Declarative Processing of Semistructured Web Data

Michael Hanus

Institut für Informatik, CAU Kiel, D-24098 Kiel, Germany.

mh@informatik.uni-kiel.de

Technical Report 1103, March 2011

Abstract

In order to give application programs access to data stored in the web in semistructured formats, in particular, in XML format, we propose a domain-specific language for declarative processing such data. Our language is embedded in the functional logic programming language Curry and offers powerful matching constructs that enable a declarative description of accessing and transforming XML data. We exploit advanced features of functional logic programming to provide a high-level and maintainable implementation of our language. Actually, this paper contains the complete code of our implementation so that the source text of this paper is an executable implementation of our language.

1 Motivation

Nowadays, huge amounts of information are available in the world-wide web. Much of this information is also available in semistructured formats so that it can be automatically accessed by application programs. The extensible markup language (XML) is often used as an exchange format for such data. Since data in XML format are basically term structures, XML data can be (in principle) easily processed with functional or logic programming languages: one has to define a term representation of XML data in the programming language, implement a parser from the textual XML representation into such terms, and exploit pattern matching to implement the specific processing task.

In practice, such an implementation causes some difficulties due to the fact that the concrete data formats are complex or evolve over time:

- For many application areas, concrete XML languages are defined. However, they are often quite complex so that it is difficult or tedious to deal with all details when one is interested in extracting only some parts of the given data.

```
<contacts>
  <entry>
    <name>Hanus</name>
    <first>Michael</first>
    <phone>+49-431-8807271</phone>
    <email>mh@informatik.uni-kiel.de</email>
    <email>hanus@acm.org</email>
  </entry>
  <entry>
    <name>Smith</name>
    <first>William</first>
    <nickname>Bill</nickname>
    <phone>+1-987-742-9388</phone>
  </entry>
</contacts>
```

Figure 1: A simple XML document

- For more specialized areas without standardized XML languages, the XML format might be incompletely specified or evolves over time. Thus, application programs with standard pattern matching must be adapted if the data format changes.

For instance, consider the XML document shown in Figure 1 which represents the data of a small address book. As one can see, the two entries have different information fields: the first entry contains two email addresses but no nickname whereas the second entry contains no email address but a nickname. Such data, which is not uncommon in practice, is also called "semistructured" [1]. Semistructured data causes difficulties when it should be processed with a declarative programming language by mapping the XML structures into data terms of the implementation language. Therefore, various distinguished languages for processing XML data have been proposed.

For instance, the language XPath[1] provides powerful path expressions to select sub-documents in XML documents. Although path expressions allow flexible retrievals by the use of wildcards, regular path expressions, stepping to father and sibling nodes etc, they are oriented towards following a path through the document from the root to the selected sub-documents. This gives them a more imperative rather than a descriptive or declarative flavor. The same is true for query and transformation languages like XQuery[2] or XSLT[3] which are based on the XPath-oriented style to select the required sub-documents.

As an alternative to path-oriented processing languages, the language Xcerpt [8, 10] is a proposal to exploit ideas from logic programming in order to provide a declarative method to select and transform semistructured data in XML format. In contrast to pure

[1] http://www.w3.org/TR/xpath
[2] http://www.w3.org/XML/Query/
[3] http://www.w3.org/TR/xslt

logic programming, Xcerpt proposes matching with partial term structures for which a specialized unification procedure, called "simulation unification" [9], has been developed. Since matching with partial term structures is a powerful feature that avoids many problems related to the evolution of web data over time, we propose a language with similar features. However, our language is an embedded domain-specific language (eDSL). Due to the embedding into the functional logic programming language Curry [21], our language for XML processing has the following features and advantages:

- The selection and transformation of incompletely specified XML data is supported.
- Due to the embedding into a universal programming language, the selected or transformed data can be directly used in the application program.
- Due to the use of advanced functional logic programming features, the implementation is straightforward and can be easily extended with new features. Actually, this paper contains the complete source code of the implementation.
- The direct implementation in a declarative language results in immediate correctness proofs of the implementation.

In the following, we present our language for XML processing together with their implementation. Since the implementation exploits features of modern functional logic programming languages, we review them in the next section before presenting our eDSL.

2 Functional Logic Programming and Curry

Curry [21] is a declarative multi-paradigm language combining features from functional programming (demand-driven evaluation, parametric polymorphism, higher-order functions) and logic programming (computing with partial information, unification, constraints). Recent surveys are available in [6, 18]. The syntax of Curry is close to Haskell[4] [23]. In addition, Curry allows free (logic) variables in conditions and right-hand sides of defining rules. The operational semantics is based on an optimal evaluation strategy [2] which is a conservative extension of lazy functional programming and (concurrent) logic programming.

