biXid: A Bidirectional Transformation Language for XML

Shinya KAWANAKA
IBM Tokyo Research Laboratory
shinyak@arbre.is.s.u-tokyo.ac.jp

Haruo HOSOYA
The University of Tokyo
hahosoya@is.s.u-tokyo.ac.jp

Abstract

Often, independent organizations define and advocate different XML formats for a similar purpose and, as a result, application programs need to mutually convert between such formats. Existing XML transformation languages, such as XSLT and XDuce, are unsatisfactory for this purpose since we would have to write, e.g., two programs for the forward and the backward transformations in case of two formats, incur high developing and maintenance costs.

This paper proposes the *bidirectional* XML transformation language biXid, allowing us to write only one program for both directions of conversion. Our language adopts a common paradigm *programming-by-relation*, where a program defines a relation over documents and transforms a document to another in a way satisfying this relation. Our contributions here are specific language features for facilitating realistic conversions whose target formats are loosely in parallel but have many discrepancies in details. Concretely, we (1) adopt XDuce-style regular expression patterns for describing and analyzing XML structures, (2) fully permit *ambiguity* for treating formats that do not have equivalent expressivenesses, and (3) allow *non-linear* pattern variables for expressing non-trivial transformations that cannot be written only with linear patterns, such as conversion between unordered and ordered data.

We further develop an efficient evaluation algorithm for biXid, consisting of the "parsing" phase that transforms the input document to an intermediate "parse tree" structure and the "unparsing" phase that transforms it to an output document. Both phases use a variant of finite tree automata for performing a one-pass scan on the input or the parse tree by using a standard technique that "maintains the set of all transitable states." However, the construction of the "unparsing" phase is challenging since ambiguity causes different ways of consuming the parse tree and thus results in multiple possible outputs that may have different structures.

We have implemented a prototype system of the biXid language and confirmed that it has enough expressiveness and a linear-time performance, through experiments with several realistic bidirectional transformations including one between vCard-XML and ContactXML.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structure; Constraints

General Terms Algorithms, Design, Language, Theory

ICFP'06 September 16–21, 2006, Portland, Oregon, USA.

Keywords XML, tree automata

1. Introduction

1.1 Bidirectional transformation

The world has seen XML's exceptionally rapid and wide acceptance as a de facto standard for structured data. One of the reasons is its openness to any community for freely defining their own new format by means of schemas. However, one of its consequences is that different communities propose formats that are structurally different but represent similar kinds of information. As a typical example, both vCard-XML [18] and ContactXML [7] are completely different formats representing address books, the former being an XML-ized version of a non-XML format defined as an international standard and the latter being a format that has newly been defined by an independent Japanese organization. In such situation, relevant application programs often need to support both formats, where the most common way to implement it is to mutually convert between them. For this purpose, existing XML transformation languages, such as XDuce [16] and XSLT [4], are unsatisfactory since they consider only conversions in one way and therefore, in order to transform back and forth between two formats, we must write two programs; maintaining their consistency often incurs a high development cost and easily becomes error prone.

In the present work, we address this problem by designing a language called biXid dedicated to the bidirectional transformation between XML documents, where we only need to write one program for converting data in both the forward and the backward directions. The basic design paradigm is *programming by relation*, in which the programmer defines a relation over documents and the system transforms a given document to another in such a way that these two documents satisfy the relation. This paradigm itself is quite common, from classical logic programming [22] and relational query languages [8] to more recent research languages for XML-text mutual conversion [3] and view updating [10, 19]. Our aim is, however, to design a language that specifically focuses on the transformation between two different XML formats, together with an efficient evaluation algorithm devoted to this language. In particular, we seriously consider conversion on formats that are defined by independent communities, which typically have structures loosely in parallel but various discrepancies in details. Since conversion between such formats is often quite complicated, effective linguistic supports are highly desirable.

The novelties in the biXid language design are threefold: (1) adoption of XDuce-style regular expression patterns, (2) full allowance of ambiguity, and (3) support for non-linear bound variables. Regular expression patterns [15] have first been proposed for the unidirectional transformation language XDuce [16], which the second author has been engaged in, for succinct description of the structure of an XML document and the extraction of its subparts. In biXid, we deploy this feature for both describing the structures of two documents and *relating* their subparts. For example, we can

write the following relation in biXid for converting between fragments of two kinds of address books.

```
relation item =
  person[name[var n as String],
         address[var a]*]
<->
  card[person-name[var n as String],
       location[var l]*]
where
  address_content(a, l)
```

This relates a `person` element and a `card` element, where the pattern on the left-hand-side (LHS) of the `<->` sign requires the `person` to contain a `name` element followed by a sequence of `address` elements, and similarly the right-hand-side (RHS) describes the structure of the `card`. Then, the bound variable `n`, which appears in both patterns, specifies that its corresponding subparts, the contents of `name` and `person-name`, are the same. Also, the variables `a` and `l` on both sides together with the `where` clause specify that the contents of `address` and `location` are constrained by another relation `address_content` (whose definition is omitted here).

Notice that, in the example, since there can be several `address` elements in the document on the left, the variable `a` can be matched several times; such use of variable is called *non-linear* and is one of our important design choices. When corresponding non-linear variables appear on both sides, each pair of n-th matching values corresponds to each other. In the above example, the content of the n-th `address` corresponds to that of the n-th `location`. Since regular expressions frequently contain Kleene stars, this feature naturally arises and is quite useful for concisely describing relations without resorting to writing explicit recursions. Further, it is one of our findings that non-linearity is occasionally indispensable in real applications. For example, we have observed, in our application of conversion between vCard-XML and ContactXML, that data pieces that are scattered on one side need to be put together as a chunk on the other side; this involves "reordering" of data, which seems not expressible only with linear variables (Appendix B).

Another design choice that we have made is that we impose no restriction on writing ambiguous relations, that is, a relation where a single document on one side may correspond to multiple documents on the other side. This is because, in real applications, one of the formats happens to have a richer mark-up structure than the other. For example, when one format has two representations for a telephone number while the other has only one, the best we could do is to write, e.g., the following relation.

```
relation phone_number =
  home_phone[var p as String] |
  work_phone[var p as String]
<->
  tel[var p as String]
```

In the forward direction, two documents can transform to a single document, while, in the reverse direction, two outputs are possible from one input. In addition to this "multiple representation" case, we have identified two more use cases where allowing ambiguity is critical, namely, "freedom of ordering" and "unused data." We will explain these in detail in Section 2.3.

We further develop an efficient evaluation algorithm for transformations written in our language. Our strategy uses a form of non-deterministic tree automata called *parsing automata* equipped with facilities for supporting variable binding and `where`-clause constraints. In the forward conversion (and symmetrically in the backward), we build two parsing automata corresponding to the left-hand-side and the right-hand-side of the relation defined by the program. We then perform the conversion in two phases, namely, the "parsing" phase, which generates an intermediate "parse tree" structure from an input document, and then the "unparsing" phase, which produces an output document from the parse tree. Both phases work in one pass in the sense that the former traverses the input document only once and the latter scans the parse tree also only once. For achieving this non-backtracking behavior, we use the classical method that collects all possible states where we can transit at each step. In the algorithm obtained in this way, while the "parsing" phase is only a slight variant of a well-known one-pass membership algorithm (cf. [20]), the "unparsing" phase is novel and quite involved, where the main complication lies in that ambiguity raises different ways of consuming the parse tree with multiple possible outputs that can in general have completely different structures.

We have already implemented a prototype system of the biXid language, and experimented with several realistic applications, including conversion between vCard-XML and ContactXML. From these experiments, we have successfully confirmed that our language features—in particular, non-linear variables and ambiguity—are quite useful in practice and that our algorithm has a linear-time performance and enough efficiency even for relatively large inputs.

1.2 Related Work

XSugar [3] is a language proposed by Brabrand, Møller, and Schwartzbach for the mutual conversion between XML and text data. Their language is also based on programming-by-relation and the program structure has a quite similar flavor to biXid. However, besides the obvious difference in XML-text or XML-XML conversion, the details of the designs are different since the intended application domains are different. XSugar is mainly targeted to the situation where both an XML format and its corresponding non-XML version are provided, where the former is for mechanical processing and the latter is for human reading and writing. In such case, the two formats are typically defined by the same organization and, naturally, have a tight correspondence in their structures. With such application requirement, a simpler language design is sufficient. For example, their *templates*, which correspond to biXid's patterns, do not allow non-linear variables, but they would probably not need this feature since the tight structural correspondence requires no data reordering, unlike our case. Furthermore, XSugar forbids any ambiguity in order to guarantee that the result of one round-trip transformation (either a forward followed by a backward or the other way) is the same as the original document. This requirement reasonably arises for the document maintenance in their case, while ambiguity is inherent in our applications, as argued above. Note that allowing ambiguity is not just a matter of omitting the check, but gives rise to a technical challenge in the evaluation algorithm, as already mentioned.

Another direction of related work is language design for view updating. The view-updating problem, which has originally been pursued in the database community, is to solve how we reflect (update) the change to the source data when we take its summary (view) and then modify it. Several groups of researchers have separately proposed languages for facilitating view updating by simultaneous description of both the source-to-view and the view-to-source transformations. We are aware of two such languages, Focal [10] developed by Foster, Greenwald, Moore, Pierce, and Schmitt, and Inv [19] by Mu, Hu, and Takeichi. Both are based on functional combinators—Focal provides combinators called "lenses" that are designed to be bidirectional and Inv supports transformation combinators that can automatically be inverted by a special operation. It is rather delicate to compare their languages and ours since the design principles are quite different, though a few remarks can be

made. First, both Focal and Inv assume classical data structures as the target of transformation (Focal uses records and Inv uses standard tuples, lists, disjoint unions, and so on). This means that, in order to process actual XML data, one needs to resort to either data-binding [2] or binary-tree representation [17], which often imposes a feeling of "impedance mismatch" on the user. In contrast, biXid's programming style with regular expression patterns gives a more direct flavor and thus, we believe, has a stronger appeal to novice programmers. However, in a more technical perspective, it seems that two approaches are fundamentally incomparable. On one hand, it is usually impossible for combinators to express a case analysis on deep structures because of its "taggedness," while this is naturally expressible with regular expression patterns since they allow full nondeterminism. On the other hand, the combinator-based approach can easily support composition of multiple bidirectional transformations, while this is not straightforward at all in the pattern-matching-based approach.

Prolog is one of the most classical languages based on programming-by-relation. Recently, Coelho and Florido [6] have proposed an extension of Prolog with regular expression patterns for adapting it to XML processing. In terms of defining relations, their language essentially includes ours, modulo non-linear patterns. However, our language has a specific focus on bidirectional transformation between XML documents and therefore does not need usual facilities provided by logic programming languages, such as unification of partially constructed XML fragments. As such, we can concentrate on developing an efficient evaluation algorithm that takes a single document for one side and emits a single document for the other side.

