Markup Languages for Functional-Logic Programming

Harold Boley
Deutsches **F**orschungszentrum fuer **K**uenstliche **I**ntelligenz GmbH
Email: boley@dfki.de

Abstract

A 10-step strategy for sharing integrated functional-logic programs or knowledge bases as XML documents is given. Four well-known functional-logic design essentials are revisited for the Web. An XML-based 'vanilla' Prolog (XmlLog) is defined and developed into the Relational-Functional Markup Language (RFML). "Content markup" in MathML is regarded as a (higher-order) sublanguage for Lisp-like notations in XML and is compared to RFML. XML assertion and query languages are unified in this functional-logic setting. XmlLog and subsets of RFML are specified by XML document type definitions. All notions are illustrated with examples, culminating in binary-tree relabeling.

1 Introduction

The Web offers many possibilities to publish and reuse declarative – in particular, functional-logic (FL) – programs or knowledge bases (KBs). Perhaps the most important Web development after HTML has been the Extensible Markup Language (XML) [Har99, Bol00]. An interactive 10-step strategy can share FL programs/KBs as XML documents, as follows:

1. Specify the FL language by an XML document type definition (DTD) at a URL/URI.
2. Convert any to-be-published FL program from its source syntax to an XML document according to the DTD.
3. Upload such an FL/XML document to a Web server for publication.
4. Also offer the FL programs for server-side querying (e.g. CGI) and advertise their XML-document version to search engines etc., ideally using metadata markup (e.g. RDF/XML).
5. Distribute these documents to requesting clients via standard Web protocols (e.g. HTTP).
6. If necessary, transform such an XML document to an FL target language having a different DTD, possibly using an (XSLT) stylesheet.
7. Download any requested XML document at the client site.
8. Convert this XML document to the client's target syntax, possibly using a (CSS or XSLT) stylesheet.
9. Query the target version via the client's program interpreter and optionally download the server's source-program interpreter (once) for client-side querying, ultimately as a browser plug-in.
10. Reuse the target version, say by incorporating it into existing FL programs.

The first five steps have been investigated using the Relfun-derived [Bol99a] Relational-Functional Markup Language (RFML) [Bol99b]. The remaining five steps constitute a challenge for the maturing FL and XML technologies.

Since the Prolog-like source syntax of Relfun can be switched to a Lisp-like intermediate syntax that is very close to the underlying tree-structured abstract syntax, the definition of RFML markup should appear quite natural to Relfun users. The 'markup abstraction' of non-declarative programming languages has to span a larger distance, but can also appear as natural as in the Java-to-JavaML transition [http://www.cs.washington.edu/homes/gjb/JavaML/]. Of course, because of steps 2 and 8 of the above strategy, sharing partners need not see any markup at all.

Besides the Web-based program-sharing advantages of internal markups, XML tools will permit program markups with DTDs to be *analyzed and transformed* in various useful ways. Because of their declarative semantics, this should again be simplified by FL languages. XML program markup will also help *literate programming*, which combines source and documentation [http://www.oasis-open.org/cover/xmlLitProg.html]. For FL languages this should be even easier, since a declarative source is itself a kind of mathematical documentation.

After this introduction, the paper discusses, in section 2, design issues for functional-logic Web languages on the basis of the RFML experience. It then compares, in section 3, FL markup languages with logic-only (XmlLog) and, in section 4, functional-only (MathML) markup languages. It subsequently discusses, in section 5, RFML as an FL-unified XML assertion and query language. Finally, in section 6 and appendix A, it uses the relabeling of binary trees as a larger FL example. The conclusions, in section 7, deal with further and related work.

2 FL Design Essentials Revisited for the Web

The combination of two essential properties of relational (logic) programming (R1, R2) and of functional programming (F1, F2) has been explored by Relfun [Bol99a]. Here we revisit these 'essentials' for the Web, showing that they acquire some added significance in this context.