A Curry program consists of the definition of data types and operations on these types. Note that in a functional logic language operations might yield more than one result on the same input due to the logic programming features. Thus, Curry operations are not functions in the classical mathematical sense so that they are sometimes called "nondeterministic functions" [14]. Nevertheless, a Curry program has a purely declarative semantics where nondeterministic operations are modeled as set-valued functions (to be more precise, down-closed partially ordered sets are used as target domains in order to cover non-strictness, see [14] for a detailed account of this model-theoretic semantics).

For instance, Curry contains a *choice* operation defined by:

[4]Variables and function names usually start with lowercase letters and the names of type and data constructors start with an uppercase letter. The application of f to e is denoted by juxtaposition ("$f\ e$").

```
x ? _ = x
_ ? y = y
```

Thus, the expression "`0 ? 1`" has two values: `0` and `1`. If expressions have more than one value, one wants to select intended values according to some constraints, typically in conditions of program rules. A *rule* has the form "$f\ t_1 \ldots t_n$ `|` c `=` e" where the (optional) condition c is a *constraint*, i.e., an expression of the built-in type `Success`. For instance, the trivial constraint `success` is a value of type `Success` that denotes the always satisfiable constraint. Thus, we say that a constraint c is *satisfied* if it can be evaluated to `success`. An *equational constraint* e_1 `=:=` e_2 is satisfiable if both sides e_1 and e_2 are reducible to unifiable values. Furthermore, if c_1 and c_2 are constraints, c_1 `&` c_2 denotes their concurrent conjunction (i.e., both argument constraints are concurrently evaluated).

As a simple example, consider the following Curry program which defines a polymorphic data type for lists and operations to compute the concatenation of lists and the last element of a list:[5]

```
data List a = [] | a : List a    --[a] denotes "List a"

-- "++" is a right-associative infix operator
(++) :: [a] → [a] → [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)

last :: [a] → a
last xs | (ys ++ [z]) =:= xs
        = z                    where ys,z free
```

Logic programming is supported by admitting function calls with free variables (e.g., `(ys++[z])` in the rule defining `last`) and constraints in the condition of a defining rule. In contrast to Prolog, free variables need to be declared explicitly to make their scopes clear (e.g., "`where ys,z free`" in the example). A conditional rule is applicable if its condition is satisfiable. Thus, the rule defining `last` states in its condition that `z` is the last element of a given list `xs` if there exists a list `ys` such that the concatenation of `ys` and the one-element list `[z]` is equal to the given list `xs`.

The combination of functional and logic programming features has led to new design patterns [3] and better abstractions for application programming, e.g., as shown for programming with databases [7, 13], GUI programming [15], web programming [16, 17, 20], or string parsing [12]. In this paper, we show how to exploit these combined features to implement an eDSL for XML processing. To make this implementation as simple as possible, we exploit two more recent features described in the following: functional patterns [4] and set functions [5].

A fundamental requirement in functional as well as logic languages is that patterns in the left-hand sides of program rules contain only variables and data constructors. This

[5]Note that lists are a built-in data type with a more convenient syntax, e.g., one can write `[x,y,z]` instead of `x:y:z:[]` and `[a]` instead of the list type "`List a`".

excludes rules like

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

stating the associativity property of list concatenation. This restriction is the key to construct efficient evaluation strategies [18]. However, in a functional logic language one can relax this requirement and allow expressions containing defined operations in patterns as an abbreviation for a (potentially infinite) set of "standard" patterns. A pattern containing defined operations is called *functional pattern*. For instance,

```
last (xs ++ [e]) = e
```

is a rule with the functional pattern `(xs++[e])` stating that `last` is reducible to `e` provided that the argument can be matched against some value of `(xs++[e])` where `xs` and `e` are free variables. By instantiating `xs` to arbitrary lists, the value of `(xs++[e])` is any list having `e` as its last element. The semantics of functional patterns can be defined in terms of standard pattern by interpreting a functional pattern as the set of all constructor terms that is the result of evaluating (by narrowing [2]) the functional pattern. Thus, the above rule abbreviates the following (infinite) set of rules:

```
last [e] = e
last [x1,e] = e
last [x1,x2,e] = e
...
```

As we will see in this paper, functional patterns are a powerful feature to express arbitrary selections in term structures. In order to assign a reasonable semantics to functional patterns, one need syntactic conditions (like stratification) to ensure meaningful definitions (e.g., the above rule stating associativity of "`++`" is not allowed). Detailed requirements and a constructive implementation of functional patterns by a demand-driven unification procedure can be found in [4].

