CLP(Flex): Constraint Logic Programming Applied to XML Processing

Jorge Coelho and Mário Florido

Technical Report Series: DCC-2004-6

CLP(Flex): Constraint Logic Programming Applied to XML Processing

Jorge Coelho and Mário Florido
DCC-FC & LIACC
University of Porto, Portugal
{jcoelho,amf}@ncc.up.pt

Abstract

In this paper we present an implementation of a constraint solving module, CLP(Flex), for dealing with unification in an equality theory for terms with flexible arity function symbols. Then we present an application of CLP(Flex) to XML-processing where XML documents are abstracted by terms with flexible arity symbols. This gives a highly declarative model for XML processing yielding a substantial degree of flexibility in 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.

With the previous motivation, in this paper we present 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. Moreover, XML documents are denoted by terms with flexible arity symbols and XML processing uses the new unification mechanism yielding a substantial degree of flexibility in programming.

We now give a simple example of the kind of processing involved.

Example 1.1 *The XML document:*

```
<teachers>
   <name>Jorge Coelho</name>
   <office>403</office>
   <email>jcoelho@isep.ipp.pt</email>
   <name>Mario Florido</name>
   <office>202</office>
   <name>Alan Jones</name>
   <office>214</office>
   <email>ajones@ncc.up.pt</email>
</teachers>
```

corresponds to the term with flexible arity function symbol:

```
teachers(name(‘‘Jorge Coelho’’),
         office(‘‘403’’),
         email(‘‘jcoelho@isep.ipp.pt’’),
         name(‘‘Mario Florido’’),
         office(‘‘202’’),
         name(‘‘Alan Jones’’),
         office(214),
         email(‘‘ajones@ncc.up.pt’’))
```

The query (which uses the non-standard notion of unification here denoted by $=*=$*):*

```
?- teachers(_,name(N),office(O),email(E),_) =*=
                      teachers(name(‘‘Jorge Coelho’’),
                               office(‘‘403’’),
                               email(‘‘jcoelho@isep.ipp.pt’’),
                               name(‘‘Mario Florido’’),
                               office(‘‘202’’),
                               name(‘‘Alan Jones’’),
                               office(214),
                               email(‘‘ajones@ncc.up.pt’’))
```

gives the following answers:

```
N = ‘‘Jorge Coelho’’,
O = ‘‘403’’,
E = ‘‘jcoelho@isep.ipp.pt’’ ? ;

N = ‘‘Alan Jones’’,
O = ‘‘214’’,
E = ‘‘ajones@ncc.up.pt’’ ?
```

giving by backtracking every sequence of the form name(N), office(O), email(E).

This example shows one relevant point of our framework in the context of XML processing. The XML document in the example can be easily validated by a DTD that defines sequences of the form *name(N), office(O), email(E)*, where information about the email is optional. This is a common practice in XML. Being able to extract sequences where a specific element has to exist (in this case the email) is a standard procedure in every standard manipulation language for XML but it can be rather tedious in logic programming if the document is translated to a term with fixed arity. Our notion of unification of terms with function symbols of flexible arity overcomes this difficult giving a quite declarative way of dealing with this cases in logic programming.

Unification with flexible arity symbols is no new notion. An unification algorithm for these terms was defined in [14] where it was used as a *Mathematica* package incorporated in the *Theorema* system (see [5]). Here we changed the algorithm presented in [14] 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:

1. to motivate the use of constraint programming, in this case based on unification of terms with function symbols of flexible arity, as a highly declarative model for XML processing;

2. a constraint solving module for unification of terms with flexible arity symbols, added to Prolog;

3. an implementation of Kutsia algorithm [14] in Prolog which takes advantage of Prolog unification. 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.

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. The web site of our project:

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

includes more examples and the complete distribution of the system.