(R1) The relational essential of *logic variables and non-ground terms* allows for partial data structures, which can be completed incrementally. It has been applied early on to message passing in agent environments [Kor83, Sha84]: A message consisting of a non-ground term is like one consisting of a form, but when the receiver binds subterm variables, analogous to filling out form slots, this is 'immediately' visible to the sender, if the sender and receiver are concurrent processes sharing these logic variables. The experience with Concurrent Prolog influenced the 'Virtual Places' technology for the Web [Sha95] [http://www.ubique.com/]. However, it seems that this 'non-ground message' use of logic variables has not yet been fully exploited for Web languages. A more recent application of logic variables has been in server-side Web scripting [Han00].

(R2) The relational essential of *"don't know" non-determinism* as implemented by backtracking permits high-level algorithms in a sequential environment [Kow83], but the ensuing implementation incompleteness [Llo87] becomes problematic when clause definitions are highly distributed over the Web [Bol96]. (Committed-choice languages such as Concurrent Prolog employ the notion of *"don't care" non-determinism.*) On the other hand, Web users cannot avoid certain interactive kinds of "don't know" non-determinism. Two extant examples are: (1) Mirror sites require users to choose one out of several URLs and come back to this 'choice point' should problems occur during subsequent navigation. (2) Search-engine results are often only partially ranked and require users to select from long lists of hits, coming back to unexplored list items after possibly long further 'navigation subtrees'. For expressing some kind of non-determinism in Web metadata, RDF has thus introduced an *Alternative* container [LS99].

(F1) The functional essential of *application values* enables nestings such that inner applications return their values to outer applications. Nestings can be principally evaluated using the simpler call-by-value (eager) strategy or the more expressive call-by-need (lazy) strategy, under which, however, programs can be harder to debug [Mor82]. In the Web, while lazy evaluation may be used to model concurrent processes, eager evaluation facilitates remote function calls: Since references to nested parameters are hard to manage over a network, it is advantageous to evaluate all parameters at the sender's site and copy their values into the message to the receiver.

(F2) The functional essential of *higher-order operators* introduces an abstraction layer via operators that take operators as parameters and/or values. For the Web this may be used to provide versions of a (say, sorting) operator by just transmitting various – source or byte code – parameter operators (say, comparison relations) instead of transmitting near-copies of the entire operator. (A parameterized `qsort` is discussed in [Bol99a].) For containers in RDF as well as Java (or rather Pizza [http://www.cis.unisa.edu.au/~pizza]), higher-order operators provide a convenient element-processing technique. Moreover, specialized operators can be generated by a server-side higher-order operator on client demands. There will certainly be more Web uses of this powerful abstraction method, perhaps even for higher-order unification as implemented in Lambda-Prolog [NM98] [http://www.cse.psu.edu/~dale/lProlog/].

3 Purely Relational Markup in XmlLog and RFML

XmlLog is a 'vanilla' markup language for purely relational specifications in XML [Bol00] [http://www.dfki.uni-kl.de/~boley/xmlp-engl.ps], realizing the non-groundness/non-determinism essentials (R1) and (R2) from section 2. Here we discuss its (essentially, specialization) kinship with RFML.

An individual (or variable or predicate) in XmlLog becomes an `ind` (or `var` or `relator`) element of the form `<ind> ... </ind>` (or analogous). A `relationship` element applying a `relator` to `ind`/`var` arguments is usable as a query.

In XmlLog a `relationship` becomes a Datalog (i.e., constructorless) **fact** through a superimposed `hn` (i.e., Horn-clause) element. Let us consider a fact corresponding to the tuple `(eurostar,fred)` from a `carry(Carrier,Person)` table of a relational database: `carry(eurostar,fred).` It is marked up in XmlLog as follows (on the left):

```
<hn>
  <relationship>
    <relator> carry </relator>
    <ind> eurostar </ind>
    <ind> fred </ind>
  </relationship>
</hn>
```

```
<hn>
  <pattop>
    <con> carry </con>
    <con> eurostar </con>
    <con> fred </con>
  </pattop>
</hn>
```

