BRICS NS-05-6 Castagna & Raghavachari (eds.): PLAN-X 2006 Informal Proceedings

Basic Research in Computer Science

PLAN-X 2006 Informal Proceedings

Charleston, South Carolina, January 14, 2006

Giuseppe Castagna
Mukund Raghavachari
(editors)

BRICS Notes Series **NS-05-6**

ISSN 0909-3206 **December 2005**

See back inner page for a list of recent BRICS Notes Series publications. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `NS/05/6/`

PLAN-X 2006 Informal Proceedings
Charleston, South Carolina
14 January 2006

Invited Talk

Papers

Demos

Program Committee

Gavin Bierman (Microsoft Research)
Giuseppe Castagna (CNRS Ecole Normale Supérieure de Paris), **chair**
Alain Frisch (INRIA Roquencourt)
Giorgio Ghelli (University of Pisa)
Tova Milo (Tel Aviv University)
Makoto Murata (IBM Japan)
Dan Olteanu (Saarland University)
Benjamin Pierce (University of Pennsylvania)
Mukund Raghavachari (IBM T.J. Watson Research Center), **demo chair**
Helmut Seidl (Technische Universität München)

General Chair

Anders Møller (BRICS, University of Aarhus)

Service Interaction Patterns (Invited Talk)

John Evdemon
John.Evdemon@microsoft.com

Abstract

The traditional method for building a service requires a developer to ensure that business logic is not hosted directly within the service itself. While this approach helps make the service more flexible it does not address the biggest architectural gap facing web services today: service interaction patterns (SIPs). A SIP occurs when services engage in concurrent and interrelated interactions with other services. Traditional web service architectures are designed to accommodate simple point-to-point interactions - there is no concept of a logical flow or series of steps from one service to another. Standards such as WS-BPEL are being developed to address this gap. In this session we will discuss a "manifesto" for workflow-enabled solutions, review emerging standards (BPEL, others) and address possible misconceptions regarding these standards.

Statically Typed Document Transformation: An XTATIC Experience

Vladimir Gapeyev
University of Pennsylvania

François Garillot
École Normale Supérieure

Benjamin C. Pierce
University of Pennsylvania

Abstract

XTATIC is a lightweight extension of C♯ with native support for statically typed XML processing. It features XML trees as built-in values, a refined type system based on *regular types* à la XDUCE, and *regular patterns* for investigating and manipulating XML. We describe our experiences using XTATIC in a real-world application: a program for transforming XMLSPEC, a format used for authoring W3C technical reports, into HTML. Our implementation closely follows an existing one written in XSLT, facilitating comparison of the two languages and analysis of the costs and benefits—both significant—of rich static typing for XML-intensive code.

1 Introduction

A profusion of recent language designs, including XDUCE [17, 18, 19], CDUCE [11, 2], XACT [25, 8], XQUERY [4, 10], XJ [15], XOBE [23], and XTATIC [14, 26, 12, 13, 27], are founded on the belief that rich static type systems based on regular tree languages can offer significant benefits for XML-intensive programming. Though attractive, this belief can be questioned on a number of counts. Are familiar XML processing idioms from untyped settings easy to enrich with types, or are there important idioms for which static typing is awkward or unworkable? Is it feasible to reimplement untyped applications in a statically typed language in a "bug-for-bug compatible" fashion? Does the need to please the typechecker lead to too much repetitive boilerplate or too many type annotations? Our aim is to put these questions to the test by a detailed comparison of a non-trivial application originally written in XSLT 1.0 [9] and a faithful reimplementation of the same application in XTATIC.

For this experiment, we chose a task that has also been used as a case study in the standard XSLT reference [20, 21]: translation of structured documents from a high-level document description language, XMLSPEC, into XHTML. XMLSPEC is the format used for authoring official W3C recommendations and drafts. This example is non-trivial but of manageable size: the DTD for XMLSPEC defines 102 elements and 57 type-like entities, while the XHTML DTD defines 89 elements and 65 entities; the XSLT stylesheet implementing the transformation is 770 lines long. Besides styling XMLSPEC elements as HTML, its functions include formatting BNF grammars, section numbering, setting up cross-references, and generating the table of contents. A useful effect of emulating a finished untyped application is that both costs and benefits are visible all at once, rather than arising and being dealt with incrementally, throughout the design and development process. To maximize the opportunities for comparison, our XTATIC implementation closely follows not only the behavior, but also, as far as possible, the structure of the original XSLT implementation.

The contributions of the paper are as follows. First, we draw attention to the XMLSPEC problem itself. This problem offers a good balance of size, complexity, and familiarity, and we hope that it can be re-used by others as a common benchmark for XML processing languages. Second, we present a detailed analysis of the costs and benefits of expressive static types for XML manipulation, both of which were substantial in this application. The main cost is the difficulty of inferring appropriate types for multiple, mutually recursive transformations. The main benefit is the expected one: design flaws in the XMLSPEC DTD—which show up in the XSLT stylesheet as behavioral bugs—are instead exposed as type inconsistencies. Third, we demonstrate that the type system and processing primitives of XTATIC are sufficiently powerful and flexible to fix (or gracefully work around) these bugs without modifying the XMLSPEC DTD. Fixing some of them in the XSLT stylesheet appears more difficult. Finally, reimplementing an existing stylesheet gives us many opportunities for head-to-head comparisons of XSLT and XTATIC, highlighting areas where each shines. In particular, we observe that XTATIC-style regular pattern matching is more natural than XSLT's style—structural recursion augmented with "context probing"—when processing structures, such as the BNF grammar descriptions found in XMLSPEC, where ordering is important. Conversely, XSLT is very convenient for straightforward structural traversions with local transformations, where XTATIC requires a heavier explicit-dispatch control flow. Also, XSLT's data model, which treats the original document as a resource for the computation, is more natural for certain tasks, though we can mimic some of its uses with generic libraries in XTATIC.

Section 2 summarizes XMLSPEC and gives a high-level explanation of the transformation task. Section 3 describes the main challenges of expressing the core XSLT processing model in XTATIC. Section 4 compares the processing of structured data such as BNF grammars in XSLT

and XTATIC. Section 5 describes the auxiliary data structures that our application uses in place of the global document access primitives offered by XSLT. We close with an overview of other evaluations of XML processing languages in Section 6 and some concluding thoughts in Section 7. The paper is intended to be self-contained, but it does not present the motivations or technical details of the XTATIC design in depth; for these, the reader is referred to our earlier papers, especially [14, 13], and to Gapeyev's forthcoming PhD dissertation.

2 The Problem

The history of both XSLT and XMLSPEC goes back to 1998, when the standards for XML and XSLT themselves were still under development. Newer versions of the DTD and the stylesheet (available from the XMLSPEC web page, `http://www.w3.org/2002/xmlspec/`) continue to be used for developing W3C specifications.

Our development is based on the 1998 version of XMLSPEC—the one used for the original XML Recommendation. Our primary reason for using this somewhat dated version was the public availability of the XML sources for the Recommendation, which we used as testing data; more recent W3C specifications, developed with newer versions of XMLSPEC, were only available only in formatted (HTML, PS, PDF) form when we began the project. (Starting in September 2005, XML sources of some newer W3C specification drafts—e.g., XPATH, XSLT, and XQUERY—have again become available.) The XMLSPEC Guide [29] is a useful resource for understanding the XMLSPEC DTD (although it describes a later, slightly more feature-rich version). The original 1998 XSLT stylesheet is described in detail in the 2nd edition of Michael Kay's XSLT reference [20]. Both the DTD and the stylesheet are available from the book's web page (`http://www.wrox.com`).

XMLSPEC is similar to more elaborate XML-based document schemas, such as DOCBOOK (`http://www.docbook.org/`) and the Text Encoding Initiative (`http://www.tei-c.org/`), in that it encodes the "logical" structure of a document so that the same information can be presented in different styles and media. Here, we consider only the task of transforming an XMLSPEC document into a single HTML page, as shown in Figure 1.

Since both XMLSPEC and XHTML are used for document markup, there are many similarities between their DTDs. In both, a valid document file has distinct sections for meta-data and for the content proper. The content has three kinds of markup: top-level, or sectional, for hierarchical document organization; medium-, or paragraph-level, for chunks of actual content; and low-, or phrase-level, for the content flow itself. More interesting for our task, though, are the differences, which stem from differences in purpose between logical and presentation markup: XMLSPEC uses markup to indicate the role of a piece of text in the discussion of a subject matter, while HTML uses markup to instruct a browser how a piece of text should be visually presented to the reader.

For example, the hierarchical document structure is represented explicitly in XMLSPEC by nested sectional elements `div1`, ..., `div4`, while in HTML it is implied by heading elements `h1`, ..., `h6` that interrupt the flow of paragraph-level markup. Both formats include generic paragraph-level elements—for example, enumerated and bulleted lists (`ol` and `ul` in HTML vs `olist` and `ulist` in XMLSPEC) and paragraphs (`p` in both). But XMLSPEC also defines special-purpose variants like `blist`, which is a list containing only bibliographic entries, and `vcnote`—a special kind of paragraph for technical snippets called validity constraint notes. Finally, at the phrase level, where HTML elements like `em`, `i`, `b` decorate the flow of character text with visual emphasis and the anchor element `a` provides simple linking points and links for external resources or locations in the document, their XMLSPEC counterparts play more semantically-loaded roles. For example, a `termdef` element encloses a phrase that defines the meaning of a term (whose occurence in the definition is marked by a `term` element) and can be linked to from other parts of the document by element `termref`. There are many more elements for specific roles, such as language keywords (`kw`), references (`specref`) to other parts of the specification, etc. This semantic specialization of elements allows one to vary independently not only their visual representation, but also additional processing such as creation of indexes and glossaries.

Another category is XMLSPEC elements containing "structured data" of various kinds. The most interesting example is the `scrap` element, which encapsulates BNF rules for grammar productions; its formatting is discussed in detail in Section 4.

Most of the task of an XMLSPEC to HTML transformer is thus a straightforward (often literally one-to-one) mapping from XMLSPEC to HTML element tags. But there are several aspects that are more interesting, including displaying structured data in a readable form, computing section numbers based on the hierarchical positioning of `div` elements, creating a table of contents, with entries hyperlinked to the corresponding sections, and formatting the cross-references occurring in the document so that they properly mention features of the referent, such as title or its computed section number.

3 Structural Recursion

The processing model of XSLT is rather different from the explicit control flow of traditional programming languages, including XTATIC, being based on an *implicit* recursive traversal of the input document. After introducing the XSLT processing model and sketching how an XTATIC application can simulate it explicitly, this section discusses the main challenges of making this implementation strictly typed: (1) structuring its code to accomodate the constraints of typing and (2) fixing the typing bugs inherited from the stylesheet.

3.1 Implicit Structural Recursion in XSLT

An XSLT *stylesheet* is a collection of *templates*, each specifying a computation to be performed on document nodes

```
<body>
<div1 id='sec-intro'> <head>Introduction</head>
<p>XML is an application profile or restricted form of SGML
<bibref ref='ISO8879'/>.</p>

<div2 id='sec-origin-goals'> <head>Origin and Goals</head>
<p>The design goals for XML are:
<olist>
<item><p>XML shall be usable over...</p></item>
<item><p>XML shall support...</p></item>
</olist></p>
</div2>
<div2 id='sec-terminology'> <head>Terminology</head>
<p>The terminology used to describe XML...</p>
</div2></div1>
</body>

<back>
<div1 id='sec-bibliography'> <head>References</head>
<blist>
<bibl id='ISO8879' key='ISO 8879'>
ISO. <emph>ISO 8879:1986(E). Standard Generalized Markup
Language (SGML).</emph> First edition 1986-10-15.
[Geneva]: ISO, 1986. </bibl>
</blist>
</div1>
</back>
```

1 Introduction

XML is an application profile or restricted form of SGML [ISO 8879].

1.1 Origin and Goals

The design goals for XML are:

1. XML shall be usable over...
2. XML shall support...

1.2 Terminology

The terminology used to describe XML...

A References

ISO 8879

ISO. *ISO 8879:1986(E). Standard Generalized Markup Language (SGML).* First edition 1986-10-15. [Geneva]: ISO, 1986.

Figure 1. A sample XMLSPEC document fragment and its rendering via HTML

that satisfy a specified test condition. The execution of a stylesheet proceeds in a single recursive pass over the input document, in document order. For each node encountered during the traversal, the run time system selects the most specific template whose test is satisfied by the node and executes it. Consider, for example, the following template:[1]

```
<xsl:template match="olist">
  <ol> <xsl:apply-templates/> </ol>
</xsl:template>
```

The test (`match="olist"`) says that the template is applicable to XMLSPEC `olist` elements; for each such element, the template produces an HTML `ol` element. The contents of the `ol` are the result of a further recursive traversal of the input: the instruction `<xsl:apply-templates>` designates the location receiving the result of applying the same procedure of selecting and executing an appropriate template, to each child node of the `olist`, in order. Since, according to the XMLSPEC DTD, the only possible children of `olist` are `item` elements, the template that gets invoked on them is

```
<xsl:template match="item">
  <li> <xsl:apply-templates/> </li>
</xsl:template>
```

which similarly constructs an HTML `li` element from each XMLSPEC `item` element. The recursive descent of the traversal terminates either on the document's text nodes, which get copied into the output, or on templates that do not call others via `<xsl:apply-templates/>` or a similar instruction.

[1]The XML-based syntax is a controversial aspect of XSLT. Readers unfamiliar with the language only need to know that elements starting with the `xsl:` prefix are XSLT instructions, while others are literal elements constructing the output.

In general, the test condition in a template's `match` attribute is specified by an *XSLT pattern*, which is written in the downward subset of XPATH. A template is applicable to an element when its pattern *matches* it, i.e., there is an ancestor node, starting from which the pattern (as a path) would select the element. More than one template can be applicable to a document node, but there is always at least one, since XSLT predefines a default template applicable to any element, whose action is to proceed with the traversal without producing any output. In the case of multiple applicable templates, only one of them gets selected for execution according to a set of priority rules whose particulars are not important for this discussion.

The bulk of the XMLSPEC stylesheet consists of templates similar to these, performing simple tag-to-tag transformations—sometimes augmented with other output whose generation depends only on the current element. This processing style, known as *structural recursion* [1, 5, 6], is the backbone of the XSLT processing model. However, since a simple one-pass structural recursion alone would not be sufficient for many applications, XSLT augments it with more features, some of which we will see later.

3.2 Types and Patterns in XTATIC

Before describing our implementation of the formatter, let us pause, briefly, to review the XML types and patterns found in XTATIC.

XTATIC's types are composed from XML element tags using the familiar regular expression operators of concatenation (“,”), alternation (“|”), repetition (“*”), and non-empty repetition (“+”). They can also contain type names, which are bound to their definitions by top-level `regtype` declarations. For example, here is a fragment of the XTATIC type declarations corresponding to the XMLSPEC DTD:

```
regtype s_olist   [[ <olist> s_item+ </> ]]
regtype s_item    [[ <item> s_obj_mix+  </> ]]
regtype s_obj_mix [[ s_p | s_olist | s_ulist |
                     ... ]]
```

We use the prefix `s_` for type names coming from XMLSPEC, and, later, `h_` for names coming from XHTML. The double square brackets are used to separate regular types, patterns, and XML values from surrounding C♯ code. (The ellipsis ... is not part of XTATIC syntax; it just indicates that the definition of `s_obj_mix` is larger than shown.)

The semantics of XML types is similar to that of regular expressions on strings: a type is the set of values described by the type's definition, except that the values are XML document fragments—i.e. sequences of trees built from XML element tags and characters. For example, the values of type `s_item` are single XML elements of the form `<item>...</item>` whose contents are non-empty sequences of elements described by the union type `s_obj_mix`. The predefined type `xml` describes all well-formed XML values. The brackets with no content, `[[]]`, denote the type containing only the empty sequence (when used where a type is expected), as well as the empty sequence value itself (when used where a value is expected).

A regular pattern is a type annotated with variables. For example,

```
[[<olist> s_item first, s_item+ rest </>]]
```

is a pattern with variables `first` and `rest` that will be bound to values of types `s_item` and `s_item+` after a successful match. An XML value matches a pattern when the value belongs to the type obtained by erasing the bound variables. These patterns are the main construct that XTATIC programs use to analyze XML.

3.3 Explicit Structural Recursion in XTATIC

Implementing an *untyped* equivalent of the XMLSPEC stylesheet's behavior in XTATIC is straightforward: it can be written as a collection of mutually recursive static class methods, one per template, plus a dispatcher method that simulates the role of the XSLT run-time system. Figure 2 shows the fragments of this implementation corresponding to the two XSLT templates discussed above.

The two template methods have a similar structure. `TemplateItem`, for example, declares `s_item`, the type of XMLSPEC elements on which it can operate, as its input type; then, relying on the fact that the argument `elt` can only contain an `item` element, it uses a pattern assignment to extract the element's content into the variable `cont`; finally, it builds and returns the resulting HTML `li` element. The contents of `li` come from a call to the method `Dispatch`, which plays the role of the `<xsl:apply-templates>` instruction. Note that the pattern in the assignment follows the definition of the type `s_item`.

The `Dispatch` method uses a combination of C♯ `while` and XTATIC `match` statements to consume the input sequence from the variable `seq` and produce the output sequence in the variable `res`. XTATIC's `match` statement is similar to C♯'s `switch`, but its `case` tests are patterns and therefore can assign fragments of the input to variables for use in the clause's body. The full code of `Dispatch` contains a `case` for each XMLSPEC element, except for elements involved in presenting structured data, which are not covered by the dispatching framework (see Section 4).

3.4 Typing the Recursion

Our goal, however, is to implement a well-typed formatter—i.e., one whose output is, by construction, valid HTML for any valid XMLSPEC input. Therefore, we need to give more precise output types to our methods.

Almost every template method returns a sequence of one or more HTML elements that it creates itself; in these cases, the precise output type for the method can be inferred from its code alone. For example, `TemplateItem` is intended to return values of type `h_li`. Precise template method types induce a precise result type for `Dispatch`, which, instead of `xml`, now yields the union of the result types of all the templates it invokes.

This type, however, is too large. For example, in order for `TemplateOlist` to return a valid `ol` element, the static result type of the recursive call to `Dispatch` at this point must contain only `li` elements. Thus, instead of a single `Dispatch` method, we need to define several dispatchers, each invoking only the subset of template methods suitable for a particular context and therefore ensuring appropriate input and output types. For example, the typed version of `TemplateItem` becomes:

```
static [[h_li]] TemplateItem ([[s_item]] elt){
  [[<item>s_obj_mix+ cont</>]] = elt;
  return [[<li>DispatchInItem(cont)</>]];
}
```

Besides the precise return type and the call to the custom dispatcher `DispatchInItem`, it also analyzes input `elt` by a pattern that strictly follows the definition of type `s_item` and therefore gives the variable `cont` a more precise type, on which `DispatchInItem` can rely.

In general, the dispatcher used by a template must be prepared to handle any input that the template can pass to it, and its output must be acceptable for the use the template has for it. Any collection of dispatchers that satisfy these constraints for all templates would give a type-correct formatter. For a few of the templates, however, it is not possible to compose a well-typed dispatcher from the template methods that would faithfully reproduce the operation of the stylesheet's templates. These are instances of genuine processing bugs in the original XSLT application, which can only be fixed by modifying existing or writing additional template code.

In a few cases, the bugs are caused by subtle incompatibilities between XMLSPEC and HTML that are possible

```
static [[xml]] TemplateOlist ([[s_olist]] elt){
  [[<olist>xml cont</>]] = elt;
  return [[<ol>Dispatch(cont)</>]];
}

static [[xml]] TemplateItem ([[s_item]] elt){
  [[<item>xml cont</>]] = elt;
  return [[<li>Dispatch(cont)</>]];
}
```

```
static [[xml]] Dispatch ([[xml]] seq) {
  [[xml]] res = [[]];
  while (!seq.Equals([[]])) {
    match (seq) {
      case [[s_olist elt, xml rest]]:
        res = [[res, TemplateOlist(elt)]];
        seq = rest;
      case [[s_item elt, xml rest]]:
        res = [[res, TemplateItem(elt)]];
        seq = rest;
      //......     }}
  return res; }
```

Figure 2. A fragment of the untyped structural recursion code in XTATIC.

(though a bit tricky) to smooth out in XTATIC, but apparently not in XSLT, so it is instructive to discuss them and our solutions in some detail.

3.5 Bugs and Fixes

XMLSPEC defines an element `ednote` for recording editorial remarks. The DTD allows `ednote` to appear in both paragraph- and phrase-level contexts, but the XSLT stylesheet contains only one template for `ednote`, which formats it as `blockquote`, a paragraph-level HTML element presented in browsers as an indented paragraph. Clearly, appearances of ednote in phrase-level contexts (e.g., inside `head` elements of section titles) should be formatted differently. To handle this, we implement a second template method for `ednote`, with a phrase-level-friendly return type. A dispatcher that has the `ednote` element in its input type processes it with whichever of the two template methods that is compatible with the dispatcher's return type.

A similar, but trickier, problem arises in the formatting of another phrase-level XMLSPEC element, `quote`. This element is different from most others: rather than creating a new HTML element or two, the corresponding template just surrounds the result of recursively formatting the `quote`'s contents with quotation mark characters. The content type of `quote` is such that it gets transformed into output belonging to the most general HTML phrase-level type, `h_Inline`. One of the elements that can occur inside `h_Inline` is the anchor element `a`, and the content of the latter is described by the subtype `h_a_content` of `h_Inline`, which disallows `a` elements, prohibiting nested anchors. The `quote` element itself, however, can occur in an XMLSPEC context that ends up formatted inside an `a` element, possibly producing a nested anchor. The resolution in XTATIC is similar to the one for `ednote`: we write two template methods for `quote`, both just adding quotation marks, but to the results coming from two different dispatchers.

The solutions for these two problems work because, by explicitly implementing the recursive traversal as a combination of calls to several *distinct* dispatcher methods, our algorithm tracks (static) information about its current context in the input document. In principle, an XSLT stylesheet could also implement processing alternatives for `ednote` and `quote` elements, but making the context-dependent decision of which one of them to invoke would be more difficult. (None of the several possibilities we can see is completely satisfactory. Using more complex path patterns in `match` attributes, such as `div1/ednote` and `head/ednote`, which test for the parent element, would require writing as many templates for `ednote` as there are possible parents—each such template's body duplicating one of the only two handlers. We can write a single template for `ednote` that accesses the parent node and determines its type via a `<xsl:choose>` or a chain of `<xsl:if>` instructions, which again have to list all the possibilities. Other options include use of template modes and template parameters, but these are also quite heavy.)

The typing bug that required the most sophisticated fix in our reimplementation is caused by one of the most straightforward-looking templates in the stylesheet:

```
<xsl:template match="p">
  <p> <xsl:apply-templates/> </p>
</xsl:template>
```

This template transforms the XMLSPEC paragraph element `p` into an HTML element of the same name. The trouble is, an HTML `p` can contain only character data and phrase-level elements, while an XMLSPEC `p` can also contain select paragraph-level elements. Consequently, this template can produce an HTML `p` with paragraph-level elements, such as lists (`ol`, etc.), as children.

The sources of the XML Recommendation actually contain quite a few instances of `p` elements that tickle this bug. Since it affects validity of the generated HTML, the bug was addressed in the later versions of the stylesheet by a hack: when an element like `ol` appears inside a paragraph, the stylesheet adds to the output tree a *text* node whose content is "`</p>`", then formats the `ol`, and then generates another text node whose content is "`<p>`". This does not restore the validity of the in-memory tree produced by the stylesheet, but only of its textual serialization, implying that the stylesheet cannot be used in pipelining scenarios without re-parsing and re-validation of its output. We do not see any natural way to fix this bug in XSLT without changing the XMLSPEC DTD.

Our method `TemplateP` implements the above fix in a fully typed way. It uses a dispatcher that transforms the contents of XMLSPEC `p` into a sequence of text and phrase- and paragraph-level HTML elements, and then processes it to find (with the use of XTATIC patterns)

```
static [[h_block*]] FlowIntoBlocks ([[h_Flow]] flow) {
  [[h_block*]] res = [[]];
  while (!flow.Equals([[]])) {
  match (flow) {
    case [[(pcchar | h_inline | h_misc_inline)+ inl, h_Flow rest]]:
      res = [[res, <p>inl</>]];
      flow = rest;
    case [[h_block+ blocks, h_Flow rest]]:
      res = [[res, blocks]];
      flow = rest;
    case [[(h_form | h_noscript) unexpected, h_Flow rest ]]:
      Error("Unexpected input in FlowIntoBlocks");
      flow = rest;
    case [[]]:Error("empty case");
  }}
  return res;
}
```

Figure 3. The method performing an HTML processing pass to detect implicit paragraphs.

longest subsequences of text and phrase-level elements and wrap them as HTML `p` elements. Figure 3 shows the method that performs the HTML processing pass. The final result of `TemplateP` is paragraph-level content.

From what we have said so far, it might appear that there is another way to implement `TemplateP`, not involving HTML post-processing: we could use patterns to find longest subsequences of XMLSPEC elements and text to be transformed into phrase-level HTML, apply an appropriate dispatcher to them, and wrap the results as `p` elements. In fact, the approach we sketched above is the only one that works, because of another problem—this one caused by XMLSPEC `termdef` elements occurring in the content of `p`. These elements are used to designate boundaries of formal definitions in a specification. As with `quote`, the processing of a `termdef` does not create an HTML element—it just returns an anchor element `a` followed by the sequence resulting from processing the contents. This sequence can contain both phrase- and paragraph-level elements. If `termdef` elements only occurred surrounded by paragraph-level elements, we could implement `TemplateTermdef` like `TemplateP`. However, when an occurrence of `termdef` in `p` is directly preceded by phrase-producing content and the result produced by the `termdef` also starts with phrase-level content, the two must be joined into a single HTML paragraph. Therefore, to avoid creating spurious paragraph breaks, we define `TemplateTermdef` to just return the result of recursive processing of its contents. The latter joins the surrounding HTML and gets processed in `TemplateP` to detect the paragraphs.

Along with these significant typing difficulties, XTATIC's typechecker uncovered several more minor bugs in the stylesheet that also affected validity, but that were easy to fix by small changes to the output.

4 Structured Data

XMLSPEC defines several collections of elements for structured data. This section employs the most sophisticated of these—elements for representing BNF grammars—as an example showing how XTATIC and XSLT handle the challenges of rendering structured data for visual presentation.

[1]	lhsA	::=	rhsA	/*comA*/
[2]	lhsB	::=	rhsB1	/*wfcB1*/
			| rhsB2	
			| rhsB3	/*comB31*/
				/*comB32*/
[3]	lhsC	::=	rhsC	

Figure 4. An HTML table generated from an XMLSPEC grammar

4.1 BNF Productions

A grammar fragment is represented in XMLSPEC as a sequence of production elements `prod`, each having the structure described by the following DTD declaration:

```
<!ELEMENT prod (lhs, (rhs, (com|wfc|vc)*)+)>
```

That is, a production consists of a left-hand side containing exactly one `lhs` element, which introduces the non-terminal defined by the production, and a right-hand side, which defines the unfolding of the non-terminal and consists of a sequence of one or more element groups. Each group contains exactly one `rhs` element, which represents a fragment of the unfolding (usually, an alternative BNF clause), possibly accompanied by side conditions in the form of a comment (`com`), or a reference to a well-formedness (`wfc`) or validity (`vc`) constraint. It is not important to know about the internals of the elements inside `prod`. Each of them gets formatted in the usual way as an HTML fragment to be placed inside a table cell; the layout of this table is our present concern.

Figure 4 shows an example. The generated table has five columns containing, respectively, an automatically generated sequence number for the production, the name of the non-terminal being defined, the symbol `::=`, the frag-

ments of the non-terminal's definition, and the comments and constraints.

The challenge here is assigning appropriate contents to the table's cells based on the relative positioning of various elements in the flat sequence of `prod`'s children, rather than by simply reflecting a nested structure that is already present in the input.

4.2 XTATIC Solution

XTATIC's patterns address this challenge naturally. Note that, in each production, the element `lhs` contributes only to the starting of the first table row corresponding to the production, while the rest of the first row, as well as each of the remaining rows, is generated from a small "chunk" of `prod`'s children containing at most one `rhs` element and at most one `com`, `wfc` or `vc` element. This chunk can be described by the type

```
regtype xs_rhschunk
   [[(s_rhs, xs_constr_mix?) | xs_constr_mix]]
regtype xs_constr_mix
   [[s_com | s_wfc | s_vc]]
```

and, using this type, we can easily write patterns that split the sequence of `prod` children into the chunks necessary for creating the table row-by-row; the full code appears in Figure 5. The method `TemplateProd` starts by extracting from the production the name (`lhs`) of its non-terminal and the first `chunk` of the definition. It uses these to construct the first table row corresponding to the production in the newly created variable `res`. The number placed in the first table cell is extracted, based on the production's identifier (`prodid`), from an index data structure created before processing the document (this process is described in Section 5). The contents of `chunk` is processed by a separate method, `MkRhsChunk`, which performs a straightforward `match` on the two alternatives in the definition of `xs_rhschunk` type and invokes `TemplateRhs` and `DispatchFlow` to process the chunk's elements. The second part of `TemplateProd` is a `foreach` statement that iterates over the `rest` of the production by cutting consecutive chunks off it with the `[[xs_rhschunk chunk]]` pattern, while adding to `res` a new table row for each chunk.

4.3 XSLT Solution

Performing the same computation in XSLT is more difficult. We start with a high-level outline of the stylesheet's structure.

A child element of an instance of `prod` can be classified as a "starter" element if it provides data for the first non-empty cell in the HTML table's row; otherwise as a "follow-up" element. Accordingly, the stylesheet defines two templates for each child element type of `prod`: a "cell" template that just performs formatting inside the HTML table's cell (in other words, a cell template is an ordinary structural recursion template in the sense of Section 3), and a "starter" template that is supposed to be executed only on starter elements, performing, among other things, row padding with empty cells.

Now, the order of template execution on an instance of `prod` is as follows. First, the template for `prod` detects all the starter elements among the `prod`'s children and invokes a type-appropriate starter template on each. The starter template pads the row with empty cells (or, in case of `lhs`, starts a new row, and makes cells with a running sequence number and the `::=` symbol), calls an appropriate cell template on the current element to format its own cell, and finally formats any remaining cells in the row by applying cell templates to the appropriate following siblings of the current element.

This algorithm requires features of XSLT that go beyond structural recursion—the ability to control selection of both templates and nodes during traversal (to invoke either starter or cell templates as appropriate) and to obtain information about the surroundings of the current node. The next few paragraphs review these XSLT features.

The selection of templates to be considered for application when executing `xsl:apply-templates` can be controlled in XSLT by template *modes*. A template's definition can contain (in the start tag of `xsl:template` element) an attribute `mode` specifying the mode of this template. E.g., the "cell" templates in our stylesheet are headed by tags like

```
<xsl:template match="rhs" mode="cell">
```

Then, an `xsl:apply-templates` instruction that also mentions the `mode` attribute, e.g.

```
<xsl:apply-templates mode="cell">
```

considers only the templates marked by the same mode.

To control the selection of nodes to be processed by further traversal, the XSLT `xsl:apply-templates` instruction can be augmented with the attribute `select` specifying the sequence of nodes to be processed next, instead of the default children sequence of the current element. For example, the `prod` template restricts further processing to starter elements only by executing the instruction

```
<xsl:apply-templates
   select="child::*
     [self::lhs
      or (self::rhs
          and not(preceding-sibling::*
                    [1][self::lhs]))
      or ((self::vc or self::wfc or self::com)
           and not(preceding-sibling::*
                    [1][self::rhs]))
     ]"/>
```

The contents of `select` is an XPATH *path expression* that, when applied to a node, produces a sequence (possibly empty) of nodes from the document that are related to the original node as specified by the path.