Regular expression patterns adopted in the biXid language have originally been proposed for the unidirectional XML transformation language XDuce [16]. In the present work, we extend this with non-linear variables for easily capturing repeated substructures. Although XDuce's patterns themselves do not allow non-linearity, it has a separate extension called *filters* for composing "regular expressions over pattern clauses" for a similar purpose. In the early stage of biXid, we have considered similar "regular expressions over relation definitions" but dismissed it since it has a crucial deficiency, which is inherited from the original filter proposal, that it is incapable of the above-mentioned data reordering. CDuce [1] extends XDuce's patterns with non-linear variables with a different semantics. In their case, when a non-linear variable matches multiple values, it is bound to the *concatenation* of these values. While this is often convenient for data extraction, it loses the information that the matched values used to be separate substructures in the original document. Since we intend to relate each individual substructure with another, this semantics is not suitable for our purpose.

1.3 Outline

The rest of this paper is organized as follows. In the next section, we illustrate our language design by using a series of examples and then, in Section 3, we formalize it. Section 4 presents our one-pass evaluation algorithm. Then, we will give results of some preliminary performance experiments in Section 5. Appendix A gives a translation procedure from our surface language programs to automata representations. We show some lessons learned from programming a conversion between vCard-XML and ContactXML in Appendix B. Proofs of theorems are omitted from this abstract and will be provided in the full version.

2. Examples

This section aims at illustrating biXid's language design. We first give a brief overview of our relation-based programming style and then explain design details, in particular, our supports for non-linear variables and ambiguity.

2.1 Programming by relation

We consider a simple biXid application for the conversion between two different bookmark formats, the Netscape and the XBEL [21] formats (both with slight simplifications). Let us first see instances of these formats in Figures 1 and 2. In the Netscape document, some heading information like the title in `h1` is followed by the main content in `dl`, where bookmark items (`dt`) and folders (`dd`) are listed. Each folder has a title in `h3` and then a `dl` element recursively containing an inner content consisting of bookmark items and folders. The XBEL document has a similar structure except that, besides the difference in tag names, it has no element corresponding to `head` and each content list begins right after a `title` element (i.e., XBEL has no tag corresponding to `dl`).

```
<html><head>My Bookmarks</head>
  <body>
    <h1>my bookmarks</h1>
    <dl><dt><a href="foo.com">Foo's</a></dt>
        <dd><h3>my folder</h3>
            <dl> inner folder... </dl><dd>
        <dt><a href="bar.edu">Bar's</a></dt>
    </dl>
  </body>
</html>
```

Figure 1. A Netscape bookmark

```
<xbel>
  <title>my bookmarks</title>
  <bookmark href="foo.com">
     <title>Foo's</title></bookmark>
  <folder><title>my folder</title>
          inner folder... </folder>
  <bookmark href="bar.edu">
     <title>Bar's</title></bookmark>
</xbel>
```

Figure 2. A XBEL bookmark

Let us write a program for converting between these formats step by step. We first define the `top` relation to describe the correspondence of the whole documents in both formats.

```
relation top =
  html[head[String],
       body[h1[var t as String], dl[var nc]]]
<->
  xbel[title[var t as String], var xc]
where
  contents(nc, xc)
```

The structure of each document is described by a regular expression pattern. That is, the Netscape document is an `html` element containing a `head` and a `body` subelements where the `head` contains a string and the `body` contains an `h1` and a `dl`; the XBEL document is simpler but quite similar.

In addition to the structure description, patterns extract substructures of documents and assign them to variables. We have two kinds of variables, *terminal* and *non-terminal* variables. A terminal variable is one that comes with a pattern constraint (in the form `... as P`) with which the substructure to be extracted must match. Also, a terminal variable appearing on one side must appear on the

other side, and the substructures corresponding to these variables are required to coincide. In the example, the content of the `h1` on the left equals to that of the `title` on the right, and must be a string. (A pattern constraint most typically used is `String` but it could in general be any structured one. Also, the pattern constraints on the same variable are usually the same on both sides, but could in general be different.)

A non-terminal variable is one that does not have a pattern constraint. Instead, two non-terminal variables bound on both sides are constrained by another relation specified in the `where` clause. (For avoiding confusion, we forbid the use of the same non-terminal variables on both sides.) In the above relation definition, the variables `nc` and `xc`, each bound to the content of the `dl` element and the remainder of the `title` element, are related by the `contents` relation defined below.

We next describe a relation between the "contents" parts of the two formats, which are, on each side, a sequence of bookmark items and folders.

```
relation contents =
  (var nb | var nf)*
<->
  (var xb | var xf)*
where
  bookmark(nb, xb),
  folder(nf, xf)
```

In both patterns, since each variable appears under a Kleene closure `*`, it may be matched more than once. As mentioned in the introduction, these are called non-linear variables and work in such a way that the n-th value matched by a variable on the LHS corresponds to the n-th value matched by the corresponding variable on the RHS.[1] In the above example, since the non-terminal variables `nb` and `xb` are related by `bookmark`, the n-th values matched by `nb` and `xb` correspond to each other; similarly for the variables `nf` and `xf`. Note that each pattern in this relation definition has no ambiguity thanks to the relation constraints specified in the `where` clause. That is, as we will see below, the first component of the `bookmark` relation matches only a `dt` element and that of the `folder` matches only a `dd` element. Therefore, for any input value, `nb` and `xb` have a unique way of matching. This is also the case for `nf` and `xf`.

What remains is to specify relations for "bookmark" parts and "folder" parts on both sides. For the former, we write the following.

```
relation bookmark =
  dt[a[@href[var url as String],
       var title as String]]
<->
  bookmark[@href[var url as String],
           title[var title as String]]
```

Here, we have no `where` clause since it uses only terminal variables. A label starting with `@` represents an XML attribute. For example, on the LHS of the `bookmark` relation, the `a` element must contain an attribute `href` with a string value. Next, for folder parts, we define the following.

```
relation folder =
  dd[h3[var title as String], dl[var nc]]
<->
  folder[title[var title as String], var xc]
where
  contents(nc, xc)
```

Here, we call the `contents` relation recursively. In this way, we can describe relations over arbitrarily nested documents.

[1] Unlike Prolog, we do not require that all the values matched by the same variable on one side are *equal*.

Now, let us see how we can transform documents by the relation defined above. We only consider the forward transformation from the Netscape document in Figure 1; the backward one is symmetric. Starting with the `top` relation, we match the input document with the LHS pattern and obtain the binding

```
title  ↦  "my bookmarks"
nc     ↦  <dt>...</dt><dd>...</dd><dt>...</dt>
```

where `nc`'s value is the whole sequence contained in the top `dl` element. The variable `title` is terminal and therefore can be used directly on the RHS, whereas `nc` is non-terminal and has to be consulted to the `contents` relation to figure out what value the corresponding variable `xc` is bound to. Therefore we match `nc`'s value with the LHS pattern of the `contents` relation. This time, since this relation uses non-linear patterns, we bind each variable to a list of all the matched values sorted in the document order.

```
nb  ↦  <dt>...Foo's...</dt>; <dt>...Bar's...</dt>
nf  ↦  <dd>...</dd>
```

Then, we recursively perform this procedure using the relation `bookmark` for converting `nb`'s each value and `folder` for converting `nf`'s value, and eventually obtain the following binding of the variables `xb` and `xf` used on the RHS of the `contents` relation.

```
xb  ↦  <bookmark>...Foo's...</bookmark>;
       <bookmark>...Bar's...</bookmark>
xf  ↦  <folder>...</folder>
```

Using this binding together with the RHS pattern, we reconstruct the following sequence of elements as the "output" for the `contents` relation

```
<bookmark>...Foo's...</bookmark>
<folder>...</folder>
<bookmark>...Bar's...</bookmark>
```

Winding further back to the `top` relation, we bind the variable `xc` on the RHS to the above sequence and, finally, we emit the XBEL document in Figure 2 as the result of the whole transformation.

2.2 Non-linear variables

As in the example in Section 2.1, non-linear variables are convenient for concisely describing correspondences between repeated elements. However, one might argue that relations written with non-linear variables could be rephrased by using only linear variables. For example, the `contents` relation given above could be rewritten as follows.

```
relation contents_rec =
  (var nb | var nf), var nrest
<->
  (var xb | var xf), var xrest
where
  bookmark(nb, xb),
  folder(nf, xf),
  contents_rec(nrest, xrest)
| () <-> ()
```

(The example uses a "union" of relations whose semantics would be the obvious one. We actually do not support this; the reason will be explained shortly.) This relation uses a right recursion to establish the correspondence between elements on both sides in a one-by-one manner from head to tail. (Note that this slightly changes the original semantics since it enforces the orderings among bookmarks and folders to be the same on both sides. Of course, such a small change would entirely be acceptable in practice.) However, what if we change the original `contents` relation in the following?

```
relation contents_reorder =
```

```
  (var nb | var nf)*
<->
  (var xb)*, (var xf)*
where ...
```

On the LHS, bookmarks and folders can appear any times in any order, while, on the RHS, bookmarks must appear before folders. Therefore, in the case where a folder occurs before a bookmark on the LHS, we must *reorder* these elements on the RHS. Therefore this relation does not allow us to use the above-mentioned rewriting with the combination of right recursion and linear variables. This idiom occasionally happens in real applications and Appendix B shows such an instance found in our mutual converter between vCard-XML and ContactXML.

A subtle semantics on non-linear variables is that the numbers of times that each variable matches must be equal on both sides. This has perfectly made sense for the examples seen already. For example, in the `contents_reorder` relation above, the forward transformation always yields exactly the same number of bookmark items as the input and similarly for folders. However, this same-number requirement may interact with the constraints expressed in regular expressions. For example, consider the following relation.

```
relation many_tels_one_phone =
  tel[var t as String]*
<->
  phone[var t as String]
```

The LHS accepts an empty sequence, in which case `t` is bound to zero values; also, the LHS accepts a sequence with length more than one, in which case `t` is bound to more than one value. In either case, transformation fails since we cannot fulfill the requirement on the RHS that `t` is bound to exactly one value. (Note that, currently, failure occurs at run time in such case. Alternatively, we could signal a warning at compile time since such failure is likely to be a programming mistake. Investigation in this direction is left for future work.)

We may think of a more relaxed semantics that may additionally supply arbitrary values or may cut off extra values for adjusting the numbers of bound values on both sides. However, this would lead to a complicated design problem. For example, consider again the `contents` relation given above. If we allow some bound values to be cut off, this would permit the forward transformation to always yield an empty sequence since it is a perfectly legitimate value resulting from the RHS by consuming no bound values. This is obviously not what we want. How about allowing some bound values to be added? Since allowing this on one direction of transformation implies allowing values to be cut off on the other direction. Therefore this also raises the same problem.

A positive by-product of the same-number requirement is that we can exploit it for relating optional elements or choices of elements. For example, in the transformation for the following relation

```
relation email_info =
  email[var t as String]?
<->
  internet[var t as String]?
```

whether `email` is present or not is conveyed by the number of `t`'s values. Likewise, in the following

```
relation phone_info =
  home[var h as String] |
  work[var w as String]
<->
  private[var h as String] |
  official[var w as String]
```

whether `home` or `work` is used is indicated by the numbers of `h`'s and `w`'s values. These can (perhaps more cleanly) be realized, instead, by introducing unions of relations, e.g.:

```
relation email_rel =
  email[var t as String]
<->
  internet[var t as String]
|
  () <-> ()
```

However, we want non-linear variables in the language in any case and, as far as we can tell, all examples using unions of relations can be rewritten by non-linear variables. Therefore our current language design does not support unions of relations.