We assume that the reader is familiar with logic programming ([15]) and CLP ([13, 12]), and knows the fundamental features of XML ([22]). We start in section 2 by presenting a simple introduction to XML. In section 3 we present the notions of terms with flexible arity symbols and sequence variables. In section 4 we give examples of the use of CLP(Flex) to process XML documents. Then, in section 5 we describe the implementation modules and present the non-standard unification algorithm. We then give an overview of the related work and finally we conclude and outline some future work.

2 XML

XML ([22]) 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>
        <record>
                <name>John</name>
                <address>New York</address>
                <email>john.ny@mailserver.com</email>
        </record>
        <record>
                <name>Sofia</name>
                <address>Rio de Janeiro</address>
                <phone>123456789</phone>
                <email>sofia.brasil@mail.br</email>
        </record>
</addressbook>
```

3 Terms with Flexible Arity Symbols and Sequence Variables

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 3.1 *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 3.1 *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, the set of constants, the set of fixed arity function symbols and the set of flexible arity function symbols.

Definition 3.2 *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.*

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 3.3 *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 and after that assignment the terms become equal.

Remark 3.1 *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.*

In [14] Kutsia extended the standard notions from unification theory to deal with substitution of sequence variables by sequences of terms. Based on this extension of the notion of *substitution*, *more general substitution* and *unifier*, Kutsia defines the new notion of *Minimal Complete Set of Unifiers* of an equation E, ($MCU(E)$) as a minimal set of substitutions with respect to the set of variables of E such that:

1. Every substitution in $MCU(E)$ is an unifier of E.
2. For any unifier $\tau \in E$, there is a $\theta \in MCU(E)$ such that θ is more general than τ.
3. For all θ,$\tau \in MCU(E)$, if θ is more general than τ then $\theta = \tau$.

Example 3.2 *Given the sequence variable X, $f(a, X, c, d)$ is a flexible arity term. X can be instantiated by a sequence of terms, leading for instance to the terms $f(a, a, a, c, d)$ or $f(a, c, d)$ (corresponding respectively to $X = a, a$ and $X = \varepsilon$).*

Example 3.3 *The minimal complete set of unifiers of the equation*

$$f(g(a, X), g(Y, c)) = * = f(U, g(b, V))$$

is: $\{\{U = g(a, X),\ Y = b,\ V = c\}, \{X = \varepsilon,\ U = g(a),\ Y = b, V = c\}, \{U = g(a, X),\ Y = b, Y,\ V = Y, c\},\ \{X = \varepsilon,\ U = g(a),\ Y = b, Y,\ V = Y, c\}\}$

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.

3.1 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 [14]. As motivation we present some examples of unification:

Example 3.4 *Given the terms $a(X, b, Y)$ and $a(a, b, b, b)$ where X and Y are sequence variables, $a(X, b, Y) = * = a(a, b, b, b)$ 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 3.5 *Given the terms $a(b, X)$ and $a(Y, d)$ where X and Y are sequence variables, $a(b, X) = * = a(Y, d)$ 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.*

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 following simple XML file, describing an address book:

```
<addressbook>
        <record>
                <name>John</name>
                <address>New York</address>
                <email>john.ny@mailserver.com</email>
        </record>
        <record>
                <name>Sofia</name>
                <address>Rio de Janeiro</address>
                <phone>123456789</phone>
                <email>sofia.brasil@mail.br</email>
        </record>
</addressbook>
```

The equivalent term is:

```
addressbook(record(
               name('John'),
                address('New York'),
                email('john.ny@mailserver.com')),
             record(
                name('Sofia'),
                 address('Rio de Janeiro'),
                 phone('123456789'),
                 email('sofia.brasil@mail.br')))
```

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 4.1 *Suppose that the term Doc is the CLP(Flex) representation of the document "addressbook.xml". If we want to gather the names of the people living in New York we can simply solve the following constraint:*

$$Doc = * = addressbook(\_, record(name(N), address(\text{'New York'}), \_), \_).$$

All the solutions can then be found by backtracking.

Example 4.2 ***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 [10]):*

```
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 4.3 ***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. 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.