For our current purposes, we can think of an XPATH path as an expression of the form[2] $a::n[q_1]\ldots[q_k]$ where a

[2]More precisely, the construction described here is a

```
static [[h_tr+]] TemplateProd ([[s_prod]] markup) {
  [[<prod id=prodid> <lhs>pcdata lhs</>, (s_rhs, xs_constr_mix?) chunk,
                    xs_rhschunk* rest </prod>]] = markup;
  [[h_tr+]] res =
    [[ <tr valign='baseline'>
         <td><a name=prodid/>, '[',prodindex.Number(prodid),']'</>,
         <td>lhs</>, <td>'::='</>,  MkRhsChunk(chunk)  </> ]];
  foreach ([[xs_rhschunk chunk]] in rest) {
    res = [[ res, <tr valign='baseline'>
                    <td/>, <td/>, <td/>, MkRhsChunk(chunk) </tr> ]]; }
  return res;
}
static [[h_td, h_td]] MkRhsChunk ([[xs_rhschunk]] chunk) {
  match (chunk) {
    case [[ s_rhs rhs, xs_constr_mix? constrOPT ]]:
      return [[ <td>TemplateRhs(rhs)</>,  <td>DispatchFlow(constrOPT)</>]];
    case [[            xs_constr_mix constr ]]:
      return [[ <td/>,  <td>DispatchFlow(constr)</>]];  }
}
```

Figure 5. BNF production formatting in XTATIC.

is an *axis*, n is a *node test*, and q_i are *predicates*. The execution of a path consists of taking the sequence of nodes specified by the axis a and successively pruning it to contain only the nodes satisfying both the node test n and all the predicates q_i. XPATH predefines several kinds of axes. The ones relevant to our examples are `self`, that produces the single-element sequence consisting of the current node, `child`, that gives the children of the current node, and `preceding-sibling` and `following-sibling` that give the corresponding sibling elements of the current node. The `preceding-sibling` axis produces the nodes in reverse document order, i.e. the closest sibling comes first. A node test n is either an element name (as in, e.g., `self::lhs`), which leaves the node in the result only if the node's name is the same as the test's, or a wildcard `*` (as in `child::*`), which is satisfied by any node. A predicate q can be numeric or boolean. A numeric predicate specifies a 1-based index of the node to be selected from the current sequence. E.g., the path `preceding-sibling::*[1]` selects the closest sibling preceding the current node in the document (or the empty sequence if the current node is the first child of its parent). A boolean predicate is built, using traditional boolean connectives `and`, `or` and `not`, from elementary predicates, which coincide with path expressions. When interpreted as a predicate, a path expression is false when it returns the empty sequence, and is true otherwise.

Taking these explanations into account, one can see why the above `select` expression restricts operation of `xsl:apply-templates` to elements that would start a new row in the HTML table. Technically, the path selects (by `child::*`) all children of `prod` that are (according to the following predicate) either the `lhs` element, or an `rhs` element not immediately preceded by the `lhs`, or a side condition element not immediately preceded by an `rhs`. The templates that get invoked on the elements so selected are starter templates, since they, as well as the `xsl:apply-templates` instruction, do not specify a `mode` attribute. Since `mode` is specified by cell templates corresponding to the same elements, the cell templates are only invoked by instructions at the end of starter templates, like this one in the starter template for `rhs`:

```
<td><xsl:apply-templates  mode="cell"
   select="following-sibling::*
     [1][self::vc or self::wfc or self::com]"/>
</td>
```

More detailed explanation of BNF formatting in the stylesheet can be found in [20, 21].

4.4 Observations

The path expressions from the BNF formatting task shown above are quite complicated—expressions of such complexity rarely appear in document-oriented stylesheets and their occurrences seem to indicate processing of the islands of structured data embedded inside documents. The XPATH fragment needed for handling structured data is more complicated and difficult to master, we believe, than regular patterns, but it can be learned. But even knowing this fragment, the major difficulty for someone trying to understand how BNF formatting works in the XSLT stylesheet comes from the fact that processing of a contiguous piece of data has to be distributed across several non-contiguous pieces of code, connections between which are only loosely indicated. By contrast, the ability of XTATIC's `match` statement to keep together inspection and transformation of a piece of data constituting a logical unit allowed us to write processing methods (Figure 5) whose responsibilities can be clearly specified in terms of their input-output behavior and whose code explicitly indicates dependencies as method calls.

Another small convenience available with regular patterns but not with XPATH paths is the ability to name type fragments and later reference them in patterns. For example, our definition of the type `xs_constr_mix` could have improved clarity of the later patterns, where it is used multiple times, while no similar XPATH shortcut is available for `[self::vc or self::wfc or self::com]`, which is also used several times in the stylesheet.

step expression s, and a general path expression p is either a step s, or an expression of the form p/s.

5 Gathering Global Information

The data model of XSLT is more complex than the one of XTATIC, supporting the notion of a *document* as a container of interconnected nodes and a corresponding assortment of basic operations that take advantage of the richer data model. Several parts of the XMLSPEC stylesheet rely on these additional XSLT features. This section explains how we handled these tasks in XTATIC, sometimes finding a generic reusable solution, other times relying on properties specific to XMLSPEC.

In XTATIC, XML values are lightweight, immutable, shareable trees, which must be inspected in a top-down fashion. By contrase, given a node in XSLT, one can retrieve the root of the document it belongs to, explore the document in any direction—including towards ancestors and siblings—and randomly access nodes that have been marked by special ID attributes, which are specified to be globally unique within a valid document. Supporting all this structure makes run-time representations of XSLT values more heavyweight, but it also provides behind-the-scenes infrastructure for several common document-processing tasks that require information about the document as a whole. These include generation of section numbers, creating the table of contents, and formatting cross-references. An XTATIC version of the XMLSPEC formatter has to handle these tasks by explicitly computing a good deal of information that is automatically provided to a stylesheet by the XSLT run-time system.

The XMLSPEC cross-referencing elements can be classified into three groups, depending on the computational needs of their formatting: "hard-wired," "fetched," and "synthesized."

The XMLSPEC element for a *hard-wired* reference like `<termref def="dt-xml-doc"> XML documents </termref>` contains all the data that needs to appear in its HTML representation, which is `<a href = "#dt-xml-doc"> XML documents </a>`. Such references are straightforward to process both in XSLT and XTATIC.

In a *fetched* reference, data for the HTML presentation must be retrieved from the location in the input document to which the reference points. The elements `wfc` and `vc` (which appeared in Section 4) are fetched elements. For example, the element `<wfc def="NoExternalRefs"/>` points with its `def` attribute to the element

```
<wfcnote id="NoExternalRefs">
  <head>No External Entity References</head>
  <p>Attribute values cannot contain
     external entity references.</p>
</wfcnote>
```

and should be formatted as an HTML anchor

```
<a href="#NoExternalRefs">No External
                          Entity References</a>
```

whose contents are a heading fetched from the `wfcnote` element. The stylesheet obtains the heading with the XSLT instruction

```
<xsl:value-of select="id(@def)/head"/>
```

Here, `@def` is the value of the `wfc`'s `def` attribute, and `id()` is a built-in XSLT function that, given a token, returns the node of the current element that carries a so-called ID attribute with the token as its value. In this example, `id()` returns the above `wfcnote` element, and the following XPATH expression extracts the contents of its `head` child.

To replicate the functionality of `id()` in XTATIC, our formatter explicitly builds an index datastructure that maps IDs to elements. Fortunately, this indexing procedure is completely generic: our implementation is encapsulated in an `IdIndex` class that can be reused in other applications requiring similar ID support. Its usage consists of creating an `IdIndex` object, say `idindex`, at the beginning of processing by passing the document's root element to the `IdIndex` constructor and then using method calls like `idindex.Id(x)` to retrieve elements from the internally maintained index.

A *synthesized* reference is yet more complicated: its HTML formatting contains computed data not directly present in the source document. For example, the element `<specref ref="sec-predefined-ent"/>` uses the ID mechanism discussed above to point to the sectional element that starts as follows:

```
<div2 id="sec-predefined-ent">
  <head>Predefined Entities</head>
```

The HTML formatting of this reference,

```
<a href="#sec-predefined-ent">
        [4.6 Predefined Entities]</a>
```

includes a computed section number.

The XSLT stylesheet computes the section number by fetching the above `div2` element via the `id()` function, and then invoking on it the instruction

```
<xsl:number level="multiple"
      count="inform-div1|div1|div2|div3|div4"
      format="1.1 "/>
```

(which appears to have been specially designed for this purpose!). This instruction uses the specifications in its attributes to produce a formatted number.

To approximate the behavior of this instruction, we create another index, encapsulated by the class `NumberIndex`, that, for each sectional element in the document, maps the element's ID to the section's number. Again, our implementation is re-usable: the index's creation is parameterized by a boolean function that recognizes section-forming elements, and by an object of the `Countkeeper` class that provides an ADT for keeping track of hierarchical section numbers and formatting them as strings. These parameters roughly mimic the above three parameter attributes of `xsl:number`. We use another instance of `NumberIndex` to keep track of the sequence numbers of BNF grammar productions.

The correct operation of `NumberIndex` depends on the existence of a unique identity for each sectional element—something that comes for free from the data model in the XSLT version. We use the `id` attribute, when one is provided, for this purpose. Otherwise (in XMLSPEC, `id` is optional on `div` elements), we create the identity by concatenating the words from the (mandatory) `head` element located under the `div`. In general, this does not guarantee uniqueness. However, the stylesheet uses the same trick to generate hyperlinks from the table of contents to titles in the main body, so we assume it is sufficient for the present application. With more effort, it should be possible to implement the interface of `NumberIndex` so that it mimics the `xsl:number` instruction with better fidelity, but we did not yet pursue this direction.

Besides reference formatting, computed sequence numbers stored in `NumberIndex` objects are used when creating section titles and grammar production entries while formatting the body of the document, as well as during creation of the table of contents. This differs from the stylesheet, which invokes the `xsl:number` instruction anew whenever a number needs to be generated—indeed, a naive XSLT implementation could end up repeating the same computation many times.

The table of contents itself is created by a separate document traversal, after the creation of the indexes but before formatting the document. It is implemented by three nested `foreach` loops, one for each of the three sectional levels (`div1`, `div2`, `div3`) that need to be reflected in the table of contents. This almost literally repeats the code in the stylesheet, which also uses explicit traversal (`xsl:for-each` instructions) for this task.

Each of the tasks discussed in this section (creating the table of contents and the indexes for IDs, section numbers, and production numbers) requires a pass over the document in addition to the main formatting pass. We experimented with combining some of these traversals (e.g., formatting the table of contents concurrently with the body of the specification, or creating all the indexes together), but concluded that increased complexity of the application code did not justify the minor efficiency gains.

Overall, we have been satisfied, in this application, with the ability of the general-purpose facilities of XTATIC (those it inherits from C#) to simulate the whole-document features of XSLT, even without direct support from XTATIC's XML data model.

6 Related Work

Further details about XTATIC can be found in several earlier papers. The core language design is presented in [14], which shows how to integrate the object and tree data models and establishes basic soundness results. A technique for compiling regular patterns based on *matching automata* is described in [26] and extended to include type-based optimization in [28]. The run-time system of XTATIC is described in [12]. A critical evaluation of the main language design choices can be found in [13]. These papers, particularly [13], also offer detailed comparisons between XTATIC and a number of related language designs; we refer the interested reader to these existing discussions, rather than repeating them here.

Most of the recent crop of statically typed XML processing languages have been tested on non-trivial applications, but only rarely have these experiences been recorded in print. A notable exception is the XQUERY Use Cases[7]—a collection of small examples specifically designed to illustrate typical tasks for which XQUERY is expected to be used. Although they were created to illustrate the capabilities of particular features of XQUERY rather than to address a particular large application, they reflect the practical experience of the XQUERY editors and cover a usefully diverse set of small transformation and extraction tasks. In the absence of more practical benchmarks, the XQUERY use cases have been used to demonstrate capabilities of competing technologies, such as XSLT and CQL [3].

Kay's book [20, 21], which suggested the XMLSPEC application for our project, contains two more case studies of substantial XSLT applications: HTML-based browsing of structured genealogical data and an XSLT solution to the classic problem of a knight's tour of the chessboard. A brief overview of typing errors from more than a dozen real-life XSLT stylesheets appears in [30].

7 Conclusions

XSLT and XTATIC are quite different animals. XSLT is a high-level language with processing model founded in structural recursion, path-based XML manipulation primitives, and no static type system. XTATIC extends a general-purpose programming language with a more familiar imperative processing model and processes XML using regular patterns, which are tightly coupled to its static XML type system. The experience described in this paper shows that, like XSLT, XTATIC is well suited for implementing at least some document-processing applications, and that, unlike XSLT, its flexible static XML type system is capable of exposing a range of design and implementation errors and facilitating fixes. We have also observed that, even in this single application, there are some practical programming tasks that are much better served by XSLT than by XTATIC and some where XTATIC is significantly better than XSLT.

Designing a strictly typed structural recursion to match the behavior of an existing untyped implementation turned out to be a surprisingly labor-intensive process. The discussion in Section 3 demonstrates that mimicking the implicit structural recursion of XSLT by explicit recursive code is possible, even while ensuring strict static typing, smoothing architectural incompatibilities of the input and output DTDs, and maintaining a clean program organization that bears a close resemblance to the original XSLT code. Unfortunately, the details of the solution described there required significant effort to discover, over multiple cycles of trial and error. A major difficulty that one faces during type debugging is finding answers to lots of questions about relationships between types from a large collection that a DTD like XMLSPEC or XHTML constitutes. We hope that reading about our

experience could help programmers facing similar typing tasks to find their solutions faster. It is likely, however, that the amount and difficulty of work needed to figure out a correct solution can be intimidating and prohibitive for a typical XML-literate C♯ programmer in a typical project. If worst comes to worst, XTATIC lets one to escape typing quandaries (and postpone discovery of typing problems till run time) by using the generic `xml` type and unsafe casts. Taking these difficulties into account, the refined typing of XTATIC might be considered overkill for one-time-use scripts, where it may be easier to just fix the bugs upon running into their manifestations. On the other hand, the benefits of early error discovery and safety guarantees of well-typed code—compared to the current mainstream technologies where only testing is available—can outweigh the development costs in projects aiming to create reusable document processing tools.

Another conclusion from our experience is that XSLT templates—especially in their simplest form, unburdened by other XSLT features like non-downward paths—are a very convenient approach to programming structural recursion. A template-like construct for implementing *local* structural recursion (i.e., traversals that can be explicitly applied to chosen document fragments as opposed to being a carrier of the whole program's computation) would be a very useful addition to XML processing languages with explicit control flow. It would be necessary, however, for this construct to be accompanied by expressive and flexible typing rules that minimally burden the programmer.

However, having discussed these difficulties with XTATIC, we should also emphasize that XSLT, for its part, turned out to be convoluted (or worse) when faced with the need to deviate from straightforward structural recursion. For the most significant typing bugs discussed in Section 3, we do not see how they could be eliminated from the XSLT stylesheet in a natural and type-safe way without revising the XMLSPEC DTD; also, processing of structured data (Section 4) is much trickier in XSLT.

It would also be interesting to see how the original XMLSPEC stylesheet might be adapted to a statically typed variant of XSLT. Even though XSLT 1.0 [9] is officially untyped, there is now a proposal [30] and implementation of data-flow based typechecking of XSLT stylesheets. The draft of XSLT 2.0 [22] describes only a dynamic type system, leaving possible static variants to the discretion of implementations.

Some of the observations made in this paper in the context of XTATIC and XSLT may be applicable to other XML programming languages. The advantages of regular patterns over paths for processing structured data (Section 4) would also hold in XDUCE and CDUCE, which also use patterns as the primary data inspection mechanism, compared to XQUERY, XJ, XACT, or Cω, which use paths. Any language in the XDuce family will have to mimic the implicit structural recursion of XSLT by explicit traversal via mutually recursive functions, as we did in Section 3. And since obtaining statically typed transformations in these languages requires specifying types of functions, they are likely to experience similar difficulties statically typing this recursion. It is possible, however, that other features found in current descendants of the original XDuce, such as Hosoya's regular pattern filters [16] and CDUCE's overloaded functions with dynamic dispatch, could mitigate some of the difficulties. On the other hand, we suspect that the nominal character of the type systems of XQUERY, XJ, and Cω might complicate the task. XACT citexact2003, by contrast, might have an easier time with typing structural recursion, thanks to its typechecking via data flow analysis, which requires typing specifications only for inputs and outputs of whole programs. (A new paper on XACT, presented at this workshop, introduces optional type annotations [24] with the goal to improve modularity of typechecking; these might interfere with the ease of the task in question.) Programs in all these languages, except XJ, XQUERY, and, possibly, Cω, would need to maintain auxiliary data structures for global document information similar to ours in Section 5, since all of them have chosen light-weight shared tree representations for XML data. These are, of course, only speculations on our part—real observations can only come from implementations of similar applications in these languages.

Acknowledgments

We are grateful to Michael Levin and Alan Schmitt, our collaborators on the XTATIC design and implementation, for numerous discussions about practical programming in XTATIC, and in particular to Alan Schmitt for comments on an earlier draft of the paper. Remarks from the PLAN-X reviewers were very helpful in revising the paper.

Work on XTATIC has been supported by the National Science Foundation under Career grant CCR-9701826 and ITR CCR-0219945, and by gifts from Microsoft.

8 References

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

[2] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-centric general-purpose language. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, pages 51–63, 2003.

[3] V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *Practical Aspects of Declarative Languages (PADL), Long Beach, CA*, volume 3350 of *LNCS*, pages 235–252. Springer, Jan. 2005.

[4] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Siméon. XQuery 1.0: An XML Query Language. Working draft, W3C, Sept. 2005. `http://www.w3.org/TR/xquery/`.

[5] P. Buneman, M. Fernandez, and D. Suciu. UnQL: A Query Language and Algebra for Semistructured Data Based on Structural Recursion. *VLDB Journal*, 9(1):76–110, 2000.

[6] P. Buneman, S. Naqvi, V. Tannen, and L. Wong. Principles of programming with complex objects and collection types. *Theoretical Computer Science*, 149(1):3–48, September 1995.

[7] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query use cases. Working draft, W3C, Sept. 2005. http://www.w3.org/TR/xquery-use-cases/.

[8] A. S. Christensen, C. Kirkegaard, and A. Møller. A runtime system for XML transformations in Java. In Z. Bellahsène, T. Milo, and e. a. Michael Rys, editors, *Database and XML Technologies: International XML Database Symposium (XSym)*, volume 3186 of *Lecture Notes in Computer Science*, pages 143–157. Springer, Aug. 2004.

[9] J. Clark. XSL Transformations (XSLT) Version 1.0. Recommendation, W3C, Nov. 1999. http://www.w3.org/TR/xslt.

[10] D. Draper, P. Fankhauser, M. F. Fernández, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Working draft, W3C, Sept. 2005. http://www.w3.org/TR/xquery-semantics/.

[11] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *IEEE Symposium on Logic in Computer Science (LICS)*, 2002.

[12] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. XML goes native: Run-time representations for Xtatic. In *14th International Conference on Compiler Construction*, Apr. 2005.

[13] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.

[14] V. Gapeyev and B. C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[15] M. Harren, M. Raghavachari, O. Shmueli, M. G. Burke, R. Bordawekar, I. Pechtchanski, and V. Sarkar. XJ: facilitating XML processing in Java. In *International World Wide Web Conference*, pages 278–287, 2005.

[16] H. Hosoya. Regular expression filters for XML. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004.

[17] H. Hosoya and B. C. Pierce. Regular expression pattern matching. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004.

[18] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[19] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(1):46–90, Jan. 2005. Preliminary version in ICFP 2000.

[20] M. Kay. *XSLT Programmer's Reference*. Wrox, 2nd edition, 2003.

[21] M. Kay. *XSLT 2.0 Programmer's Reference*. Wrox, 3rd edition, 2004.

[22] M. Kay. XSL Transformations (XSLT) Version 2.0. Working draft, W3C, Sept. 2005. http://www.w3.org/TR/xslt20.

[23] M. Kempa and V. Linnemann. On XML objects. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2003.

[24] C. Kirkegaard and A. Møller. Type checking with XML Schema in Xact. Technical Report RS-05-31, BRICS, Sept. 2005. Presented at PLAN-X 2006.

[25] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, Mar. 2004.

[26] M. Y. Levin. Compiling regular patterns. In *ACM SIGPLAN International Conference on Functional Programming (ICFP), Uppsala, Sweden*, 2003.

[27] M. Y. Levin. *Run, Xtatic, Run: Efficient Implementation of an Object-Oriented Language with Regular Pattern Matching*. PhD thesis, University of Pennsylvania, 2005.

[28] M. Y. Levin and B. C. Pierce. Typed-based optimization for regular patterns. In *First International Workshop on High Performance XML Processing*, 2004.

[29] E. Maler. Guide to the W3C XML specification ("XMLspec") DTD, version 2.1. Technical report, W3C Consortium, 1998. http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm.

[30] A. Møller, M. O. Olesen, and M. I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, Oct. 2005.

Type Checking with XML Schema in XACT

Christian Kirkegaard and Anders Møller*
†
BRICS
Department of Computer Science
University of Aarhus, Denmark
{ck,amoeller}@brics.dk

Abstract

XACT is an extension of Java for making type-safe XML transformations. Unlike other approaches, XACT provides a programming model based on XML templates and XPath together with a type checker based on data-flow analysis.

We show how to extend the data-flow analysis technique used in the XACT system to support XML Schema as type formalism. The technique is able to model advanced features, such as type derivations and overloaded local element declarations, and also datatypes of attribute values and character data. Moreover, we introduce optional type annotations to improve modularity of the type checking.

The resulting system supports a flexible style of programming XML transformations and provides static guarantees of validity of the generated XML data.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.4 [**Software Verification**]: Validation; I.7.2 [**Document and Text Processing**]: Computing Methodologies

General Terms

Languages, Design, Verification

Keywords

XML, XML Schema, Java, language design, static analysis

1 Introduction

The overall goal of the XACT project is to integrate XML into general-purpose programming languages, in particular Java, such that programming of XML transformations can become easier and safer than with the existing approaches. Specifically, we aim for a system that supports a high-level and flexible programming style, permits an efficient runtime model, and has the ability to statically guarantee validity of generated XML data.

In previous papers, see [15, 14], we have presented the first steps of our proposal for a system that fulfills these requirements. Our language, XACT, is an extension of Java where XML fragments can be manipulated through a notion of *XML templates* using XPath for navigation. Static guarantees of validity are provided by a special data-flow analysis that builds on a lattice structure of *summary graphs*.

The existing XACT system has two significant weaknesses: first, it only supports DTD as schema language, and it is generally agreed that DTD has insufficient expressiveness for modern XML applications; second, the data-flow analysis is a whole-program analysis that has poor modularity properties and hence does not scale well to larger programs. In this paper, we present an approach for attacking these issues.

Contributions

We have previously shown a connection between summary graphs and regular expression types [4, 10]. Also, it is known how regular expression types are related to RELAX NG schemas [7] and how schemas written in XML Schema [22, 2] can be translated into equivalent RELAX NG schemas [12]. We exploit these connections in this paper. Our main contributions are the following:

- We present a translation from XML Schema to summary graphs and an algorithm for validating summary graphs relative to schemas written in XML Schema, all via RELAX NG. This provides the foundation for using XML Schema as type formalism in XACT.
- We introduce optional typing in XACT so that XML template variables can be optionally typed with schema constructs (element names and simple or complex types). We show how this can lead to a validity analysis which is more modular, in the sense that it avoids iterating over the whole program.

Together, these improvements effectively remedy the weaknesses mentioned earlier. Furthermore, the results can be seen as indications of the strength of summary graphs and the use of data-flow analysis for validating XML transformations.

As an additional contribution, we identify a subset of RELAX NG that is sufficient for translation from XML Schema and where language inclusion checking is tractable.

Example

The resulting XACT language can be illustrated by a small toy program that uses the new features. This program converts a list of business cards represented in a special XML language into XHTML, considering only the cards where a phone number is present:

*Supported by the Carlsberg Foundation contract number 04-0080.

†Basic Research in Computer Science (www.brics.dk), funded by the Danish National Research Foundation.

```
import dk.brics.xact.*;
import java.io.*;

public class PhoneList {
  static {
    String[] ns =
      {"b", "http://businesscard.org",
       "h", "http://www.w3.org/1999/xhtml",
       "s", "http://www.w3.org/2001/XMLSchema"};
    XML.setNamespaceMap(ns);
  }

  XML<h:html[s:string TITLE, h:Flow MAIN]> wrapper;

  void setWrapper(String color) {
    wrapper =
      [[<h:html>
          <h:head>
            <h:title><[s:string TITLE]></h:title>
          </h:head>
          <h:body bgcolor={color}>
            <h:h1><[s:string TITLE]></h:h1>
            <[h:Flow MAIN]>
          </h:body>
        </h:html>]];
  }

  XML<h:ul> makeList(XML<b:cardlist> x) {
    XML r = [[<h:ul><[CARDS]></h:ul>]];
    XMLIterator i =
        x.select("//b:card[b:phone]").iterator();
    while (i.hasNext()) {
      XML c = i.next();
      r = r.plug("CARDS",
          [[<h:li>
              <h:b><{c.select("b:name/text()")}></h:b>,
              phone: <{c.select("b:phone/text()")}>
            </h:li>
            <[CARDS]>]]);
    }
    return r;
  }

  XML<h:html> transform(String url) {
    XML cardlist = XML.get(url, "b:cardlist");
    setWrapper("white");
    return wrapper.plug("TITLE", "My Phone List")
                  .plug("MAIN", makeList(cardlist));
  }

  public static void main(String[] args) {
    XML<h:html> x = new PhoneList().transform(args[0]);
    System.out.println(x);
  }
}
```

The general syntax for XML template constants and the meaning of the methods `select`, `plug`, `get`, and various others are described further in Section 2.

In the first part of the program, some global namespace declarations are made. Schemas for these namespaces are supplied externally (the schema for the business card XML language is shown in Section 3). Then a field `wrapper` is defined, holding an XML template that must be an `html` tree, potentially with `TITLE` gaps and `MAIN` gaps, which may occur in place of fragments of type `string` and `Flow`, respectively (all of appropriate namespaces). The method `setWrapper` assigns such an XML template to the `wrapper` field. This template has two gaps named `TITLE` and one named `MAIN`. Additionally, it has one code gap where the value of the `color` parameter is inserted. The method `makeList` iterates through a list of `card` elements that have `phone` children and builds an XHTML list. The method `main` loads in an XML document containing a list of business card, invokes the `setWrapper` method, then constructs a complete XHTML document by plugging values into the `TITLE` and `MAIN` gaps using the `makeList` method, and finally outputs this document.

As an example, the program transforms the input

```
<cardlist xmlns="http://businesscard.org">
  <card>
    <name>John Doe</name>
    <email>john.doe@widget.inc</email>
    <phone>(202) 555-1414</phone>
  </card>
  <card>
    <name>Zacharias Doe</name>
    <email>zach@notmail.com</email>
  </card>
  <card>
    <name>Jack Doe</name>
    <email>jack@mailorder.edu</email>
    <email>jack@geemail.com</email>
    <phone>(202) 456-1414</phone>
  </card>
</cardlist>
```

into an XHTML document that looks as follows:

My Phone List

- **John Doe**, phone: (202) 555-1414
- **Jack Doe**, phone: (202) 456-1414

Note that some XML variables in the program are declared by the type `XML`, which represents all possible XML templates, and others use a more constrained type, such as, the declaration of `wrapper` or the signature of `makeList`. XACT now allows the programmer to combine these two approaches. The static type checker uses data-flow analysis to reason about variables that are declared using the former approach, and it conservatively checks that the annotated types are preserved by execution of the program. For this program, one consequence is that the `makeList` method, whose signature is fully annotated, can be type checked separately, and invocations of this method can be type checked without considering its body. (We discuss fields and side-effects in Section 7.) Also note that the type checker can now reason about XML Schema types rather that being limited to DTD.

Related Work

There are numerous other projects having similar goals as XACT; the paper [19] contains a general survey of different approaches. The ones that are most closely related to ours are XJ [9], Cω [1], and XDuce and its descendants [10]. XACT is notably different in two ways: first, although variants of XML templates are widely used in Web application development frameworks, this paradigm is not supported by other type-safe XML transformation languages, which typically allow only bottom-up XML tree construction; second, the annotation overhead is minimal since schema types are only required at input and output, whereas the others require schema type annotations at all XML variable declarations. We believe that both aspects in many cases makes the XACT programming style more flexible. Furthermore, our data-flow analysis also tracks all Java string operations via a separate analysis [6], which enables XACT to reason about validity of attribute values and character data. (In fact, an additional consequence of the extensions

described here is that our static analyzer can also model computed names of elements and attributes.)

With the extensions proposed in this paper, XACT becomes closer to XJ [9], which also uses XML Schema as type formalism and XPath for navigation. Still, our use of *optional* type annotations avoids a problem that can make the XJ type checker too rigid: with mandatory type annotations at all variable declarations in XJ it is impossible to type check a sequence of operations that temporarily invalidates data. The types that are involved in XML transformations are often exceedingly complicated and difficult to write down, and types for intermediate results often do not correspond to named constructs in preexisting schemas. The benefits of type annotations are that they can serve as documentation in the programs and they can lead to faster type checking. By now supporting optional annotations, XACT gets the best from the two worlds.

Moreover, XJ represents XML data as *mutable* trees, which incurs a need for expensive runtime checks to preserve data validity. In XJ, subtyping is nominal, whereas our approach gives semantic (or structural) subtyping. A discussion of subtyping can be found in [8]. Note that although XML Schema does contain mechanisms for declaring subtyping relationships nominally, the choice of supporting XML Schema as type formalism in XACT does not force us to use nominal subtyping. We use schemas only as notation for defining sets of XML values—the internal structure of the notation being used is irrelevant.

The XDuce language family is based on the notion of regular expression types. As mentioned earlier, a connection between regular expression types and a variant of the summary graphs used in our program analysis is shown in [4]. Also, the formal expressiveness of regular expression types and RELAX NG both correspond to that of regular tree languages. We return to these relations in Sections 5 and 6. As XACT, the XTATIC language [8], which is one of the descendants of XDuce, incorporates XML into an object-oriented language in an immutable style.

The Cω language adds XML support to C♯ by combining structural sequences, unions, and products with objects and simple values. The basic features of XML Schema may be encoded in the type system, however little documentation of this is available. Rather than use full XPath for navigation in XML trees as in XACT, Cω uses a reminiscent notion of generalized member access that is closer to ordinary programming notation.

The paper [18] describes a validity analysis for XSLT transformations, which is also based on summary graphs. The techniques we present here for handling XML Schema as type formalism can be transferred seamlessly to that analysis.

The type annotations we introduce are reminiscent of the notion of programmer–designer contracts proposed in [3]. In both cases, static declarations constrain how XML templates may be combined in the programs.

The paper [20] contains a useful classification of schema languages in terms of categories of tree grammars: DTD corresponds to *local tree grammars* where the content model of an element can only depend on the name of the element; XML Schema corresponds to the larger category of *single-type tree grammars* where elements that are siblings and have the same name must have identical content models; and RELAX NG corresponds to the even more general category of *regular tree grammars*, which is equivalent to tree automata. With our new results, XACT supports single-type tree grammars as type formalism.

Overview

In Sections 2 and 3 we begin by briefly recapitulating the design of XACT and RELAX NG, and we characterize a subset of RELAX NG, called *Restricted RELAX NG*, that we will use as an intermediate language in the program analysis. Then, in Section 4 we introduce a variant of summary graphs. In Sections 5 and 6 we explain how schemas written in XML Schema can be converted into summary graphs via Restricted RELAX NG, how to check validity of summary graphs relative to Restricted RELAX NG schemas, and how these results can be used in XACT to provide static guarantees of XML transformations. In Section 7 we introduce optional typing using XML Schema constructs and discuss the resulting language design. Finally, we present our conclusions in Section 8.

Note that we here report on work in progress, and not all of what we present has yet been implemented and tested in practice so we cannot at this stage present experimental results. Also, the limited space prevents us from going into details of our algorithms and of the systems we build upon—instead, this paper aims to present an informal overview of our ideas.