If nondeterministic programming techniques are applied, it is sometimes useful to collect all the values of some expression, e.g., to accumulate all results of a query. A "set-of-values" operation applied to an arbitrary argument might depend on the degree of evaluation of the argument, which is difficult to grasp in a non-strict language. Hence, *set functions* [5] have been proposed to encapsulate nondeterministic computations in non-strict functional logic languages. For each defined function f, $f_{\mathcal{S}}$ denotes the corresponding set function. In order to be independent of the evaluation order, $f_{\mathcal{S}}$ encapsulates only the nondeterminism caused by evaluating f except for the nondeterminism caused by evaluating the arguments to which f is applied. For instance, consider the operation `decOrInc` defined by

```
decOrInc x = (x-1) ? (x+1)
```

Then "$\texttt{decOrInc}_{\mathcal{S}}$ `3`" evaluates to (an abstract representation of) the set $\{2, 4\}$, i.e., the nondeterminism caused by `decOrInc` is encapsulated into a set. However, "$\texttt{decOrInc}_{\mathcal{S}}$ `(2?5)`" evaluates to two different sets $\{1, 3\}$ and $\{4, 6\}$ due to its nondeterministic argu-

ment, i.e., the nondeterminism caused by the argument is not encapsulated.

As already mentioned, this paper contains the complete source code of our implementation. Actually, it is a literate program [22], i.e., the paper's source text is directly executable. In a literate Curry program, all real program code starts with the special character "`>`". Curry code not starting with "`>`", e.g., the example code shown so far, is like a comment and not required to run the program. To give an example of executable code, we show the declaration of the module `XCuery` for XML processing in Curry developed in this paper:

```
> module XCuery where
>
> import XML
```

Thus, we import the system module `XML` which contains an XML parser and the definition of XML structures in Curry that are explained in the next section.

3 XML Documents

There are two basic methods to represent XML documents in a programming language: a type-based or a generic representation [25]. In a type-based representation, each tagged XML structure (like `contacts`, `entry`, `name` etc) is represented as a record structure of appropriate type according to the XML schema. The advantage of this approach is that schema-correct XML structures correspond to type-correct record structures. On the negative side, this representation depends on the given XML schema. Thus, it is hardly applicable if the schema is not completely known. Moreover, if the schema evolves, the data types representing the XML structure must be adapted.

Due to these reasons, we prefer a generic representation where any XML document is represented with one generic structure. Since any XML document is either a structure with a tag, attributes and embedded XML documents (also call *child nodes* of the document), or a text string, one can define the following datatype to represent XML documents:[6]

```
data XmlExp = XText String
            | XElem String [(String,String)] [XmlExp]
```

For instance, the second `entry` structure of the XML document shown in Figure 1 can be represented by the data term

```
XElem "entry" []
  [XElem "name"     [] [XText "Smith"],
   XElem "first"    [] [XText "William"],
   XElem "nickname" [] [XText "Bill"],
   XElem "phone"    [] [XText "+1-987-742-9388"]]
```

[6]For the sake of simplicity, we ignore other specific elements like comments.

Since it could be tedious to write XML documents with these basic data constructors, one can define some useful abstractions for XML documents:

```
xtxt   :: String  → XmlExp
xtxt s = XText s

xml :: String  → [XmlExp]  → XmlExp
xml t xs = XElem t [] xs
```

Thus, we can specify the previous document a bit more compact:

```
xml "entry" [xml "name"     [xtxt "Smith"],
             xml "first"    [xtxt "William"],
             xml "nickname" [xtxt "Bill"],
             xml "phone"    [xtxt "+1-987-742-9388"]]
```

These definitions together with operations to parse and pretty-print XML documents are contained in the system module `XML` of the PAKCS programming environment for Curry [19]. In principle, these definitions are sufficient for XML processing, i.e., to select and transform XML documents. For instance, one can extract the name and phone number of an `entry` structure consisting of a name, first name and phone number by the following operation:

```
getNamePhone
  (XElem "entry" []
     [XElem "name"  [] [XText name],
      _,
      XElem "phone" [] [XText phone]])  =  name++": "++phone
```

This can be also implemented in a similar way in other functional or logic programming languages. However, functional logic languages support a nicer way to write such matchings. Whereas typical functional or logic languages require the use of data constructors in patterns, functional patterns allow also to use already defined abstractions in patterns so that we can define the previous operation also in the following form:

```
getNamePhone
  (xml "entry" [xml "name"  [xtxt name],
                _,
                xml "phone" [xtxt phone]]) = name++": "++phone
```

This shows how functional patterns improves the readability of pattern matching by reusing already defined abstractions also in patterns and not only to construct new data in right-hand sides of program rules.

Apart from these advantages, XML processing operations as defined above have several disadvantages:

- The exact structure of the XML document must be known in advance. For instance, the operation `getNamePhone` matches only entries with three components, i.e., it fails on both entries shown in Figure 1.

- In large XML documents, many parts are often irrelevant if one wants to select only some specific information entities. However, one has to define an operation to match the complete document.
- If the structure of the XML document changes (e.g., due to the evolution of the web services providing these documents), one has to update all patterns in the matching operations which could be tedious and error prone for large documents.

As a solution to these problems, we propose in the next section appropriate abstractions that can be used in patterns of operations for XML processing.