5 Implementation

Our implementation has three main modules:

1. Translating XML documents to terms;
2. Constraint solving module;
3. Translating the resulting CLP(Flex) 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 [19]. As we said before constraint solving is based on Kutsia algorithm [14].

5.1 The Unification Algorithm

The unification algorithm, as presented in [14], 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 5.1 *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 5.1 *Let $T_1 = f(b, Y, f(X))$ and $T_2 = f(X, f(b, Y))$ then $V = \{X, Y\}$ and let A be one element of the set $\{\{\}, \{X\}, \{Y\}, \{X, Y\}\}$. 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 5.2 *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 5.3 *Given a sequence term $\bar{t}$, the length of $\bar{t}$ is:*

$$\begin{array}{rcl} \textit{length(empty)} & = & 0 \\ \textit{length(seq(A,B))} & = & 1 + \textit{length(B)} \end{array}$$

Definition 5.4 *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 5.2 *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 3.1. 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 5.5 *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:*

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

Definition 5.6 *Given a sequence term, we define sequence term* normalization *as:*

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

Lemma 5.1 *The normalization procedure always terminates yielding a sequence in normal form.*

Proof 5.1 *This can be easily proved by induction on the length of the sequence.*
The base case is trivial:

$$\begin{array}{lcl}
\textit{normalize(empty)} & = & \textit{empty} \\
\textit{normalize(A)} & = & \textit{seq(A,empty)} \quad \textit{if A is a constant or variable.}
\end{array}$$

The induction hypothesis can be directly applied for the third case, by definition normalize $(seq(A, X))$ = normalize (A) ++ normailze (X) and by the induction hypothesis both normalize (A) and normailze (X) terminate. normalize (A) ++ normailze (X) also terminates by the termination of ++.

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 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 [14] 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. We also have the same restriction in the number of occurrences of a variable but we don't need to implement the cycle check since we use Prolog backtracking to obtain all solutions.

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

Example 5.3 *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* $Y = b, b$
- $X = c, c, b, b$ *and* $Y = b$

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

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

Success					
(1)	t	$=*=$	s	$\Longrightarrow$	True, if $t == s$ [1]
(2)	X	$=*=$	t	$\Longrightarrow$	$X = t$ if X does not occur in t.
(3)	t	$=*=$	X	$\Longrightarrow$	$X = t$ if X does not occur in t.
Eliminate					
(4)	$f(\bar{t})$	$=*=$	$f(\bar{s})$	$\Longrightarrow$	$\bar{t} = * = \bar{s}$
(5)	$seq(t_1, \bar{t}_n)$	$=*=$	$seq(s_1, \bar{s}_m)$	$\Longrightarrow$	$t_1 = * = s_1$,
					$\bar{t}_n = * = \bar{s}_m$
(6)	$seq(X, \bar{t})$	$=*=$	$seq(s_1, \bar{s})$	$\Longrightarrow$	$X = s_1$, if X does not occur in s_1,
					$\bar{t} = * = \bar{s}$.
				$\Longrightarrow$	$X = seq(s_1, seq(X_1, empty))$,
					if X does not occur in s_1,
					$normalize(seq(X_1, \bar{t})) = * = normalize(\bar{s})$.
(7)	$seq(t_1, \bar{t})$	$=*=$	$seq(X, \bar{s})$	$\Longrightarrow$	$X = t_1$, if X does not occur in t_1,
					$\bar{t} = * = \bar{s}$.
				$\Longrightarrow$	$X = seq(t_1, seq(X_1, empty))$,
					if X does not occur in t_1.
					$normalize(\bar{t}) = * = normalize(seq(X_1, \bar{s}))$,
(8)	$seq(X, \bar{t})$	$=*=$	$seq(Y, \bar{s})$	$\Longrightarrow$	$X = Y$
					$\bar{t} = * = \bar{s}$.
				$\Longrightarrow$	$X = seq(Y, seq(X_1, empty)$, where
					$normalize(seq(X_1, \bar{t})) = * = normalize(\bar{s})$,
					X, Y and X_1 are distinct variables.
				$\Longrightarrow$	$Y = seq(X, seq(Y_1, empty))$, where
					$normalize(\bar{t}) = * = normalize(seq(Y_1, \bar{s}))$,
					X, Y and Y_1 are distinct variables.
Split					
(9)	$seq(t_1, \bar{t})$	$=*=$	$seq(s_1, \bar{s})$	$\Longrightarrow$	if $t_1 = * = s_1 \Longrightarrow r_1 = * = q_1$ then
					$normalize(seq(r_1, \bar{t})) = * = normalize(seq(q_1, \bar{s}))$
					$\vdots$
				$\Longrightarrow$	if $t_1 = * = s_1 \Longrightarrow r_w = * = q_w$,
					$normalize(seq(r_w, \bar{t}_n)) = * = normalize(seq(q_w, \bar{s}))$,
					where t_1 and s_1 are compound terms.