2 The XACT Programming Language

We begin with a brief overview of the XACT language as it looks before adding our new extensions. In XACT, XML data is represented as *templates*, which are well-formed XML fragments that may contain gaps in place of elements or attribute values. A gap is either a name or a piece of code that evaluates to a string or an XML template. As an example, the following XML template contains four gaps: two named `TITLE`, one named `MAIN`, and one containing the expression `color`:

```
<h:html>
  <h:head>
    <h:title><[TITLE]></h:title>
  </h:head>
  <h:body bgcolor={color}>
    <h:h1><[TITLE]></h:h1>
    <[MAIN]>
  </h:body>
</h:html>
```

The special immutable class `XML` corresponds to the set of all possible XML templates. The central operations on this class are the following:

`constant:` a static method that creates a template from a constant string (the syntax `[[foo]]` is sugar for `XML.constant("foo")` where quotes, whitespace, and gaps have been transformed);

`plug:` inserts a given string or template into all gaps of a given name in this template;

`select:` returns the sub-templates of this template that are selected by a given XPath expression;

`get:` a static method that creates a template from a non-constant string and checks (at runtime) that it is valid relative to a given constant schema type;

`cast:` performs a runtime check of validity of this template relative to a given constant schema type;

`analyze:` instructs the static type checker to verify that this template will always be valid relative to a given schema type when the program runs; and

`toString:` converts this template to its textual representation.

A *schema type* is the name of an element (or, with our extension from DTD to XML Schema, a simple type or a complex type) that is declared in a schema. The *language* of a schema type is defined

as the set of XML documents or document fragments that are valid relative to the schema type. Note that in this version of XACT, before incorporating the extensions suggested in this paper, schema types appear only at `get`, `cast`, and `analyze` operations. In particular, declarations use the general type `XML`.

The primary job of the static type checker is to verify that only valid XML data can occur at program locations marked by `analyze` operations, under the assumption that `get` and `cast` operations always succeed. (It also checks properties of `plug` and `select` operations, which is less relevant here.)

3 Defining a Subset of RELAX NG

A RELAX NG schema [7] is essentially a top-down tree automaton that accepts a set of valid XML trees. It is described by a grammar consisting of recursively defined *patterns* of various kinds, including the following: `element` matches one element with a given name and with contents and attributes described by a sub-pattern; `attribute` similarly matches an attribute; `text` matches any character data or attribute value; `group`, `optional`, `zeroOrMore`, `oneOrMore`, and `choice` correspond to concatenation, zero or one occurrence, zero or more occurrences, one or more occurrences, and union, respectively; `empty` matches the empty sequence of nodes; and `notAllowed` corresponds to the empty language. In addition, the pattern `interleave` matches all possible mergings of the sequences that match its sub-patterns.

Note that attributes are described in the same expressions as the content models. Still, attributes are considered unordered, as always in XML, and syntactic restrictions prevent an attribute name from occurring more than once in any element. Mixing attributes and contents in this way is useful for describing attribute–element constraints.

To ensure regularity, there is an important restriction on recursive pattern definitions: recursion is only allowed if passing through an `element` pattern.

Element and attribute names can be described with *name classes*, which can consist of lists of possible names and wildcards that match all names, potentially restricted to a certain namespace or excluding specific names.

To describe datatypes more precisely than with the `text` pattern, RELAX NG relies on an external language, usually the datatype part of XML Schema. Using the `data` pattern, such datatypes can be referred to, and datatype facets can be constrained by a parameter mechanism.

Furthermore, RELAX NG contains various modularization mechanisms, which we can ignore here. As all other type-safe XML transformation languages, we also ignore `ID` and `IDREF` attributes from DTD and the equivalent compatibility features in RELAX NG.

As mentioned in the introduction, we handle XML Schema via a translation to RELAX NG, thus using RELAX NG as a convenient intermediate language that avoids the many complicated technical details of XML Schema. However, we only use a subset of RELAX NG, which we call *Restricted RELAX NG*, being characterized as follows.

First, we define some terminology that we need. We say that a pattern p *top-level-contains* a pattern q if p and q are identical or p contains q (as a child or further descendant) where contents of `element` and `attribute` patterns are ignored. A *content pattern* is a pattern that top-level contains one or more `element`, `data`, or `text` patterns (or `list` or `value` patterns, which we otherwise ignore here for simplicity). An *attribute list pattern* is a pattern that top-level contains one or more `attribute` patterns.

A Restricted RELAX NG schema satisfies the following syntactic requirements:

[single-type grammar] For every `element` pattern p, any two `element` patterns that are top-level-contained by the child of p and have non-disjoint name classes must have the same (identical) content. (This requirement limits the notation to single-type grammars.)

[attribute context insensitivity] No attribute list pattern can be a `choice` pattern. Also, every `optional` attribute list pattern must have an `attribute` pattern as child. (This requirement prohibits context sensitive `attribute` patterns.)

[interleaved content] Every pattern that has a child that top-level contains an `interleave` content pattern must be a `group` or `element` pattern. Also, a `group` pattern that top-level contains an `interleave` content pattern must have only one content pattern child. (This requirement makes it easier to check inclusion of `interleave` patterns, as explained in Section 6.)

We here consider `ref` patterns as abbreviations of the patterns being referred to. For every `element` and `optional` pattern that has more than one child pattern, we treat the children as implicitly enclosed by a `group` pattern. (Also, all `mixed` patterns are implicitly desugared to `interleave` patterns in the usual way.)

Restricted RELAX NG has two important properties: first, it is sufficient for making an exact and simple embedding of XML Schema; second, it makes the summary graph validation in Section 6 more tractable than using XML Schema directly or supporting full RELAX NG.

The following schema written in XML Schema may be used to describe the input to the example program shown in Section 1:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org"
        elementFormDefault="qualified">

  <element name="cardlist">
    <complexType>
      <sequence>
        <element ref="b:card"
                 minOccurs="0" maxOccurs="unbounded"/>
      </sequence>
    </complexType>
  </element>

  <element name="card" type="b:card_type"/>

  <complexType name="card_type">
    <sequence>
      <element name="name" type="string"/>
      <element name="email" type="string"
               maxOccurs="unbounded"/>
      <element name="phone" type="string"
               minOccurs="0"/>
    </sequence>
  </complexType>

</schema>
```

Assuming `cardlist` as root element name, this can be translated into the following Restricted RELAX NG schema (here using the compact RELAX NG syntax):

```
default namespace = "http://businesscard.org"
```

```
start = element cardlist { card* }
card = element card { card_type }
card_type = element name { xsd:string },
            element email { xsd:string }+,
            element phone { xsd:string }?
```

The translation from XML Schema to Restricted RELAX NG is exact and the size of the output schema is proportional to the size of the input schema. Most XML Schema constructs map directly to RELAX NG, and we will not here explain the details of the translation. However, a few points are worth mentioning.

First, the `all` construct maps to the `interleave` pattern. Because of the limitations on the use of `all` in XML Schema, this does not violate the [interleaved content] requirement.

Second, we can ignore default declarations since we only care about validation and not of normalization of the input—except that we treat an attribute or content model as optionally absent if a default is declared.

Third, wildcards can be converted into name classes. If `processContents` of an element wildcard is set to `skip`, then we make a recursive pattern that matches any XML tree.

Fourth, the most tricky parts of the translation involve type derivations and substitution groups. Assume that an element e has type t and there exists a type t' that is derived by extension from t. In this case, an occurrence of e must match either t or t', and in the latter case e must have a special attribute `xsi:type` with the value t' (in the former case, the attribute is permitted but not required). We handle this situation by encoding the `xsi:type` information in the element name. More precisely, we create a new element pattern whose name is the name of e followed by the string `%`t' and whose content corresponds to the definition of t'. Each reference to e is then replaced by a choice between e and the variants with extended types. The `xsi:nil` feature is handled similarly. Now assume that another element f has type t' and is declared as in the substitution group of e. This means that f elements are permitted in place of e elements. In Restricted RELAX NG, this is expressed simply by replacing all references to e elements by choices of e and f elements. Again, because of limitations on the `all` construct and the substitution group mechanism in XML Schema, this cannot lead to violations of the [single-type grammar] requirement, nor of the general RELAX NG requirement that `interleave` branches must be disjoint.

By the translation to Restricted RELAX NG, a schema type corresponds to a pattern definition:

- a simple type corresponds to a pattern, which we call a *simple-type pattern*, that can only contain the constructs `data`, `choice`, `list`, and `value`;
- a complex type corresponds to a pattern, which we call a *complex-type pattern*, that consists of a group of two subpatterns—one describing a content model and one describing attributes; and
- an element declaration corresponds to an `element` pattern that contains a simple-type pattern or a complex-type pattern.

We use these observations in Section 6.

4 Summary Graphs in Validity Analysis

The static type checker in XACT works in two steps. First, a data-flow analysis of the whole program is performed, using the standard data-flow analysis framework [11] but with a highly specialized lattice structure where abstract values are *summary graphs*. A summary graph is a finite representation of a potentially infinite set of XML templates, much like a schema but tailor-made for use in the program analysis [15]. Second, when the fixed point has been computed, we check that the sets of templates represented by the resulting summary graphs are valid relative to the respective schemas.

To allow a smooth integration of XML Schema as a replacement for DTD, we slightly modify the definition of summary graphs as explained below and change the summary graph validation algorithm accordingly and to work with Restricted RELAX NG (the old algorithm supported DTD via an embedding into DSD2 [17]).

A summary graph, as it is defined in [15], has two parts: one that is set up for the given program and remains fixed during the iterative data-flow analysis, and one that changes monotonically during the analysis.

The fixed part contains finite sets of nodes of various kinds: element nodes ($N_{\mathcal{E}}$), attribute nodes ($N_{\mathcal{A}}$), chardata nodes ($N_{\mathcal{C}}$), and template nodes ($N_{\mathcal{T}}$). These node sets are determined by the use of schemas, template constants, and `XML` operations in the program. The former three sets represent the possible elements, attributes, and chardata sequences that may arise when running the program. The template nodes represent sequences of template gaps, which either occur explicitly in template constants or implicitly due to XML operations or schemas. Additionally, the fixed part specifies a number of maps: *name* assigns a name to each element node and attribute node; $attr : N_{\mathcal{E}} \to 2^{N_{\mathcal{A}}}$ associates attribute nodes with element nodes; $contents : N_{\mathcal{E}} \to N_{\mathcal{T}}$ connects element nodes with descriptions of their contents; and $gaps : N_{\mathcal{T}} \to G^*$ associates a sequence of gap names from a finite set G with each template node.

The changing part of a summary graph consist of:

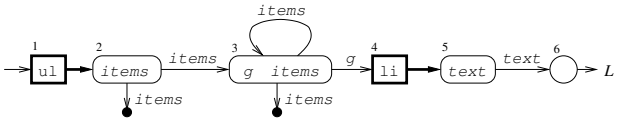
- a set of root nodes $R \subseteq N_{\mathcal{E}} \cup N_{\mathcal{T}}$;
- template edges $T \subseteq N_{\mathcal{T}} \times G \times (N_{\mathcal{T}} \cup N_{\mathcal{E}} \cup N_{\mathcal{C}})$;
- string edges $S : N_{\mathcal{C}} \cup N_{\mathcal{A}} \to REG$ where REG are all regular string languages over the Unicode alphabet; and
- a gap presence map $P : G \to 2^{N_{\mathcal{A}} \cup N_{\mathcal{T}}} \times 2^{N_{\mathcal{A}} \cup N_{\mathcal{T}}} \times \Gamma \times \Gamma$ where $\Gamma = 2^{\{\mathsf{OPEN},\mathsf{CLOSED}\}}$.

The *language* of a summary graph is intuitively the set of XML templates that can be obtained by unfolding it, starting from a root node and plugging elements, templates, and strings into gaps according to the edges. A template edge $(n_1, g, n_2) \in T$ informally means that n_2 may be plugged into the g gaps in n_1, and a string edge $S(n) = L$ means that every string in L may be plugged into the gap in n. The gap presence map, which we will not explain in further detail here, is needed during the data-flow analysis to determine where template gaps and attribute gaps occur. (For the curious reader, this is all formalized in [15].) We also define the language of an individual node n in a summary graph: this is simply the language of the modified summary graph where R is set to $\{n\}$.

As an example (borrowed from [15]), we can define a summary graph whose language is the set of `ul` lists with zero or more `li` items that each contain a string from some language L. Assume that the fixed structure is given by $N_{\mathcal{E}} = \{1,4\}$, $N_{\mathcal{A}} = \emptyset$, $N_{\mathcal{T}} = \{2,3,5\}$ (where all three are sequence nodes), $N_{\mathcal{C}} = \{6\}$, $contents(1) = 2$, $contents(4) = 5$, $attr(1) = attr(4) = \emptyset$, $name(1) = \{\texttt{ul}\}$, $name(4) = \{\texttt{li}\}$, $gaps(2) = \texttt{items}$, $gaps(3) = g \cdot \texttt{items}$, and $gaps(5) = \texttt{text}$. The remaining components are as follows:

$$R = \{1\}$$
$$T = \{(2, \texttt{items}, 3), (3, \texttt{items}, 3), (3, g, 4), (5, \texttt{text}, 6)\}$$
$$S(6) = L$$

(For simplicity, we ignore the gap presence map.) This can be illustrated as follows:



The boxes represent element nodes, rounded boxes are template nodes, the circle is a chardata node, and the dots represent potentially open template gaps.

For a given program, the family of summary graphs forms a finite-height lattice, which is used in the data-flow analysis. To determine the regular string languages used in the string edges, we use a separate program analysis that provides conservative approximations of the possible values of all string expression in the given program [6].

We now introduce two small modifications to the definition of summary graphs:

1. We let the *name* function return a regular set of names, rather than a single name. This will be used to more easily model name classes in Restricted RELAX NG. The definition of unfolding is generalized accordingly: unfolding an element node n yields an element whose name can be any string in $name(n)$, and similarly for attribute nodes. In case an unfolding leads to an element with two attributes of the same name, one of them is chosen arbitrarily and overrides the other.

   To accommodate attribute declarations that have infinite name classes and are repeated using `zeroOrMore` or `oneOrMore`, we define the unfolding of an attribute node n where $name(n)$ is infinite such that it may produce more than one attribute.

2. We distinguish between two kinds of template nodes: *sequence nodes* and *interleave nodes*. The former have the meaning of the old template nodes; the latter will be used to model `interleave` patterns. We define the unfolding of an interleave node as all possible interleavings of the unfoldings of its gaps.

The data-flow transfer functions for operations remain as explained in [15] with only negligible changes as consequence of the modifications of the summary graph definition, the only exceptions being the ones we address in the following section.

Reflecting the [interleaved content] requirement in Restricted RELAX NG, interleave nodes never appear nested within content model descriptions[1]. The translation from Restricted RELAX NG to summary graphs presented in the next section and the transfer functions maintain this property of interleave nodes as an invariant.

With the generalization of the *name* function, we can in fact now easily model computed names of elements and attributes—provided that we add operations for this in the `XML` class, of course, and we leave that to future work.

5 A Translation from Restricted RELAX NG to Summary Graphs

To define the transfer functions for the operations `get` and `cast`, we need an algorithm for translating the given schema type into a summary graph that has the same language. In [15], it is shown how this can be done for DTD schemas; we now present a modified algorithm that supports Restricted RELAX NG and then rely on the translation from XML Schema to Restricted RELAX NG to map from schema types to patterns.

Intuitively, this translation is straightforward: we may simply view summary graphs as a graphical representation of Restricted RELAX NG patterns, provided that we ignore the gap presence component of the summary graphs and the regularity requirement in Restricted RELAX NG. Due to the connection between RELAX NG and regular expression types, this translation can also be seen as a variant of the translation between regular expression types and summary graphs shown in [4].

Given a Restricted RELAX NG pattern, we construct a summary graph fragment as follows:

- First, we observe that name classes and simple-type patterns all define regular string languages[2]. Namespaces are handled by expanding qualified names according to the applicable namespace declarations.

- For an `element` pattern, we exploit the syntactic restrictions described in Section 4. An `element` pattern generally consists of a name class, a content model, and a collection of attribute declarations. Thus, we convert it to an element node e and a template node t with $contents(e) = t$. We define $name(e)$ as the regular string language corresponding to the name class. The attribute declarations are converted recursively into attribute nodes (as explained below), and $attr(e)$ is set accordingly. The content models is converted recursively into a summary graph fragment rooted by t.

- An `attribute` pattern is converted into an attribute node a. We define $name(a)$ in the same way as for `element` patterns, and $S(a)$ is set to the regular string language corresponding to the sub-pattern describing the attribute values. If the attribute is declared as optional using the `optional` pattern, the gap presence map is set to record this (as in [15]).

- For patterns describing content models of elements, the patterns `text`, `group`, `optional`, `zeroOrMore`, `oneOrMore`, `choice`, and `empty` are handled exactly as the equivalent constructs in DTD content model definitions in the way explained in [15]. Intuitively, each pattern corresponds to a tiny summary graph fragment that unfolds to the same language. A `data` pattern becomes a chardata node s where $S(s)$ is the corresponding regular string language. The `interleave` pattern is translated in the same way as `group`, except that an interleave node is used instead of a sequence node.

- Finally, the `notAllowed` pattern can be modeled as a template node t where $gaps(t) = g$ for some gap name g and t has no outgoing template edges.

The set of root nodes R contains the single node that corresponds to the whole pattern being translated. Recursion in pattern definitions simply results in loops in the summary graph. The constructs from RELAX NG that we have omitted in the description in Section 3 can be handled in a similar way as those mentioned here. Note that the translation is exact: the language of the pattern is the same as the language of the resulting summary graph.

As an example, translating the pattern

[1] To state this more precisely, we first define that a node A *top-level-contains* a node B if A and B are identical or B is reachable from A where contents of element nodes and attribute nodes are ignored, and a *content node* is a node that top-level-contains at least one element node or chardata node. We now require the following: every node that has a child that top-level contains an interleave content node must be a sequence or element node, and a sequence node that top-level contains an interleave content node must have only one content node child.

[2] We here ignore a few constraining facets that may be used on the datatypes `float` and `double`. These are uncommon cases that can be accommodated for without losing precision by slightly augmenting the definition of string edges.

```
element ul { element li { xsd:integer }* }
```

results in the summary graph shown in Section 4, assuming that L is the language of strings that match `xsd:integer`.

6 Validating Summary Graphs

When the data-flow analysis has computed a summary graph for each XML expression in the XACT program, we check for each `analyze` operation that the language of its summary graph is included in the language of the specified schema type. If the check fails, appropriate validity warnings are emitted. The entire analysis is sound: if no validity warnings show up, the programmer can be sure that, at runtime, the XML values that appear at the program points marked by `analyze` operations will be valid relative to the given schema types.

The old summary graph analyzer used in XACT is described in [5]. That algorithm, which supports DTD through an embedding into DSD2, as mentioned earlier, has proven successful in practice. We here describe a variant that works with Restricted RELAX NG instead of DSD2.

Given a summary graph node $n \in N_{\Sigma} \cup N_{\mathcal{T}}$ and a Restricted RELAX NG pattern p where p is an `element` pattern, a simple-type pattern, or a complex-type pattern (as defined in Section 3), we wish to determine whether the language of n is included in the language of p.

We begin by considering the case where n is not an interleave node and p is not an `interleave` pattern. First, a context-free grammar C is constructed from the part of the summary graph that is top-level contained by n, considering element and chardata nodes as terminals, template nodes as nonterminals, and ignoring attribute nodes. Each chardata node terminal c is then replaced by a regular grammar equivalent to $S(c)$. If C is not linear, we apply a regular over-approximation [16] (which we also use in [6]). Thus, we have a regular string language L_n over element nodes and Unicode characters that describes the possible unfoldings of n (ignoring attributes). Similarly, p defines a regular string language L_p over `element` patterns and Unicode characters. To obtain a common vocabulary, we now replace each element node n' in L_n by $\langle name(n') \rangle$ (where $\langle$ and $\rangle$ are some otherwise unused characters), and similarly for the `element` patterns in L_p. Then, we check that L_n is included in L_p with standard techniques for regular string languages. (This works because of the restriction to single-type grammars.) If this check fails, a suitable validity error message is generated. Otherwise, for each pair (n', p') of an element node in L_n and an `element` pattern in L_p where $name(n')$ and $name(p')$ are non-disjoint, we perform two checks. First, we check recursively that the language of $contents(n')$ is included in the language of the content model of p'. Second, we check that the attributes of n' match those of p': for each attribute node $a \in attr(n')$, each name $x \in name(a)$, and each value $y \in S(a)$, a corresponding `attribute` pattern must occur in p'—that is, one where x is in the language of its name class and y is in the language of its sub-pattern; also, `attribute` patterns occurring in p' that are not enclosed by `optional` patterns must correspond to one of the non-optional attribute nodes. Again, a suitable validity error message is generated if the check fails.

For interleave nodes and `interleave` patterns, we exploit the restriction on these constructs: they cannot appear nested within content model descriptions. Additionally, in RELAX NG, the sub-patterns of an `interleave` pattern must be disjoint (that is, no element name or `text` pattern occurs in more than one sub-pattern). Thus, if p is an `interleave` pattern, we simply test each sub-pattern in turn, projecting L_n onto the element names occurring in the sub-pattern, and then check that all element names occurring in L_n also occur in one of the sub-patterns. If n is an interleave node, we use a generalized product construction to check inclusion (specifically, the `shuffleSubsetOf` operation in [21]).

To avoid redundant computations (and to ensure termination, in case of loops in the summary graph or recursive definitions in the schema) we apply memoization such that a given pair (n, p) is only processed once. If a loop is detected, we can coinductively assume that the inclusion holds.

With this algorithm, we check for each root node $n \in R$ that its language is included in the language of the pattern corresponding to the given schema type.

As an example of the case with an element node and an `element` pattern, let n be element node 1 in the summary graph from Section 4 and let p be the pattern shown in Section 5:

$p =$ `element ul { element li { xsd:integer }* }`

The context-free grammar for the contents of L_n has the following productions (where N_2 is the start nonterminal and N_4 is the only terminal):

$$
\begin{aligned}
&N_2 \to N_2^{items} \\
&N_2^{items} \to N_3 \mid \varepsilon \\
&N_3 \to N_3^{g}\, N_3^{items} \\
&N_3^{g} \to N_4 \\
&N_3^{items} \to N_3 \mid \varepsilon
\end{aligned}
$$

This grammar is linear, so the regular approximation is not applied. The pattern p contains a single sub-pattern

$p' =$ `element li { xsd:integer }`

and by recursively comparing node 4 and p' we find out that the language of node 4 is included in the language of p'. We now see that $L_n \subseteq L_p$, so we conclude that the language of element node 1 is in fact included in the language of the pattern.

With the exception of the regular approximation of the context-free grammars mentioned above, the inclusion check is exact. Also, since the schemas already define only regular languages, the approximation can only cause a loss of precision if the XML transformation defined by the XACT program introduces non-regularity in the summary graphs, and our experience from [15] and [5] indicate that this rarely results in false errors. In particular, the trivial identity function, which inputs XML data using `get` with some schema type and immediately after applies `analyze` with the same schema type, is guaranteed to type check without warnings for any schema type. Moreover, we could replace the approximation by an algorithm that checks inclusion of a context-free language in a regular language, if full precision is considered more important than performance.

An obvious alternative approach to the algorithm explained above would be to exploit the connection with regular expression types and apply the results from the XDuce project for checking subtyping between general regular expression types [10] or to build on Antimirov's algorithm as in the XOBE project [13]. Our main argument for choosing the algorithm explained above is that it has been shown earlier that this approach is efficient for XACT programs. Also, unlike [23], our algorithm behaves much like existing XML Schema validators, but validating summary graphs instead of individual XML documents. Still, the relation between these different inclusion checking algorithms is worth a further investigation.

As an interesting side-effect of our approach, we get an inclusion checker for Restricted RELAX NG and hence also for XML Schema and DTD: given two schemas, S_1 and S_2, convert S_1 to a summary graph *SG* using the algorithm described in Section 5 and

then apply the algorithm presented above on SG and S_2. (Alternatively, the algorithm presented above could be modified to work directly with Restricted RELAX NG schemas instead of summary graphs.) Preliminary results indicate that our approach is efficient: on a standard PC, our implementation finds in a few seconds the elements in XHTML 1.0 Transitional that are invalid according to XHTML 1.0 Strict (and conversely, it reveals that Strict does not imply Transitional, to our surprise). For schemas that go beyond local tree grammars and use type derivations and `all` model groups, we observe a similarly acceptable performance. Moreover, the validator provides precise error messages in case validation fails.

As an interesting bonus feature, our validator can trivially be extended to precisely check element prohibitions (for example, that `form` elements must not contain `form` elements in XHTML): in XACT, we already have a technique for evaluating XPath location paths on summary graphs, and element prohibitions can be expressed as (simple) XPath location paths.

7 Optional Type Annotations

We will now extend XACT with optional type annotations such that programmers may declare the intended schema types for XML template variables, method parameters, and return values. Besides being useful as in-lined documentation of programmer intentions, type annotations can lead to better modularity properties of the validity analysis.

Every `XML` type may now optionally be annotated in the following way where S and $T_1, \ldots, T_n$ are schema types and $g_1, \ldots, g_n$ are gap names:

$$\texttt{XML<}S\texttt{[}T_1\ \ g_1, \ldots, T_n\ \ g_n\texttt{]>}$$

The semantics of an annotated type is the language described by S under the assumption that every occurrence of gap g_i has been plugged with a value in the language of schema type T_i.

In XML template constants, every template gap must now have the form `<[`T g`]>`, where T is a schema type and g is the gap name. This allows us to, at runtime, tag each gap g in an XML template with a schema type.

In gap annotations in `XML` declarations and template constants, we permit Kleene star of a schema type, T`*`, meaning that the gap can be filled with a sequence of values from the language of T. Kleene star annotations are occasionally needed because we cannot always find existing schema types for sequences of values. As an example, the XML Schema description of XHTML has no named content type describing a sequence of `li` elements. Theoretically, we could permit type annotations to be arbitrary regular expressions over schema types or even small inlined XML Schema fragments, but we have not yet observed the need for this.

Every assignment of an XML template v to a variable x whose type annotation is $t = S\texttt{[}T_1\ \ g_1, \ldots, T_n\ \ g_n\texttt{]}$ must, at runtime, satisfy three constraints:

- All gaps occurring in v must be declared in t.
- For every gap g occurring in v, the language of its type tag must be included in the language of the schema type for g as declared in t.
- The value v must, under the assumption that all gaps were plugged according to their type tags, belong to the language of S.

We put similar constraints on return statements and method invocations, except that for return statements the return value is compared with the declared return type, and for method invocations every actual parameter value is compared with the corresponding declared parameter type. Moreover, every `plug` operation must respect gap tags, that is, the value being plugged in to a gap g must belong to the language of the tag of g.

The following describes a modification of our existing static program analysis to support checking of the extra constraints introduced by annotations.

First, the abstract representation of sets of XML templates is extended to also keep track of the declared schema types of gap names. For a given XACT program, we let $\mathcal{T}$ denote the finite set of all types mentioned by gap annotations in template constants, and we introduce a new summary graph component $D : G \to \mathcal{T}$ mapping gap names to their declared type. The language of a summary graph is not affected by this change.

This leads to extending the data-flow transfer function for the `constant` operation to generate a summary graph with mappings $D(g) = T$ for every gap `<`T g`>` occurring in the given XML template constant. (A simple syntactical check ensures that in each template constant all gaps of the same name are declared with identical schema types.) The transfer function for the `plug` operation simply unions the D mappings of its arguments. (Conflicts are avoided by a check mentioned below.) All other transfer functions act as the identity on the new D component.

To ensure type consistency of variables declared with annotated `XML` types, we must validate all assignments to such variables. We check, using the validation algorithm described in Section 6, that the language of the inferred summary graph for the right-hand side of an assignment is a subset of the language permitted by the schema type annotation. However, this inclusion check is modified to treat gaps as if they were plugged with values corresponding to their declared types. More precisely, for every gap g in the inferred summary graph we apply the algorithm described in Section 5 to construct a summary graph fragment SG_g corresponding to the schema type $D(g)$ and then add template edges from all occurrences of g to the roots of SG_g.

To ensure type consistency of template gaps, we perform an additional check of every x`.plug(`g`,`y`)` operation using the summary graphs SG_x and SG_y inferred by the data-flow analysis for x and y, respectively. First, we check that the language of SG_y is a subset of the language of $D_x(g)$ declared for g in SG_x using the inclusion algorithm presented in Section 6. Then, we check that all gap names h occurring in both SG_x and SG_y are declared with identical types, that is, $D_x(h) = D_y(h)$.

As a product of the guaranteed type consistency of variables declared with annotated `XML` types, reading from a variable can now use the declared type instead of the inferred one. More precisely, for every read from an `XML` typed variable x we normally use an inferred summary graph to describe the set of possible template values at that program point, but now, since all assignments to x have already been checked for validity with respect to the declared schema type for x, we can instead apply the algorithm from Section 5 to obtain the summary graph corresponding to the declared schema type.

Note that the support for type annotations leads to a programming style where the explicit `analyze` operation is rarely needed—instead, one may request a static type check by assigning to an annotated variable. This is the style required in other XML transformation languages.

It is well-known that type annotations in programming languages enable more modular type checking. A component, whose interface is fully annotated, can be type checked independently of its context, and type checking the context can be performed without considering the body of the component. In our setting, this, for example, corresponds to methods where all `XML` typed parameters and return

types are annotated, and further, every non-local assignment and read within the method body involves fields declared with annotated types (the latter to constrain side-effects through field variables). As discussed in Section 1, annotations also have drawbacks, however, in XACT, type annotations are optional. This allows the programmer to mix annotated and unannotated `XML` types to get the best from both worlds.

8 Conclusion

We have presented an approach for generalizing the XACT system to support XML Schema as type formalism and permit optional type annotations. Compared with other programming languages for type-safe XML transformations, type annotations are permitted but not mandatory, which allows the programmer to balance between the pros and cons of type annotations.

The extension to XML Schema takes advantage of connections between XML Schema, RELAX NG, and summary graphs. In particular, it involves a tractable subset of RELAX NG that we use as an intermediate language in the static analysis.

The ideas presented in this paper will become available in the next version of the XACT implementation.

References

[1] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In *Proc. 19th European Conference on Object-Oriented Programming, ECOOP '05*, volume 3586 of *LNCS*. Springer-Verlag, July 2005.

[2] Paul V. Biron and Ashok Malhotra. XML Schema part 2: Datatypes second edition, October 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-2/`.

[3] Henning Böttger, Anders Møller, and Michael I. Schwartzbach. Contracts for cooperation between Web service programmers and HTML designers. *Journal of Web Engineering*, 5(1), 2006.

[4] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Static analysis for dynamic XML. Technical Report RS-02-24, BRICS, May 2002. Presented at Programming Language Technologies for XML, PLAN-X '02.

[5] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level Web service construction. *ACM Transactions on Programming Languages and Systems*, 25(6):814–875, 2003.

[6] Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Precise analysis of string expressions. In *Proc. 10th International Static Analysis Symposium, SAS '03*, volume 2694 of *LNCS*, pages 1–18. Springer-Verlag, June 2003.

[7] James Clark and Makoto Murata. RELAX NG specification, December 2001. OASIS. `http://www.oasis-open.org/committees/relax-ng/`.

[8] Vladimir Gapeyev, Michael Y. Levin, Benjamin C. Pierce, and Alan Schmitt. The Xtatic experience. Technical Report MS-CIS-04-24, University of Pennsylvania, October 2004. Presented at Programming Language Technologies for XML, PLAN-X '05.

[9] Matthew Harren, Mukund Raghavachari, Oded Shmueli, Michael G. Burke, Rajesh Bordawekar, Igor Pechtchanski, and Vivek Sarkar. XJ: Facilitating XML processing in Java. In *Proc. 14th International Conference on World Wide Web, WWW '05*, pages 278–287. ACM, May 2005.

[10] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[11] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977. Springer-Verlag.

[12] Kohsuke Kawaguchi. Sun RELAX NG Converter, April 2003. `http://www.sun.com/software/xml/developers/relaxngconverter/`.

[13] Martin Kempa and Volker Linnemann. Type checking in XOBE. In *Proc. Datenbanksysteme für Business, Technologie und Web, BTW '03*, volume 26 of *LNI*, February 2003.

[14] Christian Kirkegaard, Aske Simon Christensen, and Anders Møller. A runtime system for XML transformations in Java. In *Proc. Second International XML Database Symposium, XSym '04*, volume 3186 of *LNCS*. Springer-Verlag, August 2004.

[15] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[16] Mehryar Mohri and Mark-Jan Nederhof. *Robustness in Language and Speech Technology*, chapter 9: Regular Approximation of Context-Free Grammars through Transformation. Kluwer Academic Publishers, 2001.

[17] Anders Møller. Document Structure Description 2.0, December 2002. BRICS, Department of Computer Science, University of Aarhus, Notes Series NS-02-7. Available from `http://www.brics.dk/DSD/`.

[18] Anders Møller, Mads Østerby Olesen, and Michael I. Schwartzbach. Static validation of XSL Transformations. Technical Report RS-05-32, BRICS, 2005.

[19] Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *Proc. Tenth International Conference on Database Theory, ICDT '05*, volume 3363 of *LNCS*, pages 17–36. Springer-Verlag, January 2005.

[20] Makoto Murata, Dongwon Lee, and Murali Mani. Taxonomy of XML schema languages using formal language theory. In *Proc. Extreme Markup Languages*, August 2001.

[21] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2005. `http://www.brics.dk/automaton/`.

[22] Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. XML Schema part 1: Structures second edition, October 2004. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1/`.

[23] Akihiko Tozawa and Masami Hagiya. XML Schema containment checking based on semi-implicit techniques. In *Proc. 8th International Conference on Implementation and Application of Automata, CIAA '03*, volume 2759 of *LNCS*, July 2003.

PADX : Querying Large-scale Ad Hoc Data with XQuery

Mary Fernández
Kathleen Fisher
AT&T Labs Research
{mff,kfisher}@research.att.com

Robert Gruber*
Google
gruber@google.com

Yitzhak Mandelbaum
Princeton University
yitzhakm@cs.princeton.edu

Name : Use	Representation
Web server logs (CLF): Measure web workloads	Fixed-column ASCII records
AT&T provisioning data: Monitor service activation	Variable-width ASCII records
Call detail: Fraud detection	Fixed-width binary records
AT&T billing data: Monitor billing process	Various Cobol data formats
Netflow: Monitor network performance	Data-dependent number of fixed-width binary records
Newick: Immune system response simulation	Fixed-width ASCII records in tree-shaped hierarchy
Gene Ontology: Gene-gene correlations	Variable-width ASCII records in DAG-shaped hierarchy
CPT codes: Medical diagnoses	Floating point numbers
SnowMed: Medical clinic notes	Keyword tags

Figure 1. Selected ad hoc data sources.

Abstract

This paper describes our experience designing and implementing PADX, a system for querying large-scale ad hoc data sources with XQuery. PADX is the synthesis and extension of two existing systems: PADS and Galax. With PADX, an analyst writes a declarative data description of the physical layout of her ad hoc data, and the PADS compiler produces customizable libraries for parsing the data and for viewing it as XML. The resulting library is linked with an XQuery engine, permitting the analyst to view and query her ad hoc data sources using XQuery.

1 Introduction

Although enormous amounts of data exist in "well-behaved" formats such as XML and relational databases, massive amounts also exist in non-standard or *ad hoc* data formats. Figure 1 gives some sense of the range and pervasiveness of such data. Ad hoc data comes in many forms: ASCII, binary, EBCDIC, and mixed formats. It can be fixed-width, fixed-column, variable-width, or even tree-structured. It is often quite large, including some data sources that generate over a gigabit per second [6]. It frequently comes with incomplete and/or out-of-date documentation, and there are almost always errors in the data. Sometimes these errors are the most interesting aspect of the data, *e.g.*, in log files where errors indicate that something is going wrong in the associated system.

The lack of standard tools for processing ad hoc data forces analysts to roll their own tools, leading to scenarios such as the following. An analyst receives a new ad hoc data source containing potentially interesting information and a list of pressing questions about that data. Could she please provide the answers to the questions as quickly as possible, preferably last week? The accompanying documentation is outdated and missing important information, so she first has to experiment with the data to discover its structure. Eventually, she understands the data well enough to hand-code a parser, usually in C or PERL. Pressed for time, she interleaves code to compute the answers to the supplied questions with the parser. As soon as the answers are computed, she gets a new data source and a new set of questions to answer.

Through her heroic efforts, the data analyst answered the necessary questions, but the approach is deficient in many respects. The analyst's hard-won understanding of the data ended up embedded in a hand-written parser, where it is difficult for others to benefit from her understanding. The parser is likely to be brittle with respect to changes in the input sources. Consider, for example, how tricky it is to figure out which $3's should be $4's in a PERL parser when a new column appears in the data. Errors in the data also pose a significant challenge in hand-coded parsers. If the data analyst thoroughly checks for errors, then the error checking code dominates the parser, making it even more difficult to understand the semantics of the data format. If she is not thorough, then erroneous data can escape undetected, potentially (silently!) corrupting downstream processing. Finally, during the initial data exploration and in answering the specified questions, the analyst had to code *how to compute* the questions rather than being able to express the queries in a declarative fashion. Of course, many of these pitfalls can be avoided with careful design and sufficient time, but such luxuries are not available to the analyst. However, with the appropriate tool support, many aspects of this process can be greatly simplified.

We have two tools, PADS [2, 8] and Galax [1, 7], each of which addresses aspects of the analyst's problem in isolation. The PADS system allows analysts to describe ad hoc data sources declaratively and then generates error-aware parsers and tools for manipulating the sources, including statistical profiling tools. Such support allows the analyst to produce a robust, error-aware parser quickly. The Galax system supports declarative querying of XML via XQuery. If Galax could be applied to ad hoc data, it would allow the analyst first to explore the data and then to produce answers to her questions.

In this work, we strive to integrate PADS and Galax to solve the analyst's data-management problems for the large ad hoc data sources that we have seen in practice. One approach would be to have PADS produce a tool for converting ad hoc data to XML and then ap-

*Work carried out while at AT&T Labs Research.

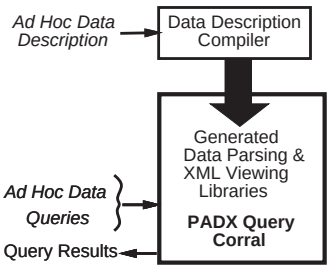


Figure 2. Data analyst's view of PADX

ply Galax to the resulting document. (In fact, PADS provides this ability.) However, the typical factor of eight space blow up in this conversion yields an unacceptable slowdown in performance. Consequently, we chose to design and implement PADX[1], a synthesis and extension of PADS and Galax. Figure 2 depicts PADX from the analyst's perspective. The analyst provides a PADS description of her ad hoc source, which is compiled into a library of components for parsing her data and for viewing and querying it as XML. The resulting libraries are linked together with the PADS and Galax runtime systems into one PADX query executable, called a "query corral.[2]" At query time, the analyst provides her ad hoc data sources and her query written in XQuery, and PADX produces the query's results.

Building PADX presented several problems. The first was semantic: We had to decide how to view ad hoc data as XML and how to express this view as a mapping from the PADS type system to XML Schema, the basis of XQuery's type system. A second problem involved systems design and engineering. Building PADX required evolving PADS and Galax in parallel, modifying the implementation of Galax to support an abstract data model so that Galax could view non-XML sources as XML, and augmenting PADS with the ability to generate concrete instances of this data model. Our solutions to these problems, which were necessary to build a working system, are described in Sections 3 and 4. A third problem involves the scale of data and efficiency of queries, in particular, how to efficiently evaluate complex queries over large sources. Section 5 describes how PADX currently handles large sources and the problems that we face with respect to data scale and query performance.

We begin with a more detailed account of a scenario that illustrates the data management tasks faced by AT&T data analysts and how PADX simplifies these tasks. We then crack open the PADX architecture, first describing PADS and Galax in isolation, and then describing our solutions to the problems described above. We conclude with related work and a discussion of open problems.

1.1 Data-management scenario

In the telecommunications industry, the term *provisioning* refers to the process of converting an order for phone service into the actual service. This process is complex, involving many interactions with other companies. To discover potential problems proactively, the Sirius project tracks AT&T's provisioning process by compiling weekly summaries of the state of certain types of phone service orders. These summaries, which are stored in flat ASCII text files, can contain more than 2.2GB of data per week.

The summaries store the processing date and one record per order. Each order record contains a header followed by a nested sequence of events. The header has 13 pipe separated fields: the order number, AT&T's internal order number, the order version, four different telephone numbers associated with the order, the zip code, a billing identifier, the order type, a measure of the complexity of the order, an unused field, and the source of the order data. Many of these fields are optional, in which case nothing appears between the pipe characters. The billing identifier may not be available at the time of processing, in which case the system generates a unique identifier, and prefixes this value with the string "no_ii" to indicate the number was generated. The event sequence represents the various states a service order goes through; it is represented as a new-line terminated, pipe separated list of state, timestamp pairs. There are over 400 distinct states that an order may go through during provisioning. It may be apparent from this description that English is a poor language for describing data formats!

The analyst's first task is to write a parser for the Sirius data format. Like many ad hoc data sources, Sirius data can contain unexpected or corrupted values, so the parser must handle errors robustly to avoid corrupting the results of analyses. With PADS, the analyst writes a declarative data description of the physical layout of her data. The language also permits the analyst to describe expected semantic properties of her data so that deviations can be flagged as errors. The intent is to allow an analyst to capture in a PADS description all that she knows about a given data source.

Figure 4 gives the PADS description for the Sirius data format. In PADS descriptions, types are declared before they are used, so the type that describes the entire data source, `summary_t`, appears at the bottom of the description (Line 42). In the next section, we use this example to give an overview of the PADS language. Here, we simply note that the data analyst writes this description, and the PADS compiler produces customizable C libraries and tools for parsing, manipulating, and summarizing the data. The fact that useful software artifacts are generated from PADS descriptions provides strong incentive for keeping the descriptions current, allowing them to serve as living documentation.

Analysts working with ad hoc data often want to query their data. Questions posed by the Sirius analyst include "Select all orders starting within a certain time window," "Count the number of orders going through a particular state," and "What is the average time required to go from a particular event state to another particular event state". Such queries are useful for rapid information discovery and for vetting errors and anomalies in data before that data proceeds to a down-stream process or is loaded into a database.

With PADX, the analyst writes declarative XQuery expressions to query her ad hoc data source. Because XQuery is designed to manipulate semi-structured data, its expressiveness matches ad hoc data sources well. As a Turing-complete language, XQuery is powerful enough to express all the questions above. For example, Figure 5 contains an XQuery expression that produces all orders that started in October, 2004. In Section 4, we discuss in more detail why XQuery is an appropriate query language for ad hoc data. One benefit is that XQuery queries may be statically typed, which helps detect common errors at compile time. For example, static typing would raise an error if the path expression in Figure 5 referred to `ordesr` instead of `orders`, or if the analyst erroneously compared the timestamp value in `tstamp` to a string.

[1] Pronounced "paddocks", an enclosed area for exercising race horses.

[2] The equestrian metaphor is intentional: Getting these systems to work together is like corralling race horses!