4 Abstractions for XML Processing

In order to define reasonable abstractions for XML processing, we start with a wish list. Since we have seen that exact matchings are not desirable to process semistructured data, we want to develop a language supporting the following features for pattern matching:

- *Partial patterns:* allow patterns where only some child nodes are known.
- *Unordered patterns:* allow patterns where child nodes can appear in any order.
- *Patterns at arbitrary depth:* allow patterns that are matched at an arbitrary position in an XML document.
- *Negation of patterns:* allow patterns defined by the absence of tags or provide default values for tags that are not present in the given XML document.
- *Transformation:* generate new structures from matched patterns.
- *Collect matchings:* accumulate results in a newly generated structure.

In the following, we show how these features can be supported by the use of carefully defined abstractions as functional patterns and other features of functional logic programming.

4.1 Partial Patterns

As we have seen in the example operation `getNamePhone` above, one would like to select some child nodes in a document independent of the availability of further components. Thus, instead of enumerating the list of *all* child nodes as in the definition above, it would be preferable to enumerate only the relevant child nodes. We support this by putting the operator "`with`" in front of the list of child nodes:

```
getNamePhone
  (xml "entry" (with [xml "name"  [xtxt name],
                      xml "phone" [xtxt phone]]))  =  name++": "++phone
```

The intended meaning of "`with`" is that the given child nodes must be present but in between any number of other elements can also occur.

We can directly implement this operator as follows:[7]

```
> with :: Data a => [a]  → [a]
> with []     = _
> with (x:xs) = _ ++ x : with xs
```

Thus, an expression like "`with [1,2]`" reduces to any list of the form

x_1`:`...`:`x_m`:1:`y_1`:`...`:`y_n`:2:`zs

where the variables x_i, y_j, zs are fresh logic variables. Due to the semantics of functional patterns, the definition of `getNamePhone` above matches any `entry` structure containing a `name` and a `phone` element as children. Hence, the use of the operation `with` in patterns avoids the exact enumeration of all children and makes the program robust against the addition of further information elements in a structure.

A disadvantage of a definition like `getNamePhone` above is the fact that it matches only XML structures with an empty attribute list due to the definition of the operation `xml`. In order to support more flexible matchings that are independent of the given attributes (which are ignored if present), we define the operation

```
> xml' :: String  → [XmlExp]  → XmlExp
> xml' t xs = XElem t _ xs
```

For instance, the operation `getName` defined by

```
getName (xml' "entry" (with [xml' "name" [xtxt n]]))  =  n
```

returns the name of an `entry` structure independent of the fact whether the given document contains attributes in the `entry` or `name` structures.

4.2 Unordered Patterns

If the structure of data evolves over time, it might happen that the order of elements changes over time. Moreover, even in some given XML schema, the order of relevant elements can vary. In order to make the matching independent of a particular order, we can specify that the required child nodes can appear in any order by putting the operator "`anyorder`" in front of the list of child nodes:

```
getNamePhone
  (xml "entry"
     (with (anyorder [xml "phone" [xtxt phone],
                      xml "name"  [xtxt name]]))  =  name++": "++phone
```

[7]The symbol "`_`" denotes an anonymous variable, i.e., each occurrence of "`_`" in the right-hand side of a rule denotes a fresh logic variable.

Obviously, the operation `anyorder` should compute any permutation of its argument list. In a functional logic language, it can be easily defined as a nondeterministic operation by inserting the first element of a list at an arbitrary position in the permutation of the remaining elements:

```
> anyorder :: [a]  → [a]
> anyorder []     = []
> anyorder (x:xs) = insert (anyorder xs)
>  where insert []     = [x]
>        insert (y:ys) = x:y:ys  ?  y : insert ys
```

Thus, the previous definition of `getNamePhone` matches both `entry` structures shown in Figure 1.

4.3 Patterns at Arbitrary Depths

If one wants to select some information in deeply nested documents, it would be tedious to define the exact matching from the root to the required elements. Instead, it is preferable to allow matchings at an arbitrary depth in a document. Such matchings are also supported in other languages like XPath since they ease the implementation of queries in complex structures and support flexibility of the implementation w.r.t. to future structural changes of the given documents. We support this feature by an operation "`deepXml`": if `deepXml` is used instead of `xml` in a pattern, this structure can occur at an arbitrary position in the given document. For instance, if we define

```
getNamePhone
  (deepXml "entry"
     (with [xml "name"  [xtxt name],
            xml "phone" [xtxt phone]]))  =  name++": "++phone
```

and apply `getNamePhone` to the complete document shown in Figure 1, two results are (nondeterministically) computed (methods to collect all those results are discussed later).

The implementation of `deepXml` is similar to `with` by specifying that `deepXml` reduces to a structure where the node is at the root or at some nested child node:

```
> deepXml :: String  → [XmlExp]  → XmlExp
> deepXml tag elems = xml tag elems
> deepXml tag elems = xml' _ (_ ++ [deepXml tag elems] ++ _)
```

Thus, an expression like "`deepXml` t cs" reduces to "`xml` t cs" or to a structure containing this element at some inner position.