Figure 1: Transformation rules

Example 5.4 *Given $\bar{t} = f(b, Y, f(X))$ and $\bar{s} = f(X, f(b, Y))$ we have the following equations after projection:*

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

The first and third cases succeed with solution $X = b, Y$ and $X = b, Y = \varepsilon$ respectively.

Example 5.5 *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. In the first case we have $X = a$ and in the second $X = a, X_1$ leading to the new problem $X_1, a = * = a, X_1$ that is exactly the same as the original.

Note that the transformation rules in figure 1 deal with sequence terms. This increases the expressiveness of CLP(Flex) enabling the explicit use of sequence terms in XML processing. The following example illustrates this feature.

Example 5.6 *We defined a syntactic sugar for sequence term in normal form: $<>$ stands for the empty sequence term and $< t_1, \ldots, t_n >$ stands for the sequence term in normal form, $seq(t_1, seq(t_2, \ldots, seq(t_n, empty) \ldots))$.*

In this example we have two XML files, a simple text.xml *file conforming to the following DTD:*

```
<!ELEMENT ref (#PCDATA)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT text (#PCDATA | ref | b)*>
```

and a bib.xml *file with simple references conforming to the DTD:*

```
<!ELEMENT bib (author, name)>
<!ELEMENT bibliography (bib+)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT name (#PCDATA)>
```

Our text.xml *$< ref >$ elements contain author names which appear in the $< author >$ element of* bib.xml. *Our goal is to process* text.xml *and* bib.xml *and generate new* text2.xml *and* bib2.xml *files. The* text2.xml *file is almost the same than* text.xml *but $< ref >$ elements are replaced by new $< i >$ elements where the content is replaced by a number. This number represents the index of that reference ordered by author within the document. The* bib2.xml *contains only the references appearing in* text.xml, *ordered by author, and with a* label *element with the corresponding number that occurs in the* text2.xml. *The program follows:*

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

```
run:-
        xml2pro('text.xml','text.dtd',T),
        process(T).

process(text(T)):-
        process2(T,T2,[],Refs),
        xml2pro('bib.xml','bib.dtd',B),
        add_bib(Refs,B,BibXML,1),
        newdoc(bibliography,BibXML,NewBIB),
        pro2xml(text(T2),'text2.xml'),
        pro2xml(NewBIB,'bib2.xml').

process2(<>,<>,L,L).

process2(A,B,L,RRefs):-
        A =*= <X,ref(R),Y>,
        B =*= <X,i(NewVar),Y2>,!,
        insert_sorted(R,NewVar,L,Refs),
        process2(Y,Y2,Refs,RRefs).

process2(S,S,L,L).

add_bib([],_,[],_).

add_bib([(A,AI)|Refs],B,[bib(label(AI),
                   author(A),name(N))|XML],I):-
        B =*= bibliography(_,bib(author(A),name(N)),_),
        I1 is I + 1,
        atom_chars(I,AI),
        add_bib(Refs,B,XML,I1).
```