```
0|15/Oct/2004:18:46:51
9152|9152|1|9735551212|0||9085551212|07988|no_ii152272|EDTF_6|0|APRL1|DUO|10|16/Oct/2004:10:02:10
9153|9153|1|0|0|0|0||152268|LOC_6|0|FRDW1|DUO|LOC_CRTE|1001476800|LOC_OS_10|17/Oct/2004:08:14:21
```

Figure 3. Tiny example of Sirius provisioning data.

```
 1. Precord Pstruct summary_header_t {
 2.  "0|";
 3.  Punixtime tstamp;
 4. };

 5. Pstruct no_ramp_t {
 6.  "no_ii";
 7.  Puint64 id;
 8. };

 9. Punion dib_ramp_t {
10.   Pint64     ramp;
11.   no_ramp_t  genRamp;
12. };

13. Pstruct order_header_t {
14.        Puint32             order_num;
15.  '|'; Puint32             att_order_num;
16.  '|'; Puint32             ord_version;
17.  '|'; Popt pn_t           service_tn;
18.  '|'; Popt pn_t           billing_tn;
19.  '|'; Popt pn_t           nlp_service_tn;
20.  '|'; Popt pn_t           nlp_billing_tn;
21.  '|'; Popt Pzip           zip_code;
22.  '|'; dib_ramp_t          ramp;
23.  '|'; Pstring(:'|':)      order_type;
24.  '|'; Puint32             order_details;
25.  '|'; Pstring(:'|':)      unused;
26.  '|'; Pstring(:'|':)      stream;
27. };

28. Pstruct event_t {
29.        Pstring(:'|':)    state;
30.   '|'; Punixtime         tstamp;
31. };

32. Parray event_seq_t {
33.   event_t[] : Psep('|') && Pterm(Peor);
34. };

35. Precord Pstruct order_t {
36.        order_header_t  order_header;
37.   '|'; event_seq_t     events;
38. };

39. Parray orders_t {
40.   order_t[];
41. };

42. Psource Pstruct summary_t{
43.   summary_header_t  summary_header;
44.   orders_t          orders;
45. };
```

Figure 4. PADS description for Sirius provisioning data.

```
(: Return orders started in October 2004 :)
$pads/Psource/orders/elt[events/elt[1]
  [tstamp/rep >= xs:dateTime("2004-10-01:00:00:00")
and tstamp/rep < xs:dateTime("2004-11-01:00:00:00")]]
```

Figure 5. Query applied to Sirius provisioning data.

2 Using PADS to Access Ad Hoc Data

In this section, we give a brief overview of PADS, focusing on its data description language and the portions of the libraries it generates that are relevant to PADX. More information about PADS is available [2, 8].

2.1 PADS: The language

A PADS specification describes the physical layout and semantic properties of an ad hoc data source. The language provides a type-based model: basic types specify atomic data such as integers, strings, dates, *etc.*, while structured types describe compound data built from simpler pieces. The PADS library provides a collection of useful base types. Examples include 8-bit signed integers (`Pint8`), 32-bit unsigned integers (`Puint32`), IP addresses (`Pip`), dates (`Pdate`), and strings (`Pstring`). By themselves, these base types do not provide sufficient information for parsing because they do not indicate how the data is coded, *i.e.*, in ASCII, EBCDIC, or binary. To resolve this ambiguity, PADS uses the *ambient* coding. By default, the ambient coding is ASCII, but programmers can customize it as appropriate.

To describe more complex data, PADS provides a collection of structured types loosely based on C's type structure. In particular, PADS has **`Pstruct`**s, **`Punion`**s, and **`Parray`**s to describe record-like structures, alternatives, and sequences, respectively. **`Penum`**s describe a fixed collection of literals, while **`Popt`**s provide convenient syntax for optional data. A type may have an associated predicate that determines whether a parsed value is indeed a legal value for the type. For example, a predicate might require that one field of a **`Pstruct`** is bigger than another or that the elements of a sequence are sorted. Programmers can specify such predicates using PADS expressions and functions, written in a C-like syntax. Finally, PADS **`Ptypedef`**s allow programmers to define new types that add further constraints to existing types.

PADS types can be parameterized by values. This mechanism reduces the number of base types and permits the format and properties of later portions of the data to depend upon earlier portions. For example, the base type `Puint16_FW(:3:)` specifies an unsigned two byte integer physically represented by exactly three characters, while the type `Pstring(:'|':)` (*e.g.*, Line 29) describes a string terminated by a vertical bar. Parameters can be used with compound types to specify the size of an array or the appropriate branch of a union.

`Pstruct`s describe ordered sequences of data with unrelated types. In Figure 4, the type declaration for the **`Pstruct`** `order_t` (Lines 35–38) contains an order header (`order_header_t`) followed by the literal character **`'|'`**, followed by an event sequence (`event_seq_t`). PADS supports character, string, and regular expression literals.

`Punion`s describe alternatives in the data format. For example, the `dib_ramp_t` type (Lines 9–12) indicates that the `ramp` field in a Sirius record can be either a `Puint_64` or a string `"no_ii"` followed by a `Puint_64`. During parsing, the branches of a **`Punion`** are tried in order; the first branch that parses without error is taken.

The order_header_t type (Lines 13–27) contains several anonymous uses of the **Popt** type. This type is syntactic sugar for a stylized use of a **Punion** with two branches: the first with the indicated type, and the second with the "void" type, which always matches but never consumes any input.

PADS provides **Parray**s to describe varying-length sequences of data all with the same type. The event_seq_t type (Lines 32–34) uses a **Parray** to characterize the sequence of events an order goes through during processing. This declaration indicates that each element in the sequence has type event_t. It also specifies that the elements will be separated by vertical bars, and that the sequence will be terminated by an end-of-record marker (**Peor**). In general, PADS provides a rich collection of array-termination conditions: reaching a maximum size, finding a terminating literal (including end-of-record and end-of-source), or satisfying a user-supplied predicate over the already-parsed portion of the **Parray**.

Finally, the **Precord** (Line 35) and **Psource** (Line 42) annotations deserve comment. The first indicates that the annotated type constitutes a record, while the second means that the type constitutes the totality of a data source. The notion of a record varies depending upon the data encoding. ASCII data typically uses newline characters to delimit records, binary sources tend to have fixed-width records, while COBOL sources usually store the length of each record before the actual data. PADS supports each of these encodings of records and allows users to define their own encodings.

2.2 PADS: The generated library

From a description, the PADS compiler generates a C library for parsing and manipulating the associated data source. From each type in a PADS description, the compiler generates

- an in-memory representation,
- parsing and printing functions,
- a mask, which allows customization of generated functions, and
- a parse descriptor, which describes syntactic and semantic errors detected during parsing.

To give a feeling for the library that PADS generates, Figure 6 includes a fragment of the generated library for the Sirius event_t declaration.

The C declarations for the in-memory representation (Line 1–4), the mask (Line 5–9), and the parse descriptor (Line 10–17) all share the structure of the PADS type declaration. The mapping to C for each is straightforward: **Pstruct**s map to C structs with appropriately mapped fields, **Punion**s map to tagged unions coded as C structs with a tag field and an embedded union, **Parray**s map to a C struct with a length field and a dynamically allocated sequence, **Penum**s map to C enumerations, **Popt**s to tagged unions, and **Ptypedef**s to C typedefs. Masks include auxiliary fields to control behavior at the level of a structured type, and parse descriptors include fields to record the state of the parse, the number of detected errors, the error code of the first detected error, and the location of that error.

The parsing functions, *e.g.* event_t_read on Line 19, take a mask as an argument and returns an in-memory representation and a parse descriptor. The mask allows the user to specify which constraints the parser should check and which portions of the in-memory representation it should fill in. This control allows the description-writer to specify all known constraints about the data without worrying about the run-time cost of verifying potentially expensive constraints for time-critical applications.

Appropriate error-handling is as important as processing error-free data. The parse descriptor marks which portions of the data contain errors and specifies the detected errors. Depending upon the nature of the errors and the desired application, programmers can take the appropriate action: halt the program, discard parts of the data, or repair the errors. If the mask requests that a data item be verified and set, and if the parse descriptor indicates no error, then the in-memory representation satisfies the semantic constraints on the data.

Because we generate a parsing function for each type in a PADS description, we support multiple-entry point parsing, which accommodates larger-scale data. For a small file, a programmer can call the parsing function for the PADS type that describes the entire file (*e.g.* summary_t_read) to read the whole file with one call. For larger-scale data, programmers can sequence calls to parsing functions that read manageable portions of the file, *e.g.*, reading one record at a time in a loop. The parsing code generated for **Parray**s allows users to choose between reading the entire array at once or reading it one element at a time, again to support parsing and processing very large data sources. We return to the use of multiple-entry point parsing functions in Section 5.

3 Using XQuery and Galax

In this section, we give a brief overview of XML, XQuery, and Galax, focusing on Galax's data-model support for viewing non-XML data as XML. Given the subject of this workshop, we assume the reader is already familiar with XML, XQuery, and XML Schema.

XML [18] is a flexible format that can represent many classes of data: structured documents with large fragments of marked-up text; homogeneous records such as those in relational databases; and heterogeneous records with varied structure and content such as those in ad hoc data sources. XML makes it possible for applications to handle all these classes of data simultaneously and to exchange such data in a standard format. This flexibility has made XML the "lingua franca" of data integration and exchange.

XQuery [20] is a typed, functional query language for XML that supports user-defined functions and modules for structuring large queries. Its type system is based on XML Schema [21]. XQuery contains XPath 2.0 [19] as a proper sub-language, which supports navigation, selection, and extraction of fragments of XML documents. XQuery also includes expressions to construct new XML values and to integrate or join values from multiple documents.

XQuery is a natural choice for querying ad hoc data. Like XML data, ad hoc data is semi-structured, and XQuery is tailored to such data. XQuery's static type system detects type errors at compile time, which is valuable when querying ad hoc sources: Long-running queries on large ad hoc sources do not raise dynamic type errors, and queries made obsolete by schema evolution are identified at compile time. XQuery is also ideal for specifying integrated views of multiple sources. Although here we focus on querying one ad hoc source at a time, XQuery supports simultaneous querying of multiple sources. Lastly, XQuery is practical: It will soon be a standard; numerous manuals already exist [5]; and it is widely

```
 1. typedef struct {     // In-memory representation
 2.   order_header_t order_header;
 3.   event_seq_t    events;
 4. } event_t;

 5. typedef struct {     // Mask
 6.   Pbase_m           compoundLevel;   // Struct-level controls
 7.   order_header_t_m   order_header;
 8.   event_seq_t_m      events;
 9. } event_t_m;

10. typedef struct {     // Parse descriptor
11.   Pflags_t   pstate;       // Normal, partial, or panicking
12.   Puint32    nerr;         // Number of detected errors
13.   PerrCode_t errCode;      // Error code of first detected error
14.   Ploc_t     loc;          // Location of first error
15.   order_header_t_pd order_header;     // Nested header information
16.   event_seq_t_pd    events; // Nested event sequence information
17. } event_t_pd;

18. /* Parsing and printing functions */
19. Perror_t event_t_read     (P_t *pads, event_t_m *m, event_t_pd *pd, event_t *rep);
20. ssize_t  event_t_write2io (P_t *pads, Sfio_t *io,   event_t_pd *pd, event_t *rep);
```

Figure 6. Fragment of the library generated for the `event_t` declaration from Sirius data description.

implemented in commercial databases.

Galax is a complete, extensible, and efficient implementation of XQuery 1.0 that supports XML 1.0 and XML Schema 1.0 and that was designed with database systems research in mind. Its architecture is modular and documented [15], which makes it possible for other researchers to experiment with a complete XQuery implementation. Its compiler produces evaluation plans in the first complete algebra for XQuery [13], which permits experimental comparison of query-compilation techniques. Lastly, its query optimizer produces efficient physical plans that employ traditional and novel join algorithms [13], which makes it possible to apply non-trivial queries to large XML sources. Lastly, its abstract data model permits experimenting with various physical representations of XML and non-XML data sources. Galax's abstract data model is the focus of the the rest of this section.

3.1 Galax's Abstract Data Model

Galax's abstract data model is an object-oriented realization of the XQuery Data Model. The XQuery Data Model [17] contains tree nodes, atomic values, and sequences of nodes and atomic values. A tree node corresponds to an entire XML document or to an individual element, attribute, comment, or processing-instruction. Algebraic operators in a query-evaluation plan produced by Galax's query compiler access documents by applying methods in the data model's object-oriented interface.

Figure 7 contains part of Galax's data model interface[3] for a node in the XQuery Data Model. Node accessors return information such as a node's name (`node_name`), the XML Schema type against which the node was validated (`type`), and the node's atomic-valued data if it was validated against an XML Schema simple type (`typed_value`). The `parent`, `child`, and `attribute` methods navigate the document and return a node sequence containing the respective parent, child, or attribute nodes of the given node.

The first six methods in Figure 7 (Lines 5–11) access the physical representation of a document. Therefore, a concrete instance of the data model must provide their implementations. Galax provides default implementations for the four descendant and ancestor axes (Lines 13–16), which are defined recursively in terms of the child and parent methods. These defaults may be overridden in concrete data models that can provide more efficient implementations than the defaults. For example, some representations permit axes to be implemented by range queries over relational tables [11].

All the axis methods take an optional node-test argument, which is a boolean predicate on the names or types of nodes in the given axis. For example, the XQuery expression `descendant::order` returns nodes in the descendant axis with name `order`. Galax compiles this expression into a single axis/node-test operator that invokes the corresponding methods in the abstract data model, delegating evaluation of node tests to the concrete data model. Some implementations, like PADX, can provide fast access to nodes by their name. We describe PADX's concrete data model in Section 4.

One other important feature of Galax's abstract data model is that sequences are represented by *cursors* (also known as streams), non-functional lists that yield items lazily. Accessing the first item in a sequence does not require that the entire sequence be materialized, *i.e.*, evaluated eagerly. Galax's algebraic operators produce and consume cursors of values, which permits pipelined and short-circuited evaluation of query plans.

In addition to the concrete data model for PADX,which we describe in the next section, Galax has three other concrete data models: a DOM-like representation in main memory and two "shredded" representations, one in main memory and one in secondary storage for very large documents (*e.g.* $>$ 100MB). The shredded data model partitions a document into tables of elements, attributes, and values that can be indexed on node names and values [16].

4 Using PADX to Query Ad Hoc Data

Figure 8 depicts an internal view of the PADX architecture first shown in Figure 2. Pre-existing components (in grey boxes) include the PADS compiler, the Galax query engine, and the PADS runtime system. In this section, we focus on the new components (in white

[3] Galax is implemented in O'Caml, so these signatures are in O'Caml.

```
 1. type sequence = cursor
 2. class virtual node :
 3. object
 4.   (* Selected XQuery Data Model accessors *)
 5.   method virtual node_name   : unit -> atomicQName option
 6.   method virtual type        : unit -> (schema * atomicQName)
 7.   method virtual typed_value : unit -> atomicValue sequence

 8.   (* Required axes *)
 9.   method virtual parent      : node_test option -> node option
10.   method virtual child       : node_test option -> node sequence
11.   method virtual attribute   : node_test option -> node sequence

12.   (* Other axes *)
13.   method descendant_or_self  : node_test option -> node sequence
14.   method descendant          : node_test option -> node sequence
15.   method ancestor_or_self    : node_test option -> node sequence
16.   method ancestor            : node_test option -> node sequence

  ... Other accessors in XQuery Data Model ...
```

Figure 7. Signatures for methods in Galax's abstract node interface

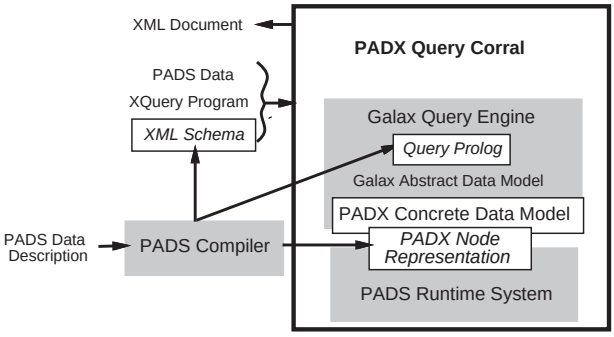


Figure 8. Internal view of PADX Architecture

```
 1. <xs:simpleType name="base_Puint32">
 2.  <xs:restriction base="xs:unsignedInt"/>
 3. </xs:simpleType>
 4. <xs:complexType name="val_Puint32">
 5.  <xs:choice>
 6.   <xs:element name="rep" type="p:base_Puint32"/>
 7.   <xs:element name="pd"  type="p:Pbase_pd"/>
 8.  </xs:choice>
 9. </xs:complexType>
10. <xs:complexType name="Pbase_pd">
11.  <xs:sequence>
12.   <xs:element name="pstate"  type="p:Pflags_t"/>
13.   <xs:element name="errCode" type="p:PerrCode_t"/>
14.   <xs:element name="loc"     type="p:Ploc_t"/>
15.  </xs:sequence>
16. </xs:complexType>
```

Figure 9. Fragment of XML Schema for PADS base types.

boxes) and describe the compiler and run-time support for viewing PADS data as XML. From a PADS description, the compiler generates an XML Schema description that specifies the virtual XML view of the corresponding PADS data, an XQuery prolog that imports the generated schema and that associates the input data with the correct schema type, and a type-specific library that provides the virtual XML view of PADS values necessary to implement PADX's concrete data model.

Note that a query corral is *customized* for a particular PADS description, in particular, its concrete data model only supports views of data sources that match the PADS description. To maintain the correct correspondence between a description, XML Schema, queries, and data, the query corral explicitly contains the generated query prolog, which imports the XML Schema that corresponds to the underlying type-specific library. This guarantees that the user's XQuery program is statically typed, compiled, and optimized with respect to the correct XML Schema and that the underlying data model is an instance of this XML Schema. At runtime, the query corral takes an XQuery program and a PADS data source and produces the query result in XML. We discuss the problem of producing native PADS values in Section 6.

4.1 Viewing PADS data as XML

The mapping from a PADS description to an XML Schema is straight-forward. The interesting aspect of this mapping is that both PADS values that are error free and those containing errors are accessible in the XML view. We begin with the mapping of PADS base types.

A default XML Schema, `pads.xsd`, contains the schema types that represent the PADS base types shared by all PADS descriptions. Figure 9 contains a fragment of this schema. Every PADS base type is mapped to the schema simple type that most closely subsumes the value space of the given PADS base type. For example, the `Puint32` base type maps to the schema type `xs:unsignedInt` (Lines 1–3). Recall that all parsed PADS values have an in-memory representation and a parse descriptor, which records the state of the parse, the error code for detected errors, and the location of those errors. The XML view of a parsed value is a choice of the in-memory representation (`rep`), if no error occurred, or of the parse descriptor (`pd`), if an error occurred (Lines 4–8). This light-weight view exposes the parse descriptor only when an error occurs. The parse-descriptor type for all base types is represented by the schema type `Pbase_pd` (Line 10–14).

The fragment of the XML Schema in Figure 10 corresponds to the description in Figure 4. Note that the schema imports the schema for PADS base types (Line 5). Each compound type is mapped to a complex schema type with a particular content model. A **`Pstruct`** is mapped to a complex type that contains a sequence of local el-