4.4 Negation of Patterns

As mentioned above, in semistructured data some information might not be present in a given structure, like the email address in the second entry of Figure 1. Instead of failing on missing information pieces, one wants to have a constructive behavior in application

programs. For instance, one could select all entries with a missing email address or one puts a default nickname in the output if the nickname is missing.

In order to implement such behaviors, one could try to negate matchings. Since negation is a non-trivial subject in functional logic programming, we propose a much simpler but practically reasonable solution. We provide an operation "`withOthers`" which is similar to "`with`" but has a second argument that contains the child nodes that are present but not part of the first argument. Thus, one can use this operation to denote the "unmatched" part of a document in order to put arbitrary conditions on it. For instance, if we want to get the name and phone number of an entry that has no email address, we can specify this as follows:

```
getNamePhoneWithoutEmail
  (deepXml "entry"
     (withOthers [xml "name" [xtxt name], xml "phone" [xtxt phone]]
                 others))
  | "email" `noTagOf` others   =  name++": "++phone
```

The useful predicate `noTagOf` returns true if the given tag is not a tag of all argument documents (the operation `tagOf` returns the tag of an XML document):

```
> noTagOf :: String → [XmlExp] → Bool
> noTagOf tag = all ((/=tag) . tagOf)
```

Hence, the application of `getNamePhoneWithoutEmail` to the document in Figure 1 returns a single value.

The implementation of `withOthers` is slightly different from `with` since we have to accumulate the remaining elements that are not part of the first arguments in the second argument:

```
> withOthers :: Data a => [a] → [a] → [a]
> withOthers ys zs = withAcc [] ys zs
>  where -- Accumulate remaining elements:
>    withAcc prevs [] others | others=:=prevs++suffix = suffix
>                                                 where suffix free
>    withAcc prevs (x:xs) others =
>       prefix ++ x : withAcc (prevs++prefix) xs others
>                                                 where prefix free
```

Thus, an expression like "`withOthers [1,2]` *os*" reduces to any list of the form

$$x_1 : \ldots : x_m : 1 : y_1 : \ldots : y_n : 2 : zs$$

where $os = x_1 : \ldots : x_m : y_1 : \ldots : y_n : zs$. If we use this expression as a pattern, the semantics of functional patterns ensures that this pattern matches any list containing the elements `1` and `2` where the variable *os* is bound to the list of the remaining elements.

4.5 Transformation of Documents

Apart from the inclusion of data selected in XML documents in the application program, one also wants to implement transformations on documents, e.g., transform an XML document into a corresponding HTML document. Such transformation tasks are almost trivial to implement in declarative languages supporting pattern matching by using a scheme like

transform pattern = *newdoc*

and applying the *transform* operation to the given document. For instance, we can transform an `entry` document into another XML structure containing the phone number and full name of the person by

```
transPhone (deepXml "entry"
                    (with [xml "name"  [xtxt n],
                           xml "first" [xtxt f],
                           xml "phone" phone])) =
  xml "phonename" [xml "phone" phone,
                   xml "fullname" [xtxt (f++' ':n)]]
```

If we apply `transPhone` to the document of Figure 1, we nondeterministically obtain two new XML documents corresponding to the two entries contained in this document.

4.6 Collect Matchings

If we want to collect all matchings in a given document in a single new document, we have to encapsulate the nondeterministic computations performed on the input document. For this purpose, we can exploit set functions described above. Since set functions return an unordered set of values, we have to transform this value set into an ordered list structure that can be printed or embedded in another document. This can be done by the predefined operation `sortValues`. Thus, if c denotes the XML document shown in Figure 1, we can use our previous transformation operation to create a complete table of all pairs of phone numbers and full names by evaluating the expression[8]

```
xml "table" (sortValues (transPhone_S c))
```

which yields the representation of the XML document

```
<table>
  <phonename>
    <phone>+1-987-742-9388</phone>
    <fullname>William Smith</fullname>
  </phonename>
  <phonename>
```

[8] In the implementation of set functions in the PAKCS environment [19], one has to write `(set`n f`)` for the set function corresponding to the n-ary operation f.

```
    <phone>+49-431-8807271</phone>
    <fullname>Michael Hanus</fullname>
  </phonename>
</table>
```

Similarly, one can also transform XML documents into HTML documents by exploiting the HTML library of Curry [16]. Furthermore, one can also nest set functions to accumulate intermediate information. As an example, we want to compute a list of all persons together with the number of their email addresses. For this purpose, we define a matching rule for an `entry` document that returns the number of email addresses in this document by a set function $\texttt{emailOf}_{\mathcal{S}}$:

```
getEmails (deepXml "entry" (withOthers [xml "name" [xtxt name]] os))
   = (name, length (sortValues (emailOf_S os)))
 where
   emailOf (with [xml "email" email]) = email
```

In order to compute a complete list of all entries matched in a document c, we apply the set function $\texttt{getEmails}_{\mathcal{S}}$ to collect all results in a list structure:

```
sortValues (getEmails_S c)
```

For our example document, this evaluates to `[("Hanus",2),("Smith",0)]`.