The run *predicate starts the translation process. In a first step, the* process2 *predicate translates the original text into a new one where the* $< ref >$ *elements are replaced by new* $< i >$ *elements and the content is replaced by a new free variable. At the same time pairs* $(Ref\_content, FreeVariable)$ *(corresponding to the author in* $< ref >$ *and the new variable in* $< i >$ *respectively) are inserted in a list ordered by* $Ref\_content$*. At the end we have a list of pairs ordered by author. In a second step,* add_bib *queries the bibliography file, retrieving the references found in the text and replacing the free variables with their definite content, thus, avoiding a second processing of the text file. For example, if* text.xml *corresponds to:*

```
<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>
```

and the bib.xml *to:*

```
<bibliography>
<bib>
   <author>Coelho and Florido</author>
   <name>Type-based XML Processing in Logic Programming</name>
</bib>
<bib>
```

```
   <author>W3C</author>
   <name>XSL Transformations (XSLT).
                       http://www.w3.org/TR/xslt/</name>
</bib>
<bib>
   <author>Hosoya</author>
   <name>XDuce: A Typed XML processing language</name>
</bib>
<bib>
   <author>Frish Casagna and Benzaken</author>
   <name>CDuce an XML-centric
                 general-purpose language</name>
</bib>
<bib>
   <author>Pierce</author>
   <name>Xtatic.
    http://www.cis.upenn.edu/~bcpierce/xtatic/</name>
</bib>
</bibliography>
```

the resulting text2.xml *will be:*

```
<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 bib2.xml*:*

```
<bibliography>
<bib>
  <label>1</label>
  <author>
     Frish Casagna and Benzaken
  </author>
  <name> CDuce an XML-centric general-purpose language
  </name>
</bib>
<bib>
  <label>2</label>
  <author>Hosoya</author>
  <name> XDuce: A Typed XML processing language</name>
</bib>
<bib>
  <label>3</label>
  <author>Pierce</author>
  <name> Xtatic.
       http://www.cis.upenn.edu/~bcpierce/xtatic/</name>
</bib>
<bib>
  <label>4</label>
  <author>W3C</author>
  <name> XSL Transformations (XSLT).
           http://www.w3.org/TR/xslt/ </name>
```