As a final note, non-linear variables become not as useful as we might want once a pattern has a complex structure. For example, consider the following relation.

```
relation time_seq =
  time[day[var d as String],
       hour[var h as String]?]*
<->
  temps[jour[var d as String],
        heure[var h as String]?]*
```

Since the variable `d` holds all the `day`s contents and, independently, the variable `h` holds all the `hour`'s contents, we lose the information that each `day` and the `hour` that optionally follows are a pair. We can overcome this problem by hoisting the inner structure to another relation as follows:

```
relation time_seq =
  (var e)*  <->  (var f)*
where
  time(e,f)

relation time =
  time[day[var d as String],
       hour[var t as String]?]
<->
  temps[jour[var d as String],
        heure[var t as String]?]
```

This indicates a certain ad-hoc-ness in our design choice and suggests a further refinement by introducing more structures in variable bindings, e.g., a list of pairs of values, rather than just a list of values. However, such situations are relatively infrequent and can often be rephrased in another way like the above example. Moreover, such extension would considerably complicate the language definition and implementation. For these reasons, we have decided to stay with the current design.

2.3 Ambiguity

When a relation allows a document on one side to correspond to multiple documents on the other side, we say the relation to be *ambiguous*. It is one of our design choices that we fully permit ambiguity. In transformation, when several outputs are possible from an input, we simply choose one of them as an actual result. Ambiguity arises for various reasons in practice for applications involving two independently defined formats. We discuss below typical cases.

Choice of multiple representations Ambiguity arises when a certain kind of information can be represented only in one way on one side but in several ways on the other side. We have already shown an example in the introduction where one format has several representations for a telephone number while the other has only one. Another typical situation is when one of the formats allows a choice

of an element or an attribute to put a certain kind of information (this idiom is also known as *attribute-element constraints*; see [14] for related discussions).

```
relation identification =
  person[
    (@id[var i as String] | id[var i as String]),
    name[var n as String]
<->
  employee[idnum[var i as String],
           name[var n as String]
```

Freedom of ordering Ambiguity also arises when either side involves a repetition of alternation. For example, consider again the `contents` relation given above.

```
relation contents =
  (var nb | var nf)*
<->
  (var xb | var xf)*
where
  bookmark(nb, xb),
  folder(nf, xf)
```

Recall that, in the discussion of the transformation in Section 2.1, at the point where we reconstruct an output for the RHS of `contents` from the binding of `xb` and `xf`, our previous explanation has silently chosen the same ordering of bookmarks and folders as the input and yielded:

```
<bookmark>...Foo's...</bookmark>
<folder>...</folder>
<bookmark>...Bar's...</bookmark>
```

However, this is actually only one possibility. What the semantics of non-linear variables implies is that the ordering among bookmarks and the ordering among folders are *each* preserved; any output satisfying this constraint is possible. Therefore the following ordering is also possible, for example.

```
<folder>...</folder>
<bookmark>...Foo's...</bookmark>
<bookmark>...Bar's...</bookmark>
```

For this particular example, we can rewrite it by using right recursion for eliminating ambiguity as in Section 2.2. However, in the same section, we have shown the following variant of the `contents` relation

```
relation contents_reorder =
  (var nb | var nf)*
<->
  (var xb)*, (var xf)*
where ...
```

which does not permit such rewriting. Note that, in this example, the backward transformation has the same ambiguity on ordering as we have discussed. Therefore allowing ambiguity is of critical importance in our language design.

Unused data Yet another cause of ambiguity is that a certain kind of information representable in one format does not appear in the other. For example, in the `top` relation given above

```
relation top =
  html[head[String],
       body[h1[var t as String], dl[var nc]]]
<->
  xbel[title[var t as String], var xc]
where ...
```

the content of `head` on the left is not used on the right. This is not a problem in the forward transformation, but it is in the backward direction since we need to fill in something there. Fortunately, we have specified that, whatever `head` contains, the relation holds; therefore we can put an arbitrary string in it. In practical applications, more structured patterns than `String` can appear for unused data. For example, the above `head` pattern can be made more complicated like

```
head[meta[String]*, title[String], base[String]]
```

in which case an arbitrary structured value matching this pattern can be emitted for this part.

As already mentioned, our semantics for choosing one of multiple possibilities is nondeterministic. This design choice is for simplifying the definition and the implementation of the language and thus focusing on studying the issues specific to bidirectional transformations. There can be other choices introducing some kind of priority rules (the most typical one is "first-match," which takes the earliest matching pattern [15]; more discussions can be found in [13]). We have not yet examined any implication in the programming nor the algorithmic aspect, though.

3. Language Definition

This section formalizes the syntax and semantics of our language. For simplicity, we elide XML attributes and texts from our formalism, though we continue to use them in examples. Details on our treatment of attributes will be given in the full version of this paper.

3.1 Syntax

We assume a set $\mathcal{L}$ of *labels*, ranged over by `e`. We define *values* t by the following grammar.

$$t \ ::= \ \varepsilon \mid \texttt{<e>}t\texttt{</>} \mid t\ t$$

That is, a value is a sequence of *elements*, each consisting of a label and another value (an empty sequence is written ε). We abbreviate an element $\texttt{<e>}\varepsilon\texttt{</>}$ with an empty content by `<e/>`. The set of all values is denoted by $\mathcal{T}$.

We next define the syntax of programs. We assume a set $\mathcal{V}$ of *variables*, a set $\mathcal{V}_T$ of *terminal variables*, and a set $\mathcal{V}_N$ of *nonterminal variables* such that $\mathcal{V} = \mathcal{V}_T \cup \mathcal{V}_N$ and $\mathcal{V}_T \cap \mathcal{V}_N = \emptyset$. We use x, x_T, and x_N to range over $\mathcal{V}$, $\mathcal{V}_T$, and $\mathcal{V}_N$, respectively. We define *patterns* P by the following grammar.

$$\begin{aligned} P \ ::= \ & () \mid \mathbf{var}\ x_T\ \mathbf{as}\ P \mid \mathbf{var}\ x_N \mid \\ & \texttt{e}[P] \mid P, P \mid (P \mid P) \mid P* \end{aligned}$$

We define $\mathrm{Var}(P)$ as the set of all variables appearing in the pattern P, and $\mathrm{Var}_N(P) = \mathrm{Var}(P) \cap \mathcal{V}_N$. As usual, we use other regular expression operators for shorthands, namely, $P?$ for $(P \mid ())$ and $P+$ for $(P, P*)$. A *program* Π is a finite set of *relation definitions*, each of which has the form

$$\mathbf{relation}\ r = P \leftrightarrow P\ \mathbf{where}\ W$$

where r is a *relation name* and W, called *where-clause*, is a finite set of *relation constraints* each having the form $r(x_N, x_N)$. We assume any program to declare each relation name at most once and thus can regard it as a finite mapping from relation names to triples $(P \leftrightarrow P\ \mathbf{where}\ W)$. Also, we require the program to declare the special relation name `top` (from which the interpretation begins). Further, in each relation definition

$$\mathbf{relation}\ r = P \leftrightarrow Q\ \mathbf{where}\ \{r_1(x_1, y_1), \ldots, r_n(x_n, y_n)\}$$

the pattern on each side must contain a disjoint set of nonterminal variables, each constrained by a relation constraint exactly once, i.e., we require that $\mathrm{Var}_N(P) \cap \mathrm{Var}_N(Q) = \emptyset$, that

$\{x_1, \ldots, x_n\} = \mathrm{Var}_N(P)$, that $\{y_1, \ldots, y_n\} = \mathrm{Var}_N(P)$, and that $x_i \neq x_j$ and $y_i \neq y_j$ for any $i \neq j$. Lastly, all relation names that are referred to must be declared: $r_i \in \mathbf{dom}(\Pi)$ for each i. From now on, we assume a fixed program Π.

We impose three algorithmically motivated restrictions on the syntax, namely (1) regularity, (2) non-overlapping of pattern variables, and (3) non-nullability of repeated patterns. Regularity is for ensuring that each side of the whole relation defined by a program accepts only a regular tree language. For example, in the following relation definition

```
relation r =
  (a[], var x, b[]) | ()  <->  ...
where
  r(x, ...)
```

the LHS accepts the set of sequences consisting of the same number of `a`'s and `b`'s, which goes beyond regularity. Confining ourselves to regularity allows us to use a finite-automata-based framework, greatly simplifying the algorithmics. For ensuring regularity, we use the standard condition that "any recursion must be enclosed by a label." A formal definition is omitted from this abstract.

The second restriction, non-overlapping of pattern variables, is for ensuring that the same element is never bound to more than one variable. This restriction particularly simplifies the part of the algorithm generating the output. For example, if we write the following relation

```
relation r =
  (var x as a[String]), (var y as a[String])
<->
  var x as (var y as a[String])
```

the forward transformation would have to check that the two `a`'s are identical since these must become the same element in the output. The formal statement for the restriction is simple: for any occurrence of the form $\mathbf{var}\ x\ \mathbf{as}\ P$, we require that $\mathrm{Var}(P) = \emptyset$.

The third restriction, non-nullability of repeated patterns, is for avoiding the possibility that a variable may be bound to a list of an arbitrary number of empty sequences, which would substantially complicate the evaluation algorithm. For example, consider the pattern `(var x as a[]?)*`. Given an empty sequence as input, the length of the list that may be assigned to `x` has no upper bound. Our actual restriction is slightly stronger: it forbids any repetition of a pattern that may match an empty sequence. This also rejects patterns with no variable at all, e.g., `(a[]?)*`, but this seems to cause little problem in practice. A formal definition is elided from this abstract.

3.2 Semantics

As already mentioned, a variable is bound to a list of values. Since a value itself is a sequence, we introduce a notation for avoiding the confusion between lists and sequences. A list of n objects o_1, o_2, …, and o_n is written by $o_1; o_2; \ldots; o_n$. We also use ; for the operation to concatenate two lists. An empty list is written $\bullet$.

A binding β is a total mapping from variables to lists of values. We write $\{x \mapsto l\}$ for the binding that maps the variable x to the list l and any other variable to an empty list. Analogously, we write $\{\}$ for the binding that maps every variable to an empty list. In addition, we define the binary operator $\oplus$ for combining two bindings in a way that concatenates the two lists that each variable is bound to. That is, $(\beta_1 \oplus \beta_2)(x) = \beta_1(x); \beta_2(x)$ for each variable x.