4.7 Attribute Matchings

So far we have only defined matchings of XML structures where the attributes are not taken into account. If we want to match on attribute values, we can also use the generic matching operators like `with`, `anyorder`, or `withOthers` for this purpose. For instance, if the `first` structure of an XML document contains an attribute `sex` to indicate the gender, we can select all male first names by the operation

```
getMaleFirstNames
  (deepXml "entry"
     (with [XElem "first" (with [("sex","male")]) [xtxt f]]))  =  f
```

Here, we use the pattern `(with [("sex","male")])` for the attribute list in order to match on any occurrence of the attribute `sex` with value `male`.

5 Properties of the Implementation

5.1 Correctness

As shown in the previous section, the matching operations are quite powerful and can be directly implemented in a functional logic language. This has the advantage that the correctness of the implemented matching operations is a direct consequence of the

correctness results for functional logic programming. We demonstrate this reasoning by a simple example.

Consider the following operation to select a name in an `entry` document:

```
getName (xml "entry" (with [xml "name" [xtxt n]]))  =  n
```

In order to show the correctness of this operation, we have to show the following property ($\rightarrow^*$ denotes the evaluation relation):

Proposition: If $xdoc$ = `xml "entry" [...,xml "name" [xtxt` n`],...]`, then `getName` $x \rightarrow^* n$.

Since the formal definition of the semantics of functional logic programming is outside the scope of this paper, we provide only a proof sketch. The definition of `with` implies that the expression `(with [xml "name" [xtxt n]])` evaluates to

```
x1:...:xm:xml "name" [xtxt n]:ys
```

for any $m \geq 0$. Hence, by the semantics of functional patterns,

```
getName (xml "entry" (x1:...:xm:xml "name" [xtxt n]:ys)) = n
```

is a rule defining `getName` for any $m \geq 0$ (more precisely, we must also evaluate the operations `xml` and `xtxt`, but we omit this detail here). Thus,

```
getName xdoc  →*  n
```

is a valid rewrite step.

5.2 Termination

A functional pattern like `(with [xml "name" [xtxt n]])` denotes an infinite set of constructor patterns, i.e., it denotes all constructor patterns of the form

```
x1:...:xm:xml "name" [xtxt n]:ys
```

for any $m \geq 0$. Thus, it is not obvious that a search for all possible matchings, which is usually performed by set functions in order to collect all results, will ever terminate. In principle, general termination criteria for functional logic programs with functional patterns are not yet known. However, it should be noted that the set of constructor patterns represented by a functional patterns is not blindly enumerated. Actually, the corresponding constructor patterns are generated in a demand-driven manner, i.e., new constructor patterns are computed only if they are demanded to match the actual argument. Thus, the structure of the actual argument determines how far the operations in the functional patterns are evaluated (see [4] for more details about the demand-driven unification procedure). Hence, the finite size of the actual arguments (i.e., the XML documents) implies the finiteness of the set of constructor patterns that are computed to match the actual

argument.[9] Therefore, the search space is finite in all our examples.

5.3 Performance

Our implementation heavily exploits nondeterministic computations, e.g., when matching partially specified or deep structures, a nondeterministic guessing of appropriate patterns takes place. This raises the question whether this approach can be used in practice. Since our main emphasis is on expressiveness (i.e., we want to be able to express selections and transformations in a declarative rather than navigational manner), we do not intend to compete in performance with specialized languages for XML processing. For our purpose it is sufficient, to be practically useful, that there is a reasonable relation between the time to read an XML document and the time to process it, because each XML document must be read from a file or network connection before processing it. Our first practical experiments (using the PAKCS environment [19] which compiles Curry programs into Prolog programs that are executed by SICStus-Prolog) indicate that the processing time to select or transform documents is almost equal or smaller than the parsing time. Since the XML parser is implemented by deterministic operations without any nondeterministic steps, this shows that the nondeterminism used to implement our matching operators does not hinder the practical application of our implementation.

6 Related Work

Since the processing of semistructured data is a relevant issue in current application systems, there are many proposals for specialized languages or embedding languages in multi-purpose programming languages. We discuss some related approaches in this section.

We have already mentioned in the beginning the languages XPath, XQuery, and XSLT for XML processing supported by the W3C. These languages provide a different XML-oriented syntax and use a navigational approach to select information rather than the pattern-oriented approach we proposed. Since these are separate languages, it is more difficult to use them in application programs written in a general purpose language where one wants to process data available in the web.