```
</bib>
</bibliography>
```

The previous example emphasizes the advantages of CLP(Flex) for XML processing, it shows an highly declarative and compact code and the advantages of using logic variables that permit to solve the problem by processing the text.xml document only once.

5.2 Correctness

We now prove the correctness of our implementation of Kutsia algorithm. In [14] Kutsia proved the correctness of his algorithm with respect to a given semantics for the non-standard unification. We show that our implementation of Kutsia algorithm is correct, i.e, both give the same set of solutions for a given equation. Before the main theorem we present several definitions and lemmas which basically relate different aspects between our implementation and the original presentation of the unification algorithm in [14].

Definition 5.7 *Given a sequence term $\bar{t}$. $\mathcal{T}$ translates $\bar{t}$ into a sequence $\tilde{t}$ (as defined by Kutsia in [14]). $\mathcal{T}$ is defined as:*

$$\begin{array}{rcll} \mathcal{T}(empty) & = & \varepsilon & \\ \mathcal{T}(X) & = & X & \text{if } X \text{ is a constant or variable} \\ \mathcal{T}(f(\bar{t})) & = & f(\mathcal{T}(\bar{t})) & \text{if } f \text{ is function symbol and } f \neq seq \\ \mathcal{T}(seq(A,B)) & = & \mathcal{T}(A),\mathcal{T}(B) & \end{array}$$

Proposition 5.1 *Let $\bar{s}$ be a sequence, $[X/t]$ a standard substitution and $X \leftarrow t$ a substitution as defined by Kutsia in [14]. If t is not a sequence term then*

$$\mathcal{T}(s[X/t]) = \mathcal{T}(s)[X \leftarrow t]$$

Lemma 5.2 *Given the sequence terms, $\bar{t}$ and $\bar{s}$, both in normal form, then:*

$$\mathcal{T}(\bar{t} + +\bar{s}) = \mathcal{T}(\bar{t}), \mathcal{T}(\bar{s})$$

Proof 5.2 *The proof follows by structural induction on $\bar{t}$. When $\bar{t}$ is* empty *the lemma is trivially true. Let's now see that $\mathcal{T}(\bar{t} + +\bar{s}) = \mathcal{T}(\bar{t}), \mathcal{T}(\bar{s})$ also holds when $\bar{t}$ is of the form $seq(A,B)$.*

$$\begin{array}{rcl} \mathcal{T}(\bar{t} + +\bar{s}) & = & \mathcal{T}(seq(A, B + +\bar{s})) \\ & = & \mathcal{T}(A), \mathcal{T}(B + +\bar{s}), \text{ by definition of } \mathcal{T} \\ & = & \mathcal{T}(A), \mathcal{T}(B), \mathcal{T}(\bar{s}), \text{ by the induction hypothesis} \\ & = & \mathcal{T}(\bar{t}), \mathcal{T}(\bar{s}), \text{ since } \bar{t} = seq(A,B) \\ & & \text{and } \mathcal{T}(seq(A,B)) = \mathcal{T}(A), \mathcal{T}(B) \end{array}$$

Lemma 5.3 *Let $X \leftarrow \tilde{t}$ be the substitution defined by Kutsia in [14]. Given the sequence terms $\bar{t}$ and $\bar{s}$ (both in normal form) and a variable X:*

$$\mathcal{T}(normalize(\bar{s}[X/\bar{t}])) = \mathcal{T}(\bar{s})[X \leftarrow \mathcal{T}(\bar{t})]$$

Proof 5.3 *Let's use induction on the structure of $\bar{s}$ to prove lemma 5.3:*

- *If $\bar{s}$ is empty then the result is trivially true.*
- *If $\bar{s}[X/t]$ is in normal form then t is not a sequence term and $\mathcal{T}(t) = t$ and by Proposition 5.1:*

$$\mathcal{T}(s[X/t]) = \mathcal{T}(s)[X \leftarrow t]$$

- *If $\bar{s}[X/\bar{t}]$ is not in normal form then $\bar{s} \equiv seq(A, R)$ ($\equiv$ stands for syntactic equality).*
  $normalize(seq(A, R)[X/t])) =$
  $normalize(A[X/t]) + +normalize(R[X/t])$ *, by the definition of normalize.*
  Then,
  $\mathcal{T}(normalize(seq(A, R)[X/t])) =$
  $= \mathcal{T}(normalize(A[X/t]) + +normalize(R[X/t]))$
  $= \mathcal{T}(normalize(A[X/t])), \mathcal{T}(normalize(R[X/t]))$, *by Lemma 5.2*
  $= \mathcal{T}(A)[X \leftarrow \mathcal{T}(t)], \mathcal{T}(R)[X \leftarrow \mathcal{T}(t)]$, *by the induction hypothesis*
  $= \mathcal{T}(A), \mathcal{T}(R)[X \leftarrow \mathcal{T}(t)]$
  $= \mathcal{T}(seq(A, R))[X \leftarrow \mathcal{T}(t)]$, *by the definition of $\mathcal{T}$*