We then define the *matching judgment* $t \triangleright P \Rightarrow \beta$, read "value t matches pattern P with binding β," by the following set of rules.

$$\text{B-UNIT}\ \frac{}{\varepsilon \triangleright () \Rightarrow \{\}} \qquad \text{B-NON}\ \frac{}{t \triangleright \mathbf{var}\ x_N \Rightarrow \{x_N \mapsto t\}}$$

$$\text{B-TERM}\ \frac{t \triangleright P \Rightarrow \{\}}{t \triangleright \mathbf{var}\ x_T\ \mathbf{as}\ P \Rightarrow \{x_T \mapsto t\}} \qquad \text{B-LAB}\ \frac{t \triangleright P \Rightarrow \beta}{\texttt{<e>}t\texttt{</>} \triangleright \texttt{e}[P] \Rightarrow \beta}$$

$$\text{B-CAT}\ \frac{t_1 \triangleright P_1 \Rightarrow \beta_1 \qquad t_2 \triangleright P_2 \Rightarrow \beta_2}{t_1\ t_2 \triangleright P_1, P_2 \Rightarrow \beta_1 \oplus \beta_2} \qquad \text{B-ORL}\ \frac{t_1 \triangleright P_1 \Rightarrow \beta}{t_1 \triangleright P_1 | P_2 \Rightarrow \beta}$$

$$\text{B-ORR}\ \frac{t_1 \triangleright P_2 \Rightarrow \beta}{t_1 \triangleright P_1 | P_2 \Rightarrow \beta} \qquad \text{B-REP}\ \frac{t_1 \triangleright P \Rightarrow \beta_1 \qquad \ldots \qquad t_n \triangleright P \Rightarrow \beta_n}{t_1\ \ldots\ t_n \triangleright P* \Rightarrow \beta_1 \oplus \cdots \oplus \beta_n}$$

In B-CAT and B-REP, the same variable can be bound to different lists of values in the subpatterns, in which case the resulting binding maps the variable to the concatenation of these lists. In B-TERM, the subpattern always returns an empty binding since, as we have required, it contains no variable. On top of the matching judgment, we define the *relating* judgment $\Pi \vdash r(t, u)$ by the following rule

$$\frac{\Pi(r) = (P \leftrightarrow Q\ \mathbf{where}\ W) \qquad t \triangleright P \Rightarrow \beta \qquad u \triangleright Q \Rightarrow \gamma \qquad \forall x \in \mathcal{V}_T.\ \beta(x) = \gamma(x) \qquad \forall r'(x, y) \in W.\ \Pi \vdash r'(\beta(x), \gamma(x))}{\Pi \vdash r(t, u)}$$

where we write $\Pi \vdash r((t_1; \ldots; t_n), (u_1; \ldots; u_n))$ when $\Pi \vdash r(t_i, u_i)$ for $i = 1, \ldots, n$. That is, trees t and u are related by r when these are matched by the corresponding patterns. Further, the yielded bindings map each terminal variable to the same list of values and each non-terminal variable to the same length of lists whose each corresponding pair of elements is related accordingly to the where-clause. Finally, we define the whole program's relating judgment $\Pi \vdash (t, u)$ simply by $\Pi \vdash \mathtt{top}(t, u)$.

4. Evaluation Algorithm

This section presents an algorithm whose goal is to produce an output value t' from a program Π and an input value t such that $\Pi \vdash (t, t')$. We focus here only on the forward transformation since the other direction is symmetric. Our strategy, as usual, consists of the static and the dynamic parts. In the static part, we convert the program into two instances of *parsing automata*, where one of these ("LHS automaton") represents a "package" of the patterns appearing on the left-hand-sides of all the relations and the other ("RHS automaton") represents one on all the right-hand-sides. The dynamic part executes the automata with the input in two phases. The first phase is called "parsing" and generates an intermediate "parse tree" data structure from the input value and the LHS automaton; the second phase is called "unparsing" and produces an output value from the intermediate structure and the RHS automaton. Both phases are one-pass; this is achieved by a standard technique that "collects all possible states." As a result of using this, the parsing phase actually yields a *set* of all possible parse trees, from which the unparsing phase produces a *set* of all possible outputs, as explained in the sequel.

Below, we first give our automata model and then present our one-pass algorithms for the parsing and the unparsing phases. An automaton construction from programs in the surface language is presented in Appendix A.

4.1 Parsing Automata

Overview To see what parse trees are, consider first the following relation definitions.

```
relation top =
  (var x as a[]), var y  <->  ...
where
  sub(y, ...)

relation sub =
  b[], (var z as c[])+  <->  ...
```

According to the semantics formalized in Section 3.2, we first obtain a binding by matching the input with the `top` relation's LHS and then, for the non-terminal variable `y`, we evaluate the LHS of the corresponding sub-relation `sub` with the value to which `y` is bound. For efficiency, our strategy evaluates both `top`'s LHS and `sub`'s LHS at the same time and produces an intermediate structure that combines bindings obtained from these. However, we need to be able to identify which relation each binding belongs to. Therefore our intermediate structures, a.k.a. parse trees, have the form of a binding that not only maps terminal variables to values but also recursively maps non-terminal variables to other bindings. In the above example, from the input value

`<a/><b/><c/><c/>`

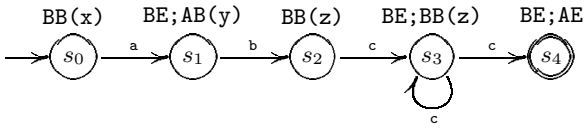
we generate the following parse tree.

$$\{\texttt{x} \mapsto \texttt{<a/>},\ \texttt{y} \mapsto \{\texttt{z} \mapsto (\texttt{<c/>};\ \texttt{<c/>})\}\}$$

That is, the parse tree holds, in addition to a binding of the terminal variable `x` to a value, a binding of the non-terminal variable `y` to a "sub-binding" that records values for the terminal variable `z` used in the `sub` relation.

Parsing automata aim at representing a package of patterns that yields such parse tree while accepting a value. They are actually exactly the same as standard (nondeterministic) finite tree automata except that we devise a mechanism called *markers* for treating variable binding and relation constraints. Before explaining markers, what an "accepting" automaton corresponding to the above `top` example should look like, besides binding and relation constraints? We can easily see that such an automaton can be obtained by constructing an automaton for `top`'s LHS where an automaton for `sub`'s LHS is embedded as the "sub-automaton" corresponding to the pattern `var y`.

On top of such automaton, we add markers to states in order to be able produce an appropriate parse tree. We have four kinds of markers. Two have the form $\texttt{BB}(x_T)$ ("binding begins") and `BE` ("binding ends") and are related to terminal variables. That is, for a pattern of the form $\mathbf{var}\ x_T\ \mathbf{as}\ P$, we add the mark $\texttt{BB}(x_T)$ to the starting state of the sub-automaton for P and `BE` to its final state. When the automaton runs, it generates a binding of x_T to the sequence accepted between these states. Two other kinds of markers have the form $\texttt{AB}(x_N)$ ("automaton begins") and `AE` ("automaton ends") and are related to non-terminal variables. That is, for a pattern of the form $\mathbf{var}\ x_N$ and the sub-relation that constrains x_N (note that it is uniquely determined thanks to the syntactic restrictions in Section 3.1), we add the mark $\texttt{AB}(x_N)$ to the starting state of the sub-automaton representing the sub-relation's LHS; we also mark `AE` to its final state. When the automaton runs, it generates a binding of x_N to a sub-binding yielded by the sub-automaton. As an example, a parsing automaton corresponding to the relation definitions above looks like the following.



(For brevity, each transition here has only one "destination" state corresponding to the sequence remaining after a label. Later in the formalization, we will augment each transition with another destination state for matching the content of the label.) The whole automaton corresponds to the `top` relation and the sub-automaton between s_1 and s_4 to the relation `sub`. Note here that each state is associated with a *list* of markers. This is because a single state can be the end of a binding (or a sub-automaton) and the beginning of another at the same time. Also, when a variable binds an empty sequence, markers for both the beginning and the end of the same variable are put to a single state.

A parsing automaton can be used also in the unparsing phase by swapping the roles of the (input) value and the parse tree. Note that, in this, we follow transitions for *producing* nodes of a value that are matched by the transitions, rather than for consuming such nodes. For example, from the parse tree

$$\{\texttt{x} \mapsto \texttt{<a/>},\ \texttt{y} \mapsto \{\texttt{z} \mapsto (\texttt{<c/>};\ \texttt{<c/>})\}\}$$

the above automaton first consumes the value `<a/>` in `x`'s binding, checks that it is accepted by the transition between $\texttt{BB(x)}$ and `BE`, and emits it. After the marker $\texttt{AB(y)}$, we proceed the execution with the sub-parse-tree $\{\texttt{z} \mapsto (\texttt{<c/>};\ \texttt{<c/>})\}$, where we first silently emit `<b/>` (since we follow the transition with label `b`) and then proceed to consume each of `z`'s values. Eventually, the automaton outputs the value `<a/><b/><c/><c/>`. Using parsing automata symmetrically both for "parsing" and "unparsing" is one of the simplicities of our evaluation strategy.

Syntax and semantics A *parsing automaton* M is a tuple $(S, S_0, F, \Delta, \chi)$ where S is a finite set of *states*, $S_0 \subseteq S$ is a set of *initial states*, $F \subseteq S$ is a set of *final states*, $\Delta \subseteq S \times \mathcal{L} \times S \times S$ is a set of *transition rules*, and $\chi \in S \to \mathcal{K}^*$ associates each state with a list of *markers*. Here, markers $v \in \mathcal{K}$ are defined by the following:

$$v \ ::= \ \texttt{BB}(x_T) \mid \texttt{BE} \mid \texttt{AB}(x_N) \mid \texttt{AE}$$

For visual appeal, we sometimes write $s \xrightarrow{\texttt{e}[s_1]} s_2$ for a transition rule $(s, \texttt{e}, s_1, s_2)$ and call s_1 *content state* and s_2 *remainder state*.

Parsing automata constructed from biXid programs obey a certain nesting constraint on markers and the algorithms given in the sequel assume only such automata. Namely, `BB` and `BE` makers and `AB` and `AE` markers must appear in a nested way in the same "horizontal" level, i.e., an end marker is reachable from its beginning marker by taking remainder states. Moreover, since the "non-overlapping requirement" says that the pattern constraint of each variable pattern must not contain other variables, no marker can appear between a `BB` and its corresponding `BE` even in "deeper" levels, i.e., anywhere reachable by taking both content and remainder states. To summarize these, suppose that there is a sequence $s_0 \xrightarrow{\texttt{e}_1[s'_1]} s_1 \xrightarrow{\texttt{e}_2[s'_2]} \cdots \xrightarrow{\texttt{e}_n[s'_n]} s_n \in \Delta$ where $s_n \in F$ and either $s_0 \in S_0$ or $(s, \texttt{e}, s_0, s') \in \Delta$ for some $s, s', \texttt{e}$. Then, we require that the list $\chi(s_0); \chi(s_1); \ldots; \chi(s_n)$ satisfies the following grammar.

$$V \ ::= \ \texttt{BB}(x_T); \texttt{BE} \ \mid \ \texttt{AB}(x_N); V; \texttt{AE} \ \mid \ \bullet$$

In addition, suppose that there is a sequence $s_1 \xrightarrow{\texttt{e}_2[s'_2]} \cdots \xrightarrow{\texttt{e}_n[s'_n]} s_n$ such that $\chi(s_1) = (\ldots; \texttt{BB}(x_T))$, $\chi(s_2) = \bullet, \ldots, \chi(s_{n-1}) = \bullet$, and $\chi(s_n) = (\texttt{BE}; \ldots)$. Then, we require that $\mathrm{marks}(s'_i) = \emptyset$ for each $i = 2, \ldots, n$, where $\mathrm{marks}(s)$ is the set of markers appearing in the states reachable from s. In this case, the states $s_2, \ldots, s_{n-1}$ are said to be in the *scope* of x_T.