More generally, A Datalog **rule** in XmlLog is asserted as an `hn` element that displays two or more subelements, namely, the head-`relationship` element followed by at least one body-`relationship` element. For example, the rule with exactly one body `relationship`

```
travel(Someone,channel-tunnel) :- carry(eurostar,Someone).
```

in XmlLog becomes (on the left)

```
<hn>
  <relationship>
    <relator> travel </relator>
    <var> someone </var>
    <ind> channel-tunnel </ind>
  </relationship>
  <relationship>
    <relator> carry </relator>
    <ind> eurostar </ind>
    <var> someone </var>
  </relationship>
</hn>
```

```
<hn>
  <pattop>
    <con> travel </con>
    <var> someone </var>
    <con> channel-tunnel </con>
  </pattop>
  <callop>
    <con> carry </con>
    <con> eurostar </con>
    <var> someone </var>
  </callop>
</hn>
```

Let us answer the query travel(fred,Where) in the XmlLog version (on the left)

```
<relationship>
  <relator> travel </relator>
  <ind> fred </ind>
  <var> where </var>
</relationship>
```

```
<callop>
  <con> travel </con>
  <con> fred </con>
  <var> where </var>
</callop>
```

It unifies with the rule head via the bindings <var> someone </var> = <ind> fred </ind> and <var> where </var> = <ind> channel-tunnel </ind>. The resulting internal XmlLog query corresponding to carry(eurostar,fred) is then successful with the fact. The entire inference is carried out as an SLD-resolution step [Llo87].

Our XmlLog markup for Datalog can be extended to full Prolog and formalized as follows. A Horn clause hn has one (fact) or more (rule) relationship subelements. A relationship element applies a relator to zero or more arguments that can now also be structures. A struc element applies a constructor to arguments identical to those of relator applications.

The DTD for XmlLog, then, just consists of these ELEMENT declarations for a knowledge base kb of zero or more hn clauses (also at URL/URI [http://www.relfun.org/xmllog/xmllog.dtd]):

```
<!ELEMENT kb           (hn*) >
<!ELEMENT hn           (relationship, relationship*) >
<!ELEMENT relationship (relator,     (ind | var | struc)*) >
<!ELEMENT struc        (constructor, (ind | var | struc)*) >
<!ELEMENT relator      (#PCDATA) >
<!ELEMENT constructor  (#PCDATA) >
<!ELEMENT ind          (#PCDATA) >
<!ELEMENT var          (#PCDATA) >
```

RFML is an FL markup language integrating Prolog- and Lisp-like specifications according to the design principles of section 2. Its relational subset was already illustrated above (on the right) along with the XmlLog examples. Its functional subset will be exemplified in section 4.

RFML enriches XmlLog by using *valued* clauses, which on success return some value, besides yielding possible bindings. A *Horn-like* clause, again marked up by hn, implicitly returns the success value true. A *footed* clause, marked up by ft, has an extra last 'foot' premise, which evalutates to an arbitrary value. (Thus, hn-clauses could be replaced by true-footed ft-clauses.) Another RFML characteristic, following from the valuedness essential

(F1) of section 2, is differentiating XmlLog's `relationships` into `callops`, in the body/foot, and `pattops`, in the head: A `call` of an `operator` may use as its (mandatory) operator and (optional) arguments a `constant`, `variable`, `structure`, or again a `callop`. A (definition) `pattern` of an `operator` uses the same content as `callop` except that it may not contain any `callop`, reflecting the "constructor discipline" [O'D85] of valued clauses. A `structure` has the same content as `pattop`. Apart from `callops`, both kinds of clauses may also have `con/var/struc` terms as body/foot premises. Because of the higher-order essential (F2) from section 2, `structures` and `variables` may occur in operator position, and `constants` need not be XmlLog's `individuals`, but can also be its `relators` and `constructors`.

A DTD for an RFML subset (omitting `rest` arguments, `anonymous` variables, and `tuples`, as well as any built-ins etc.) consists of these `ELEMENT` declarations for an `rfml` document with zero or more `hn/ft` clauses (cf. test URL [http://www.relfun.org/rfml/tiny-rfml.dtd]):

```
<!ELEMENT rfml          (hn | ft)* >
<!ELEMENT hn            (pattop, (con | var | struc | callop)*) >
<!ELEMENT ft            (pattop, (con | var | struc | callop)+) >
<!ELEMENT callop        ((con | var | struc | callop)+) >
<!ELEMENT pattop        ((con | var | struc)+) >
<!ELEMENT struc         ((con | var | struc)+) >
<!ELEMENT con           (#PCDATA)>
<!ELEMENT var           (#PCDATA)>
```

An extended DTD for RFML will be introduced in section 6. The full DTD and a detailed description with many examples can be found in [Bol99b] [http://www.relfun.org/rfml/].