In the formalization of the unification algorithm, Kutsia aggregates arguments using a dummy function symbol, where we use a sequence. When we have $seq(t_1, seq(t_2, \ldots, seq(t_n, empty) \ldots)$, Kutsia has $g(t_1, \ldots, t_n)$, where g is a new function symbol. The next function relates both notations.

Definition 5.8 *$\mathcal{K}$ translates sequence terms into Kutsia original notation and is defined as follows:*

$\mathcal{K}(t)$	$=$	t	*if t is a constant or variable*
$\mathcal{K}(seq(t, empty))$	$=$	$\mathcal{T}(t)$	
$\mathcal{K}(f(t))$	$=$	$f(\mathcal{T}(t))$	
$\mathcal{K}(seq(t_1, \bar{t_2}))$	$=$	$g(\mathcal{T}(seq(t_1, \bar{t_2})))$	*where g is a new function symbol*

Theorem 5.1 (Correctness) *Let A be the set (possible infinite) of answer substitutions for the query $t = * = s$. Let $\overset{?}{=}$ be the unification operator and $\leftarrow$ be the substitution operator defined by Kutsia in [14]. Let B be the set of Kutsia substitutions for the equation:*

$$\mathcal{K}(t) \overset{?}{=} \mathcal{K}(s)$$

Then,

$$X = q \in A \Leftrightarrow X \leftarrow \mathcal{T}(q) \in B$$

Proof 5.4 *Note that when t and s are not sequence terms then it is obvious by proposition 5.1 that if $t = s \in A$ then $t \leftarrow \mathcal{T}(s) \in F$. Let us prove the theorem for the case of t or s being sequence terms. The proof will proceed by induction on the number of steps of the algorithm.*
The base case corresponds to rules 1,2 and 3. The result follows by direct observation that these rules correspond to similar rules in Kutsia algorithm and by Lemma 5.3. Let us follow to the other rules.

Rule 4 *By the induction hypothesis the property holds for $\mathcal{K}(\bar{t}) \overset{?}{=} \mathcal{K}(\bar{s})$ then it also holds for $f(\bar{t}) = * = f(\bar{s})$.*

Rule 5 *For $t_1 = * = s_1$ the result holds trivially because t_1 and t_2 are not sequence terms and by the induction hypothesis the property also holds for $\bar{t} = * = \bar{s}$.*

Rule 6 *This rule has two possible cases:*

- *$X = s_1$ and $normalize(\bar{t}) = * = normalize(\bar{s})$ (please recall that we are using standard Prolog unification to propagate substitutions and $X = s_1$ will have immediate effect in $\bar{t}$ and $\bar{s}$). Since, by lemma 5.3 $\mathcal{T}(normalize(\bar{t}[X/s_1]) = \mathcal{T}(\bar{t})[X \leftarrow \mathcal{T}(s_1)]$ and $\mathcal{T}(normalize(\bar{s}[X/s_1]) = \mathcal{T}(\bar{s})[X \leftarrow \mathcal{T}(s_1)]$ and $\mathcal{T}$ translates sequence terms in the corresponding sequences we have that, by the induction hypothesis this rule outputs the same set of solutions than Kutsia algorithm.*
- *$X = seq(s_1, seq(X_1, empty))$ and $normalize(seq(X_1, \bar{t})) = * = normalize\ (\bar{s})$ (please recall that X_1 is a new variable). This rule results in a set S of solutions:*

$$S = S' \cup \{X = normalize(seq(s_1, seq(X_1, empty)))\}$$

where S' is normalized by the final normalization step of the algorithm. The result follows by induction for S' noting that for

$$X = normalize(seq(s_1, seq(X_1, empty)))$$

we have the following cases:

- *X_1 is a constant or variable. In this case Kutsia algorithm generates the solution $X \leftarrow s_1, X_1$. Note that, by definition of $\mathcal{T}$, $X \leftarrow s_1, X_1 \Leftrightarrow X \leftarrow \mathcal{T}(seq(s_1, seq(X_1, empty)))$.*
- *X_1 is a sequence. In this case the normalization step will translate the new sequence to its normal form, $X = seq(s_1, \bar{t})$ and $X = \mathcal{T}(seq(s_1, \bar{t}))$ is the same than $X = \mathcal{T}(s_1), \mathcal{T}(\bar{t})$ which originates the substitution $X \leftarrow s_1, \bar{t}$. But this is precisely the result obtained by the same step in Kutsia algorithm.*

Rule 7 *This case is analogous to the previous one.*

Rule 8 *Here we have two variables and three possible cases:*