```
 1. <xs:schema targetNamespace="file:/example/sirius.p"
 2.            xmlns="file:/example/sirius.p"
 3.            xmlns:xs="http://www.w3.org/2001/XMLSchema"
 4.            xmlns:p="http://www.padsproj.org/pads.xsd">
 5. <xs:import namespace = "http://www.padsproj.org/pads.xsd".../>
 6. ...
 7. <xs:complexType name="order_header_t">
 8.  <xs:sequence>
 9.   <xs:element name="order_num"     type="p:val_Puint32"/>
10.   <xs:element name="att_order_num" type="p:val_Puint32"/>
11.   <xs:element name="ord_version"   type="p:val_Puint32"/>
12.   <!-- More local element declarations -->
13.   <xs:element name="pd"            type="p:PStruct_pd" minOccurs="0"/>
14.  </xs:sequence>
15. </xs:complexType>
16. <!-- More complex type declarations -->
17. <xs:complexType name="orders_t">
18.  <xs:sequence>
19.   <xs:element name="elt"    type="order_t" maxOccurs="unbounded"/>
20.   <xs:element name="length" type="p:Puint32"/>
21.   <xs:element name="pd"     type="p:Parray_pd" minOccurs="0"/>
22.  </xs:sequence>
23. </xs:complexType>
    ...
24. <xs:element name="Psource" type="summary_t"/>
25. </xs:schema>
```

Figure 10. Fragment of XML Schema for Sirius PADS description.

ements, each of which corresponds to one field in the **Pstruct**. For example, the **Pstruct** order_header_t is mapped to the complex type order_header_t (Lines 7–15), which contains an element declaration for the field order_num, among others. A **Punion** is mapped to a complex type that contains a choice of elements, each of which corresponds to one field in the **Punion**.

Each complex type also includes an optional pd element that corresponds to the type's parse descriptor (Lines 13 and 21). All parse-descriptor types contain the parse state, error code, and location. The parse-descriptor for compound types contain additional information, *e.g.*, Pstruct_pd contains the number of nested errors and Parray_pd contains the index of the array item in which the first error occurred. The pd element is absent if no errors occurred during parsing, but if present, permits an analyst to easily identify the kind and location of errors in the source data. For example, the following XQuery expression returns the locations of all orders that contain at least one error: $pads/Psource/orders/elt/pd/loc.

The schema types for some compound types contain additional fields from the PADS in-memory representation, *e.g.*, arrays have a length (Line 20). Note that Parray types do not associate a name with each individual array item, so in the corresponding schema type, the default element elt encapsulates each array item.

The PADS compiler generates a query prolog that specifies the environment in which all XQuery programs are typed and evaluated. Figure 11 contains the query prolog for the schema in Figure 10. The import schema declaration on Line 1 imports the schema in Figure 10. This declaration puts all global element and type declarations in scope for the query. The variable declaration on Line 2 specifies that the value of the variable $pads is provided externally and that its type is a document whose top-level element is of type Psource, defined on Line 24 in Figure 10. This declaration guarantees that the query is statically typed with respect to the correct input type.

At run time, the user can specify the input data as a command-line argument or by calling the XQuery fn:doc function on a PADS source, *e.g.* pads:/example/sirius.data.

4.2 PADX Concrete Data Model

In Figure 8, the interface between Galax and PADS consists of two modules: the generic PADX concrete data model, which implements the Galax abstract data model, and a compiler-generated module, in which each PADS type has a corresponding, type-specific node representation providing the XML view of values of that type. We note that the generic concrete data model is implemented in O'Caml and the compiler-generated module is implemented in C, but to simplify exposition, we present the compiler-generated module in O'Caml syntax.

Figure 13 contains a fragment of the PADX concrete data model for a node. This object provides a thin wrapper around the type-specific node representation, padx_node_rep, whose interface is in Figure 12. A node representation contains references to a PADS value's in-memory representation and parse descriptor. The node representation interface returns the XML view of the PADS value, including the value's element name, its typed value, and parent. The kth_child and kth_child_by_name methods return all of the PADS value's children in order and those with a given name in order, respectively.

For some methods in Figure 13 (Lines 4–5), the concrete data model simply invokes the corresponding type-specific methods. One exception is the child axis method (Lines 7–17), which we describe in detail as it illustrates how the XML view of a PADS source is materialized lazily. The child method takes an optional name-test argument. We describe the case when the name-test is absent, which corresponds to the common expression child::*. The child method creates a mutable counter k (Line 8), which contains the index of the last child accessed, and a continuation function lazy_child (Lines 11–16), which is invoked each time the child

```
1. import schema default element namespace "file:/example/sirius.p";
2. declare variable $pads as document-node(Psource) external;
```

Figure 11. PADX generated query prolog

```
class virtual padx_node_rep :
  object
    (* Private data includes parsed value's rep & pd *)
    method node_name   : unit -> string
    method typed_value : unit -> item
    method parent      : unit -> padx_node_rep option
    method kth_child   : int -> padx_node_rep option
    method kth_child_by_name : int -> string -> padx_node_rep option
  end
```

Figure 12. The PADX node representation

```
 1. class pads_node (nr : padx_node_rep) =
 2. object
 3.   inherit Galax.node
 4.   method node_name   () = nr#node_name()
 5.   method typed_value () = nr#typed_value()
 6.   (* ... Other data model accessors ... *)
 7.   method child name_test =
 8.     let k = ref 0 in
 9.     match name_test with
10.     | None ->
11.       let lazy_child () =
12.        (incr k;
13.         match nr#kth_child !k with
14.         | Some cnr ->  Some(new pads_node(cnr))
15.         | None -> None)
16.       in Cursor.cursor_of_function lazy_child
17.     | Some (NameTest name) ->
            (* Same as above, but call nr#kth_child_named *)
18.   (* ... Other axes ... *)
19. end
```

Figure 13. Fragment of the PADX concrete data model

cursor is poked. On each invocation, `lazy_child` increments the counter and delegates to the `kth_child` method of the type-specific node representation. For some PADS types, accessing the virtual k^{th} child does not require reading or parsing data, *e.g.*, if the virtual child is part of a complete PADS record. For other PADS types, *e.g.*, **`Parray`**s that contain file records, accessing the virtual k^{th} child may require reading and parsing data. The `kth_child` method provides a uniform interface to all types and delegates the problem of when to read and parse data to the underlying type-specific node representation.

To illustrate type-specific compilation, we give the compiler-generated node representation of an `order_header_t` value in Figure 14. The object takes the name of the field that contains the `order_header_t` value, which corresponds to the XML node name, and the in-memory representation and parse descriptor of the value. The `kth_child` method (Lines 9–15) takes an index and returns the node representation of the field at that index. For example, the first child (Line 11) corresponds to the field `order_num`, which contains a **`Puint32`** value. The `kth_child_by_name` method (Lines 16–21) provides constant-time lookup of a child with a particular name: It looks up the index of the name in the associative map `name_map` and then delegates to `kth_child`. Note that this XML view of an `order_header_t` value corresponds to the schema type `order_header_t` in Figure 10.

To summarize, the PADX concrete data model completely implements the Galax data model, making it possible to evaluate any XQuery program over a PADS data source. Due to limited space, we have omitted some details, such as how PADX guarantees that each virtual node has a unique, immutable identity, as is required by the Galax abstract data model. The data model's most important features are that it provides lazy access to virtual XML nodes in the PADS source, it delegates navigation to type-specific node representations, and it separates navigation of the virtual nodes from data loading, which is discussed next.

4.3 Loading PADS data

The PADX abstract data model provides Galax with a random-access view of a PADS data source. In particular, any virtual node may be accessed in any order at any time during query evaluation regardless of its physical location in the PADS data. This abstraction permits the PADX concrete data model to decide when and how to read and parse, or *load*, a data source.

PADX has three strategies for loading data, each of which use the multiple-entry parsing functions generated by the PADS compiler. The *bulk* strategy loads a complete PADS source before query evaluation begins, populating all the in-memory representations and parse descriptors. With all data pre-fetched, bulk loading is the simplest strategy to implement random access. However, because each PADS value has a lot of associated meta-data, bulk loading incurs a high memory cost and is only feasible for smaller data sources.

The *on-demand, random-access* strategy loads PADS data when Galax accesses virtual nodes via the abstract data model. The strategy maintains a fixed size buffer for loaded values and when the buffer is filled, expels values in LIFO order. The default units loaded are any PADS types annotated with **`Precord`**, which indicates that the type denotes an atomic physical unit in the ambient coding. This default works well in practice, because many PADS sources contain a header, one (or more) very large array(s) of records, and a trailer. This strategy loads all the data before the record array(s) and then loads each array item on demand, expelling old records when the buffer is filled. A small amount of meta-data is preserved for each expelled record, so that the virtual node containing that data can be reconstructed on subsequent accesses.

The *on-demand, sequential* strategy is a restriction of the on-demand, random-access strategy. It loads data on demand, but its fixed-size buffer stores only one record at a time, and it supports strictly sequential access to records, *i.e.*, accessing records out of order is prohibited. Given that the Galax abstract data model requires random access, it is not obvious when this strategy can be used, even though it has the smallest memory footprint of all three and therefore could scale to very large sources. It turns out that many common XQuery queries can be evaluated in *one* sequential scan over the input document, and in these cases, the sequential strategy is both semantically correct and time and space efficient. We give examples of "one-scan" queries and their performance in Section 5.

4.4 Ways to use PADX

Our focus so far has been on describing PADX's internal architecture to demonstrate the feasibility of viewing and querying ad hoc data sources as though they were XML. We expect this use of PADX to be convenient, because it supports rapid querying of transient data and does not require an analyst to convert the data into another format or load it into a database before being able to ask simple queries. PADX can be used in other ways. For example, an analyst might prefer to materialize a PADS source in XML and query her data using a high-performance, commercial XML query engine. To do this, the analyst simply runs the query "`$pads`", which returns the entire source materialized in XML, and then provides the resulting XML document to the query engine. Another use is to transform the PADX view of a PADS source into the XML view required by a database by some down-stream application. Such transformations can be easily expressed in XQuery and can be statically type checked against the PADX and target XML schemata.

We note, however, that the size of an ad hoc data source is significantly smaller than its representation in XML. For our two example PADS sources, the ratio of the size of the original PADS data to its size in XML using the mapping described in Section 4.1 ranges from 1:7 to 1:8. Of course, this size increase depends on the PADS types and field names in the PADS description, but even a reasonable choice of names like those in Figure 4 results in a significant size increase. We mention this size increase to give the reader some sense of the relative scale of data sources that PADX can query compared to those supported by native XML query engines.

5 Performance

Query performance in PADX depends on the efficiency of the underlying concrete data model; therefore its performance must be well understood before we can understand the performance of particular query plans. We focus on the performance of the concrete data model and measure the cost of accessing data via the PADS type-specific parsing functions, the PADX type-specific node representations, and the generic PADX concrete data model. At the end of this section, we give preliminary measurements on query performance.

We measured data model and query performance for two PADS sources, Sirius and the Web server logs in Figure 1, on data sources of 1 to 50MB in size. Our measurements were taken on an 1.67GHz Intel Pentium M with 500MB real memory running Linux Redhat

```
 1. class order_header_t_node_rep
 2.       (field_name : string)
 3.       (rep : order_header_t)
 4.       (pd  : order_header_t_pd) =
 5. object
 6.   inherit padx_node_rep
 7.   method name() = field_name
 8.   ...
 9.   method kth_child idx =
10.     match idx with
11.     |  1 -> Some(new val_Puint32_node_rep("order_num", rep.order_num, pd.order_num_pd))
12.     |  2 -> Some(new val_Puint32_node_rep("att_order_num", rep.att_order_num, pd.att_order_num_pd))
13.     | ...
14.     | 14 -> Some(new Pstruct_pd_node_rep("pd", pd))
15.     | _  -> None

16.   (* Chidren's name map *)
17.   let name_map = Associative_array.create [("order_num", 1); ("att_order_num", 2); ...; ("pd", 14)]
18.   method kth_child_by_name child_name =
19.     match Associative_array.lookup name_map child_name with
20.     | None -> Cursor.empty_cursor()
21.     | Some idx -> kth_child idx
22. end
```

Figure 14. Fragment of compiler-generated node representation for `order_header_t`

	Data size (MB)				
Source	1	5	10	20	50
Sirius	0.25	0.23	0.23	0.22	10.64
Web server	0.70	0.67	0.67	1.18	6.14

Table 1. Bulk strategy: load time per byte in μs

9.0. Each test was run five times, the high and low times were dropped, and the mean of the remaining three times is the reported time.

Source	Data size	PADS read	PADX node rep	PADX generic DM
Sirius	5MB	0.07	0.27	0.61
	10MB	0.06	0.26	0.56
	50MB	0.06	0.25	0.56
Web server	5MB	0.54	0.78	1.63
	10MB	0.53	0.74	1.61
	50MB	0.53	0.74	1.58

Table 2. Sequential strategy: load time per byte in μs

5.1 Concrete Data Model

We first measured the time to bulk load data sources of 5, 10, 20, and 50MB by calling the PADS parsing functions, *i.e.*, the lowest level in the PADX data model. Table 1 gives the load time per byte in microseconds. For smaller sources, load time is constant, but eventually increases. For Sirius, the increasing load time is observed at 50MB and for the Web server data at 20MB. We note that for a PADS source, the memory overhead of a PADS parsed value can be four to sixteen times the size of the raw data, depending on the value's type. In the cases where non-linear load time occurs, the processes' physical memory usage is close to or exceeds real memory, CPU utilization plummets, and the process begins to thrash. These measurements indicate that the bulk strategy is only feasible for smaller data sources.

Next, we measured load time using the on-demand sequential stategy on sources of 5, 10, and 50 MB. We were particularly interested in the overhead introduced at each level in the concrete data model. Table 2 gives the load time per byte in microseconds (μs) for three levels: reading the source by calling the PADS parsing functions directly, a depth-first walk of the virtual XML document by calling the PADX node-representation functions, and a depth-first walk of the virtual XML document by calling the PADX generic data model. Recall that the node-rep functions are in C and the generic data model is in O'Caml.

We observe that the load time per byte at each level is near constant for increasing source size, but that each level incurs a substantial cost compared to the lower levels. For the Sirius source, the PADX node-representation is four times slower than the native PADS parsing functions, but for the Web-server source, the PADX node representation is only 44% slower. Understanding the source of this difference requires further experiments with other sources.

For both sources, the generic concrete data model (in O'Caml) is twice as slow as the node representation (in C). The interface from the generic data model to the node representation crosses the O'Caml-C boundary and uses data marshalling functions generated by the O'Caml IDL tool. We have noticed similar per-byte read costs in the Galax secondary storage system [16], whose data-model architecture is similar to that of PADX.

We also measured the time to load using the on-demand, random-access strategy. In general, it was 10–15% slower than the on-demand, sequential strategy.

These measurements indicate that the on-demand, sequential strategy scales with increasing data size, and that there is a constant overhead incurred at each level in the data model. Ideally, we would like the cost of accessing data via the generic concrete data model to be close to the PADS read cost, but this will require more engineering effort.

Data size (MB)	1	5	10	20	50
Time (seconds)	1.0	4.8	10.7	24.0	90.0

Table 3. PADX query evaluation time in seconds

5.2 Querying

Ultimately, PADX's query performance depends on Galax, because the Galax compiler produces and executes the query plans. Currently, Galax's query compiler includes a variety of logical optimizations for detecting joins and re-grouping constructs in XQuery expressions. Another important optimization is detecting when a query can be evaluated in one scan over the input document. Path expressions that contain only descendant axes and no branches are one example of the kind of queries that can be evaluated in one scan. For example, the following query, which returns the locations of all records containing some error in a Sirius source, can be evaluated in one scan:

```
$pads/Psource/orders/elt/pd/loc
```

Detecting and evaluating one-scan queries (also known as streamable queries) is necessary in XML environments in which the XML data is an infinite or bursty stream. Several query processors already exist in which streamable queries are evaluated directly over a stream of tokens produced by SAX-style parsers [9, 14].

Streamable queries are important for PADX, because the resulting plans can be evaluated on large PADS sources that are loaded on-demand and sequentially. Table 3 contains the time in seconds to evaluate the query above when applied to PADS data sources into which we injected errors randomly in the file (12 errors per 1MB). The query plan produced by Galax is not perfectly pipelined, thus the execution time is super linear.

To understand the costs and benefits of other evaluation strategies, we materialized the 1MB PADS source in Table 3, which yielded a 7.4MB XML document. We then used Galax to execute the above query, using the same query execution plan, and applied it to the 7.4MB XML document loaded into the main-memory data model. The execution time was 13.1s of which 12.9 was spent in document parsing. To amortize the cost of document parsing time, we often store documents in Galax's secondary storage system. To compare with this strategy, we stored the 7.4MB XML document in Galax's secondary storage system, which required 166MB of disk space. We then ran the above query on the stored document. The execution time was 2.9s, almost three times slower than PADX applied to the PADS data directly. For comparison with an independent query processor, we evaluated the above query using Saxon [12], a popular XSLT and XQuery engine, applied to the 7.4MB document and it executed in 6.3s.

In summary, our initial impressions are that evaluating streamable XQuery expressions directly on a PADS source is feasible, efficient, and convenient.

6 Discussion

The PADX system solves important data-management tasks: it supports declarative description of ad hoc data formats, its descriptions serve as living documentation, and it permits exploration of ad hoc data and vetting of erroneous data using a standard query language. The resulting PADS descriptions and queries are robust to changes that may occur in the data format, making it possible for more than one person to profitably use and understand a PADX description and related queries.

A PADX query corrall is an example of partially compiled query engine, because its concrete data model is customized for a particular data format, but its queries are interpreted over an abstract data model that delegates to the concrete model. This architecture places PADX on the continuum between query architectures that provide fully interpreted query plans applied to generic data models to architectures that provide fully compiled query plans applied to customized data model instances [10]. The latter architectures provide very high performance on large scale data. PADX has some of the benefits of such architectures but does not have the overhead of a complete database system.

Others share our interest in declarative descriptions of ad hoc data formats. Currently, the Global Grid Forum is working on a standard data-format description language for describing ad hoc data formats, called DFDL [3, 4]. Like PADS, DFDL has a rich collection of base types and supports a variety of ambient codings. Unlike PADS, DFDL does not support semantic constraints on types nor dependent types, *e.g.*, it is not possible to specify that the length of an array is determined by some field in the data. DFDL is an annotated subset of XML Schema, which means that the XML view of the ad hoc data is implicit in a DFDL description. DFDL is still being specified, so no DFDL-aware parsers or data analyzers exist yet. We expect that bi-directional translation between PADS and DFDL to be straightforward. Such a translation would make it possible for DFDL users to use PADX to query their ad hoc data sources.

The steps in a data-management workflow that PADX addresses typically precede the steps that require a high-performance database system, *e.g.*, asking complex OLAP queries applied to long-lived, archived data. Commercial database products do provide support for parsing data in external formats so the data can be imported into their database systems, but they typically support a limited number of formats, *e.g.*, COBOL copybooks, no declarative description of the original format is exposed to the user for their own use, and they have fixed methods for coping with erroneous data. For these reasons, PADX is complementary to database systems.

We continue to focus on improving the usability and scalability of PADX. Currently, PADX is not compositional, because the result of evaluating a query is in native XML, not in a PADS format. Given an arbitrary XQuery expression over a PADX source, an open problem is being able to infer a reasonable PADS format for the result and produce the results in this format. We have already mentioned the important problem of detecting when a query can be evaluated in a single scan over an input document and of producing a fully pipelined execution plan. Interestingly, this problem is important in XML environments in which the XML data is an infinite or bursty stream. We are working on improving Galax's ability to detect one-scan queries and to produce query plans that are indeed fully pipelined and that use limited memory.

7 References

[1] Galax user manual. http://www.galaxquery.org.

[2] PADS user manual. http://www.padsproj.org/.

[3] Data format description language (DFDL) a Proposal, Working Draft, Global Grid Forum. https://forge.gridforum.org/projects/dfdl-wg/document/DFDL_Proposal/en/%2, Aug 2005. Global Grid Forum.

[4] M. Beckerle and M. Westhead. GGF DFDL primer. http://www.ggf.org/Meetings/GGF11/Documents/DFDL_Primer_v2.pdf,

May 2004. Global Grid Forum.

[5] M. Brundage. *XQuery: The XML Query Language*. Addison-Wesley, 2004.

[6] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: High performance network monitoring with an SQL interface. In *SIGMOD*. ACM, 2002.

[7] M. Fernández, J. Siméon, B. Choi, A. Marian, and G. Sur. Implementing XQuery 1.0: The Galax Experience. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 1077–1080, Berlin, Germany, Sept. 2003.

[8] K. Fisher and R. Gruber. PADS: A domain-specifi c language for processing ad hoc data. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, June 2005.

[9] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, and A. Sundararajan. The BEA streaming XQuery processor. *VLDB J.*, 13(3):294–315, 2004.

[10] R. Greer. Daytona and the fourth generation language cymbal. In *Proceedings of ACM Conference on Management of Data (SIGMOD)*, 1999.

[11] T. Grust, M. van Keulen, and J. Teubner. Staircase join: Teach a relational DBMS to watch its axis steps. In *Proceedings of International Conference on Very Large Databases (VLDB)*, pages 524–535, Berlin, Germany, Sept. 2003.

[12] M. Kay. SAXON 8.0. SAXONICA.com. `http://www.saxonica.com/`.

[13] C. Ré, J. Siméon, and M. Fernández. A complete and effi cient algebraic compiler for XQuery. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, April 2006.

[14] K. Rose and L. Villard. Phantom XML. In *XML Conference and Exhibition*, 2005.

[15] J. Siméon and M. F. Fernández. Build your own XQuery processor. EDBT Summer School, Tutorial on Galax architecture, Sept 2004. `http://www.galaxquery.org/slides/edbt-summer-school2004.pdf`.

[16] A. Vyas, M. F. Fernández, and J. Siméon. The simplest XML storage manager ever. In *XIME-P 2004*, pages 37–42, Paris, France, June 2004.

[17] W3C. XQuery 1.0 and XPath 2.0 data model, Oct. 2005. `http://www.w3.org/TR/query-datamodel/`.

[18] Extensible markup language (XML) 1.0. W3C Recommendation, Feb. 2004. `http://www.w3.org/TR/2004/REC-xml-20040204/`.

[19] XPath 2.0. W3C Working Draft, Oct. 2005. `http://www.w3.org/TR/xpath20`.

[20] XQuery 1.0: An XML query language. W3C Working Draft, Oct. 2005. `http://www.w3.org/TR/xquery/`.

[21] XML schema part 1: Structures. W3C Recommendation, Oct. 2004. http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/.

OCaml + XDuce

Alain Frisch
INRIA Rocquencourt
Alain.Frisch@inria.fr

January 2006

Abstract

This paper presents the core type system and type inference algorithm of OCamlDuce, a merger between OCaml and XDuce. The challenge was to combine two type checkers of very different natures while preserving the best properties of both (principality and automatic type reconstruction on one side; very precise types and implicit subtyping on the other side). Type inference can be described by two succesive passes: the first one is an ML-like unification-based algorithm which also extracts data flow constraints about XML values; the second one is an XDuce-like algorithm which computes XML types in a direct way. An optional preprocessing pass, called strenghtening, can be added to allow more implicit use of XML subtyping. This pass is also very similar to an ML type checker.

1 Introduction

This paper presents the core type system of OCamlDuce, a merger between OCaml [L$^+$01] and XDuce [Hos00, HP00, HP03, HVP00]. OCamlDuce source code, documentation and sample applications are available at `http://www.cduce.org/ocaml`.

OCaml is a widely-used general-purpose multi-paradigm programming language with automatic type reconstruction based on unification techniques. XDuce is a domain specific and type-safe functional language adapted to writing transformations of XML documents. It comes with a very precise and flexible type system based on regular expression types and a natural notion of subtyping. The basic type-checking primitives for XDuce constructions are rather involved, but the structure of the type checker is simple: types are computed in a bottom-up way along the abstract syntax tree; the input and output types of functions are explicitly provided by the programmer. The high-level objective of the OCamlDuce project is to enrich OCaml with XDuce features in order to provide a robust development platform for applications that need to deal with XML but which are not necessarily focused on XML.

The challenge was to combine two type checkers of very different natures while preserving as much as possible the best properties of both (principality and automatic type reconstruction on one side; very precise types and implicit subtyping on the other side).

Our main guideline was to design a type system which can be implemented by reusing existing implementations of OCaml and CDuce [BCF03, Fri04]. (CDuce can be seen as a dialect of XDuce with first-class and overloaded functions – for the merger with OCaml, we don't consider these extra features). Because of the complexity of OCaml's type system, it was out of question to reimplement it. The typing algorithm we describe in this paper has been successfully implemented simply by combining a slightly modified OCaml type checker with the CDuce type checker, and by adding some glue code. As a result, OCamlDuce is a strict extension of OCaml: programs which don't use the new features will be treated exactly the same by OCaml and OCamlDuce. It is thus possible to compile any existing OCaml library with OCamlDuce. Also, we believe our modifications to the OCaml compiler are small enough to make it easy to maintain OCamlDuce in sync with future evolutions of OCaml. Our experience so far confirms that.

Another guideline in the design of OCamlDuce was that XDuce programs should be easily translatable to OCamlDuce in a mechanical way. In XDuce, all the functions are defined at the toplevel and comes with an explicit signature. We can obtain an OCamlDuce program by some minor syntactical modifications (the new constructions in the language are delimited to avoid grammatical overloading of notations). Explicit function signatures are simply translated to type annotations.

The design goals pushed us into the direction of simplicity. We choosed to segregate XDuce values from regular ML values. Of course, a constructed ML value can contain nested XDuce values, but from the point of view of ML, XDuce values are black boxes, and similarly for types. Also, we de-

cided not to have parametric polymorphism on XDuce types. A type variable can of course be instantiated to an XDuce type (or to a type which contains a nested XDuce type), but it is not possible to force a generalized variable to be instantiated only to XDuce types or to use a type variable within an XDuce type. The technical presentation introduces a notion of foreign type variables, but they are nothing more than a technical device for inferring ground XDuce types.

Overview In Section 2, we give some intuitions about the behavior of OCamlDuce's type-checker.

The formalization of the type system will be developped by abstracting away from details about XDuce. In Section 3, we introduce an abstract notion of *extension* (foreign types and foreign operators) and show of XDuce can be seen as an extension. In Section 4, we present the type-system and type inference algorithm for a calculus made of ML [Mil78, Dam85] plus an arbitrary extension. The basic idea is to rely on standard techniques for ML type inference. Indeed, we start from a type system which is an instance of ML where foreign types are considered as atomic types and foreign operators are explicitly annotated with their input and output types. Then we present an algorithm to *infer* these annotations. This algorithm is described as two successive passes: the first one is a slightly modified version of an ML type-checker, and the second one is a simple forward computation on foreign types.

In Section 5, we present a preprocessing pass, called strengthening, whose purpose is to make more programs accepted by the type system by allowing implicit use of subtyping.

In Section 6, we present other details of the concrete integration in OCaml. In Section 7, we compare our approach to related works.

2 An example

In this section, we illustrate the behavior of OCamlDuce's type-checker on the following code snippet:

```
let f x = match x with
  {{ [ (y::<a>_ | _)* ] }} -> {{ y @ y }}
let z1 =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ] }}
let z2 =
  List.map f
    [ {{ [ <a>[<a>[]] ] }};
      {{ [ <a>[<c>[]] ] }} ]
```

The exemple is intended to illustrate the use of the OCaml type checker to perform a data-flow analysis of XML values, and also how OCaml features (here, higher-order functions and data-structures) interact with XDuce features.

Double curly-braces {{...}} are used in OCamlDuce only to avoid ambiguities in the grammar; they carry no typing information. For instance, the symbol `@` used for list concatenation in OCaml is re-used for denote XML sequence concatenation. Similarly, the square brackets `[...]` are used both to denote OCaml list literals (whose elements are separated by semi-colons) and XML sequences literals when used within double curly braces (their elements are separated by whitespace). XML element literals are written in the form `<tag>content`.

The first line of the program above declares a function `f` which consists of an XML pattern matching on its argument, with a single branch. The XML pattern `p = [ (y::<a>_ | _)* ]` extracts from an XML sequence all the elements with a tag `<a>` and put them (in order) in the capture variable `y`. The function is then used twice, including once indirectly through a call to the function `List.map` (from the OCaml standard library) of type $\forall\alpha,\beta.(\alpha \to \beta) \to \alpha$ `list` $\to \beta$ `list`. For the purpose of explaining type-checking, we will rewrite the body of the function `f` as:

```
let f x =
  let y = match[y;p](x) in
  {{ y @ y }}
```

The `y` and `p` parameters of the `match` operator represent the capture variable under consideration and the pattern itself.

In OCamlDuce, XML values (elements, sequences, ...) and regular OCaml values are kept appart. An XML value can of course appear as part of an OCaml value (e.g. the XML elements which are put into an OCaml list), but an OCaml value cannot appear within an XML value. The same applies to types: an XML type can appear as part of a complex OCaml type expression, but the converse is impossible. XML operators can be applied to XML values and return new XML values. In the example, we can see three kind of XML operators: XML literals (no argument), XML concatenation (two arguments), and XML pattern matching (one argument).

The basic idea of the OCamlDuce type system is to infer XML types for the inputs and outputs of XML operators. This is done by introducing internally a new kind of type variables, called XML type variables. Before proper type-checking starts, each XML operator used in the program is annotated with fresh XML type variables (in subscript position for the inputs, and in superscript position for the outputs):

```
let f x =
  let y = match[y;p]^{ι2}_{ι1}(x) in
  {{ y @^{ι5}_{ι3,ι4} y }}
let z1 =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ]^{ι6} }}
let z2 =
  List.map f
    [ {{ [ <a>[<a>[]] ]^{ι7} }};
      {{ [ <a>[<c>[]] ]^{ι8} }} ]
```

The regular OCaml type-checker is then applied. It gives to each XML operator an arrow type following the annotations and then proceeds as usual (generalizes types of `let`-bound identifiers, instantiates ML type-schemes when an identifier is used, and performs unifications to make type compatible).

For instance, the concatenation operator in our example is given the type $\iota_3 \to \iota_4 \to \iota_5$, and the type-checker performs the following unifications: $\iota_2 = \iota_3 = \iota_4$ (the type for `y`), $\iota_1 = \iota_6 = \iota_7 = \iota_8$ (the type for the argument of `f`). It also produces the following types for the top-level identifiers:

```
val f  : ι1  →  ι5
val z1 : ι5
val z2 : ι5 list
```

Of course, we must still instantiate the XML type variables with ground XML types. Each occurence of an XML operator in the program gives one constraint on the instantiation. Indeed, we can interpret each n-ary operator as as n-ary function from XML types to XML types. If we choose ι_1 and ι_2 as representatives for their classes of equivalence modulo unification, the program is:

```
let f x =
  let y = match[y;p]^{ι2}_{ι1}(x) in
  {{ y @^{ι5}_{ι2,ι2} y }}
let z1 =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ]^{ι1} }}
let z2 =
  List.map f
    [ {{ [ <a>[<a>[]] ]^{ι1} }};
      {{ [ <a>[<c>[]] ]^{ι1} }} ]
```

from which we read the following constraints:

$$
\begin{array}{lcl}
\iota_2 & \geq & \texttt{match[y;p]}(\iota_1) \\
\iota_5 & \geq & \iota_2 @ \iota_2 \\
\iota_1 & \geq & \texttt{[<a>[] <b>[] <a>[<b>[]]]} \\
\iota_1 & \geq & \texttt{[<a>[<a>[]]]} \\
\iota_1 & \geq & \texttt{[<a>[<c>[]]]}
\end{array}
$$

In this system, we consider `match[y;p]` as a function from XML types to XML types, given by XDuce's type inference algorithm for pattern matching. Similarly, the operator `@` is now intrepreted as a function from pair of types to types.

The set of constraints generates dependencies between variables. We say that a variable on a left-hand side of a constraint depends on variables of the right-hand side. In our example, the graph of dependencies between variables is acyclic. In this case, we can topologically order the variables and find the least possible ground XML type for each of them: we assign to a variable the union of all its lower bounds. In the example, we will thus compute the following instantiation:

$$
\begin{array}{lcl}
\iota_1 & = & \texttt{[R1]} \\
\iota_2 & = & \texttt{match[y;p]([R1])} = \texttt{[R2]} \\
\iota_5 & = & \iota_2 @ \iota_2 = \texttt{[R2 R2]}
\end{array}
$$

where `R1` is the regular epxression `(<a>[]<b>[]<a>[<b>[]])|<a>[<a>[]|<c>[]]` and `R2` is the regular expression `(<a>[]<a>[<b>[]])|<a>[<a>[]|<c>[]]`.

Type-checking is over: we have found an instantiation for XML type variables which satisfies all the constraints. In essence, the type-checker has collected all the XML types that can flow to the input of the function, and then type-checked the body of the function with the union of all these types. In general, the OCaml type-checker is used to infer the data flow of XML values in the programs. The way to solve the resulting set of constraints by forward computation corresponds roughly to the structure of the XDuce type-checker.

Implicit subtyping Let's see what happens if we add an explicit type constraint for `z1`:

```
let z1 : {{[ <a>_* ]}} =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ] }}
```

The algorithm described above will infer a much less precise type for `z2` as well, which is unfortunate. The reason is that the OCaml type-checker unifies ι_5 with `[ <a>_* ]`. Basically, the unification-based type system forgets about the direction of the data flow. There is some dose of implicit subtyping in the algorithm, but only for the result of XML operators (because of the way we interpet them as subtyping - not equality - constraints).

In order to address this lack of implicit subtyping, we use a preprocessing pass whose purpose is to detect which subexpressions are of kind XML and to introduce around them a special unary XML operator `id` which behaves semantically as the identity, but allows subtyping. This preprocessing pass would rewrite the definition for `z1` as:

