \Longrightarrow t_1 =*= s_1, \\
 & & & & \quad \bar{t}_n =*= \bar{s}_m \\
(6) & seq(X, \bar{t}) & =*= & seq(s_1, \bar{s}) & \Longrightarrow X = s_1, \text{ if } X \text{ does not occur in } s_1, \\
 & & & & \quad \bar{t} =*= \bar{s}. \\
 & & & & \Longrightarrow X = seq(s_1, seq(X_1, empty)), \\
 & & & & \quad \text{if } X \text{ does not occur in } s_1, \\
 & & & & \quad normalize(seq(X_1, \bar{t})) =*= normalize(\bar{s}), \\
 & & & & \quad \text{where } X_1 \text{ is a new variable.} \\
(7) & seq(t_1, \bar{t}) & =*= & seq(X, \bar{s}) & \Longrightarrow X = t_1, \text{ if } X \text{ does not occur in } t_1, \\
 & & & & \quad \bar{t} =*= \bar{s}. \\
 & & & & \Longrightarrow X = seq(t_1, seq(X_1, empty)), \\
 & & & & \quad \text{if } X \text{ does not occur in } t_1, \\
 & & & & \quad normalize(\bar{t}) =*= normalize(seq(X_1, \bar{s})), \\
 & & & & \quad \text{where } X_1 \text{ is a new variable.} \\
(8) & seq(X, \bar{t}) & =*= & seq(Y, \bar{s}) & \Longrightarrow X = Y \\
 & & & & \quad \bar{t} =*= \bar{s}. \\
 & & & & \Longrightarrow X = seq(Y, seq(X_1, empty), \\
 & & & & \quad normalize(seq(X_1, \bar{t})) =*= normalize(\bar{s}), \\
 & & & & \quad \text{where } X_1 \text{ is a new variable and } X, Y \text{ are} \\
 & & & & \quad \text{distinct.} \\
 & & & & \Longrightarrow Y = seq(X, seq(Y_1, empty)), \\
 & & & & \quad normalize(\bar{t}) =*= normalize(seq(Y_1, \bar{s})), \\
 & & & & \quad \text{where } Y_1 \text{ is a new variable and } X, Y \text{ are} \\
 & & & & \quad \text{distinct.}
\end{array}$$

Split

$$\begin{array}{llcll}
(9) & seq(t_1, \bar{t}) & =*= & seq(s_1, \bar{s}) & \Longrightarrow \text{if } t_1 =*= s_1 \Longrightarrow r_1 =*= q_1 \text{ then} \\
 & & & & \quad normalize(seq(r_1, \bar{t})) =*= normalize(seq(q_1, \bar{s})) \\
 & & & & \quad \vdots \\
 & & & & \Longrightarrow \text{if } t_1 =*= s_1 \Longrightarrow r_w =*= q_w, \\
 & & & & \quad normalize(seq(r_w, \bar{t}_n)) =*= normalize(seq(q_w, \bar{s})), \\
 & & & & \quad \text{where } t_1 \text{ and } s_1 \text{ are compound terms.}
\end{array}$$

Fig. 1. Transformation rules

- $X = c, c, b, b$ *and* $Y = b$

  And for $f(X, b) = * = f(c, c, b, b, b, b)$ *we have:*

- $X = c, c, b, b, b$
- $Y = \varepsilon$

Example 46 *In some cases we can have an infinite set of solutions for the unification of two given terms. For example when we solve* $f(X, a) = * = f(a, X)$ *the solutions are:*

- $X = a$
- $X = a, a$
- $X = a, a, a$
- $X = a, a, a, a$
- $\ldots$

In the previous example Kutsia algorithm with the cycle check fails immediately after detecting that it is repeating the unification problem. Our implementation gives all solutions by backtracking. The correctness of the non-standard unification algorithm in figure 1 is presented in [8].