A *parse tree* b is inductively defined as a total mapping from terminal variables to lists of values and from non-terminal variables to lists of parse trees. We use notations similar to those for bindings, namely, $\{x \mapsto l\}$ for the mapping from the variable x to l and from any other variable to an empty list), and $\{\}$ for the mapping from

any variable to an empty list, and the "concatenation" $\oplus$ of two parse trees similarly defined as before. The set of all parse trees is written as $\mathcal{B}$.

A parsing automaton $M = (S, S_0, F, \Delta, \chi)$ works in terms of *configurations* of the form $(s, x_\perp, p, b)$ where $s \in S$ is the "current state," $x_\perp \in \mathcal{V}_T \cup \{\perp\}$ is the "current scope," $p \in (\mathcal{V}_N \times \mathcal{T})^*$ is the "current stack" explained below, and $b \in \mathcal{B}$ is the "current parse tree." The automaton operates on these components of configuration mostly when it steps on a state and processes the list of markers associated to the state. We formalize this behavior by the function `pms` ("process markers") defined by

$$\texttt{pms}(s, x_\perp, p, b) = \texttt{pm}(v_n, (\ldots(\texttt{pm}(v_1, (s, x_\perp, p, b)))\ldots))$$

where $(v_1; \ldots; v_n) = \chi(s)$ and each individual marker v_i is interpreted by the following function `pm`:

$$\begin{array}{lcl}
\texttt{pm}(\texttt{BB}(x_T), (s, \perp, p, b)) & = & (s, x_T, p, b \oplus \{x_T \mapsto \varepsilon\}) \\
\texttt{pm}(\texttt{BE}, (s, x_T, p, b)) & = & (s, \perp, p, b) \\
\texttt{pm}(\texttt{AB}(x_N), (s, \perp, p, b)) & = & (s, \perp, ((x_N, b); p), \{\}) \\
\texttt{pm}(\texttt{AE}, (s, \perp, ((x_N, b'); p), b)) & = & (s, \perp, p, b' \oplus \{x_N \mapsto b\})
\end{array}$$

Markers $\texttt{BB}(x_T)$ and `BE` operate on the current scope and the current parse tree. That is, the marker $\texttt{BB}(x_T)$ sets the scope to the terminal variable x_T and the marker `BE` closes it as $\perp$. The marker $\texttt{BB}(x_T)$ also appends a new empty sequence value to the list that x_T is bound to in the current parse tree. In the scope of x_t, the automaton accumulates each consumed element to this value. Markers $\texttt{AB}(x_N)$ and `AE` operate on the current stack as well as the current parse tree. Let us first explain what a stack is for. Recall that a parse tree is a tree of bindings where each inner binding is constructed while evaluating a sub-relation. At a moment where evaluation of a sub-relation proceeds, a stack holds the list of partial bindings that have been being built for the parent and ancestor relations. In our example, while evaluating the `sub` relation and building the inner parse tree $\{\texttt{z} \mapsto (\texttt{<c/>};\ \texttt{<c/>})\}$, the stack holds the partial parse tree $\{\texttt{x} \mapsto \texttt{<a/>}\}$ that has been build for the `top` relation. When we return from the `sub` relation, we need to add a binding of `y` to the constructed inner parse tree. For this, we also remember the variable `y` in the stack. In general, when the automaton encounters an $\texttt{AB}(x_N)$ marker, it pushes the pair of the variable x_N and the current parse tree onto the current stack p and, at the same time, sets up an empty parse tree for evaluating the sub-automaton that is now starting. Then, when the automaton reaches the corresponding `AE` marker, it pops the variable x_N and the parent parse tree b' from the stack, augments this parse tree with a binding of x_N to the current parse tree, and sets the resulting parse tree as the current one.

The behavior of a parsing automaton is described by the *parsing* relation of the form $M \vdash t \diamond (s, x_\perp, p, b) \to b'$, read "automaton M parses value t from configuration $(s, x_\perp, p, b)$ with parse tree b'." For brevity, the definitions in the sequel assume an automaton $M = (S, S_0, F, \Delta, \chi)$ and thus omits any mention of M without confusion. The parsing relation is defined by the following rules.

$$\frac{(s, \perp, p', b') = \texttt{pms}(s, x_\perp, p, b) \qquad s \in F}{\epsilon \diamond (s, x_\perp, p, b) \to b'} \text{ [I-ACCEPT]}$$

$$\frac{\begin{array}{c}(s, x'_\perp, p', b') = \texttt{pms}(s, x_\perp, p, b) \\ (s, \texttt{e}, s_1, s_2) \in \Delta \qquad b_2 = \texttt{chtl}(x'_\perp, b', \texttt{<e>}t_1\texttt{</>}) \\ t_1 \diamond (s_1, \perp, p', b_2) \to b_3 \\ t_2 \diamond (s_2, x'_\perp, p', b_3) \to b_4\end{array}}{\texttt{<e>}t_1\texttt{</>}t_2 \diamond (s, x_\perp, p, b) \to b_4} \text{ [I-TRANS]}$$

The rule I-ACCEPT first processes the markers and returns the resulting parse tree if the state is final. The rule I-TRANS also first processes the markers. Then, if the automaton is in the scope of x_T, we append the element $\texttt{<e>}t_1\texttt{</>}$ to the end of the *last* value in the list of x_T's binding (which is the temporary value that the automaton is building at this moment). This operation is done by using the following auxiliary function `chtl` ("change tail"):

$$\begin{array}{lcl}
\texttt{chtl}(x_T, b \oplus \{x_T \mapsto t'\}, t) & = & b \oplus \{x_T \mapsto t'\ t\} \\
\texttt{chtl}(\perp, b, t) & = & b
\end{array}$$

Then, with the parse tree b_2 obtained from this and the content state s_1, the rule I-TRANS proceeds to the content value t_1, yielding another parse tree b_3. (Note that we reset the scope to $\perp$ when entering in the content state since we must not append inner elements to the temporary value.) Further, with this parse tree and the remainder state s_2, we continue to process the remainder value t_2. We can now define the whole semantics of parsing automata: automaton M *parses* value t to parse tree b when $M \vdash t \diamond (s, \perp, \bullet, \{\}) \to b$ for some $s \in S_0$.

The already-mentioned "nesting" constraint on a parsing automaton ensures its "nesting" behavior. That is, we set the scope closed at the whole beginning, at a content state (the fourth premise of I-TRANS), or at the entry of a sub-automaton (in $\texttt{pm}(\texttt{AB}(x_N))$); then, the scope is ensured to be closed again when we reach a final state (the first premise of I-ACCEPT) or at the exit of the sub-automaton (in $\texttt{pm}(\texttt{AE})$). Similarly, we start with an empty stack at the whole beginning and we ensure that the stack returns to be empty at the end. Also, the nesting requirement guarantees that the stack at a content state and the one at a final state are the same. (The last point allows the rule I-TRANS to use the same stack p' at the content state s_1 and at the remainder state s_2.) In addition to these, the "non-overlapping" constraint ensures that, when processing markers, a $\texttt{BB}(x_T)$ and an $\texttt{AB}(x_N)$ can expect the scope to be closed.

By using the semantics of parsing automata, we can now rephrase the goal of our algorithm as "given a value t with two automata M_L and M_R, find another value t' such that M_L parses t to a parse tree b and, simultaneously, M_R parses t' to b, for some b." (Though, note that, in order to be able to use the same b here, we need to rename non-terminal variables appearing in each relation, e.g., replace each occurrence of x_N with x'_N whenever $r(x_N, x'_N)$ is in the `where`-clause.) The correspondence between the surface language semantics and the automata semantics is intuitively clear and therefore omitted from this abstract.

Simplification for unparsing The unparsing phase needs a care for ensuring termination. For example, in the following pattern

```
((var x as a[]) | b[])*
```

a naive algorithm would diverge by taking, at each repetition, the choice of emitting a `b` element without consuming `x`'s values at all. Although there could be some dynamic way of solving this problem, our approach is to statically rewrite such patterns to harmless ones. For example, we transform the pattern above to:

```
(var x as a[])*
```

This is legitimate since any output value would perfectly satisfy the original constraint even though there is no longer the possibility that the output contains any `b` element. Note that we cannot perform the same transformation for the parsing side since it would incorrectly reject some values that need to be accepted. This further implies that we need to create, in total, *four* parsing automata for a given program: two for the parsing and the unparsing phases, in each of the forward and the backward directions.

Formally, simplification transforms, from inner to outer, each pattern $P*$ that does not appear under a (**var** x **as** ...), in the follow way:

1. convert P to an equivalent pattern of the form $(P_1 \mid P_2 \mid \ldots \mid P_n)$ where $n \geq 1$ and each P_i does not have a choice at the

top level. This can be done essentially by distributing choices appearing in concatenations. For example, $P_1, (P_2 \mid P_3), P_4$ is converted to $(P_1, P_2, P_4) \mid (P_1, P_3, P_4)$;

2. rewrite $(P_1 \mid P_2 \mid \ldots \mid P_n)*$ to $(P_1*, P_2*, \ldots, P_n*)$;
3. for each i, replace P_i* with () if $\mathrm{Var}(P_i) = \emptyset$.

Note that it is incorrect to apply this transformation for a pattern under a variable binder like `(var x as (a[] | b[])*)` since it would change the pattern constraint to `()` and therefore would cause a wrong failure for valid bindings given from the parsing phase.

4.2 Parsing phase

This phase takes the "LHS" parsing automaton $M_L = (S, S_0, F, \Delta, \chi)$ and the input value. It then traverses the input value with only one visit to each node and produces the set of all possible parse trees. During the traversal, we work with sets of *parsing configurations* each of the form $c = (s, x_\perp, k, p, b)$. They are similar to configurations used in the description of the semantics except that each has a list $k \in (S \times (\mathcal{V}_T \cup \{\perp\}))^*$ of *parsing continuations*. To see why we need them, consider processing the value $\texttt{<e>}t_1\texttt{</>}t_2$ with a set of configurations one of which has a state s. For this, we first find a transition $(s, \mathtt{e}, s_1, s_2)$, then process the content value t_1 from the state s_1, and eventually obtain a parse tree b for this part. After this, we need to proceed to the remainder value t_2 with this parse tree, but from which state? At this point, the configuration loses the information that we came from the content state s_1 of the above transition and hence we should next go to s_2. In the semantics, this information is retained in the premise of I-TRANS, whereas the algorithm cannot take the same approach since it deals with a *set* of configurations that may possibly have stem from different transitions. Therefore we record this information in configurations themselves. Since element nesting can be arbitrarily deep, we keep a list of states to return. In addition, since we set the current scope to $\perp$ when entering in a content state (the fourth premise of I-TRANS), we need to revert it upon return. Therefore we also hold this information in the list. As a result, each configuration is added with a list of pairs $(s, x_\perp)$ describing the state s to continue for the remainder value and its scope $x_\perp$; each pair here is called parsing continuation.