```
let z1 : {{[ <a>_* ]}} =
  id^{ι10}_{ι9}(f {{[<a>[] <b>[] <a>[<b>[]]]^{ι1}}})
```

The variable ι_9 will then be unified with ι_5 and ι_{10} with `[ <a>_* ]`. The additional constraint corresponding to the `id` operator is thus simply:

$$\texttt{[<a>\_*]} \geq \iota_5$$

which is satisfied by the same instantiation for ι_5 as in the original example. As a consequence, the type for `z2` is not changed.

The preprocessing pass is quite simple. It consists of another run of the OCaml type-checker, where all the XML types are considered equal. This allows to identify which sub-expressions are of kind XML. Section 5 describes formally this pass.

Breaking cycles The key condition which allowed us to compute an instantiation for XML type variables in the example was the acyclicity of the constraints. This property does not always hold. For instance, let's extend the original example with the following definition:

```
let z3 = f z1
```

Without the preprocessing pass mentionned above, this line would force the OCaml type-checker to unify ι_1 and ι_5. The preprocessing pass actually replaces this definition by:

$\texttt{let z3 = f id}^{\iota_{12}}_{\iota_{11}}\texttt{(z1)}$

The type-checker then unifies ι_{11} with ι_5 and ι_{12} with ι_1; the resulting constraint for `id` is thus:

$$\iota_1 \geq \iota_5$$

which corresponds to the fact that the output of `f` can flow back to its input. We observe that the set of constraints has now a cycle between variabls ι_1, ι_5 and ι_2.

Our type-system cannot deal with such a situation. It would issue an error explaining that the inferred data flow on XML values has a cycle. The programmer is then required to break explicitly this cycle by providing more type annotations. For instance, the programmer could use the same annotation as above on `z1`:

```
let z1 : {{ [ <a>_* ] }} =
  f {{ [ <a>[] <b>[] <a>[<b>[]] ] }}
```

or maybe he will prefer to annotate the input or output type of `f`.

3 Abstract extension of ML

The previous section explained the behavior of OCamlDuce's type checker on a example. It should be clear from this example that the type system is largely independent of the actual definitions of values, types, patterns and operators from XDuce and could be applied to other extensions of OCaml as well. In this section, we will thus introduce an abstract notion of extension and show how XDuce fits into this notion. This more abstract presentation should help the reader to understand the structure of the type checker, without having to care about the details of XDuce type system.

Definition 1. *An* **extension** X *is defined by:*

- *a set of ground foreign types* $\mathcal{T}$*;*
- *a subtyping relation* $\leq$ *on* $\mathcal{T}$*, which is a partial order with a finite least-upper bound operator* $\sqcup$*;*
- *a set of foreign operators* $\mathcal{O}$*;*
- *for each operator* $\mathtt{o} \in \mathcal{O}$*: an arity* $n \geq 0$ *and an abstract semantics* $\hat{\mathtt{o}} : \mathcal{T}^n \to \mathcal{T}$ *which is monotone with respect to* $\leq$ *on each of its argument.*

We use the meta-variable τ to range over ground foreign types. The foreign operators are used to model both foreign value constructors and operations on these foreign values. Since we are not going to formalize dynamic semantics, we don't need to distinguish between these two kinds of operators.

The monotonicity requirement on the abstract semantics ensures that our resolution strategy (taking the union of lower bounds for each variables) for constraints is complete.

We don't formalize in this paper the operational semantics of operators. Instead, we assume informally that it is given and compatible with the abstract semantics.

XDuce as an extension We now show how XDuce features can be seen as an extension. We consider here a simple version of XDuce, with the following kind of expressions: element constructor $\mathtt{a}[\mathtt{e}]$ (seen as a unary operator), empty sequence $()$, concatenation $\mathtt{e}_1, \mathtt{e}_2$, and pattern matching $\mathtt{match\ e\ with\ p} \to \mathtt{e} \mid \ldots \mid \mathtt{p} \to \mathtt{e}$. OCamlDuce is actually build on CDuce, which considers for instance XML element constructors as ternary operators (the tag and a specification for XML attributes are also considered as arguments).

The meta-variable `p` ranges over XDuce patterns. We don't need to recall here what they are. We just need to know that for any pattern `p` we can define an accepted type $\lbag \mathtt{p} \rbag$,

a finite set of capture variable $\texttt{Var}(\texttt{p})$, and for any type τ and any variable $\texttt{x}$ in $\texttt{Var}(\texttt{p})$, a type $\texttt{match}[\texttt{x};\texttt{p}](\tau)$ (which represents the set of values possibly bound to $\texttt{x}$ when the input value is in τ and the pattern succeed)

Here is the formal definition of an extension X for XDuce. We take for foreign types the XDuce types quotiented by the equivalence induced by the subtyping relation (that is: types with the same set-theoretic interpretation are considered equal). The least-upper bound operator $\sqcup$ corresponds to XDuce's union type constructor (usually written $|$). We use the following families of foreign operators:

- a unary operator for each XML label $\texttt{a}$, a unary operator;
- a binary operator corresponding to the concatenation;
- a constant operator corresponding to the empty sequence;
- for any pattern $\texttt{p}$ and variable $\texttt{x}$ in $\texttt{Var}(\texttt{p})$, a unary operator written $\texttt{match}[\texttt{x};\texttt{p}]$ (its semantics is to return the value bound to $\texttt{x}$ when matching its argument against the pattern $\texttt{p}$).

The abstract semantics for all these operators follows directly from XDuce's theory.

Element constructor, concatenation and the empty sequence expressions can directly be seen as foreign operators. This is not the case for a pattern matching $\texttt{match e with } \texttt{p}_1 \rightarrow \texttt{e}_1 \mid \ldots \mid \texttt{p}_n \rightarrow \texttt{e}_n$. We are going to present an encoding of pattern-matching in terms of operators and normal ML expressions. This encoding is rather heavy; in practice, the implementation deals with pattern matching directly.

First, we define the translation $\overline{\texttt{p} \rightarrow \texttt{e}}$ of a single branch where $\texttt{Var}(\texttt{p}) = \{\texttt{x}_1, \ldots, \texttt{x}_n\}$ as the expression:

$$
\begin{array}{l}
\lambda\texttt{x}. \\
\texttt{let } \texttt{x}_1 = \texttt{match}[\texttt{x}_1;\texttt{p}]\texttt{x in} \\
\ldots \\
\texttt{let } \texttt{x}_n = \texttt{match}[\texttt{x}_n;\texttt{p}]\texttt{x in} \\
\texttt{e}
\end{array}
$$

Then, the translation of $\texttt{match e with } \texttt{p}_1 \rightarrow \texttt{e}_1 \mid \ldots \mid \texttt{p}_n \rightarrow \texttt{e}_n$ is defined as:

$$
\begin{array}{l}
\texttt{let x} = \texttt{e in} \\
\texttt{dispatch}[\tau_1, \ldots, \tau_n]\ \texttt{x x}\ (\overline{\texttt{p}'_1 \rightarrow \texttt{e}_1})\ \ldots\ (\overline{\texttt{p}'_n \rightarrow \texttt{e}_n})
\end{array}
$$

where $\tau_i = \lceil \texttt{p}_i \rfloor$ and $\texttt{p}'_i = \texttt{p}_i \backslash (\tau_1 \sqcup \ldots \sqcup \tau_{i-1})$ (the restriction of $\texttt{p}_i$ to values which do not match any pattern form an preceding branch). We have used in this translation a new built-in ML constant $\texttt{dispatch}[\tau_1, \ldots, \tau_n]$ of type scheme: $\forall\alpha.(\tau_1 \sqcup \ldots \sqcup \tau_n) \rightarrow \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \ldots \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$, which we assume to be present in the initial typing environment. Its intuitive semantics is to drop the first argument (it is used only to force the type-checker to verify that $\texttt{x}$ has type $\tau_1 \sqcup \ldots \sqcup \tau_n$, which corresponds to the XDuce's pattern matching exhaustivity condition), and to call the k^{th} functional argument ($1 \leq k \leq n$) on the second argument when k is the smallest integer such that this argument has type τ_k.

In principle, the technique described in this paper could be used to integrate many of the existing extensions to the original XDuce design (such as attribute-element constraints [HM03] or XML filters [Hos04]) without any additional theoretical complexity. In its current form, however, OCamlDuce integrates all the CDuce extensions except overloaded functions: XML attributes as extensible records, sequence and tree pattern-based iterators, strings as sequences of characters (hence string regular expression types and patterns), etc.

4 Type system

In this section, we present a type system and a type inference algorithm for a fixed extension X. Our language will be the kernel of an ML-like type system, enriched with types and operators from the extension X.

Types and expressions The syntax of types and expressions is given in Figure 1. We use a vector notation to represent tuples. E.g. $\vec{\texttt{t}}$ stands for an n-tuple $(\texttt{t}_1, \ldots, \texttt{t}_n)$.

We assume a set of ML type constructors, ranged over by the meta-variable $\texttt{P}$. Each ML type constructor comes with a fixed arity and we assume all the types to be well-formed with respect to these arities. The arrow $\rightarrow$ is considered as a distinguished binary type constructor for which we use an infix and right-associative syntax.

We assume given two infinite families of type variables and foreign type variables, respectively ranged over by the meta-variables α and ι. In an expression $\exists\alpha.\texttt{e}$, the type variable α is bound in $\texttt{e}$. Expressions are considered modulo α-conversion of bound type variables. The construction $\exists\alpha.\texttt{e}$ thus serves to introduce a fresh type variable α to be used in a type annotation somewhere in $\texttt{e}$.

Foreign operators are annotated with the type of their arguments (in subscript position) and of their result (in superscript); the number of type arguments is assumed to be coherent with the arity of the foreign operator. However, in

practice, the source language does not include the annotations: they are automatically filled with fresh foreign type variables by the compiler (we also use this convention in this paper for some examples). Putting the annotations in the syntax is just a way of simplifying the presentation. The main technical contribution of the paper is an algorithm to infer ground foreign types for the foreign type variables.

The ML(X) fragment We call ML(X) the fragment of our calculus where all the foreign types are restricted to be ground. Figure 2 defines a typing judgment $\Gamma \vdash \mathtt{e} : \mathtt{t}$ for ML(X). It is exactly an instance of the ML type system [Mil78, Dam85] if we see ground foreign types as atomic ML types and ground-annotated foreign type operators $\mathtt{o}^{\tau}_{\vec{\tau}}$ as built-in ML constants or constructors (we also introduce explicit type annotation and type variable introduction). We recall classical notions of type scheme, typing environment and generalization. A **type scheme** is a pair of a finite set $\bar{\alpha}$ of type variables and of a type $\mathtt{t}$, written $\forall\bar{\alpha}.\mathtt{t}$. The variables in $\bar{\alpha}$ are considered bound in this scheme. We write $\sigma \ll \mathtt{t}$ if the type $\mathtt{t}$ is an instance of the type scheme σ. A **typing environment** is a finite mapping from program variables to type schemes. The **generalization** of a type $\mathtt{t}$ with respect to a typing environment Γ, written $\mathtt{Gen}_{\Gamma}(\mathtt{t})$ is the type scheme $\forall\bar{\alpha}.\mathtt{t}$ where $\bar{\alpha}$ is the set of variables which are free in $\mathtt{t}$, but not in Γ.

Type-soundness of the ML(X) fragment We *assume* that a *sound* operational semantics is given for the ML(X) calculus. This amounts to defining δ-reduction rules for the $\mathtt{o}^{\tau}_{\vec{\tau}}$ operators which are coherent with the abstract semantics for the foreign operators. Well-typed expressions in ML(X) (in an empty typing environment, or an environment which contains built-in ML operators) cannot go wrong. We also assume that the operational semantics for an $\mathtt{o}^{\tau}_{\vec{\tau}}$ operator depends only on $\mathtt{o}$, not on the annotations $\vec{\tau}$, τ. This allows us to lift the semantics of ML(X) to the full calculus.

Typing problems A **substitution** ϕ is an idempotent function from types to types that maps type variables to types, foreign type variables to foreign types, ground foreign types to themselves, and that commutes with ML type constructors. We use a post-fix notation to denote a capture-avoiding application of this substitution to typing environments, expressions, types or constraints.

A substitution ϕ_1 is **more general** than a substitution ϕ_2 if $\phi_2 = \phi_2 \circ \phi_1$. (Or equivalently, because substitutions are idempotent: there exists a substitution ϕ such that $\phi_2 = \phi \circ \phi_1$.)

A **typing problem** is a tuple $(\Gamma, \mathtt{e}, \mathtt{t})$. (Usually, $\mathtt{t}$ is a fresh type variable.) A **solution** to this problem is a substitution ϕ such that $\Gamma\phi \vdash \mathtt{e}\phi : \mathtt{t}\phi$ is a valid judgment in ML(X). We will now rephrase this definition in terms of a typing judgment on the full calculus. This judgment $\Gamma \vdash_X \mathtt{e} : \mathtt{t}$ is defined by the same rules as in Figure 2, except for foreign operators, for which we take:

$$\frac{}{\Gamma \vdash_X \mathtt{o}^{\varepsilon}_{\vec{\varepsilon}} : \varepsilon_1 \to \ldots \to \varepsilon_n \to \varepsilon}$$

Typing environment and type schemes that are used in the judgment $\vdash_X$ are allowed to contain foreign type variables. We say that ϕ is a **pre-solution** to the typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$ if the assertion $\Gamma\phi \vdash_X \mathtt{e}\phi : \mathtt{t}\phi$ holds. Of course, the new rule for foreign operators forgets the constraints that relates the input and output types of foreign operators. In order to ensure type soundness, we must also enforce these constraints.

Formally, we define a **constraint** $\mathtt{C}$ as a finite set of annotated foreign operators $\mathtt{o}^{\varepsilon}_{\vec{\varepsilon}}$. We write $\Vdash \mathtt{C}$ if all the elements of $\mathtt{C}$ are of the form $\mathtt{o}^{\tau}_{\vec{\tau}}$ with $\hat{\mathtt{o}}(\vec{\tau}) \leq \tau$. For an expression $\mathtt{e}$, we collect in a constraint $\mathtt{C}(\mathtt{e})$ all the instances of foreign operators $\mathtt{o}^{\varepsilon}_{\vec{\varepsilon}}$ that appear in $\mathtt{e}$. Note that for any substitution ϕ, we have $\mathtt{C}(\mathtt{e})\phi = \mathtt{C}(\mathtt{e}\phi)$.

We are ready to rephrase the notion of solution.

Lemma 1. *A substitution ϕ is a solution to the typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$ if and only if the following three assertions hold:*

- *$\Gamma\phi$, $\mathtt{e}\phi$ and $\mathtt{t}\phi$ do not contain foreign type variables;*
- *ϕ is a pre-solution to the typing problem;*
- $\Vdash \mathtt{C}(\mathtt{e}\phi)$.

Type soundness Type soundness for our calculus is a trivial consequence of the type soundness assumption for the ML(X) fragment. Indeed, we can see a solution ϕ to a typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$ as an *elaboration* into a well-typed program in this fragment.

Type inference Let us consider a fixed typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$. We want to find solutions to this problem. Thanks to Lemma 1, we will split this task into two different steps:

- find a most-general pre-solution ϕ_0;
- instantiate the remaining foreign type variables so as to satisfy the resulting constraint.6

$\varepsilon ::=$	Foreign types:	$\mathtt{e} ::=$	Expressions:
τ	ground foreign type	$\mathtt{x}$	program variable
ι	foreign type variable	$\lambda\mathtt{x}.\mathtt{e}$	abstraction
		$\mathtt{e}\ \mathtt{e}$	application
$\mathtt{t} ::=$	Types:	$\mathtt{let\ x = e\ in\ e}$	local definition
$\mathtt{P}\,\vec{\mathtt{t}}$	constructed	$(\mathtt{e} : \mathtt{t})$	annotation
α	type variable	$\exists\alpha.\mathtt{e}$	existential variable
ε	foreign type	$\mathtt{o}^{\varepsilon}_{\vec{\varepsilon}}$	foreign operator

Figure 1: Types and expressions

$$\frac{\Gamma(\mathtt{x}) \ll \mathtt{t}}{\Gamma \vdash \mathtt{x} : \mathtt{t}} \qquad \frac{\Gamma, \mathtt{x} : \mathtt{t}_1 \vdash \mathtt{e} : \mathtt{t}_2}{\Gamma \vdash \lambda\mathtt{x}.\mathtt{e} : \mathtt{t}_1 \to \mathtt{t}_2} \qquad \frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{t}_1 \to \mathtt{t}_2 \qquad \Gamma \vdash \mathtt{e}_2 : \mathtt{t}_1}{\Gamma \vdash \mathtt{e}_1\mathtt{e}_2 : \mathtt{t}_2} \qquad \frac{\Gamma \vdash \mathtt{e}_1 : \mathtt{t}_1 \qquad \Gamma, \mathtt{x} : \mathtt{Gen}_\Gamma(\mathtt{t}_1) \vdash \mathtt{e}_2 : \mathtt{t}_2}{\Gamma \vdash \mathtt{let\ x = e}_1\ \mathtt{in\ e}_2 : \mathtt{t}_2}$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{t}}{\Gamma \vdash (\mathtt{e} : \mathtt{t}) : \mathtt{t}} \qquad \frac{\Gamma \vdash \mathtt{e}[\mathtt{t}_0/\alpha] : \mathtt{t}}{\Gamma \vdash \exists\alpha.\mathtt{e} : \mathtt{t}} \qquad \frac{\hat{\mathtt{o}}(\vec{\tau}) \leq \tau}{\Gamma \vdash \mathtt{o}^{\tau}_{\vec{\tau}} : \tau_1 \to \ldots \to \tau_n \to \tau}$$

Figure 2: Type system for the ML(X) fragment

It is almost straightforward to adapt unification-based existing algorithms for ML type inference (and their implementations) to compute a most general pre-solution if there exists a pre-solution, or to report a type error otherwise. Indeed, the typing judgment $\vdash_X$ is very close to a normal ML type system. In particular, it satisfies a substitution lemma: if $\Gamma \vdash_X \mathtt{e} : \mathtt{t}$, then $\Gamma\phi \vdash_X \mathtt{e}\phi : \mathtt{t}\phi$ for any substitution ϕ.

Of course, if the typing problem has no pre-solution, it has no solution as well. For the remaining of the discussion, we assume given a most general pre-solution ϕ_0. Let us write V for the set of foreign type variables that appear in $(\Gamma\phi_0, \mathtt{e}\phi_0, \mathtt{t}\phi_0)$ and $\mathtt{C}_0$ for the constraint $\mathtt{C}(\mathtt{e}\phi_0)$.

A solution to the typing problem is in particular a pre-solution. As a consequence, a substitution ϕ is a solution if and only if $\phi = \phi \circ \phi_0$ and if it maps foreign type variables in V to ground foreign types in such a way that $\Vdash \mathtt{C}_0\phi$. The "minimal" modification we need to bring to ϕ_0 to get a solution is to instantiate variables in V so as to validate $\mathtt{C}_0$. Formally, we define a **valuation** as a function $\rho : V \to \mathcal{T}$ such that $\Vdash \mathtt{C}_0\rho$. To any valuation ρ, we can associate a solution ϕ defined by $\mathtt{t}\phi = \mathtt{t}\phi_0\rho$ and any solution is less general than the solution obtained this way from some valuation. In particular, a solution exists if and only if a valuation exists. So we are now looking for a valuation.

We won't give a *complete* algorithm to check for the existence of a valuation. This would lead to difficult constraint solving problems which might be undecidable (this of course depends on the extension X). Even if they are decidable for a given extension, they might be intractable in practice and so we prefer to stick to our design guideline that type inference shouldn't be significantly more complicated than both ML type inference and XDuce-like type inference. XDuce computes in a bottom-up way, for each sub-expression, a type which over approximates all the possible outcomes of this sub-expression. The basic operations and their typing discipline corresponds respectively to our foreign operators and their static semantics. XDuce's type system uses subsumption only when necessary (e.g. to join the types of the branches of a pattern matching, or when calling a function). So we can say that XDuce tries to compute a minimal type for each sub-expression, by applying basic type-checking primitives. We will do the same, and to make it work, we need some acyclicity property, which corresponds to the bottom-up structure of XDuce's type checker.

Definition 2. *Let* $\mathtt{C}$ *be a constraint. We write* $\iota_1 \overset{\mathtt{C}}{\rightsquigarrow} \iota_2$ *if* $\mathtt{C}$ *contains an element* $\mathtt{o}^{\varepsilon}_{\vec{\varepsilon}}$ *such that* $\iota_2 = \varepsilon$ *and* ι_1 *appears in* $\vec{\varepsilon}$*. We say that* $\mathtt{C}$ *is* **acyclic** *if the directed graph defined by this relation is acyclic.*

Our type inference algorithm only deals with the case of an acyclic constraint $\mathtt{C}_0$ (this condition does not depend on the particular choice of the most general pre-solution). If the condition is not met, we issue an error message. It is not a type error with respect to the type system, but a situation where the algorithm is incomplete.

Remark. *The acyclicity criterion is of course syntactical (it does not depend on the semantics of constraints but on their syntax), but it is not defined in terms of a specific inference algorithm. Instead, it is defined in terms of the most-general pre-solution of an ML-like type system. In particular, it does*

not depend on implementation details such as the order in which sub-expression are type-checked.

Below we furthermore assume that $\mathtt{C}_0$ is acyclic. We define the function $\rho_0 : V \to \mathcal{T}$ by the following equation:

$$\forall \iota \in V.\ \iota\rho_0 = \bigsqcup\{\hat{\mathtt{o}}(\vec{\varepsilon}\rho_0) \mid \mathtt{o}^{\iota}_{\vec{\varepsilon}} \in \mathtt{C}_0\}$$

The acyclicity condition ensures that this definition is well-founded and yields a unique function ρ. Furthermore, this function is a valuation if and only if the typing problem has a solution. To check this property, only constraints whose right-hand side is a ground foreign type need to be considered:

$$(1)\quad \forall \mathtt{o}^{\tau}_{\vec{\varepsilon}} \in \mathtt{C}_0.\ \hat{\mathtt{o}}(\vec{\varepsilon}\rho_0) \leq \tau$$

Also, any other valuation ρ is such that:

$$\forall \iota \in V.\ \iota\rho_0 \leq \iota\rho$$

In other words, under the acyclicity condition, we can check in a very simple way whether a given typing problem has a solution, and if this is the case, we can compute the smallest valuation (for the point-wise extension of the subtyping relation). This computation only involves one call to the abstract semantics for each application of a foreign operator in the expression to be type-checked.

Remark. *In some cases, it is possible to find manifest type errors even when the constraint is not acyclic. In practice, the computation of ρ_0, the verification of (1), and the check for acyclicity can be done in parallel, e.g. with a deep-first graph traversal algorithm. It can detect some violation of (1) before a cycle. In this case, we know that the typing problem has no solution, and thus a proper type error can be issued.*

Manually working around the incompleteness When the algorithm described above infers a cyclic constraint, it cannot detect whether the typing problem $(\Gamma, \mathtt{e}, \mathtt{t})$ has a solution or not. However, we have the following property. If a solution ϕ exists, then we can always produce an expression $\mathtt{e}'$ by adding annotations to $\mathtt{e}$ such that the algorithm succeeds for the typing problem $(\Gamma, \mathtt{e}', \mathtt{t})$ and that ϕ is equivalent (for the equivalence induced by the more-general ordering) to the solution ϕ_0 computed by the algorithm.

In other words, even if the algorithm is not complete (because of the acyclicity condition) and makes a choice between most-general solutions (the smallest one for the subtyping relation), for any solution to a typing problem, the programmer can always add annotations so that the algorithm infers this very solution (or an equivalent one).

Partial operators The foreign operators were assumed to be total. This means they should apply to any foreign value. We can simulate partial operators by adding a new top element $\top$ to the set of ground foreign types $\mathcal{T}$, and by requiring the abstract semantics of operators to be such that whenever an argument is $\top$, the result is also $\top$. Since the typing algorithm infers the smallest valuation for foreign type variables, we can simply look at it and check that no foreign type variable is mapped to $\top$.

5 Strengthening

As we mentioned above, we can see the type system for the calculus as an elaboration into its $\mathrm{ML}(X)$ fragment, which immediatly gives type soundness.

In this section, we consider another elaboration from the calculus into itself. Namely, this elaboration is intended to be used as a preprocessing pass (rewriting expressions into expressions) in order to make the type system accept more programs. We call this elaboration procedure strengthening.

The issue addressed by strengthening is a lack of implicit subsumption in our calculus. We already hinted at this issue in Section 2. We will now give more examples.

Subsumption missing in action We consider the typing problem $(\Gamma_1, \mathtt{e}_1, \beta)$ where $\Gamma_1 = \{\mathtt{x} : \tau_1, \mathtt{y} : \tau_2, \mathtt{f} : \forall\alpha.\ \alpha \to \alpha \to \alpha\}$ and $\mathtt{e}_1 = \mathtt{f\ x\ y}$. It admits a solution if and only if $\tau_1 = \tau_2$. In a system with implicit subtyping, we might expect to give type $\tau = \tau_1 \sqcup \tau_2$ to both $\mathtt{x}$ and $\mathtt{y}$, so that the application succeeds and the result type is τ.

Similarly, the expression $(\lambda \mathtt{x}.\mathtt{x} : \tau_1 \to \tau_2)$ is not well-typed even if $\tau_1 \leq \tau_2$ (unless $\tau_1 = \tau_2$).

A naive solution Let us see how to implement the amount of implicit subtyping we need to make these examples typecheck. The following rule could be a reasonable candidate as an addition to the type system (we write $\vdash_{\leq}$ for the new typing judgment):

$$\frac{\Gamma \vdash_{\leq} \mathtt{e} : \tau \qquad \tau \leq \tau'}{\Gamma \vdash_{\leq} \mathtt{e} : \tau'}$$

A concrete way to see this rule is that any subexpression $\mathtt{e}'$ can be magically transformed to the application $\mathtt{id}^{\iota_2}_{\iota_1}\ \mathtt{e}'$, where $\mathtt{id}$ is a distinguished foreign operator such that $\hat{\mathtt{id}}(\tau) = \tau$ and ι_1, ι_2 are fresh foreign type variables.

The type system extended with this rule would accept the examples given above to illustrate the lack of implicit subsumption. However, this rule as it stands would add a lot

of complexity to the type inference algorithm. As a matter of fact, the type system would not admit most-general pre-solutions anymore. We can see this on a very simple example with the typing problem $(\{\mathtt{x} : \tau\}, \mathtt{x}, \alpha)$. We could argue that a more liberal definition of being more-general should allow some dose of subtyping. So let us consider the more complex example $\Gamma_3 = \{\mathtt{f} : \forall\alpha.\ \alpha \to \alpha \to \alpha\}$ and $\mathtt{e}_3 = \lambda\mathtt{x}.\lambda\mathtt{y}.\lambda\mathtt{z}.\lambda\mathtt{g}.\mathtt{g}\ (\mathtt{f}\ \mathtt{x}\ \mathtt{y})\ (\mathtt{f}\ \mathtt{x}\ \mathtt{z})$. In ML, the inferred type scheme would be $\forall\alpha, \beta.\alpha \to \alpha \to \alpha \to (\alpha \to \alpha \to \beta) \to \beta$ which forces the first three arguments to have the same type. But if the arguments turn out to be of a foreign-type, another family of types for the function is possible, namely $\forall\beta.\tau_1 \to \tau_2 \to \tau_3 \to ((\tau_1 \sqcup \tau_2) \to (\tau_1 \sqcup \tau_3) \to \beta) \to \beta$, and these types cannot be obtained as instances of the ML type scheme above (we can obtain $\forall\beta.\tau_1 \to \tau_2 \to \tau_3 \to ((\tau_1 \sqcup \tau_2 \sqcup \tau_3) \to (\tau_1 \sqcup \tau_2 \sqcup \tau_3) \to \beta) \to \beta$ but this is less precise).

A practical solution We will now describe a practical solution. Instead of modifying the type system by adding a new subsumption rule, we will formulate the extension as a rewriting preprocessing pass. The rewriting consists in inserting applications of the identity foreign operator `id`. The challenge is then to choose which sub-expressions $\mathtt{e}'$ should be rewritten to $\mathtt{id}\ \mathtt{e}'$. If we had an oracle to tell us so, the composition of the rewriting pass and the type system of Section 4 would be equivalent to the type system $\vdash_{\leq}$. Unfortunately, we don't have such an oracle. We could try all the possible choices of sub-expressions, and this would give a complete type-checking algorithm for the type system $\vdash_{\leq}$.

We prefer to use a computationally simpler solution. We also expect it to be simpler to understand by the programmer. The idea is to use an incomplete oracle. The oracle first runs a special version of an ML type-checker on the expression to be type-checked. This type-checker identifies all the foreign types together. The effect is to find out which sub-expressions have a foreign type in a principal derivation, that is, which sub-expression have necessarily a foreign type in all the possible derivations. The preprocessing pass consists in adding an application of the identity operator above all these sub-expressions and only those.

The important point here is that the oracle may be overly conservative. Let us consider a type variable which has been generalized in the principal derivation. In a non-principal derivation, it could have been instead instantiated to a foreign type. If this derivation had been considered instead of the principal one, the preprocessing pass would have added more applications of the identity operator. Maybe this would have been necessary in order to make the resulting expression type-check. An example is given by the expression $\mathtt{let\ h} = \mathtt{e}_3\ \mathtt{in}\ (\mathtt{h} : \tau_1 \to \tau_2 \to \tau_3 \to ((\tau_1 \sqcup \tau_2) \to (\tau_1 \sqcup \tau_3) \to \mathtt{t}) \to \mathtt{t})$ where $\mathtt{e}_3$ is from the example above. Here, the preprocessing pass succeeds but does not change the expression because no sub-expression has a foreign type in the principal type derivation. The type-scheme inferred for `h` is a pure ML type-schema, which makes the type-system subsequently fail on the expression.

We believe that this restriction of the $\vdash_{\leq}$ system is reasonnable. It can be implemented very simply by reusing the same type-checker as in Section 4 in a different mode (where all the foreign types can be unified). The simple examples at the beginning of this section are now accepted. Indeed, the preprocessing pass transforms the expressions to $\mathtt{f}\ (\mathtt{id}\ \mathtt{x})\ (\mathtt{id}\ \mathtt{y})$ and $((\lambda\mathtt{x}.\mathtt{id}\ \mathtt{x}) : \tau_1 \to \tau_2)$ respectively. This allows the type system $\vdash$ to use subtyping where needed.

Properties The strenghtening pass cannot transform a well-typed program into an ill-typed one. Note, however, that it might break the acyclicity condition if it was already met. See below for a way to relax the acyclicity condition.

Also, if strenghtening fails, the typing problem has no pre-solution (for the typing judgment $\vdash$), and thus no solution. However, it is not true that if it succeeds, a pre-solution necessarily exists (for the new program where applications of the `id` operators have been added). As an example, let us consider the situation where $\Gamma = \{\mathtt{x} : \tau_1 \to \tau_1, \mathtt{y} : \tau_2 \to \tau_2, \mathtt{f} : \forall\alpha.\ \alpha \to \alpha \to \alpha\}$ and $\mathtt{e} = \mathtt{f}\ \mathtt{x}\ \mathtt{y}$. The preprocessing succeeds, because all the foreign types are considered equal but does not touch the expression (because no sub-expression has a foreign type in a principal typing derivation). Still, the next pass of the type inference algorithm attempts to unifiy the types τ_1 and τ_2 and thus fails.

Relaxing the acyclicity condition Inserting applications of the `id` operator can break the acyclicity condition. We can actually relax this condition to deal with the `id` operator more carefully. Let us consider a constraint $\mathtt{C}$ with a cycle $\iota_1 \overset{\mathtt{C}}{\rightsquigarrow} \ldots \overset{\mathtt{C}}{\rightsquigarrow} \iota_1$, such that all the edges in this cycle come from elements of the form $\mathtt{id}_\iota^{\iota'}$. Clearly, any valuation ρ such that $\Vdash \mathtt{C}\rho$ will map all the ι_i in the cycle to the same ground foreign type. So instead of considering the most-general pre-solution and then face a cyclic constraint, we may as well unify all these ι_i first: all the solutions can still be obtained from this less-general pre-solution.

The relaxed condition is: There must be no cycle in the constraint except maybe cycles whose edges are all produced by the `id` operator.

To illustrate the usefulness of the relaxed condition, let us consider the expression $\mathtt{e} = \mathtt{fix}(\lambda\mathtt{g}.\lambda\mathtt{x}.\mathtt{f}\ \mathtt{c}\ (\mathtt{g}\ \mathtt{x}))$ with $\Gamma =$

$\{\texttt{fix} : \forall\alpha.(\alpha \to \alpha) \to \alpha, \texttt{f} : \forall\alpha.\alpha \to \alpha \to \alpha, \texttt{c} : \tau\}$. The strengthening pass builds a principal typing derivation for `e` in a type algebra where all the foreign types are identified. Here is such a derivation, where we write $\star$ for foreign types and $\texttt{t} = \alpha \to \star$, $\Gamma' = \Gamma, \texttt{g} : \texttt{t}, \texttt{x} : \alpha$ (we collapse rules for multiple abstraction and application):

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash \texttt{fix} : (\texttt{t} \to \texttt{t}) \to \texttt{t}}}{\Gamma' \vdash f : \star \to \star \to \star} \quad \dfrac{\dfrac{}{\Gamma' \vdash g : \texttt{t}} \quad \dfrac{}{\Gamma' \vdash x : \alpha}}{\Gamma' \vdash g\ x : \star}}{\dfrac{\dfrac{}{\Gamma' \vdash c : \star} \quad \Gamma' \vdash g\ x : \star}{\Gamma' \vdash f\ c\ (g\ x) : \star}}$$
$$\dfrac{\Gamma \vdash \lambda\texttt{g}.\lambda\texttt{x}.\texttt{f}\ \texttt{c}\ (\texttt{g}\ \texttt{x})) : \texttt{t} \to \texttt{t}}{\Gamma \vdash \texttt{e} : \alpha \to \star}$$

On this principal derivation, we observe three sub-expressions of a foreign type. Accordingly, strengthening introduces three instances of the `id` operator and thus rewrites the expression to:

$$\texttt{e}' = \texttt{fix}(\lambda\texttt{g}.\lambda\texttt{x}.\texttt{id}^{\iota_2}_{\iota_1}\ (\texttt{f}\ (\texttt{id}^{\iota_4}_{\iota_3}\ \texttt{c})\ (\texttt{id}^{\iota_6}_{\iota_5}\ (\texttt{g}\ \texttt{x}))))$$

The type-checker which is then applied performs some unifications: $\iota_1 = \iota_4 = \iota_6$, $\iota_2 = \iota_5$, $\iota_3 = \tau$. We can for instance assume that the computed most-general pre-solution maps ι_4 and ι_6 to ι_1 and ι_5 to ι_2. The first and third instances of the `id` operator in $\texttt{e}'$ thus generate the dependencies $\iota_1 \overset{\texttt{C}_0}{\rightsquigarrow} \iota_2$ and $\iota_2 \overset{\texttt{C}_0}{\rightsquigarrow} \iota_1$. Strictly speaking, the constraint is cyclic, but we can break the cycle simply by unifying ι_1 and ι_2. The smallest valuation is then given by $\iota_1\rho = \tau$. We would have obtained the same solution if we had applied the type-checker directly on `e` without the strengthening pass. In this example, strengthening is useless and the relaxed acyclicity condition is just a way to break a cycle introduced by strenghtening. We can easily imagine more complex examples where strenghtening is really necessary but introduces cycles that can be broken by the relaxed condition.

6 Integration in OCaml

We have described a type system for basic ML expressions. Of course, OCaml is much more than an ML kernel. We found no problem to extend it to deal with the whole OCaml type system, including recursive types, modules, classes, and other fancy features. The two ML-like typing passes (the one used during strengthening and the one using for the real type-checking) are done on whole compilation units (in the toplevel, they are done on each phrase). The information from the compilation unit interface (the `.cmi` file) is integrated before checking the acyclicity condition. Indeed, this information acts as additional type annotations on the values exported by the compilation unit and can thus contribute to obtaining this condition. Also, in addition to type annotations on expressions, OCaml provides several ways to introduce explicit type informations (and thus obtain the acyclicity condition): datatype definitions (explicit types for constructor and exception arguments, record fields), module signatures, type annotations on ML pattern variables.

OCaml subtyping OCaml comes with a structural subtyping relation (generated by object types and polymorphic variants subtyping and extended structurally by considering the variance of ML type constructors). The use of this subtyping relation in programs is explicit. The syntax is $(\texttt{e} : \texttt{t}_1 :> \texttt{t}_2)$ (sometimes, the type $\texttt{t}_1$ can be inferred) and it simply checks that $\texttt{t}_1$ is a subtype of $\texttt{t}_2$. Of course, the OCaml subtyping relation has been extended in OCamlDuce to take XDuce subtyping into account. For instance, if τ_1 is a XDuce subtype of τ_2 and `e` has type τ_1`list`, then it is possible to coerce it to type τ_2 `list`: $(\texttt{e} :> \tau_2\ \texttt{list})$.

Crossing the boundary In our system, XDuce values are opaque from the point of view of ML and XDuce types cannot be identified with other ML type constructors. Sometimes, we need to convert values between the two worlds. For instance, we have a foreign type `String` which is different from OCaml `string`. This foreign type conceptually represents immutable sequences of arbitrary Unicode characters, whereas the OCaml type should be thought as representing mutable buffers of bytes. As a consequence, we don't even try to collapse these two types into a single one. Instead, OCamlDuce comes with a runtime library which exports conversion functions such as `Utf8.make: string -> String`, `Utf8.get: String -> string`, `Latin1.make: string -> Latin1`, `Utf8.get: Latin1 -> string`. The type `Latin1` is a subtype of `String`: it represents all the strings which are only made of latin-1 characters (latin-1 is a subset of the Unicode character set). The function `Utf8.make` checks at runtime that the OCaml string is a valid representation of a Unicode string encoded in utf-8.

Similarly, we often need to translate between XDuce's sequences and OCaml's lists. For any XDuce type τ, we can easily write two functions of types $[\tau*] \to \tau$ `list` and τ `list` $\to [\tau*]$ (the star between square brackets denotes Kleene-star). Similarly, we can imagine a natural XDuce counterpart of an OCaml product type $\tau_1 \times \tau_2$, namely $[\tau_1\ \tau_2]$, and coercion functions. However, writing this kind of coercions by hand is tedious. OCamlDuce comes with

built-in support to generate them automatically. This automatic system relies on a structural translation of *some* OCaml types into XDuce types: lists and array are translated to Kleene-star types, tuples are translated to finite-length XDuce sequences, variant types are translated to union types, etc. Some OCaml types such as polymorphic or functional types cannot be translated. OCamlDuce comes with two magic unary operators `to_ml`, `from_ml` (both written {: ... :} in the concrete syntax). The first one takes an XDuce value and applies a structural coercion to it in order to obtain an OCaml value; this coercion is thus driven by the output type of the operator. The type-checker requires this type to be fully known (polymorphism is not allowed). Similarly, the operator `from_ml` takes an OCaml value and apply a structural coercion in order to obtain an XDuce value. Since the type of its input drives its behavior, the type-checker requires this type to be fully known.

This system can be used to obtain coercions from complex OCaml types (e.g. obtained from big mutually recursive definitions of concrete types) to XDuce types, whose values can be seen as XML documents. This gives parsing from XML and pretty-printing to XML for free.

7 Related work

The CDuce language itself comes with a typed interface with OCaml. The interface was designed to: (i) let the CDuce programmers use existing OCaml libraries; (ii) develop hybrid projects where some modules are implemented in OCaml and other in CDuce. The interface is actually quite simple: each monomorphic OCaml type `t` is mapped in a structural way to a CDuce type $\hat{\texttt{t}}$. A value defined in an OCaml module can be used from CDuce (the compiler introduces a natural translation $\texttt{t} \rightarrow \hat{\texttt{t}}$). Similarly, it is possible to provide an ML interface for a CDuce module: the CDuce compiler checks that the values exported by the module are compatible with the ML-to-CDuce translation of these types and produces stub code to apply a natural translation $\hat{\texttt{t}} \rightarrow \texttt{t}$ to these values. This CDuce/OCaml interface is used by many CDuce users and served as a basis to the `to_ml` and `from_ml` operators described in Section 6.

Sulzmann and Zhuo Ming Lu [SL05] pursue the same objective of combining XDuce and ML. However, their contribution is orthogonal to ours. Indeed, they propose a compilation scheme from XDuce into ML such that the ML representation of XDuce values is driven by their static XDuce type (implicit use of subtyping are translated to explicit coercions). Their type system supports in addition used-defined coercions from XDuce types to ML types. However, they do not describe a type inference algorithm for their abstract specification of a type system and do not study the interaction between XDuce type-checking and ML type inference (XDuce code can call ML functions but their type must be fully known). These last points are precisely the issues tackled by our contribution. For instance, our system makes it possible to avoid some type annotation on non-recursive XDuce functions. Another difference is that in our approach, the XDuce/CDuce type checker and back-end (compilation of pattern matching) can be re-used without any modification whereas their approach requires a complete reengineering of the XDuce part (because subtyping and pattern matching relations must be enriched to produce ML code) and it is not clear how some XDuce features such as the `Any` type can be supported in a scenario of modular compilation. We believe our approach is more robust with respect to extensions of XDuce and that the XDuce-to-ML translation can be seen as an alternative implementation technique for XDuce which allows some interaction between XDuce and ML (the same kind of interaction as what can be achieved with the CDuce/OCaml interface described above).

The Xtatic project [GP03] is another example of the integration of XDuce types into a general purpose language, namely C#. Since both C#'s and XDuce's type checkers operate with bottom-up propagation (explicit types for functions/methods, no type inference), the structure of Xtatic type-checker is quite simple. The real theoretical contribution is in the definition of a subtyping relation which combines C# named subtyping (inheritance) and XDuce set-theoretic subtyping. Since the resulting type algebra does not have least-upper bounds, the nice locally-complete type inference algorithm for XDuce patterns [HP02] cannot be transferred to Xtatic. In Xtatic, XDuce types and C# types are stratified, but the two algebras are mutually recursive: XDuce types can appear in class definitions and C# classes can be used as basic items in XDuce regular expression types. This does not really introduce any difficulty because C# types are not structural. The equivalent in OCamlDuce would be to allow OCaml abstract types as part of XDuce types, which would not be difficult, except for scoping reasons (abstract types are scoped by the module system).

In the last ten years, a lot of research effort has been put into developping type inference techniques for extensions of ML with subtyping and other kinds of constraints. For instance, the HM(X) framework [MOW99] could serve as a basis to express the type system presented here. The main modification to bring to HM(X) would be to make foreign-type variables global. Another way to express it is to disallow constraints in type-schemes (which is what we do in the current presentation). We have chosen to present our system in a setting closer to ML so as to make our message more explicit: our system can be easily implemented on top of existing ML implementations.

8 Conclusion and future work

We have presented a simple way to integrate XDuce into OCaml. The modification to the ML type-system is small enough so as to make it possible to easily extend existing ML type-checkers.

Realistic-sized examples of code have been written in OCamlDuce, such as an application that parses XML Schema documents into an internal OCaml form and produces an XHTML summary of its content. Compared to a pure OCaml solution, this OCamlDuce application was easier to write and to get right: XDuce's type system ensures that all possible cases in XML Schema are treated by pattern-matching and that no invalid XHTML output can be produced). We refer the reader to OCamlDuce's website for the source code of this application.

The main limitation of our approach is that it doesn't allow parametric polymorphism on XDuce types. Adding polymorphism to XDuce is an active research area. In a previous work with Hosoya and Castagna [HFC05], we presented a solution where polymorphic functions must be explictly instantiated. Integrating this kind of polymorphism into the same mechanism as ML polymorphism is challenging and left for future work. The theory recently developped by Vouillon [Vou06] could be a relevant starting point for such a task.

Another direction for improvement is to further relax the acyclicity conditions, that is, to accept more programs without requiring extra type annotations. Once the set of constraints representing XML data flow and operations have been extracted by the ML type-checker, we could use techniques which are more involved than simple forward computation over types. The static analysis algorithm used in Xact [KMS04] could serve as a starting point in this direction.

Acknowledgments The author would like to thank Didier Rémy and François Pottier for fruitful discussion about the design and formalization of type systems.

References

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM International Conference on Functional Pr ogramming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.

[Dam85] Luis Manuel Martins Damas. *Type assignment in programming languages*. PhD thesis, University of Edinburgh, Scotland, April 1985.

[Fri04] Alain Frisch. *Théorie, conception et réalisation d'un langage de programmatio n fonctionnel adapté à XML*. PhD thesis, Université Paris 7, December 2004.

[GP03] Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darms tadt, Germany*, 2003.

[HFC05] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. In *POPL*, 2005.

[HM03] Haruo Hosoya and Makoto Murata. Boolean operations and inclusion test for attribute-element constraint s. In *Eighth International Conference on Implementation and Application of Automata*, 2003.

[Hos00] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, Japan, December 2000.

[Hos04] Haruo Hosoya. Regular expression filters for XML. In *Programming Languages Technologies for XML (PLAN-X)*, 2004.

[HP00] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Data bases (WebDB2000)*, 2000.

[HP02] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 13(4), 2002.

[HP03] Haruo Hosoya and Benjamin C. Pierce. A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000.

[KMS04] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[L+01] Xavier Leroy et al. The Objective Caml system release 3.08; Documentation and user's manual, 2001.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 1978.

[MOW99] Martin Sulzmann Martin Odersky and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1), 1999.

[SL05] Martin Sulzmann and Kenny Zhuo Ming Lu. A type-safe embedding of XDuce into ML. In *The 2005 ACM SIGPLAN Workshop on ML*, 2005.

[Vou06] Jérôme Vouillon. Polymorphic regular tree types and patterns. In *POPL*, 2006. To appear.

Polymorphism and XDuce-style patterns

Jérôme Vouillon
CNRS and Université Paris 7
Jerome.Vouillon@pps.jussieu.fr

Abstract

We present an extension of XDuce, a programming language dedicated to the processing of XML documents, with polymorphism and abstract types, two crucial features for programming in the large. We show that this extension makes it possible to deal with first class functions and eases the interoperability with other languages. A key mechanism of XDuce is its powerful pattern matching construction and we mainly focus on this construction and its interaction with abstract types. Additionally, we present a novel type inference algorithm for XDuce patterns, which works directly on the syntax of patterns.

1. Introduction

XDuce [14] is a programming language dedicated to the processing of XML documents. It features a very powerful type system: types are regular tree expressions [15] which correspond closely to the schema languages used to specify the structure of XML documents. The subtyping relation is extremely flexible as it corresponds to the inclusion of tree automata. Another key feature is a pattern matching construction which extends the algebraic patterns popularized by functional languages by using regular tree expressions as patterns [13].

In this paper, we aim at extending in a seamless way the XDuce type system and pattern construction with ML-style prenex polymorphism and abstract types. These are indeed crucial features for programming in the large in a strongly typed programming language. In our extension, patterns are not allowed to break abstraction. This crucial property makes it possible to embed first class functions and foreign values in a natural way into XDuce values.

In another paper [21], we present a whole calculus dealing with polymorphism for regular tree types. Though most of the results in that paper (in particular, the results related to subtyping) can be fairly easily adapted for an extension of XDuce, a better treatment of patterns is necessary. Indeed, a straightforward application of the results would impose severe restrictions on patterns. For instance, binders and wildcards would be required to occur only in tail position. The present paper is therefore mostly focused on patterns and overcomes these limitations.

Additionally, we present a novel type inference algorithm for XDuce patterns, which works directly on the syntax of patterns, rather than relying on a prior translation to tree automata. This way, better type error messages can be provided, as the reported types are closer to the types written by the programmer. In particular, type abbreviations can be preserved, while they would be expanded by the translation into tree automata.

The paper is organized as follows. We introduce the XDuce type system (section 2) and present the extension (section 3). Then, we formalize patterns (section 4) and provide algorithms for checking patterns and performing type inference (section 5). Related works are presented in section 6.

2. A Taste of XDuce

XDuce values are *sequences* of elements, where an *element* is characterized by a *name* and a *contents*. (Elements may also contain *attributes*, both in XDuce and XML. We omit attributes here for the sake of simplicity.) This contents is itself a sequence of elements. These values corresponds closely to XML documents, such as this address book example.