4 Purely Functional Markup in MathML and RFML

MathML "content markup" [http://www.w3.org/TR/MathML2/] can be regarded as a sublanguage for purely functional, Lisp-like specifications in XML. Thus, it can be naturally re-expressed as RFML markup.

As a simple example, consider the (first-order) function `double(x) := x*2`, definable in Lisp as `(defun double (x) (* x 2))` or in Relfun as `double(X) :& *(X,2).` It can be equivalently marked up in MathML (on the left) and RFML (on the right) as follows:

```
<declare type="fn">                    <ft>
  <ci> double </ci>                      <pattop>
  <lambda>                                 <con> double </con>
    <bvar><ci> x </ci></bvar>              <var> x </var>
    <apply>                              </pattop>
      <times/>                           <callop>
      <ci> x </ci>                         <con> * </con>
      <cn> 2 </cn>                         <var> x </var>
    </apply>                               <con> 2 </con>
  </lambda>                              </callop>
</declare>                             </ft>
```

The top-level stylistic difference is the following. MathML associates the function's name with a `lambda` expression whose `bound variable` is the function's argument and whose body is the function's defining (`apply`) expression. RFML associates the function's name-argument pattern with the function's defining (`callop`) expression. A further difference concerns built-in calls. MathML uses a built-in function (e.g. for multiplication: `times`), unlike a user-defined function, as a canonical empty element. RFML uses a built-in function (e.g. for multiplication: "`*`"), like a user-defined function, as the content of generic `con` tags.

As a second example, consider the (higher-order) function `compose(f,g)(x) := f(g(x))`, in Relfun defined as `compose[F,G](X) :& F(G(X))`. Our proposed markup in MathML 2.0 (whose DTD has not yet been 'URIfied' at the point of this writing) is shown on the left; the corresponding RFML markup is shown on the right:

```
<declare type="fn" nargs="2">
  <ci> compose </ci>
  <lambda>
    <bvar><ci type="fn"> f </ci></bvar>
    <bvar><ci type="fn"> g </ci></bvar>
    <lambda>
      <bvar><ci> x </ci></bvar>
      <apply>
        <ci> f </ci>
        <apply>
          <ci> g </ci>
          <ci> x </ci>
        </apply>
      </apply>
    </lambda>
  </lambda>
</declare>
```

```
<ft>
  <pattop>
    <struc>
      <con> compose </con>
      <var> f </var>
      <var> g </var>
    </struc>
    <var> x </var>
  </pattop>
  <callop>
    <var> f </var>
    <callop>
      <var> g </var>
      <var> x </var>
    </callop>
  </callop>
</ft>
```

Here, the main difference is the following. MathML uses an outer `lambda` expression whose `b`ound `var`iables are the functional paramenters and returns an inner `lambda` expression whose `b`ound `var`iable is the object argument. RFML uses a `struc`ture for the functional paramenters and an ordinary object argument.

Since, of course, MathML (up to the current MathML 2.0) was not conceived for LP, defining and calling relations in MathML would be less obvious than shown for functions above. For example, free query variables in relation calls would have to be explicitly scoped by an existential quantifier. This would also give rise to the need for such quantifiers in MathML functional-logic programs. One solution might be to use MathML `lambda` expressions to express both universal and existential quantifiers, as demonstrated by the higher-order functional-logic language Lambda-Prolog [NM98].

5 FL-Unified XML Assertion and Query Language

There are many possibilities for cross-fertilizations between XML (incl. XML Schema and XML Query) [Har99] and FL-programming languages.