The algorithm for the parsing phase is defined by the following function `parse` taking a parsing automaton M and an input value t and returning a set of parse trees

$$\mathtt{parse}_M(t) = \{\ b \mid (s, \perp, \bullet, \bullet, b) \in \mathtt{convert}_p(C_0, t)\ \}$$

where the initial set C_0 of parsing configurations is as follows.

$$C_0 = \{\ (s, \perp, \bullet, \bullet, \{\}) \mid s \in S_0\ \}$$

(Since the parsing phase works with the fixed automaton M, we will avoid further mentioning it in the sequel. Also, we add the subscript $_p$ to each function defined here for indicating that it is related to the parsing phase.) That is, we start with the configurations each with an initial state, the closed scope, an empty parsing continuation, an empty stack, and an empty parse tree. Then, we invoke the function $\mathtt{convert}_p$ and collect all the parse trees from the resulting configurations. The $\mathtt{convert}_p$ function performs the main traversal and is defined below. It takes a set C of configurations and an input and returns a set of configurations.

$$\mathtt{convert}_p(C, \varepsilon) = \{\ (s, \perp, k, p, b) \in \mathtt{pms}_p(C) \mid s \in F\ \}$$

$$\mathtt{convert}_p(C, \texttt{<e>}t_1\texttt{</>}t_2) = \mathtt{convert}_p(C'', t_2)$$

where

$$C' = \mathtt{convert}_p(\mathtt{down}_p(\mathtt{pms}_p(C), \texttt{<e>}t_1\texttt{</>}), t_1)$$
$$C'' = \{\ (s', x_\perp, k, p, b) \mid (s, \perp, ((s', x_\perp); k), p, b) \in C'\ \}$$

In both cases, we first process markers by the `pms` function defined as

$$\mathtt{pms}_p(C) = \left\{ \mathtt{pm}_p(v_n, (\ldots(\mathtt{pm}_p(v_1, c))\ldots)) \;\middle|\; \begin{array}{l} c = (s, x_\perp, k, p, b) \in C, \\ (v_1; \ldots; v_n) = \chi(s) \end{array} \right\}$$

where

$$\begin{array}{lcl} \mathtt{pm}_p(\mathtt{BB}(x_T), (s, \perp, k, p, b)) & = & (s, x_T, k, p, b \oplus \{x_T \mapsto \varepsilon\}) \\ \mathtt{pm}_p(\mathtt{BE}, (s, x_T, k, p, b)) & = & (s, \perp, k, p, b) \\ \mathtt{pm}_p(\mathtt{AB}(x_N), (s, \perp, k, p, b)) & = & (s, \perp, k, ((x_N, b); p), \{\}) \\ \mathtt{pm}_p(\mathtt{AE}, (s, \perp, k, ((x_N, b'); p), b) & & \\ \qquad & = & (s, \perp, k, p, b' \oplus \{x_N \mapsto b\})). \end{array}$$

These definitions are the same as those in the semantics except that $\mathtt{pms}_p$ here deals with a set of configurations and $\mathtt{pm}_p$ here takes a parsing configuration (with no change to the continuation list). After the marker processing, in the case of an empty sequence value, we drop all configurations whose states are not final. In the case of a value $\texttt{<e>}t_1\texttt{</>}t_2$, we form a set of configurations for processing the content t_1 by using the $\mathtt{down}_p$ function shown below and proceed the traversal by a recursive call to $\mathtt{convert}_p$. In the resulting set C', the head of each configuration's continuation list gives the state to continue for processing the remainder sequence t_2. Therefore we form another set C'' of configurations for the continuation states and perform another traversal by $\mathtt{convert}_p$.

The function $\mathtt{down}_p$ is defined as follows.

$$\mathtt{down}_p(C, \texttt{<e>}t\texttt{</>}) = \left\{ (s_1, \perp, k', p, b') \;\middle|\; \begin{array}{l} (s, x_\perp, k, p, b) \in C, \\ (s, \mathtt{e}, s_1, s_2) \in \Delta, \\ k' = ((s_2, x_\perp); k), \\ b' = \mathtt{chtl}(x_\perp, b, \texttt{<e>}t\texttt{</>}) \end{array} \right\}$$

That is, for each configuration in C and each transition from s, we form a new configuration in which we take the content state s_1, the closed scope $\perp$, the current continuation list k added with the pair of the remainder state s_2 and the current scope $x_\perp$, the current stack p, and the current parse tree b possibly appended with the element $\texttt{<e>}t\texttt{</>}$ in case $x_\perp$ is open (by using the `chtl` function defined in the last subsection).

Although the parsing phase traverses each node only once, it can potentially generate an exponential number of parse trees. For example, matching

```
((var x as a[String]) | (var y as a[String]))*
```

against a sequence of n `a` elements will yield a set of 2^n parse trees. However, such a pattern seems to be rare and, indeed, our experimental results in Section 5 indicate no blow-up.

Theorem 4.1 (Soundness) If $b \in \mathtt{parse}_M(t)$ under a parsing automaton M, then M parses t to b.

Theorem 4.2 (Completeness) If a parsing automaton M parses a value t to a parse tree b, then $b \in \mathtt{parse}_M(t)$ under M.

Theorem 4.3 (Termination) The algorithm `parse` terminates for any input.

4.3 Unparsing phase

The unparsing phase takes the "RHS" parsing automaton $M_R = (S, S_0, F, \Delta, \chi)$ and the set of parse trees resulted from the parsing step. It then performs a traversal with only one visit to each node in the set of parse trees, and produces the set of all possible output values. This time, the algorithm uses sets of *unparsing configurations* each of the form $d = (s, t_\perp, b, l, q, t)$ where each component is explained below.

- $s \in S$ is the current state.
- $b \in \mathcal{B}$ is the current parse tree; while the parsing phase produces it, the unparsing phase *consumes* it.
- $t_\perp \in \mathcal{T} \cup \{\perp\}$ is the current value to verify; when entering in a scope of a variable, we not only need to consume a value from its binding and emit it to the output, but also need to check that the transitions within this scope accept this value; $t_\perp$ holds the value currently being checked (and thus is also a value to consume); outside any scope, $t_\perp$ is set to $\perp$.
- $l \in (S \times (\mathcal{T} \cup \{\perp\}) \times \mathcal{T} \times \mathcal{L})^*$ is the current list of *unparsing continuations* each of the form $(s, t_\perp, t, \mathtt{e})$; it is explained below.
- $q \in \mathcal{B}^*$ is the current *unparsing stack*; unlike the parsing phase, the unparsing phase records only parse trees; it is also explained below.
- $t \in \mathcal{T}$ is the current output value; while the parsing phase deconstructs a value, the unparsing phase constructs it. Furthermore, while the parsing phase takes a single input value for all the configurations, the unparsing phase keeps one parse tree for each configuration since different configurations may yield different results.

Like in the parsing phase, the roles of a list of unparsing continuations and an unparsing stack are different. Unparsing continuations are for saving some information at the entrance of a content state and for restoring them at its exit and continuing with the remainder state. For example, suppose that we encounter a transition rule $(s, \mathtt{e}, s_1, s_2)$ with a configuration $(s, t_\perp, b, l, q, t)$ and proceed to the content state s_1. From this, we eventually reach the final state and, at this point, we need to recall that the remainder state is s_2, that we had been verifying the value $t_\perp$, that we had already constructed the partial output t, and that the label of the transition rule is `e`. Therefore we record these pieces of information in the continuation list and, afterward, we can continue from the state s_2 with the current output $t\texttt{<e>}t'\texttt{</>}$, which can be constructed from the previous output t, the label `e`, and the output t' obtained from the processing from the content state s_1.

An unparsing stack is, on the other hand, for saving some information when following an $\mathtt{AB}(x_N)$ marker and for restoring it later when following its corresponding `AE`. For example, consider the situation where we follow an $\mathtt{AB}(x_N)$ with a configuration $(s, t_\perp, b, l, q, t)$ and then proceed to the sub-automaton between the $\mathtt{AB}(x_N)$ and the `AE`. At the moment exiting from this, we need to recall that we had been consuming the parse tree b. We need not memorize other components, in particular, the value $t_\perp$ to verify since the nesting constraint requires that, at the moment of seeing the $\mathtt{AB}(x_N)$, we are out of any scope and therefore $t_\perp$ must be $\perp$.

Now, we formalize the unparsing phase by the following function `unparse`

$$\mathtt{unparse}_M(B) = \left\{ t \;\middle|\; \begin{array}{l} (s, \perp, \{\}, \bullet, \bullet, t) \in \mathtt{convert}_u(D_0), \\ D_0 = \{\ (s, \perp, b, \bullet, \bullet, \varepsilon) \mid s \in S_0,\ b \in B\ \} \end{array} \right\}$$

where B is the set of parse trees obtained from the parsing phase and D_0 is the initial set of unparsing configurations each consisting of an initial state, no value to verify, a parse tree from B, an empty list of continuations, an empty stack, and an empty output value. (As before, we fix the automaton M and omit any mention of it in the following. Also, each function for unparsing has the subscript $_u$ for distinguishing it from a similar one for parsing.) We then run the $\mathtt{convert}_u$ function defined below and collect the set of the output values from the resulting configurations.

The main traversal function $\mathtt{convert}_u$ is defined by the following.

$$\mathtt{convert}_u(\emptyset) = \emptyset$$

$$\mathtt{convert}_u(D) = \{\ (s, t_\perp, b, l, q, t) \in D' \mid s \in F,\ t_\perp \in \{\epsilon, \perp\}\ \} \cup \mathtt{convert}_u(D'')$$

where

$$D \neq \emptyset$$

$$D' = \mathtt{pms}_u(D)$$

$$D'' = \left\{ \begin{array}{l} (s', t_\perp, b, l, q, t'\texttt{<e>}t\texttt{</>}) \mid \\ \quad (s, t'_\perp, b, ((s', t_\perp, t', \mathtt{e}); l), q, t) \\ \quad\quad \in \mathtt{convert}_u(\mathtt{down}_u(D')) \end{array} \right\}$$

That is, if the given set of configurations is empty, we return an empty set of results. Otherwise, we first process the markers by the $\mathtt{pms}_u$ function defined as

$$\mathtt{pms}_u(D) = \left\{ \mathtt{pm}_u(v_n, (\ldots(\mathtt{pm}_u(v_1, d))\ldots)) \;\middle|\; \begin{array}{l} d = (s, t_\perp, b, l, q, t) \in D, \\ (v_1; \ldots; v_n) = \chi(s) \end{array} \right\}$$

where

$$\begin{array}{lcl} \mathtt{pm}_u(\mathtt{BB}(x_T), (s, \perp, \{x_T \mapsto t'\} \oplus b, l, q, t)) & & \\ \qquad = (s, t', b, l, q, t) & & \\ \mathtt{pm}_u(\mathtt{BE}, (s, \varepsilon, b, l, q, t)) & = & (s, \perp, b, l, q, t) \\ \mathtt{pm}_u(\mathtt{AB}(x_N), (s, \perp, \{x_N \mapsto b\} \oplus b', l, q, t)) & & \\ \qquad = (s, \perp, b, l, (b'; q), t) & & \\ \mathtt{pm}_u(\mathtt{AE}, (s, \perp, \{\}, l, (b; q), t)) & = & (s, \perp, b, l, q, t). \end{array}$$

The unparsing phase processes markers roughly in a way symmetric to the parsing phase. For a $\mathtt{BB}(x_T)$, we remove the first value t' from the list to which x_T is bound in the current parse tree and set the value-to-verify to t'. The actual verification on t' will be done by the $\mathtt{down}_u$ function defined in the next paragraph. For a `BE`, we check the current value-to-verify to be fully consumed (ε) and then set it to the absence ($\perp$). For an $\mathtt{AB}(x_N)$, we remove the first sub-parse-tree b from x_N's binding in the current parse tree, set the current parse tree to b, and save the remaining parse tree to the stack. For an `AE`, we check the current parse tree to be fully consumed ($\{\}$), restore b from the stack, and set the current parse tree to b. Analogously to the parsing phase, the nesting and the non-overlapping constraints guarantee that a $\mathtt{BB}(x_T)$, a `BE`, and an $\mathtt{AB}(x_N)$ can assume the current value-to-verify to be absent. However, a failure may occur in `BE`'s check whether the current value-to-verify is ε and in `AE`'s check whether the current parse tree is $\{\}$, in which cases the function `pm` is not defined; in other words, the configuration being processed is dropped at this point.