- *$X = Y$ and $\bar{t} = * = \bar{s}$. The result is a set S' of solutions for $\bar{t} = * = \bar{s}$ plus $X = normalize(Y)$. The result follows for S' by the induction hypothesis. Now, $X = Y$ corresponds to $X \leftarrow \mathcal{T}(Y)$ and there are two possible cases, one is when Y is never instantiated and the output is simply $X = Y$. Then, by the definition of $\mathcal{T}$ the result holds. The other case happens when Y was instantiated to a sequence and in this case by the definition of $\mathcal{T}$ it is also translated to the equivalent result of Kutsia algorithm.*
- *$X = seq(Y, seq(X_1, empty))$ and $seq(X_1, \bar{t}) = * = \bar{s}$. In this case we have the set S' of solutions generated for $seq(X_1, \bar{t}) = * = \bar{s}$, plus the solution $X = normalize($ $seq(Y, seq(X1, empty)))$. The result follows by the induction hypothesis for S'. Now, Y can be instantiated to a variable, constant or sequence term, but every sequence term will be normalized after that. X_1 can be instantiated by different values and then this case becomes the same of rule 6, point 2.*
- *$Y = seq(X, seq(Y_1, empty))$ and $seq(Y_1, \bar{t}) = * = \bar{s}$. This case is similar to the previous one.*

Rule 9 *In this case the unification $t_1 = * = s_1$ originates new problems normalize $(seq(r_i, \bar{t})) = * =$ normalize $(seq(q_i, \bar{s}))$, for $1 \leq i \leq n$. The result follows by the induction hypothesis.*

6 Related Work

Mainstream languages for XML processing such as XSLT ([23]), XDuce ([10]), CDuce ([1]) and Xtatic ([24]) 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 ([9]) and the tool Ontolingua [8] extend first order logic with variable arity function symbols and apply it to knowledge management. Feature terms [21] 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 ([11, 20]), equations over lists of atoms with a concatenation operator ([7]) and equations over free semigroups ([16]). Kutsia ([14]) 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 5.5). 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 ([18]) 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 [6] XML was represented directly by terms of fixed arity. A last reference to some query languages for XML (such as XPathLog [17]) where Prolog style variables are used as an extension to XPath in a query language for XML.

7 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 We thank Temur Kutsia for all the help provided explaining several details of his algorithm. 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*.

References

[1] Vronique 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] 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.

[7] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–90, 1990.

[8] 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.

[9] 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.

[10] 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.

[11] J. Jaffar. Minimal and complete word unification. *Journal of the ACM*, 37(1):47–85, 1990.

[12] 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.

[13] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[14] 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.

[15] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[16] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Math. Sbornik USSR*, 103:147–236, 1977.

[17] Wolfgang May. XPathLog: A Declarative, Native XML Data Manipulation Language. In *International Database Engineering & Applications Symposium (IDEAS '01)*, Grenoble, France, 2001. IEEE.

[18] Pillow: Programming in (Constraint) Logic Languages on the Web. http://clip.dia.fi.upm.es/Software/pillow/pillow.html.

[19] SWI Prolog. http://www.swi-prolog.org/.

[20] Klaus U. Schulz. Word unification and transformation of generalized equations. *Journal of Automated Reasoning*, 11(2):149–184, 1993.

[21] Gert Smolka. Feature constraint logics for unification grammars. *Journal of Logic Programming*, 12:51–87, 1992.

[22] Extensible Markup Language (XML). http://www.w3.org/XML/.

[23] XSL Transformations (XSLT). http://www.w3.org/TR/xslt/, 1999.

[24] Xtatic. http://www.cis.upenn.edu/~bcpierce/xtatic/.