For instance, as discussed in [Bol00], returned values of FL languages can reflect the value-oriented result delivery of XQL [http://www.w3.org/TandS/QL/QL98/pp/xql.html], their non-determinism can implement multiple results, and higher-order operators can realize filter queries.

Here we exemplify the following FL suggestion for XML. Instead of separating XML-element assertion and querying, an FL/XML language should incorporate both of these aspects in a uniform manner: Asserted clauses can contain queries in their premises, and queries are also usable on an interactive top-level.

For example, the *unconditional (ground) equation*

```
pay(john,fred,17.95)  =  cheque
```

which in Relfun's left-to-right-directed version

```
pay(john,fred,17.95) :&  cheque.
```

defines a `pay` function returning a payment method, in RFML becomes the following markup:

```
<ft>
   <pattop>
      <con> pay </con>
      <con> john </con>
      <con> fred </con>
      <con> 17.95 </con>
   </pattop>
  <con>
     cheque
  </con>
</ft>
```

It can be queried (non-ground) directly via

```
<callop>
   <con> pay </con>
   <var> customer </var>
   <var> merchant </var>
   <var> price </var>
</callop>
```

binding the three variables to the corresponding constants in the definition pattern and returning the constant `cheque`.

It can also be queried indirectly by the *conditional (non-ground) equation*

```
acquire(Customer,Merchant,Product,Price)  =  pay(Customer,Merchant,Price)
                                       if satisfied(Customer,Product,Price)
```

which in Relfun's left-to-right-directed version (using the natural 'condition-action' order)

```
acquire(Customer,Merchant,Product,Price)  :-
                                          satisfied(Customer,Product,Price) &
                                              pay(Customer,Merchant,Price).
```

defines an `acquire` operation, whose RFML version below uniformly marks up the relational (`satisfied`) and functional (`pay`) subqueries via `callop` subelements. This uniformity reflects Relfun's identification of relations with characteristic (`true`-valued) functions.

```
<ft>
   <pattop>
      <con> acquire </con>
      <var> customer </var>
      <var> merchant </var>
      <var> product </var>
      <var> price </var>
   </pattop>
   <callop>
      <con> satisfied </con>
      <var> customer </var>
      <var> product </var>
      <var> price </var>
   </callop>
   <callop>
      <con> pay </con>
      <var> customer </var>
      <var> merchant </var>
      <var> price </var>
   </callop>
</ft>
```

6 FL Programming Exemplified by Binary-Tree Relabeling

Generalizing the `maxtree` problem solved purely relationally in [BM97], we consider a relabeling transformation for labeled binary trees specified thus (note partial data structure):

Let all `void` *(sub)trees have the user-provided 'binding'* `Mixvoid` *and the 'value'* `void`*. For a non-*`void` *(sub)tree with some* `Label` *assume that the bindings of its* `Left` *and* `Right` *subtrees are* `Mixleft` *and* `Mixright`*, respectively, and its values,* `Newleft` *and* `Newright`*, respectively; then its own binding,* `Mixsofar`*, is the result of applying an arbitrary ternary function,* `Terfun`*, to* `Label`*,* `Mixleft`*, and* `Mixright`*, in this order, and its own value is a* `tree` *with label* `Mix`*, whose value is specified below, and subtrees* `Newleft` *and* `Newright`*. The* `Mix` *of all non-*`void` *(sub)trees is identical to the* `Mixsofar` *of the whole tree.*

In FL programming this can be solved by defining a main function `mixtree` together with its workhorse function `walktree`, corresponding to a 'top-down' version of the 'bottom-up' problem specification (we employ Relfun's `let`-like single-assignment primitive "`.=`"):

```
mixtree[Terfun,Mixvoid](Tree) :& walktree[Terfun,Mixvoid](Tree,Mix,Mix).

walktree[Terfun,Mixvoid](void,Mix,Mixvoid) :& void.
walktree[Terfun,Mixvoid](tree[Label,Left,Right],Mix,Mixsofar) :-
  Newleft  .= walktree[Terfun,Mixvoid](Left,Mix,Mixleft),
  Newright .= walktree[Terfun,Mixvoid](Right,Mix,Mixright),
  Mixsofar .= Terfun(Label,Mixleft,Mixright)
  &
  tree[Mix,Newleft,Newright].
```

Its markup according to a "`.=`"-extended RFML DTD is shown in appendix A.