The same is true for the language Xcerpt [8, 10]. It is also a separate XML processing language without a close connection to a multi-purpose programming language. In contrast to XPath, Xcerpt proposes the use of powerful matching constructs to select information in semistructured documents. Xcerpt supports similar features as our embedded language but provide a more compact syntax due to its independence of a concrete base language. In contrast to our approach, Xcerpt requires a dedicated implementation

[9] Obviously, this need not be the case for general functional patterns. For instance, if the pattern contains a non-terminating operation like "`loop = loop`", the functional pattern unification will not terminate. However, our operations have the property that a data constructor is produced around each recursive call. Thus, an infinite recursion results in constructor terms of infinite size.

based on a specialized unification procedure [9]. The disadvantages of such separate developments become obvious if one tries to access the implementation of Xcerpt (which failed at the time of this writing due to inaccessible web pages and incompatible compiler versions).

HaXML [25] is a language for XML processing embedded in the functional language Haskell. It provides a rich set of combinators based on *content filters*, i.e., functions that map XML data into collections of XML data. This allows an elegant description of many XML transformations, whereas our rule-based approach is not limited to such transformations since we have no restrictions on the type of data constructed from successful matchings.

Caballero et al. [11] proposed the embedding of XPath into the functional logic language Toy that has many similarities to Curry. Similarly to our approach, they also exploit nondeterministic evaluation for path selection. Due to the use of a functional logic language allowing inverse computations, they also support the generation of test cases for path expressions, i.e., the generation of documents to which a path expression can be applied. Nevertheless, their approach is limited to the navigational processing of XPath rather than a rule-based approach as in our case. The same holds for FnQuery [24], a domain-specific language embedded in Prolog for the querying and transformation of XML data.

7 Conclusions

We have presented a rule-based language for processing semistructured data that is implemented and embedded in the functional logic language Curry. The language supports a declarative description to query and transform such data. It is based on providing operations to describe partial matchings in the data and exploits functional patterns and set functions for the programming tasks. Due to its embedding into a general-purpose programming language, it can be used to further process the selected data in application systems or one can combine semistructured data from different sources. Moreover, it is easy to extend our language with new features without adapting a complex implementation.

The simplicity of our implementation together with the expressiveness of our language demonstrate the general advantages of high-level declarative programming languages. In order to check the usability of our language, we applied it to extract information provided by our university information system[10] in XML format into a curricula and module information system[11] that is implemented in Curry. In this application it was quite useful to specify only partial patterns so that most of the huge amount of information contained in the XML document could be ignored.

For future work, we intend to apply our language to more examples in order to enrich the set of useful pattern combinators. Moreover, it would be interesting to generate more

[10] `http://univis.uni-kiel.de/`

[11] `http://www-ps.informatik.uni-kiel.de/~mh/studiengaenge/`

efficient implementations by specializing functional patterns (e.g., by partial evaluation w.r.t. the given definitions, or by exploiting the XML schema if it is precisely known in advance).

References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann, 2000.

[2] S. Antoy, R. Echahed, and M. Hanus. A Needed Narrowing Strategy. *Journal of the ACM*, Vol. 47, No. 4, pp. 776–822, 2000.

[3] S. Antoy and M. Hanus. Functional Logic Design Patterns. In *Proc. of the 6th International Symposium on Functional and Logic Programming (FLOPS 2002)*, pp. 67–87. Springer LNCS 2441, 2002.

[4] S. Antoy and M. Hanus. Declarative Programming with Function Patterns. In *Proceedings of the International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, pp. 6–22. Springer LNCS 3901, 2005.

[5] S. Antoy and M. Hanus. Set Functions for Functional Logic Programming. In *Proceedings of the 11th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'09)*, pp. 73–82. ACM Press, 2009.

[6] S. Antoy and M. Hanus. Functional Logic Programming. *Communications of the ACM*, Vol. 53, No. 4, pp. 74–85, 2010.

[7] B. Braßel, M. Hanus, and M. Müller. High-Level Database Programming in Curry. In *Proc. of the Tenth International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, pp. 316–332. Springer LNCS 4902, 2008.

[8] F. Bry and S. Schaffert. A gentle introduction to Xcerpt, a rule-based query and transformation language for XML. In *Proceedings of the International Workshop on Rule Markup Languages for Business Rules on the Semantic Web (RuleML'02)*, 2002.

[9] F. Bry and S. Schaffert. Towards a Declarative Query and Transformation Language for XML and Semistructured Data: Simulation Unification. In *Proceedings of the International Conference on Logic Programming (ICLP'02)*, pp. 255–270. Springer LNCS 2401, 2002.

[10] F. Bry, S. Schaffert, and A. Schroeder. A Contribution to the Semantics of Xcerpt, a Web Query and Transformation Language. In *Applications of Declarative Programming and Knowledge Management (INAP/WLP 2004)*, pp. 258–268. Springer LNCS 3392, 2005.

[11] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Integrating XPath with the Functional-Logic Language Toy. Technical Report SIC-05-10, Univ. Complutense de Madrid, 2010.

[12] R. Caballero and F.J. López-Fraguas. A Functional-Logic Perspective of Parsing. In *Proc. 4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99)*, pp. 85–99. Springer LNCS 1722, 1999.