```
<addrbook>
   <person>
      <name> Haruo Hosoya </name>
      <email> hosoya </email>
   </person>
   <person>
       <name> Jerome Vouillon </name>
      <tel> 123 </tel>
   </person>
</addrbook>
```

XDuce actually uses a more compact syntax, which we also adopt in this paper:

```
addrbook[
  person[name["Haruo Hosoya"], email["hosoya"]],
  person[name["Jerome Vouillon"], tel["123"]]]
```

The shape of values can be specified using *regular expression types*. A sequence of elements is described using a regular expression. Mutually recursive type definitions make it possible to deal with the nested nature of values. Here are the type definitions for address books.

```
type Addrbook = addrbook[Person*]
type Person   = person[Name,Email*,Tel?]
type Name     = name[String]
type Email    = email[String]
type Tel      = tel[String]
```

These type definitions can be read as follows. An `Addrbook` value is an element with name `addrbook` containing a sequence of any number of `Person` values. A `Person` value is an element with name `person` containing a `Name` value followed by a sequence of `Email` values and optionally a `Tel` value. Values of type `Name`, `Email`, and `Tel` are all composed of a single element containing a string of characters.

There is a close correspondence between regular expression types and tree automata [5]. As the inclusion problem between tree automata is decidable, the subtyping relation can be simply defined as language inclusion [15]. This subtyping relation is extremely powerful. It includes associativity of concatenation (type `A,(B,C)` is equivalent to type `(A,B),C`), distributivity rules (type `A,(B|C)` is equivalent to type `(A,B)|(A,C)`).

In order to present the next examples, we find it convenient to use the following parametric type definition for lists:

```
type List{X} = element[X]*
```

Parametric definitions are not currently implemented in XDuce, but are a natural extension and can be viewed as just syntactic sugar: all occurrences of `List{T}` (for any type `T`) can simply be replaced by the type `element[T]*` everywhere in the source code.

Another key feature of XDuce is *regular expression patterns*, a generalization of the algebraic patterns popularized by functional languages such as ML. These patterns are simply types annotated with *binders*. Consider for instance this function which extracts the names of a list of persons.

```
fun names (lst : Person*) : List{String} =
   match lst with
     () →
        ()
   | person [name [nm : String], Email*, Tel?],
     rem : Person* →
        element [nm], names (rem)
```

The function `names` takes an argument `lst` of type `Person*` and returns a value of type `List{String}`. The body of the function is a pattern matching construction. The value of the argument `lst` is matched against two patterns. If it is the empty sequence, then it will match the first pattern `()` (the type `()` is the type of the empty sequence `()`), and the function returns the empty sequence. Otherwise, the value must be a non-empty sequence of type `Person*`. Thus, it is an element of name `person` followed by a sequence of type `Person*`, and matches the second pattern. This second pattern contains two binders `nm` and `rem` which are bound to the corresponding part of the value.

Some *type inference* is performed on patterns: the type of the expression being matched is used to infer the type of the values that may be bound to a binder. By taking advantage of this, the function `names` can be rewritten more concisely using *wildcard* patterns[1] as follows. The type of the binders `nm` and `rem` are inferred to be respectively `String` and `Person*` by the compiler.

```
fun names (l : Person*) : List{String} =
   match l with
     () →
        ()
   | person [name [nm : _], _], rem : _ →
        element [nm], names (rem)
```

3. Basic Ideas

We want to extend regular expression types and patterns with ML-style polymorphism (with explicit type instantiation) and abstract types. Such an extension is interesting for numerous reasons. First, it makes it possible to describe XML documents in which arbitrary subdocuments can be plugged. A typical example is the SOAP envelop. Here is the type of SOAP messages and of a function that extracts the body of a SOAP message.

```
type Soap_message{X} =
   envelope[header[...], body[X]]
fun extract_body :
   forall{X}. Soap_message{X} → X
```

A more important reason is that polymorphism is crucial for programming in the large. It is intensively used for collection datastructures. As an example, we present a generic map function over lists. This function has two type parameters `X` and `Y`.

```
fun map{X}{Y}
       (f : X → Y)(l : List{X}) : List{Y} =
   match l with
     () →
        ()
   | element[x : _], rem : _ →
        element[f(x)], map{X}{Y}(f)(rem)
```

When using a polymorphic function, type arguments may have to be explicitly given, as shown in the following expression where the `map` function is applied to the identity function on integers and to the empty list:

```
map{Int}{Int} (fun (x : Int) → x) ().
```

Indeed, it is possible to infer type arguments in simple cases, using an algorithm proposed by Hosoya, Frisch and Castagna [12], but not in general, as a best type argument does not necessarily exist: the problem is harder in our case due to function types which are contravariant on the left.

Abstract types facilitate interoperability with other languages. Indeed, we can consider any type from the foreign language as an abstract type as far as XDuce is concerned. For instance, the ML type[2] `int` can correspond to some XDuce type `Int`. This generalizes to parametric abstract types: to the ML type `int array` would correspond the polymorphic XDuce type `Array{Int}`. Furthermore, if the two languages share the same representation of functions, ML function types can be mapped to XDuce function types (and conversely). Thus, for instance, a function of type `int→int` can be written in either language and used directly in the other language without cumbersome conversion.

In order to preserve abstraction and to deal with foreign values that may not support any introspection, some patterns should be disallowed. For instance, this function should be rejected by the type checker as it tries to test whether a value `x` of some abstract type `Blob` is the empty sequence.

```
fun f (x : Blob) : Bool =
   match x with
     () → true
   | _  → false
```

Another restriction is that abstract types cannot be put directly in sequences. Indeed, it does not make sense to concatenate two values of the foreign language (two ML functions, for instance). In order to be put into a sequence, they must be wrapped in an element. As a type variable may be instantiated to an abstract type, and as we want to preserve abstraction for type variables too, the same restrictions apply to them: a pattern `a[],X,b[]` implicitly asserts that the variable `X` stands for a sequence, and thus would limit its polymorphism.

There are different ways to deal with type variables and abstract types occurring syntactically in patterns. The simplest possibility is not to allow them. Instead, one can use wildcards and rely on type inference to assign polymorphic types to binders. This approach is taken in the related work by Hosoya, Frisch and Castagna [12]. Another possibility is to consider that type variables should behave as the actual types they are instantiated to at runtime. This is a natural approach, but this implies that patterns do not preserve abstraction. It is also challenging to implement this efficiently, though it may be possible to get good results by performing pattern compilation (and optimization) at run-time. Finally, it is not clear in this case how abstract types should behave in patterns. We propose a middle-ground, by restricting patterns so that their behaviors do not depend on what type variables are instantiated to, and on what abstract

[1] XDuce actually uses the pattern `Any` as a wildcard pattern.

[2] We consider here ML as the foreign language, as XDuce is currently implemented in OCaml. But this would apply equally well to other languages.

types stand for. In other words, patterns are not allowed to break abstraction. As a consequence, type variables can be compiled as wildcards. In other words, type variables and abstract types occurring in patterns can be considered as annotations which are checked at compile time but have no effect at run-time. We indeed feel it is interesting to allow type variables and abstract types in patterns. A first reason is that it is natural to use patterns to specify the parameters of a functions. And we definitively want to put full types there. For instance, we should be able to write such a function:

```
fun apply{X}{Y}
    (funct[f : X -> Y], arg[x : X]) : Y = f(x)
```

Another reason is that one may want to reuse a large type definition containing abstract types in a pattern, and it would be inconvenient to have to duplicate this definition, replacing abstract types with wildcards. Finally, the check can be implemented easily: the type inference algorithm can be used to find the type of the values that may be matched against any of the type variables occurring in the pattern, so one just has to check that this type is a subtype of the type variable (this usually means that the type is either empty or equal to the type variable, but some more complex relations are possible, as we will see in section 4.3).

4. Specifications

We now specify our pattern matching construction, starting from the data model, continuing with types and patterns, before finally dealing with the whole construction.

4.1 Values

We assume given a set of *names* l and a set of *foreign values* e. A *value* v is either a foreign value or a *sequence* f of *elements* $l[v]$ (with name l and *contents* v).

v	$::=$	e	foreign value
		f	sequence
f	$::=$	$l[v], \ldots, l[v]$	

We write ϵ for the *empty sequence*, and f, f' for the *concatenation* of two sequences f and f'.

Note that strings of characters can be embedded in this syntax by representing each character `c` as an element whose name is this very character and whose contents is empty: $\texttt{c}[\epsilon]$. This encoding was introduced by Gapeyev and Pierce [9].

4.2 Patterns

We start by two comments clarifying the specification of patterns. First, in all the examples given up to now, in a pattern element `L[T]`, the construction `L` stands for a single name. It actually corresponds in general to a set of names. This turns out to be extremely convenient in practice. For instance, this can be used to define character sets (remember that characters are encoded as names). Second, abstract types and type variables are very close notions. Essentially, the distinction is a difference of scope: an abstract type stands for a type which is unknown to the whole piece of code considered, while a type variable has a local scope (typically, the scope of a function). Thus, for patterns, we can unify both notions. Parametric abstract types can be handled by considering each of their instances as a distinct type variable. Thus, the two types `Array{Int}` and `Array{Bool}` correspond each to a distinct type variable in our formalization of patterns. Similarly, each function type `T2->T1` corresponds to a distinct type variable. We explain in section 4.3 how subtyping can be expressed for these types.

As a running example, we consider the pattern matching code in function `map`:

```
match l with
    () -> ...
  | element[x : _], rem : List{X} -> ...
```

where `l` has type `List{X}`.

Such a grammar-based syntax of patterns is convenient for writing patterns but typically does not reflect their internal representation in a compiler. For instance, it assumes a notion of pattern names (such as `List{X}` or `Name`) which may be expanded away at an early stage by the compiler. Binders may also be represented in a different way. Finally, this notation is not precise about subpattern identity: for instance, in the pattern `a[_]|b[_]`, it in not clear whether one should consider the two occurrences of the wildcard pattern `_` as two different subpatterns, or as a single subpattern. The distinction matters as a compiler usually does not identify expressions which are structurally equal. In particular, one should be careful not to use any termination argument that relies on structural equality. Another reason is that we need to be able to specify precisely how a value is matched by a pattern. This is especially important for type inference (section 4.9), where we get a different result depending on whether we infer a single type for both occurrences of the wildcard pattern or a distinct type for each occurrence.

Thus, we define a more abstract representation of patterns which provides more latitude for actual implementations. A pattern is a rooted labeled directed graph. Intuitively, this graph can be understood as an in-memory representation of a pattern: nodes stands for memory locations and edges specify the contents of each memory location. To be more accurate, a pattern is actually a hypergraph, as edges may connect a node to zero, one or several nodes: for instance, for a pattern `()`, there is an (hyper)edge with source the location of the whole pattern and with no target, while for a pattern `P,Q`, there is an (hyper)edge connecting the location of the whole pattern to the location of subpatterns `P` and `Q`.

We assume given a family of *name sets* L, a set of *type variables* X and a set $\mathcal{X}$ of *binders* x. Formally, a pattern is a quadruple $(\Pi, \phi, \pi_0, \mathcal{B})$ of

- a finite set Π of *pattern locations* π;
- a mapping $\phi : \Pi \to \mathcal{C}(\Pi)$ from pattern locations to *pattern components* $p \in \mathcal{C}(\Pi)$, defined below;
- a *root* pattern location $\pi_0 \in \Pi$.
- a relation $\mathcal{B} \subseteq \mathcal{X} \times \Pi$ between binders and pattern locations.

Pattern components $\mathcal{C}(\Pi)$ are defined by the following grammar, parameterized over the set Π of pattern locations.

p	$::=$	$L[\pi]$	element pattern
		ϵ	empty sequence pattern
		π, π	pattern concatenation
		$\pi \cup \ldots \cup \pi$	pattern union
		$\pi*$	pattern repetition
		$\Box$	wildcard
		X	type variable

Binders do not appear directly in patterns. Instead they are specified by a relation between binder names and pattern locations. This allows us to simplify significantly the presentation of the different algorithms on patterns. Indeed, most of them simply ignore binders.

As an example, the two patterns:

`()` and `element[x:_],rem:List{X}`

can be formally specified respectively as:

$$(\Pi, \phi, 1, \emptyset) \quad \text{and} \quad (\Pi, \phi, 2, \{(\texttt{x}, 4), (\texttt{rem}, 5)\})$$

where the set of pattern locations is:

$$\Pi = \{1, 2, 3, 4, 5, 6, 7\}$$

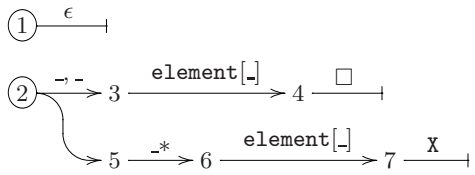


Figure 1. Graphical Depiction of Two Patterns

$$\frac{\phi(\pi) = \pi', \pi''}{\pi' \text{ seq}} \qquad \frac{\phi(\pi) = \pi', \pi''}{\pi'' \text{ seq}} \qquad \frac{\phi(\pi) = \pi'*}{\pi' \text{ seq}}$$

$$\frac{\pi \text{ seq} \qquad \phi(\pi) = \pi_1 \cup \ldots \cup \pi_n}{\pi_i \text{ seq}}$$

Figure 2. Locations in a Sequence π seq

$$\frac{\phi(\pi) = L[\pi']}{\pi \text{ wf}} \qquad \frac{\phi(\pi) = \epsilon}{\pi \text{ wf}} \qquad \frac{\pi' \text{ wf} \qquad \pi'' \text{ wf} \qquad \phi(\pi) = \pi', \pi''}{\pi \text{ wf}}$$

$$\frac{\pi_1 \text{ wf} \qquad \ldots \qquad \pi_n \text{ wf} \qquad \phi(\pi) = \pi_1 \cup \ldots \cup \pi_n}{\pi \text{ wf}} \qquad \frac{\phi(\pi) = \square}{\pi \text{ wf}}$$

$$\frac{\neg(\pi \text{ seq}) \qquad \phi(\pi) = X}{\pi \text{ wf}} \qquad \frac{\pi' \text{ wf} \qquad \phi(\pi) = \pi'*}{\pi \text{ wf}}$$

Figure 3. Location Well-Formedness π wf

and the mapping from pattern locations to pattern components is the function ϕ defined by:

$$\begin{array}{lll} \phi(1) = \epsilon & & \\ \phi(2) = 3, 5 & \phi(3) = \texttt{element}[4] & \phi(4) = \square \\ \phi(5) = 6* & \phi(6) = \texttt{element}[7] & \phi(7) = \texttt{X} \end{array}$$

(We write `element` for the name set containing only the name `element`.) A graphical depiction of the formal representation of the two patterns is given in figure 1. The two root locations 1 and 2 are circled. Edges are labeled with the corresponding component. One can see three kind of edges on this picture: the edges with labels ϵ, $\square$ and `X` have no target; one edge with label `_, _` has two targets 3 and 5 and corresponds to the component 3, 5; some edges with label `_*` or `element[_]` has a single target. Note that the locations 5 to 7 correspond to the expansion of type `List{X}`.

Not all patterns are correct. The most important restriction is that cycles are not allowed except when going through an element $L[\pi']$: for instance, the pattern