Let us consider this definition in the light of the design essentials of section 2:

(F1) The returned values of `walktree` recursions are bound to single-assignment variables `Newleft` and `Newright`, but could also be nested directly into a `tree`-constructing function.

(F2) `Terfun` is handed through as a functional parameter `[Terfun,...]` until its value is called via `Terfun(Label,Mixleft,Mixright)`.

(R1) The logic variable `Mix` occurs free in the top-level call of the `walktree` function, is put into the label position of all non-`void` subtrees (which thus constitute a partial data structure), and is eventually bound (everywhere in that data structure) to the `Mixsofar` of the whole tree via its duplicate top-level use.

(R2) While the definition itself appears (ground-)deterministic, its `Terfun` parameter can 'import' a non-deterministic function, causing overall non-determinism (see second sample call below).

For example, simulating the `maxtree` relation in [BM97], we can employ a (built-in) `maximum` function as a parameter of the `mixtree` function, as in the call

```
mixtree[max,0](tree[2,
                    tree[5,
                         tree[3,void,void],
                         tree[4,tree[7,void,void],tree[1,void,void]]],
                    tree[1,tree[4,void,void],tree[8,void,void]]])
```

returning the following `max`-relabeled `tree` (for a corresponding `minimum`-parameterized `mixtree` call replace all `8`'s by `1`'s):

```
tree[8,
     tree[8,
          tree[8,void,void],
          tree[8,tree[8,void,void],tree[8,void,void]]],
     tree[8,tree[8,void,void],tree[8,void,void]]]
```

On the other hand, with the user-defined non-deterministic `ternary choice` function

```
terchoice(X,Y,Z) :& X.
terchoice(X,Y,Z) :& Y.
terchoice(X,Y,Z) :& Z.
```

as the functional parameter, the call

```
mixtree[terchoice,epsilon](tree[alpha,
                                tree[beta,void,void],
                                tree[gamma,void,void]])
```

non-deterministically enumerates these `terchoice`-relabeled `trees`:

```
tree[alpha,tree[alpha,void,void],tree[alpha,void,void]],
tree[beta,tree[beta,void,void],tree[beta,void,void]],
tree[gamma,tree[gamma,void,void],tree[gamma,void,void]],
tree[epsilon,tree[epsilon,void,void],tree[epsilon,void,void]]
```

7 Related Work and Conclusions

The first well-known language that has provided Horn-clause markup in the Web is SHOE (Simple HTML Ontology Extensions) [HHL99]. XmlLog defines a 'vanilla' Horn language as an XML DTD, extended to RFML, a 'lightweight' FL-integration language. RFML has been implemented as a (Web-)output syntax for FL knowledge bases (cf. appendix A) and for FL computations. Further descriptions and download information are available at [http://www.relfun.org/rfml/]. This FL/XML kernel could be extended for further FL languages.

It will be a problem, of course, to share programs between the large number of FL proposals, including Curry [Han97], TOY [LFSH99], Escher [Ede99], Lambda-Prolog [NM98], Mercury [SHC96], and Oz [VHB+97]. As one approach, we could develop a merely syntactic common markup language as done by the MathML community (who, however, have the advantage of the generally agreed upon semantics of mathematical expressions); experience with different semantics for the same markup might then encourage semantic convergence. Alternatively, we could try to first converge on semantic differences such as determinism vs. non-determinism, eager vs. lazy evaluation, and first-order vs. higher-order logics; on this basis, a semantic markup language with common FL semantics could be developed. In practice a combination of both approaches may turn out to be the best solution.

In order to bridge some of the remaining differences, step 6 of our strategy for sharing FL programs in section 1 proposes to transform between FL markups having different DTDs, possibly using an (XSLT) stylesheet. This can also be used to transform between subsets of FL languages and logic-only resp. functional-only languages, as illustrated by the juxtaposed RFML and XmlLog (section 3) resp. MathML (section 4) elements.