5 Conclusion

In this paper we present a constraint solving extension for Prolog to deal with terms with flexible arity symbols. We show an application of this framework to XML processing yielding a highly declarative language for that purpose. Some points can be further developed in future work:

- an extension with further built-in predicates and constraints, such as predicates to deal with XML types (DTDs and XML-Schema);
- XML attributes are ignored in our language. We just translate them to lists of pairs. More declarative representation of attributes, such as sets of equalities, and an extension to unification to deal with this new constraints would be a relevant feature which is left for future work;
- finally we note that, CLP(Flex) may have applications in other areas different from XML-processing.

Acknowledgements The work presented in this paper has been partially supported by funds granted to *LIACC* through the *Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia* and *Programa POSI.*

[1] == denotes syntactic equality (in opposite with = which denotes standard unification)

References

1. Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.
2. H. Boley. Relationships between logic programming and XML. In *Proc. 14th Workshop Logische Programmierung*, 2000.
3. 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*. Springer Verlag, 2002.
4. 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.
5. B. Buchberger, C. Dupre, T. Jebelean, B. Konev, F. Kriftner, T. Kutsia, K. Nakagawa, F. Piroi, D. Vasaru, and W. Windsteiger. The Theorema System: Proving, Solving, and Computing for the Working Mathematician. Technical Report 00-38, Research Institute for Symbolic Computation, Johannes Kepler University, Linz, 2000.
6. A M Cheadle, W Harvey, A J Sadler, J Schimpf, K Shen, and M G Wallace. ECLiPSe: An Introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College London, London, 2003.
7. J. Coelho and M. Florido. Type-based XML Processing in Logic Programming. In V. Dahl and P. Wadler, editors, *Practical Aspects of Declarative Languages*, volume 2562 of *Lecture Notes in Computer Science*, pages 273–285, New Orleans, USA, 2003. Springer Verlag.
8. Jorge Coelho and Mario Florido. CLP(Flex): Constraint Logic Programming Applied to XML Processing. In *Ontologies, Databases and Applications of SEmantics (ODBASE)*, LNCS, Agia Napa, Cyprus, 2004. Springer Verlag.
9. A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.
10. A. Colmerauer. Prolog III Reference and Users Manual, Version 1.1. In *PrologIA*, Marseilles, 1990.
11. A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: A tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46(6):707–727, 1997.
12. M. R. Genesereth and R. E. Fikes. Knowledge Interchange Format, Version 3.0 Reference Manual TR Logic-92-1. Technical report, Stanford University, Stanford, 1992.
13. 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.
14. J. Jaffar. Minimal and complete word unification. *Journal of the ACM*, 37(1):47–85, 1990.
15. J. Jaffar and J. L. Lassez. Constraint Logic Programming. In *Proceedings of the Fourteenth Annual ACM Symp. on Principles of Programming Languages, POPL '87*, pages 111–119, Munich, Germany, 1987. ACM Press.
16. Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

17. Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. The CLP(R) Language and System. In *Trans. Program. Lang. Syst.*, volume 14, pages 339–395. ACM, 1992.
18. T. Kutsia. Unification with sequence variables and flexible arity symbols and its extension with pattern-terms. In *Artificial Intelligence, Automated Reasoning and Symbolic Computation. Proceedings of Joint AICS'2002 - Calculemus'2002 conference*, volume 2385 of *Lecture Notes in Artificial Intelligence*, pages 290–304, Marseille, France, 2002. Springer Verlag.
19. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.
20. G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik USSR*, 103:147–236, 1977.
21. Wolfgang May. XPathLog: A Declarative, Native XML Data Manipulation Language. In *International Database Engineering & Applications Symposium (IDEAS '01)*, Grenoble, France, 2001. IEEE.
22. Pillow: Programming in (Constraint) Logic Languages on the Web. http://clip.dia.fi.upm.es/Software/pillow/pillow.html.
23. Klaus U. Schulz. Word unification and transformation of generalized equations. *Journal of Automated Reasoning*, 11(2):149–184, 1993.
24. Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.
25. Extensible Markup Language (XML). http://www.w3.org/XML/.
26. XSL Transformations (XSLT). http://www.w3.org/TR/xslt/, 1999.
27. Xtatic. http://www.cis.upenn.edu/~bcpierce/xtatic/.