In the $\mathtt{convert}_u$ function, after processing markers, we collect the configurations with final states and form a set of the results that can be yielded at this moment. Then, we proceed further for finding other results. For this, we first use the following $\mathtt{down}_u$ function for constructing a set of configurations from the content states of all

possible transitions.

$$\texttt{down}_u(D) =$$
$$\left\{ (s_1, \bot, b, l', q, \varepsilon) \;\middle|\; \begin{array}{l} (s, \bot, b, l, q, t) \in D, \\ (s, \texttt{e}, s_1, s_2) \in \Delta, \\ l' = ((s_2, \bot, t, \texttt{e}); l) \end{array} \right\}$$
$$\cup \left\{ (s_1, t_1, b, l', q, \varepsilon) \;\middle|\; \begin{array}{l} (s, \texttt{<e>}t_1\texttt{</>}t_2, b, l, q, t) \in D, \\ (s, \texttt{e}, s_1, s_2) \in \Delta, \\ l' = ((s_2, t_2, t, \texttt{e}); l) \end{array} \right\}$$

Such configurations come from two clauses, depending on whether the value to verify is present or not. The first clause is for the case where it is absent: for each such configuration and for each transition rule from the current state, we form a new configuration whose state is the content state s_1, whose list of continuations is a new one l' (explained below), and whose current output is an empty sequence (the other components remain the same). In the new list l' of continuations, we save, on top of the old one l, a tuple of the remainder state s_2, the absence of a value to verify, the current parse tree, and the label `e`. The second clause is for the case where the value to verify is present and has the form `<e>`t_1`</>`t_2: we create a similar new configuration except that the value to verify is set to the content value t_1 in the new configuration and that the remainder value t_2 is saved in the head of its continuation list as the value to verify after returning to the parent level.

The $\texttt{convert}_u$ function recursively applies itself to the set of new configurations formed above and obtains a set of resulting configurations for the "content" part. After this, from each resulting configuration, we form a new configuration for obtaining results for the "remainder" part. For this, as already explained, we first remove the head tuple $(s', t_\bot, t', \texttt{e})$ from the continuation list, then set the current state to s' and the value-to-verify to $t_\bot$, and finally append the element `<e>`t`</e>` to the end of the output value t'. With th obtained set of configurations, we invoke again the $\texttt{convert}_u$ function.

An important optimization used in our implementation of the unparsing phase is that, at each call to $\texttt{convert}_u$, we drop each configuration whose parse tree has some (terminal or non-terminal) variable still bound to a non-empty list but with no possibility to be consumed in the subsequent processing. Whether each variable's binding may or may not be consumed after each state can easily be calculated statically either during the automata construction or by a simple static analysis on the constructed automata (our implementation takes the former). Without this optimization, the unparsing phase can easily blow up. A typical situation is when we have a concatenation of several repetitions each of which uses a different non-linear variable. For example, consider the following relation.

```
a[var x]*, b[var y]*, c[var z]*
```

Each time after we consume one of `x`'s values and emit an `a` element, there are two possibilities, one for going back to emit the next `a` element and the other for proceeding to emitting `b` elements. Without the above optimization, we never know that the second possibility is redundant when there still remain `x`'s values, which must be consumed. Therefore, if each variable is bound to n values and there are k repetition patterns, then this would generate $O(n^k)$ possibilities. This optimization is further effective in conjunction with simplification described in the end of Section 4.1. For example, for the pattern

```
(a[var x] | b[var y] | c[var z])*
```

the above optimization by itself does not help avoiding explosion, but it does after simplification, which translates the above pattern to `a[var x]*,b[var y]*,c[var z]*`.

Note, however, that neither the simplification or the optimization eliminates all exponential blow-ups. For example, during unparsing with the pattern

```
(var x as a[String])*, (var x as a[String])*
```

and a parse tree that binds `x` to a list of n `a` elements, up to 2^n configurations may be generated. As in the parsing phase, our experiments in Section 5 support that an actual blow-up rarely happens.

Finally, our algorithm is formalized in a way that returns the set of all possible outputs, but can stop as soon as one output is found. However, this is safe only at the top level where there is no continuation; in a deeper level, each configuration may consume parse trees in different ways and not all of them are necessarily correct—they need to further be checked in the continuation.

Theorem 4.4 (Soundness) If $t \in \texttt{unparse}_M(B)$ under a parsing automaton M, then M parses t to some $b \in B$.

Theorem 4.5 (Completeness) If a parsing automaton M parses a value t to a parse tree b, then $t \in \texttt{unparse}_M(B)$ under M for any B with $b \in B$.

Theorem 4.6 (Termination) The algorithm `unparse` terminates for any input.

The proof of termination relies on the automaton's structural constraint resulting from the simplification procedure.

5. Performance Experiments

We have implemented a prototype system of the biXid language in O'Caml and measured its run-time performance with several application programs written in biXid. The experiments have been done with the O'Caml native-code compiler (3.08.0) and an iBook G4 (1.33GHz) equipped with 512MB memory running Mac OS X (10.4.5). The applications that we have used are the following.

Bookmarks A conversion between the well-known bookmark formats Netscape and XBEL [21]. XBEL has a DTD in 94 lines. Netscape's DTD is not available but should have a similar size since both formats have structures almost in parallel. The conversion program is in 65 lines and only slightly bigger than the simplified version shown in Section 2.1.

Address books A conversion between vCard-XML [18] and ContactXML [7]. These are formats for address books independently defined by different organizations and therefore have structures with various discrepancies (see Appendix B). The DTDs of vCard-XML and ContactXML are each in 390 lines and in 110 lines, and the conversion program is in 175 lines.

BibTeX A conversion between two formats, Gundersen and Hendrikse's BibTeXML (GH-Bib) [11] and Wilde's BibTeXML (W-Bib) [23], for representing BibTeX files in XML. These are defined by different people and thus have quite different structures. GH-Bib (816 lines of DTD) uses separate tags for expressing by its schema different field requirements associated to each kind of bib-entries (`inproceedings` etc.), whereas W-Bib (71 lines of DTD) uses only a single tag with a generic content model, thus hands to the user the responsibility to check such bib-entry-wise requirements. The conversion program is in 680 lines.

For each application, we measured (by using the Unix `time` command) the average of the times spent by 10 runs in either direction of conversion, varying the size of the input document. The results are shown in Figure 3. Since each run includes the static parts of our

algorithm (e.g., automata construction), we can see some overhead taken even for 0-byte inputs. We have not made enough efforts to optimize the static part since these are pretty standard and should be a matter of work. The remaining time is for the dynamic part, for which the figure shows a clear linear-time performance w.r.t. the input size. In Sections 4.2 and 4.3, we have pointed out potential blow-ups, but we do not observe any symptom in the figure.

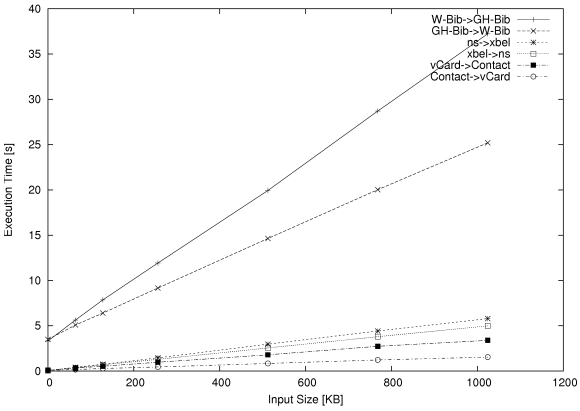


Figure 3. Times spent for the conversion

6. Future Work

This work is only the first step towards a full-fledged language design and implementation and there are a variety of possible future directions. On the design side, a few more useful features are worth investigating. Firstly, as explained in Section 3, we sometimes wish to take a part of data appearing on one side and duplicate it on the other side. This is currently not allowed since its inverse needs to check the structural equality of several separate parts, which would considerably complicate the algorithm but there could be some way to simplify it. Secondly, an important facility that is missing here is shuffle expressions like $P\&Q$, which matches any interleave of a sequence from P and one from Q. Some forms of shuffle expressions are supported by several schema languages like XML-Schema [9] and RELAX NG [5] and more and more formats actually use this feature. Thirdly, the current design has no way to transform a table-like document in a way to swap the rows and the columns. There is a substantial number of applications that would critically use such feature. However, this seems to go beyond our design principle based on finite automata and a separate, special language may be more appropriate.

On the algorithmic side, while the current evaluation algorithm has two phases, parsing and unparsing, we could think of a further improvement that bypasses the construction of a part of parse trees and directly yields the output. Also, while the present work considers only a dynamic aspect, we are eager to investigate static verification, e.g., typechecking.

Acknowledgments

We are most grateful to Makoto Murata, who participated in the start-up of the present project and made a significant influence to the current design of the biXid language. We thank members of the POPL seminar at the University of Tokyo for stimulating discussions. We had lots of valuable comments and suggestions from Zhenjiang Hu, Benjamin Pierce, Claus Brabrand, and anonymous ICFP referees. This work was partly supported by Japan Society for the Promotion of Science.

References

[1] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 51–63, 2003.

[2] R. Bourret. XML data binding resources, 2001. `http://www.rpbourret.com/xml/XMLDataBinding.htm`.

[3] C. Brabrand, A. Møller, and M. I. Schwartzbach. Dual syntax for XML languages. In *Proc. 10th International Workshop on Database Programming Languages, DBPL '05*, volume 3774 of *LNCS*, pages 27–41. Springer-Verlag, August 2005.

[4] J. Clark. XSL Transformations (XSLT), 1999. `http://www.w3.org/TR/xslt`.

[5] J. Clark and M. Murata. RELAX NG, 2001. `http://www.relaxng.org`.

[6] J. Coelho and M. Florido. Type-based XML processing in logic programming. In *PADL*, pages 273–285, 2003.

[7] ContactXML Users Group. ContactXML, 2000. `http://www.contactxml.org/`.

[8] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 2nd Edition*. Benjamin/Cummings, 1994.