[13] S. Fischer. A Functional Logic Database Library. In *Proc. of the ACM SIGPLAN 2005 Workshop on Curry and Functional Logic Programming (WCFLP 2005)*, pp. 54–59. ACM Press, 2005.

[14] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, Vol. 40, pp. 47–87, 1999.

[15] M. Hanus. A Functional Logic Programming Approach to Graphical User Interfaces. In *International Workshop on Practical Aspects of Declarative Languages (PADL'00)*, pp. 47–62. Springer LNCS 1753, 2000.

[16] M. Hanus. High-Level Server Side Web Scripting in Curry. In *Proc. of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, pp. 76–92. Springer LNCS 1990, 2001.

[17] M. Hanus. Type-Oriented Construction of Web User Interfaces. In *Proceedings of the 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP'06)*, pp. 27–38. ACM Press, 2006.

[18] M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pp. 45–75. Springer LNCS 4670, 2007.

[19] M. Hanus, S. Antoy, B. Braßel, M. Engelke, K. Höppner, J. Koj, P. Niederau, R. Sadre, and F. Steiner. PAKCS: The Portland Aachen Kiel Curry System. Available at `http://www.informatik.uni-kiel.de/~pakcs/`, 2010.

[20] M. Hanus and S. Koschnicke. An ER-based Framework for Declarative Web Programming. In *Proc. of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL 2010)*, pp. 201–216. Springer LNCS 5937, 2010.

[21] M. Hanus (ed.). Curry: An Integrated Functional Logic Language (Vers. 0.8.2). Available at `http://www.curry-language.org`, 2006.

[22] D.E. Knuth. Literate Programming. *The Computer Journal*, Vol. 27, No. 2, pp. 97–111, 1984.

[23] S. Peyton Jones, editor. *Haskell 98 Language and Libraries—The Revised Report.* Cambridge University Press, 2003.

[24] D. Seipel, J. Baumeister, and M. Hopfner. Declaratively Querying and Visualizing Knowledge Bases in XML. In *Applications of Declarative Programming and Knowledge Management (INAP/WLP 2004)*, pp. 16–31. Springer LNCS 3392, 2005.

[25] M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, pp. 148–159. ACM Press, 1999.

A Further Abstractions

This appendix contains some further abstractions that are not relevant for this paper but useful for XML processing.

The operation `deepXml` defined in Section 4.3 can be used to match XML structures without attributes at an arbitrary position. In order to match structures with a possibly non-empty list of attributes, we provide (analogously to the definition of `xml'` in Section 4.1) the following operation:

```
> deepXml' :: String → [XmlExp] → XmlExp
> deepXml' tag elems = xml' tag elems
> deepXml' tag elems = xml' _ (_ ++ [deepXml' tag elems] ++ _)
```

If we are also interested to match the attributes of a deep XML structure, we can use the following operation:

```
> deepXElem :: String → [(String,String)] → [XmlExp]
>             → XmlExp
> deepXElem tag attrs elems = XElem tag attrs elems
> deepXElem tag attrs elems = xml' _ (_ ++ [deepXElem tag attrs elems] ++ _)
```

For instance, we can use this abstraction to provide a simpler and more general definition of the operation `getMaleFirstNames` shown in Section 4.7:

```
getMaleFirstNames
  (deepXElem "first" (with [("sex","male")]) [xtxt f]) = f
```

When dealing with semistructured data, it could be the case that one wants to use a default value if some element is not present. For this purpose, we define an operation `optXml` such that "`optXml` t xs ys" evaluates to "`xml` t xs" if there is no element with tag t in ys, otherwise the first element of ys with tag t is returned:

```
> optXml :: String → [XmlExp] → [XmlExp] → XmlExp
> optXml tag elems [] = xml tag elems
> optXml tag elems (x:xs) =
>   if tag == tagOf x then x else optXml tag elems xs
```

One can apply this operation in combination with the matching operator `withOthers` to check optional occurrences in the remaining elements. As an example, we transform the entries of Figure 1 into `nickphone` structures consisting of a nickname and a phone number. The definition is similar to `transPhone` (see Section 4.5) with the difference that the nickname is assumed to be optional: if it is not present in the given `entry` structure, it is generated by concatenating the given names:

```
transNickPhone
  (deepXml "entry"
           (withOthers [xml "name"  [xtxt n],
                        xml "first" [xtxt f],
                        xml "phone" phone]
```

```
                        others)) =
  xml "nickphone" [optXml "nickname" [xtxt (f++n)] others,
                   xml "phone" phone]
```

Thus, if c denotes the XML document of Figure 1, the evaluation of

```
xml "table" (sortValues (transNickPhone_S c))
```

yields the representation of the XML document

```
<table>
  <nickphone>
    <nickname>Bill</nickname>
    <phone>+1-987-742-9388</phone>
  </nickphone>
  <nickphone>
    <nickname>MichaelHanus</nickname>
    <phone>+49-431-8807271</phone>
  </nickphone>
</table>
```