Much of the knowledge on the Web constitutes definitions of relations and functions. Markup languages for FL programming will permit such "Functional-Logic Knowledge on the Web". Their development should take into account previous experience such as with the Knowledge Interchange Format (KIF), which can also be regarded as an FL language. KIF is a full first-order logic language that has been proposed as an ANSI standard [http://logic.stanford.edu/kif/dpans.html]. However, full KIF currently seems still too big for practical knowledge sharing. For example, only a (Horn-like) subset of KIF has been employed in the recent Business Rules Markup Language (BRML) [GL99]. Thus, a complementary approach has been pursued with Relfun/RFML: gradually extending a minimal kernel towards a practical FL language.

A More RFML: `is`-Extended DTD and `mixtree` Markup

An RFML DTD extended for a (top-level) single-assignment primitive "`.=`" uses a new `is` element and modified `hn`/`ft` elements (see test URL [http://www.relfun.org/rfml/tiny-is-rfml.dtd]):

```
. . .
<!ELEMENT hn         (pattop, (con | var | struc | is | callop)*) >
<!ELEMENT ft         (pattop, (con | var | struc | is | callop)+) >
. . .
<!ELEMENT is         ((con | var | struc), (con | var | struc | callop)) >
. . .
```

Then, the RFML markup that was automatically generated from the `mixtree` function of section 6 (via Relfun's commands `style xml` and `listing`) gives us the following Web-based FL KB (use XML browser for uploaded version at URL/URI [http://www.relfun.org/lib/mixtree.rfml]):

```
<ft>
  <pattop>
    <struc>
      <con>mixtree</con>
      <var>terfun</var>
      <var>mixvoid</var>
    </struc>
    <var>tree</var>
  </pattop>
  <callop>
    <struc>
      <con>walktree</con>
      <var>terfun</var>
      <var>mixvoid</var>
    </struc>
    <var>tree</var>
    <var>mix</var>
    <var>mix</var>
  </callop>
</ft>


<ft>
  <pattop>
    <struc>
      <con>walktree</con>
      <var>terfun</var>
      <var>mixvoid</var>
    </struc>
    <con>void</con>
    <var>mix</var>
    <var>mixvoid</var>
  </pattop>
  <con>void</con>
</ft>

<ft>
  <pattop>
    <struc>
      <con>walktree</con>
      <var>terfun</var>
      <var>mixvoid</var>
    </struc>
    <struc>
      <con>tree</con>
      <var>label</var>
      <var>left</var>
      <var>right</var>
    </struc>
    <var>mix</var>
    <var>mixsofar</var>
  </pattop>
  <is>
    <var>newleft</var>
    <callop>
      <struc>
        <con>walktree</con>
        <var>terfun</var>
        <var>mixvoid</var>
      </struc>
      <var>left</var>
      <var>mix</var>
      <var>mixleft</var>
    </callop>
  </is>
  <is>
    <var>newright</var>
    <callop>
      <struc>
        <con>walktree</con>
        <var>terfun</var>
        <var>mixvoid</var>
      </struc>
      <var>right</var>
      <var>mix</var>
      <var>mixright</var>
    </callop>
  </is>
  <is>
    <var>mixsofar</var>
    <callop>
      <var>terfun</var>
      <var>label</var>
      <var>mixleft</var>
      <var>mixright</var>
    </callop>
  </is>
  <struc>
    <con>tree</con>
    <var>mix</var>
    <var>newleft</var>
    <var>newright</var>
  </struc>
</ft>
```

References

[BM97] Johan Boye and Jan Maluszynski. Directional types and the annotation method. *Journal of Logic Programming*, 33(3):179–220, December 1997.

[Bol96] Harold Boley. Knowledge Bases in the World Wide Web: A Challenge for Logic Programming. In Paul Tarau, Andrew Davison, Koen De Bosschere, and Manuel Hermenegildo, editors, *Proc. JICSLP'96 Post-Conference Workshop on Logic Programming Tools for INTERNET Applications*, pages 139–147. COMPULOG-NET, Bonn, Sept. 1996. Revised versions in: International Workshop "Intelligent Information Integration", KI-97, Freiburg, Sept. 1997; DFKI Technical Memo TM-96-02, Oct. 1997.