[9] D. C. Fallside. XML Schema Part 0: Primer, W3C Recommendation, 2001. `http://www.w3.org/TR/xmlschema-0/`.

[10] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 233–246, 2005.

[11] V. B. Gundersen and Z. W. Hendrikse. BibTeX as XML markup, 2005. `http://bibtexml.sourceforge.net/index.html`.

[12] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[13] H. Hosoya. Regular expression pattern matching — a simpler design. Technical Report 1397, RIMS, Kyoto University, 2003.

[14] H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints. In *Eighth International Conference on Implementation and Application of Automata*, volume 2759 of *Lecture Notes in Computer Science*, pages 201–212. Springer-Verlag, 2003.

[15] H. Hosoya and B. C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–80, Jan. 2001.

[16] H. Hosoya and B. C. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003. Short version appeared in Proceedings of Third International Workshop on the Web and Databases (WebDB2000), volume 1997 of Lecture Notes in Computer Science, pp. 226–244, Springer-Verlag.

[17] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27(1):46–90, 2004. Short version appeared in Proceedings of the International Conference on Functional Programming (ICFP), pp.11-22, 2000.

[18] R. Iannella. Representing vCard objects in RDF/XML, 2001. `http://www.w3.org/TR/vcard-rdf`.

[19] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *APLAS*, pages 2–20, 2004.

[20] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, 2001.

[21] Python XML Special Interest Group. The XML bookmark exchange language, 1998. `http://pyxml.sourceforge.net/topics/xbel/`.

[22] L. Sterling and E. Y. Shapiro. *The Art of Prolog—Advanced Programming Techniques*. MIT Press, 1986.

[23] E. Wilde. BibTeXML (BibTeX markup language), 2004. `http://dret.net/projects/bibtexml/`.

Appendix

A. Translation from Relations to Automata

We take two steps for the translation from a given program to automata: (1) canonicalization of patterns and (2) construction of a parsing automaton. The first step is to collect the patterns appearing on one of the two sides and convert them into a certain form that makes the second step easy.

Canonicalization First of all, we rename non-terminal variables appearing in each relation so that the corresponding ones have the same name, i.e., whenever $r(x_N, x'_N)$ is in the `where`-clause, replace each occurrence of x_N in the LHS pattern with x'_N and rewrite $r(x_N, x'_N)$ itself to $r(x'_N, x'_N)$.

Let us assume a set of *names*, ranged over by X, and define a *grammar* as a mapping from *names* to *expressions*, where expressions E are defined by the following syntax.

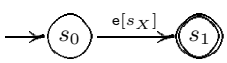
$$E \ ::= () \mid \mathbf{var}\ x\ \mathbf{as}\ E \mid \mathbf{varrel}\ x\ \mathbf{as}\ E \mid \mathtt{e}[E] \mid E, E \mid (E|E) \mid E* \mid X$$

We assume the special name X_{top} to be in the domain of any grammar. The canonical grammar G_c for the LHS of a given program Π (similarly for that for the RHS) is obtained by the following procedure.

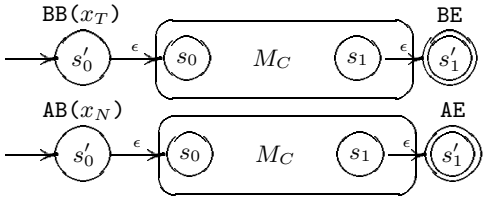
1. Build a grammar G such that $G(X_r) = E$ for each $r \in \mathbf{dom}(\Pi)$ where
$$\Pi(r) = P_L \leftrightarrow P_R\ \mathbf{where}\ W$$
and E is obtained after replacing each $(\mathbf{var}\ x_N)$ appearing in the pattern P_L by $(\mathbf{varrel}\ x_N\ \mathbf{as}\ X_{r'})$ where $r'(x_N, x_N) \in W$.
2. Convert the grammar to another equivalent one in such a way that each expression is canonical, i.e., described by the following syntax.
$$C \ ::= () \mid \mathbf{var}\ x\ \mathbf{as}\ C \mid \mathbf{varrel}\ x\ \mathbf{as}\ C \mid \mathtt{e}[X] \mid C, C \mid (C|C) \mid C*$$
In this form, a use of a name X can appear only as the content of an element and each element must have a name as its content. The conversion is entirely standard and therefore elided here.

Automata construction In this step, for simplicity, we first create a parsing automaton with ϵ-transitions and then eliminate them afterward.

Given a canonical grammar G from the previous step, for each name $X \in \mathbf{dom}(G)$, we first convert the corresponding canonical expression $\Pi(X)$ to a fragment of parsing automaton and then package all the resulting fragments into a single parsing automaton. In the conversion part, we construct an automaton fragment $(s_0^X, s_1^X, \Delta^X, \chi^X)$ where s_0^X is an initial state, s_1^X is a final state, Δ^X is a set of transition rules of the form either $(s, \mathtt{e}, s', s'')$ or (s, s'), and χ is a mapping from states to lists of markers. The construction is done in an inductive manner from inner expressions to outer expressions is mostly standard (cf. [12]) except for element and variable expressions. For an element expression $\mathtt{e}[X]$, we construct the following automaton



with freshly generated states s_0 and s_1. Here, s_X is a special state associated to each name $X \in \mathbf{dom}(X)$ and will be used for later packaging all the constructed automata. A terminal-variable expression $\mathbf{var}\ x_T\ \mathbf{as}\ C$ and a non-terminal variable expression $\mathbf{varrel}\ x_N\ \mathbf{as}\ C$ are each translated to the following



where M_C is the sub-automaton corresponding to the sub-expression C whose initial and final states are s_0 and s_1 and the states s'_0 and s'_1 are freshly generated for the resulting automaton. Except for these variable cases, the marker assignment χ associates each state with an empty list.

Then, we put together all the resulting automaton fragments into a single parsing automaton $(S, S_0, F, \Delta, \chi)$ possibly with ϵ-transitions where

$$S = \bigcup_{X \in \mathbf{dom}(G)} (\{s_0^X, s_1^X\} \cup \{s, s', s'' \mid (s, \mathtt{e}, s', s'') \in \Delta^X\})$$
$$S_0 = \{s_0^{X_{\mathrm{top}}}\}$$
$$F = \bigcup_{X \in \mathbf{dom}(G)} F^X$$
$$\Delta = \bigcup_{X \in \mathbf{dom}(G)} (\Delta^X \cup \{(s_X, s_0^X)\})$$
$$\chi = \bigcup_{X \in \mathbf{dom}(G)} \chi^X.$$

Finally, we obtain the desired parsing automaton $M' = (S', S'_0, F', \Delta', \chi')$ by ϵ-elimination:

$$S' = \left\{ (s_1; \ldots; s_n) \;\middle|\; \begin{array}{l} s_1 \in S, \\ (s_1, s_2) \in \Delta, \ldots, (s_{n-1}, s_n) \in \Delta, \\ i \neq j \Rightarrow s_i \neq s_j \end{array} \right\}$$
$$S'_0 = \{\ (s_1; \ldots; s_n) \mid s_1 \in S_0\ \}$$
$$F' = \{\ (s_1; \ldots; s_n) \mid s_n \in F\ \}$$
$$\Delta' = \left\{ (s_1, \mathtt{e}, s_2, s_3) \;\middle|\; \begin{array}{l} (s_{1l}, \mathtt{e}, s_{21}, s_{31}) \in \Delta, \\ s_1 = (s_{11}; \ldots; s_{1l}) \in S', \\ s_2 = (s_{21}; \ldots; s_{2m}) \in S', \\ s_3 = (s_{31}; \ldots; s_{3n}) \in S' \end{array} \right\}$$
$$\chi' = \{\ (s_1; \ldots; s_n) \mapsto (\chi(s_1); \ldots; \chi(s_n)) \mid (s_1; \ldots; s_n) \in S'\ \}$$

Here, we take, as a state of the result automaton, each list of original states connected by ϵ-transitions. Note that the "non-nullability of repeated patterns" constraint (Section 3.1) ensures that there is no loop by ϵ-transitions and therefore the set of all such lists is finite. Then, we associate each list of states with the list of markers that have been associated to those states.

B. Programming Experiments

In this section, we present an example of mutual conversion between vCard-XML and ContactXML for exposing the practicality of the expressiveness of our language. We here show only a part of the actual program, which is substantially longer to handle the full specifications of these formats.

As already mentioned, both are different formats for address books. A vCard-XML document holds a list of contact information under the root tag `iq`; similarly for a ContactXML document except that the root has tag `ContactXML`.

```
relation top =
  iq[(var x)*] <-> ContactXML[(var y)*]
where
  vcard-contact(x, y)
```

Naturally, each item in `iq` is related to each item in `ContactXML` by the relation `vcard-contact` defined below.

```
relation vcard-contact =
  vCard[var namepart, var body]
<->
  ContactXMLItem[PersonName[var namepart'],
                 var body']
where
  name-part(namepart, namepart'),
  body(body, body')
```

This relates the first half of the content of a `vCard` element to the content of a `PersonName`, and the second half of the `vCard` to the remainder after the `PersonName`. The former relation is described by `name-part` and the latter by `body`. The `name-part` relation is:

```
relation name-part =
  FN[var fullname as String],
  N[FAMILY[var lastname as String],
    GIVEN [var firstname as String],
    MIDDLE[var middlename as String]]
<->
  PersonNameItem[
    FullName[var fullname as String],
    FirstName[var firstname as String],
    MiddleName[var firstname as String],
    LastName[var lastname as String]
  ]
```

The `body` relation describes the main contents. For brevity, we only show parts for addresses, telephones, emails, and images.

```
relation body =
  (var adr | var email | var phone)*
<->
  Address[(var adr')*],
  Phone[(var phone')*],
  Email[(var email')*],
  Image[@..., AnyElem*]
where
  address(adr, adr'),
  email(email, email'),
  phone(phone, phone')
```

First, for image parts, there is no way to represent them on the vCard-XML side and therefore they have to be dropped on the backward transformation. For the other parts, on the vCard-XML side, each piece of information is kept in a separate element `ADR`, `EMAIL`, or `TEL` and all such elements can appear in an unordered and repeated manner. On the ContactXML side, all addresses are packed in a single element `Address` and similarly for the other kinds of information. Therefore expressing a conversion between these parts necessarily involves a data reordering. Our support for non-linear variables can exactly implement this.

In the above, the pattern `@...` allows arbitrary attributes to be present and `AnyElem` matches any element; thus, the forward transformation fills a junk for this part whereas the backward drops it. The relation on emails is given by the following. (The relations for addresses and phones are analogous and so elided here.)

```
relation email =
  EMAIL[
    INTERNET[var pc as ()]?,
    CELL[var cell as ()]?,
    WORK[var official as ()]?,
    HOME[var private as ()]?,
    USERID[var email as String]]
<->
  EmailItem[
    @emailDevice["PC", var pc as ()]?,
    @emailDevice["Cellular", var cell as ()]?,
    @usage["Official", var official as ()]?,
    @usage["Private", var private as ()]?,
    var email as String]
```

Here, the presence of an `INTERNET` element corresponds to the presence of an `emailDevice` attribute with the string content `"PC"`. For this, we use a little trick to represent a one-bit flag by a variable optionally bound to an empty sequence.