[Bol99a] Harold Boley. *A Tight, Practical Integration of Relations and Functions.* Number 1712 in Lecture Notes in Artificial Intelligence (LNAI). Springer-Verlag, Berlin, Heidelberg, September 1999.

[Bol99b] Harold Boley. The Relational-Functional Markup Language RFML. Technical report, Stanford Medical Informatics, December 1999.

[Bol00] Harold Boley. Beziehungen zwischen Logikprogrammierung und XML. In François Bry, Ulrich Geske, and Dietmar Seipel, editors, *14. WLP – Workshop Logische Programmierung*, pages 19–34. Institut für Informatik, Univ. Würzburg, GMD Report 90, January 2000.

[Ede99] Kerstin Eder. Yet another way of Set-processing: The Escher style and its implementation. In *Proceedings of the Workshop on Declarative Programming with Sets (DPS'99)*, pages 37–50, I-43100 Parma, Italy, September 1999. Quaderni del Dipartimento di Matematica, n. 200, Universita degli Studi di Parma.

[GL99] Benjamin N. Grosof and Yannis Labrou. An approach to using XML and a rule-based content language with an agent communication language. Research report RC. International Business Machines Corporation. Research Division; 21491 International Business Machines Corporation. Research Division. Research report; RC 21491 RC 21491 (96965), IBM T.J. Watson Research Center, Yorktown Heights, NY, USA, 1999.

[Han97] Michael Hanus. A Unified Computation Model for Functional and Logic Programming. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 80–93, Paris, France, January 1997.

[Han00] Michael Hanus. Server Side Web Scripting in Curry. These Proceedings, September 2000.

[Har99] Elliotte Rusty Harold. *XML Bible.* IDG Books Worldwide Inc., San Mateo, CA, USA, 1999.

[HHL99] Jeff Heflin, James Hendler, and Sean Luke. SHOE: A Knowledge Representation Language for Internet Applications. Technical Report CS-TR-4078, University of Maryland, College Park, October 1999.

[Kor83] William A. Kornfeld. Equality for Prolog. In Alan Bundy, editor, *Proceedings of the 8th International Joint Conference on Artificial Intelligence*, pages 514–519, Karlsruhe, FRG, August 1983. William Kaufmann.

[Kow83] Robert Kowalski. Logic programming. In *INFORMATION PROCESSING 83, Proc. IFIP 9th World Computer Congress*, pages 133–145, Paris, 1983.

[LFSH99] F. J. Lopez-Fraguas and J. Sanchez-Hernandez. TOY: A multiparadigm declarative system. *Lecture Notes in Computer Science*, 1631, 1999.

[Llo87] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, Heidelberg, New York, 1987.

[LS99] Ora Lassila and Ralph R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. Recommendation REC-rdf-syntax-19990222, W3C, February 1999.

[Mor82] James H. Morris. Real programming in functional languages. In J. Darlington, P. Henderson, and D. A. Turner, editors, *Functional Programming and its Applications*, pages 129–176. Cambridge University Press, 1982.

[NM98] Gopalan Nadathur and Dale Miller. Higher-order logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, chapter 8. Oxford University Press, 1998.

[O'D85] M. J. O'Donnell. *Equational Logic as a Programming Language.* MIT Press, Cambridge, Mass., 1985.

[Sha84] Ehud Shapiro. Systems programming in Concurrent Prolog. In Ken Kennedy, editor, *Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages*, pages 93–105, Salt Lake City, UT, January 1984. ACM Press.

[Sha95] Ehud Shapiro. Virtual places – A foundation for human interaction. In *Second International World-Wide Web Conference: Mosaic and the Web, Chicago, IL, October 17–20, 1994*, Urbana, IL 61801, USA, 1995. National Center for Supercomputer Applications, University of Illinois at Urbana-Champaign.

[SHC96] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3):17–64, October–November 1996.

[VHB+97] Peter Van Roy, Seif Haridi, Per Brand, Gert Smolka, Michael Mehl, and Ralf Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.