```
Balanced = a[], Balanced, b[]
```

should be rejected, while the pattern

```
Tree = leaf[] | node[Tree, Tree]
```

is accepted. This restriction ensures that the set of values matching a given pattern is a regular tree language[3]. The other restriction is that pattern variables should not occur in sequences. For instance, the patterns `a[],X` and `X*`, where `X` is a pattern variable, are rejected. Indeed, the semantics of a pattern variable may contain foreign values, which cannot be concatenated. These two restrictions are formally specified using a well-formedness condition. First, we define when a pattern location is in a sequence (figure 2). Then, we define the well-formedness condition for pattern locations (figure 3). There is one rule per pattern component. For all rules but one, in order to deduce that a pattern location is well-formed, one must first show that its subpatterns are themselves well-formed. This ensures that there is no cycle. The exception is the rule for element patterns $L[\pi']$, hence cycles going through elements are allowed. The rule for type variables X additionally requires that the pattern location is not in a sequence. Finally, a pattern is *well-formed* if all its locations are. These restrictions could also have been enforced syntactically [11, 17], but we prefer to keep the syntax as simple and uniform as possible. In the remainder of this paper, all patterns are implicitly assumed to be well-formed.

4.3 Typing Environments

In order to provide a semantics to patterns, we assume given a class of binary relations between values and types variables, which we call typing environments. Equivalently, we can consider a typing environment as a function from type variables to their semantics which is a set of values). We have two motivations for restricting ourselves to a class of such relations rather than allowing all relations. First, some type variables may have a fixed semantics, identical in all typing environments. This makes it possible to define the type of a function `T2→T1` (assuming that `T1` and `T2` are pure regular expression types, without type variables). The semantics of some type variables may also be correlated to the semantics of other type variables. For instance, the semantics of the type `Array{X}` depends on the semantics of the type variable `X`. Second, for semantic reasons, the semantics of any type, and thus the semantics of type variables, may be required to satisfy some closure properties. This is the case for instance in the ideal model [16].

4.4 Pattern Matching

In order for the algorithms presented in this paper to be implementable, the family of name sets L should be chosen so that the following predicates are decidable:

- the inclusion of a name in a name set: $l \lhd L$;
- the non-emptiness of a name set: $\lhd L$ (that is, there exists a name l such that $l \lhd L$);
- the non-disjointness of two name sets: $L_1 \bowtie L_2$ (that is, there exists a name l such that $l \lhd L_1$ and $l \lhd L_2$).

Furthermore, for technical reasons (see section 4.7), there must be a name set $\top$ containing all names.

The semantics of a pattern $(\text{II}, \phi, \pi_0, \mathcal{B})$ is given in figure 4 using inductive rules. It it parameterized over a typing environment, that is a relation $v \lhd X$ which provides a semantics to each type variable. We define simultaneously the relation $v \lhd \pi$ (the value v matches the pattern location π) and a relation $f \lhd_* \pi$ (the sequence f matches a repetition of the pattern location π). Then, a value v matches a whole pattern if it matches its root location, that is, $v \lhd \pi_0$.

A *match* of a value v against a location π is a derivation of $v \lhd \pi$. Given such a match, we define the *submatches* as the set of assertions $v' \lhd \pi'$ which occur in the derivation. These submatches indicate precisely which parts of the value is associated to each

[3] Actually this is not quite accurate due to type variables. In order to state the regularity property precisely, the semantics of patterns should be defined in two steps. The first step would be a semantics in which values contain variables matching the variables in the pattern. With this initial semantics, the denotation of a pattern would indeed be a regular tree language. The second step would correspond in substituting values for the type variables.

$$\textsc{Match-Element}\ \frac{l \lhd L \qquad v \lhd \pi' \qquad \phi(\pi) = L[\pi']}{l[v] \lhd \pi} \qquad \textsc{Match-Eps}\ \frac{\phi(\pi) = \epsilon}{\epsilon \lhd \pi}$$

$$\textsc{Match-Concat}\ \frac{f' \lhd \pi' \qquad f'' \lhd \pi'' \qquad \phi(\pi) = \pi', \pi''}{f', f'' \lhd \pi}$$

$$\textsc{Match-Union}\ \frac{v \lhd \pi_i \qquad \phi(\pi) = \pi_1 \cup \ldots \cup \pi_n}{v \lhd \pi} \qquad \textsc{Match-Wild}\ \frac{\phi(\pi) = \square}{v \lhd \pi}$$

$$\textsc{Match-Abstract}\ \frac{v \lhd X \qquad \phi(\pi) = X}{v \lhd \pi} \qquad \textsc{Match-Rep}\ \frac{f \lhd_* \pi' \qquad \phi(\pi) = \pi' *}{f \lhd \pi}$$

$$\textsc{Match-Rep-Eps}\ \epsilon \lhd_* \pi \qquad \textsc{Match-Rep-Once}\ \frac{f \lhd \pi}{f \lhd_* \pi} \qquad \textsc{Match-Rep-Concat}\ \frac{f \lhd_* \pi \qquad f' \lhd_* \pi}{f, f' \lhd_* \pi}$$

Figure 4. Matching a Value against a Pattern $v \lhd \pi$

location in the pattern. They can thus be used to define which value to associate to each binder during pattern matching.

We choose to use a non-deterministic semantics: there may be several ways to match a value against a given pattern. The reasons are twofold. First, this yields much simpler specifications and algorithms. Second, we don't want to commit ourselves to a particular semantics. Indeed, we may imagine that the programmer is allowed to choose between different semantics, such as a first-match policy (Perl style) or a longest match policy (Posix style). Our algorithms will be sound in both cases, without any adaptation needed.

4.5 Types

A pattern specifies a set of values: the set of values which matches this pattern. So, patterns can be used as types. More precisely, we define a *type* as a pattern $(\Pi, \phi, \pi_0, \mathcal{B})$ with no wildcard $\square$ (that is, $\phi(\pi)$ is different from the wildcard $\square$ for all locations $\pi \in \Pi$) and no binder (the relation $\mathcal{B}$ is empty).

The wildcard has a somewhat ambiguous status: it stands for any value when not in a sequence, but only stands for sequence values when it occurs inside a sequence. For instance, the values accepted by a pattern `_,P` are not the concatenations of the values accepted by pattern `_` and pattern `P`, as some values in pattern `_` cannot be concatenated. Due to this ambiguous status, type inference would be more complicated if the wildcard pattern was allowed in types.

4.6 Subtyping

We define a subtyping relation $<:$ in a semantic way on the locations $\pi_1 \in \Pi_1$ and $\pi_2 \in \Pi_2$ of two patterns $P_1 = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$ and $P_2 = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2)$ by $\pi_1 <: \pi_2$ if and only if, for all typing environments and for all values v, the assertion $v \lhd \pi_1$ implies the assertion $v \lhd \pi_2$. Two patterns are in a subtype relation, written $P_1 <: P_2$, if their root locations are. The actual algorithmic subtyping relation used for type checking does not have to be as precise as this semantics subtyping relation. This will simply result in a loss of precision.

4.7 Bidirectional Automata and Disallowed Matchings

In the previous section, the semantics of patterns is specified in a declarative way. In order to clarify the operational semantics

$$l[v], f \xrightarrow{\mathsf{l}} l,v,f \qquad\qquad f, l[v] \xrightarrow{\mathsf{r}} l,v,f$$

Figure 5. Value Decomposition $v \longrightarrow^{\delta} l,v,v$

$$\textsc{Label-Trans}\ \frac{\sigma \xrightarrow{\delta} L,\sigma_1,\sigma_2 \qquad v \xrightarrow{\delta} l,v_1,v_2 \qquad l \lhd L \qquad v_1 \in \sigma_1 \qquad v_2 \in \sigma_2}{v \in \sigma}$$

$$\textsc{Eps-Trans}\ \frac{\sigma \rightsquigarrow \sigma' \qquad v \in \sigma'}{v \in \sigma} \qquad \textsc{Accept}\ \frac{\sigma \downarrow v}{v \in \sigma}$$

Figure 6. Automaton Semantics $v \in \sigma$

of patterns, we now define a notion of tree automata, which we call *bidirectional automata*. These automata are used in particular to specify which patterns should be rejected. They capture the idea that a value is matched from its root and that a sequence is matched one element at a time from its extremities. Still, some freedom is left over the implementation. In particular, the automata do not mandate any specific strategy (such as left to right) for the traversal of sequences. This is achieved thanks to an original feature of the automata: at each step of their execution, the matched sequence may be consumed from either side. This symmetry in the definition of automata results in symmetric restrictions on patterns: if a pattern is disallowed, then the pattern obtained by reversing the order of all elements in all sequences is also disallowed. We believe this is easier to understand for a programmer. Additionally, this feature is a key ingredient for our type inference algorithm.

Formally, a bidirectional automaton is composed of

- a finite set Σ of *states* σ;
- an *initial state* $\sigma_0 \in \Sigma$;
- a set of *labeled transitions* $\sigma \longrightarrow^{\delta} L,\sigma,\sigma$;
- a set of *epsilon transitions* $\sigma \rightsquigarrow \sigma$;
- an *immediate acceptance relation* $\sigma \downarrow v$.

The transitions are annotated by a tag $\delta \in \{\mathsf{l}, \mathsf{r}\}$ which indicates on which side of the matched sequence they take place: either on the left (tag `l`) or on the right (tag `r`). The semantics of automata is given in figure 6: the relation $v \in \sigma$ specifies when a value v is *accepted* by a state σ of the automaton. A value is accepted by a whole automaton if it is accepted by its initial state σ_0. The rule LABEL-TRANS states that, starting from a goal $v \in \sigma$, a labeled transition $\sigma \longrightarrow^{\delta} L,\sigma_1,\sigma_2$ may be performed provided that the value v decomposes itself on side δ into a element with name l and contents v_1 followed by a value v_2 (value decomposition is specified in figure 5). The name l must furthermore be included in the name set L. One then gets two subgoals $v_1 \in \sigma_1$ and $v_2 \in \sigma_2$. The rule EPS-TRANS moves to another state of the automaton while remaining on the same part of the value. Usually, automata have a set of accepting states, which all accept the empty sequence ϵ. Here, we use an accepting relation, so that a state may accept whole values at once (rule ACCEPT). This is necessary to deal with type variables `X` that match a possibly non-regular set of values and with foreign values e which are not sequences. The use of an epsilon transition relation simplify the translation from patterns to automata. It also keeps the automata smaller. Indeed, eliminating epsilon transitions may make an automaton quadratically larger. Note that our automata are non-deterministic.

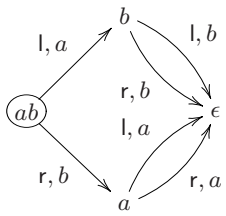


Figure 7. Example of Bidirectional Automaton

Not all patterns could be translated into deterministic automata as top down deterministic tree automata are strictly less powerful than non-deterministic ones [5].

An example of bidirectional automaton is depicted in figure 7. This automaton recognizes the sequence a[],b[]. It has four states $\Sigma = \{ab, a, b, \epsilon\}$. The initial state ab is circled. The labeled transitions are all of the form $\sigma \longrightarrow^{\delta} L,\epsilon,\sigma'$ and are represented by an arrow from state σ to state σ' with label the pair δ, L. There is no epsilon transition. The acceptance relation, not represented, is reduced to $\epsilon \downarrow \epsilon$. In order to recognize the sequence a[],b[], one can first consume a[] from the left and then the remaining part b[] from either side, or consume the part b[] before the part a[].

The automata we build below satisfy some commutation properties, which ensure that the strategy used to match a value is not important. For instance, one can choose to consume values only from the left, or only from the right, or any combination of these two strategies. In all cases, the set of accepted values remain the same. We do not state these properties.

We now specify the translation of a pattern $(\Pi, \phi, \pi_0, \mathcal{B})$ into an automaton. This translation is inspired by some algorithms by Hopkins [10] and Antimirov [2] for building a non-deterministic automaton using *partial derivatives* of a regular expression. The way we apply the same operations symmetrically on both sides of a pattern is inspired by Conway's *factors* [6, 18]. At the root of all these works is Brzozowski's notion of regular expression *derivatives* [3].

The key idea for the translation is that each state corresponds to a regular expression that exactly matches what is accepted by the state, and a transition corresponds to a syntactic transformation of a regular expression into the regular expression of the next state. In our case, one may have expected pattern locations to take the role of regular expression. As they are not flexible enough, we actually use finite sequences of pattern locations (plus some non-binding variants). Thus, a state σ of the automaton is defined by the following grammar.

s	::=	π	single pattern
		$*\pi$	non-binding pattern repetition
		$\diamond$	non-binding wildcard
σ	::=	$[s; \ldots; s]$	pattern sequence

We write $[\,]$ for the empty pattern sequence, and $\sigma;\sigma'$ for the concatenation of the pattern sequences σ and σ'. The intuition behind non-binding variants is the following. Suppose we match a value `a[],a[],a[]` against a pattern `A*`. As we will see, this pattern reduces to something akin to `A,A*` by epsilon transitions. According to the semantics of patterns, the beginning of the value `a[]` is indeed bound to the location of subpattern `A`, but the remaining part `a[],a[]` is not bound to any location. Thus, the subpattern `A*` does not correspond to a pattern location, but rather to a repetition of the location of the subpattern `A`.

The initial state of the automaton is the sequence $[\pi_0]$ containing only the root π_0 of the pattern. The epsilon transitions, the labeled transitions and the immediate acceptance relation are respectively

$$\textsc{Dec-Eps}\ \frac{\phi(\pi) = \epsilon}{[\pi] \overset{\delta}{\rightsquigarrow} [\,]} \qquad \textsc{Dec-Concat}\ \frac{\phi(\pi) = \pi', \pi''}{[\pi] \overset{\delta}{\rightsquigarrow} [\pi'; \pi'']} \qquad \textsc{Dec-Union}\ \frac{\phi(\pi) = \pi_1 \cup \ldots \cup \pi_n}{[\pi] \overset{\delta}{\rightsquigarrow} [\pi_i]}$$

$$\textsc{Dec-Rep}\ \frac{\phi(\pi) = \pi'*}{[\pi] \overset{\delta}{\rightsquigarrow} [*\pi']} \qquad \textsc{Dec-Wildcard}\ \frac{\phi(\pi) = \square}{[\pi] \overset{\delta}{\rightsquigarrow} [\diamond]}$$

$$\textsc{Dec-Rep-Left}\ [*\pi] \overset{\mathsf{l}}{\rightsquigarrow} [\pi; *\pi] \qquad \textsc{Dec-Rep-Right}\ [*\pi] \overset{\mathsf{r}}{\rightsquigarrow} [*\pi; \pi] \qquad \textsc{Dec-Rep-Eps}\ [*\pi] \overset{\delta}{\rightsquigarrow} [\,]$$

$$\textsc{Dec-Wildcard-Eps}\ [\diamond] \overset{\delta}{\rightsquigarrow} [\,]$$

$$\textsc{Dec-Left}\ \frac{\sigma \overset{\mathsf{l}}{\rightsquigarrow} \sigma'}{\sigma;\sigma'' \overset{\mathsf{l}}{\rightsquigarrow} \sigma';\sigma''} \qquad \textsc{Dec-Right}\ \frac{\sigma \overset{\mathsf{r}}{\rightsquigarrow} \sigma'}{\sigma'';\sigma \overset{\mathsf{r}}{\rightsquigarrow} \sigma'';\sigma'}$$

Figure 8. Epsilon Transitions $\sigma \rightsquigarrow^{\delta} \sigma$

$$\frac{\phi(\pi) = L[\pi']}{[\pi] \overset{\delta}{\longrightarrow} L,[\pi'],[\,]} \qquad [\diamond] \overset{\delta}{\longrightarrow} \top,[\diamond],[\diamond]$$

$$\frac{\sigma \overset{\mathsf{l}}{\longrightarrow} L,\sigma_1,\sigma_2}{\sigma;\sigma' \overset{\mathsf{l}}{\longrightarrow} L,\sigma_1,(\sigma_2;\sigma')} \qquad \frac{\sigma \overset{\mathsf{r}}{\longrightarrow} L,\sigma_1,\sigma_2}{\sigma';\sigma \overset{\mathsf{r}}{\longrightarrow} L,\sigma_1,(\sigma';\sigma_2)}$$

Figure 9. Labeled Transitions $\sigma \longrightarrow^{\delta} L,\sigma,\sigma$

$$[\,] \downarrow \epsilon \qquad [\diamond] \downarrow e \qquad \frac{v \triangleleft X \qquad \phi(\pi) = X}{[\pi] \downarrow v}$$

Figure 10. Immediate Acceptance Relation $\sigma \downarrow v$

defined in figure 8, 9, and 10. The assertion $\sigma \rightsquigarrow \sigma$ holds when either assertion $\sigma \rightsquigarrow^{\mathsf{l}} \sigma$ or $\sigma \rightsquigarrow^{\mathsf{r}} \sigma$ holds. Note that the definition of the immediate acceptance relation depends on the typing environment.

As there is an infinite number of sequences σ, we define the finite set of states Σ of the automaton as the set of sequences reachable from the initial state $[\pi_0]$ of the automaton through the transitions. The following lemma states that we define this way a finite automaton.

LEMMA 1 (Finite Automaton). *The number of states σ reachable from the initial state $[\pi_0]$ through the transition relations is finite.*

The number of states can however be exponential in the size of the pattern due to sharing. A typical example is the type definitions below.

```
type T = a[],a[] and U = T,T and V = U,U
```

We expect the state of the automata to be reasonable in practice. Indeed, for patterns without sharing of locations, the bound is much better: it is quadratic in the size of the pattern.

An example of translation is given in figure 11. The pattern `a[],b[]` is represented using the same notation as in figure 1. For the sake of simplicity, we do not represent the part of the automa-

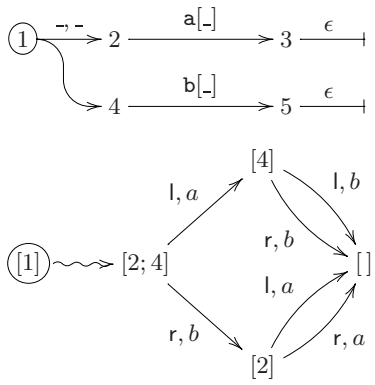


Figure 11. Pattern and its Translation (Simplified)

$$\frac{\phi(\pi) \text{ is a test}}{e \not\downarrow [\pi]} \qquad \frac{\sigma \rightsquigarrow \sigma' \qquad v \not\downarrow \sigma'}{v \not\downarrow \sigma}$$

$$\frac{\sigma \xrightarrow{\delta} L,\sigma_1,\sigma_2 \qquad v \xrightarrow{\delta} l,v_1,v_2 \qquad l \lhd L \qquad v_1 \not\downarrow \sigma_1}{v \not\downarrow \sigma}$$

$$\frac{\sigma \xrightarrow{\delta} L,\sigma_1,\sigma_2 \qquad v \xrightarrow{\delta} l,v_1,v_2 \qquad l \lhd L \qquad v_2 \not\downarrow \sigma_2}{v \not\downarrow \sigma}$$

Figure 12. Disallowed Matching $v \not\downarrow \sigma$

ton corresponding to the element contents. The initial state of the automaton is the sequence $[1]$. An epsilon transition yields from this state to the state $[2; 4]$. Then, values can either be consumed from the left or from the right. The first case correspond to a transition $[2; 4] \longrightarrow^{\mathsf{l}} a,[3],[4]$, depicted as an arrow from state $[2; 4]$ to state $[4]$ with label l, a.

Some matchings of a value against a pattern should not be allowed, either because they are not implementable, or because they would break abstraction. As the automata describe the operational semantics of patterns, they are the right tool to specify which matchings should be rejected. This *disallowed matching* relation $v \not\downarrow \sigma$ is defined in figure 12. Automaton matching can be viewed as a dynamic process: for matching a value v against a pattern sequence σ, we start from the assertion $v \in \sigma$ and try to consume the whole value by applying repeatedly the rules in figure 6. We should never arrive in a position where a test needs to be performed on an external value. Therefore, in the definition of the disallowed matching relation, there is one rule corresponding to epsilon transitions and two rules corresponding to labeled transitions, depending on whether the failure occurs in the element contents or in the sequence but outside this element. The last rule corresponds to an immediate failure, where a test is performed on an external value. The following pattern components are *tests*: $L[\pi]$, ϵ, (π, π), and $\pi*$. Basically, a test is a pattern component that only accepts sequences. For this last rule, we only need to consider the case when the pattern sequence contains a single pattern location. Indeed, one can easily show that the only way to arrive to a sequence which is not of this form is through epsilon transitions, starting from a sequence of this form. This specification of disallowed matchings is quite algorithmic. Still, we are confident it can be understood intuitively by a programmer.

$$\text{MATCH-SEQ-EPS}\quad \epsilon \lhd [\,] \qquad \text{MATCH-SEQ-SINGLE}\quad \frac{v \lhd \pi}{v \lhd [\pi]} \qquad \text{MATCH-SEQ-STAR}\quad \frac{f \lhd_* \pi}{f \lhd [*\pi]}$$

$$\text{MATCH-SEQ-WILCARD}\quad v \lhd [\diamond] \qquad \text{MATCH-SEQ-CONCAT}\quad \frac{f \lhd \sigma \qquad f' \lhd \sigma'}{f, f' \lhd \sigma; \sigma'}$$

Figure 13. Matching a Value against a Pattern Sequence $v \lhd \sigma$

We now relate the semantics of a pattern to the semantics of its translation into an automaton. It is convenient to first extend the semantics of pattern locations to pattern sequences (figure 13). We then have the following result.

LEMMA 2. *A value is matched by a pattern if and only if it is matched by the corresponding automaton, as long as the matching is allowed: if $v \not\downarrow [\pi_0]$ does not hold, then $v \in [\pi_0]$ if and only if $v \lhd \pi_0$. More generally, for any value v and any state σ such that $v \not\downarrow \sigma$ does not hold, we have $v \in \sigma$ if and only if $v \lhd \sigma$.*

The restriction to allowed matchings is important. Indeed, consider the pattern `(),_`. It matches only sequences but it is translated into an automaton that matches everything, as the empty sequence is eliminated by epsilon transitions (rule DEC-CONCAT followed by rule DEC-EPS together with rule DEC-LEFT).

Our automata are actually designed for analyzing patterns rather than for being executed. They make it possible to focus on a particular part of a pattern by consuming subpatterns from both sides. For instance, if we have a pattern `A,B,C`, we can focus on `B` by consuming `A` on the left and `C` on the right. Thus, type inference can be performed by consuming a type and a pattern in a synchronized way in order to find out which parts of the type corresponds to which parts of the pattern. For instance, if we have a type `a[],T,b[]` and a pattern `a[],(x : _),b[]`, we can compute that the type of the variable `x` is `T` by simultaneously consuming the elements `a[]` and `b[]` of the type and the pattern. For this to work, it must be possible to associate a state to each part of a value matched by a pattern. As a consequence, there is a slight mismatch between our definition and what should be an actual implementation of patterns. First, the rule for type variables in the definition of the acceptance relation is important for analyzing patterns but would not be used in an actual implementation, where matching against a type variable should always succeed. Second, when in state $[\diamond]$, only foreign values are immediately accepted while sequence values are progressively decomposed. Thus is crucial for type inference but cannot be implemented: foreign values cannot be tested and thus an implementation cannot adopt a different behavior depending on whether a value is a sequence or a foreign value. A simple change is sufficient to adapt the automaton: make the state $[\diamond]$ accept any value and remove any transition from this state. Note that this change does not affect the disallowed matching relation.

4.8 Pattern Matching Construction

We can now complete our specification of pattern matching. We are only interested in how a value is matched in a pattern matching construction: which branch is selected and which values are associated to the binders in this branch. We do not consider what happens afterwards. Thus, we can ignore the body of each branch of the construction and can formalize a pattern matching construction as a list of patterns. It turns out to be convenient to share between all patterns a set of pattern locations and a mapping from pattern lo-

cation to pattern components. Therefore, a pattern construction is characterized by:

- a set of pattern locations Π;
- a mapping $\phi : \Pi \to \mathcal{C}(\Pi)$;
- a family (π_i) of root pattern locations ($\pi_i \in \Pi$);
- a family $(\mathcal{B}_i)$ of binder relations ($\mathcal{B}_i \subseteq \mathcal{X} \times \Pi$)

The i-th pattern is defined as $P_i = (\Pi, \phi, \pi_i, \mathcal{B}_i)$. For instance, the pattern construction in the body of the function `map` presented in section 4.2 can be specified by reusing the corresponding definitions of the set Π and mapping ϕ and defining (π_i) and $(\mathcal{B}_i)$ by

$$\pi_1 = 1 \qquad \mathcal{B}_1 = \emptyset$$
$$\pi_2 = 2 \qquad \mathcal{B}_2 = \{(\texttt{x}, 4), (\texttt{rem}, 5)\}$$

In order to type-check a pattern construction, the type T of the values that may be matched by the pattern must be known. In our example, this type is `List{X}`, which can be represented as a pattern $(R, \psi, 1, \emptyset)$ with

$$R = \{1, 2, 3\}$$

and

$$\psi(1) = 2* \qquad \psi(2) = \texttt{element}[3] \qquad \psi(3) = \texttt{X}.$$

The semantics of pattern matching is as follows. Given a value v_0 belonging to the input type T, a pattern P_i is chosen such that the value v_0 matches the root location π_i of the pattern, that is, so that there exists a derivation of $v_0 \lhd \pi_i$. We then consider all submatches, that is, all assertions $v \lhd \pi$ which occur in this derivation. This defines a relation $\mathcal{M}$ between locations and values. The composition $\mathcal{M} \circ \mathcal{B} = \{(x, v) \mid \exists \pi.(x, \pi) \in \mathcal{B} \wedge (\pi, v) \in \mathcal{M}\}$ of this relation with the binder relation $\mathcal{B}$ is then expected to be a total function from the set of binders of the pattern to parts of value v. This function indicates which part of the value v is bound to each binder x.

In order to ensure that this matching process succeeds for any value of the input type T, the following checks must be performed:

- *exhaustiveness*: for all typing environments and for all values v in the input type T, there must exists a pattern P_i such that the value v matches the root location π_i of this pattern;
- *linearity*: for all typing environments, for all values v in the input type T and for all derivations $v \lhd \pi_i$ where π_i is the root location of one of the patterns P_i, the composition $\mathcal{M} \circ \mathcal{X}$ defined above must be a function.

These two checks are standard [13]. In our case, two additional checks must be performed. Indeed, some matchings are not allowed in order to preserve abstraction and for the patterns to be implementable. Furthermore, patterns are not implemented directly but only after erasure. We define the *erasure* of a pattern $(\Pi, \phi, \pi_0, \mathcal{B})$ as the pattern $(\Pi, \phi', \pi_0, \mathcal{B})$ where:

$$\phi'(\pi) = \begin{cases} \square & \text{if } \phi(\pi) \text{ is a type variable } X \\ \phi(\pi) & \text{otherwise.} \end{cases}$$

An *erased* pattern is a pattern containing no pattern variable (that is, $\phi(\pi)$ is different from any variable X for all locations π in the pattern). The semantics remain the one given above, but applied to the erased patterns. We thus have these two additional checks:

- *allowed patterns*: the erasure of each pattern P_i should be *allowed with respect to* the input type T, that is, we must not have $v \not\lhd \pi_i$ for any value v in the input type T, any erasure of pattern P_i, and any typing environment.
- *preservation of the semantics*: for all typing environments and for all values v in T, the value v is matched the same way by each pattern P_i and its erasure;

By "matched the same way", we mean that, if there is a derivation of $v \lhd \pi_i$ in one of the patterns P_i, then there must be an identical derivation in the erasure of pattern P_i (except for applications of rule MATCH-ABSTRACT which should be replaced by applications of rule MATCH-WILD), and conversely. Algorithms for performing all these checks are presented section 5. The linearity check algorithm is actually omitted as it is standard and its presentation is long.

4.9 Type Inference

An additional operation we are interested in is type inference: we want to compute for each binder a type which approximates the set of values that may be bound to it. From the semantics of the pattern matching construction above, we can derive the following characterization of this set of values. Consider an input type $(\Pi_1, \phi_1, \pi_1^0, \emptyset)$ and a pattern $(\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2)$. Then, a value v may be bound to a binder x if there exists a value v_0 and a location $\pi \in \Pi_2$ such that:

- $v_0 \lhd \pi_1^0$ (the value v_0 belongs to the input type);
- $v_0 \lhd \pi_2^0$ (the value is matched by the pattern);
- $(x, \pi) \in \mathcal{B}_2$ (the binder x is at location π in the pattern);
- there exists a derivation of $v_0 \lhd \pi_2^0$ containing an occurrence of the assertion $v \lhd \pi$ (the assertion $v \lhd \pi$ is a submatch).

Several algorithms for *precise* type inference have been proposed [7, 13, 20]. These algorithms are tuned to a particular matching policy (such as the first-match policy). With these algorithms, the semantics of the type computed for a binder is exactly the set of values that may be bound to it. (As binders are considered independently, any correlation between them is lost, though.) For instance, let us consider the following function.

```
fun f (x : (a[] | b[] | c[])) =
   match x with
     b[]                    → ...
   | y : (a[] | b[] | d[]) → ...
   | _                      → ...
```

A precise type algorithm infers the type `a[]` for the binder `y`. Indeed, values of type `b[]` are matched by the first line of the pattern. Therefore, only values of type the difference between type `a[]|b[]|c[]` and type `b[]`, that is, type `a[]|c[]` may be matched by the second pattern. Finally, the values matching the second pattern must also have type `a[]|b[]|d[]`, hence their type is the intersection of `a[]|c[]` and `a[]|b[]|d[]`, that is, `a[]`. Such a type algorithm is implemented in CDuce and was initially implemented in XDuce.

Difference is costly to implement. Besides, though this is not apparent in the example above, difference operations may need to be performed at many places in the pattern, especially when binders are deeply nested. Hosoya proposed a simpler design [11], remarking that with a non-deterministic semantics (in other words, when the matching policy is left unspecified) no difference operation needs to be performed. An intersection operation still needs to be performed, but only once per occurrence of a binder. So, in our example, the second line of the pattern still matches values of type `b[]`. Therefore, the type of `y` is the intersection of the initial type `a[]|b[]|c[]` and the type `a[]|b[]`, that is, `a[]|b[]`.

In our case, even the intersection operations must be avoided. Indeed, our types are not closed under intersection: for instance, there is no type that corresponds to the intersection of two type variables. Xtatic has the same issue [8, section 5.3]. The current

implementation of Xtatic thus computes an approximation of the intersection. Another reason to avoid intersection is that it is not a syntactic operation on types in XDuce. Thus, in order to compute an intersection, types must first be translated to automata and the intersection must be translated back from an automaton to a type. In the process, the type may become more complex. In the worst case, the size of the intersection of two automata is quadratic in the size of these automata. Also, some type abbreviations may be lost during the successive translations.

What we propose is to infer types not for binders but for wildcards `_` and compute the type of binders by substitutions. The key idea is that the intersection of a type with a wildcard is the type itself. Thus, no intersection is actually needed. Consider for instance the function below.

```
fun g (x : (a[],b[])) =
   match x with
     y : (_,(b[]|c[])) → ...
```

The type inferred for the wildcard is `a[]`. Thus, by substitution, the type inferred for the binder `y` is `a[],(b[]|c[])`. We deliberately gave an example for which the inferred type is not precise, in order to emphasize the difference with other specifications of type inference. We expect this weaker form of type inference to perform well in practice. In particular, type inference is still precise for wildcards (assuming a non-deterministic semantics). When needed, the programmer can provide explicitly a more precise type. We experimented with the examples provided with the XDuce distribution. Only some small changes were necessary to get them to compile. What we actually had to do was to replace by wildcards some explicit types which were not precise enough.

More formally, we define the *semantics of a location* π of the pattern as the set of values v such that there exists a value v_0 such that the assertions $v_0 \lhd \pi_1^0$ and $v_0 \lhd \pi_2^0$ holds and the assertion $v \lhd \pi$ is a submatch of a derivation of $v_0 \lhd \pi_2^0$. The type inference algorithm then consists in computing for each location corresponding to a wildcard a type whose semantics is the semantics of this location and substituting this type in place of the wildcard. The substitution may not preserve pattern well-formedness. In this case, the type checking fails. But we believe this is unlikely to occur in practice, as this can only happen when a wildcard location is shared in two different contexts. For instance, consider the type `a[X]` and the pattern `a[Q],Q` where `Q` $=$ `_`. The type inferred for the wildcard is `X|()` and substituting this type does not preserve well-formedness. If the resulting pattern is well-formed, then it is a type: it does not contain any wildcard. The type of a binder is the type corresponding to the union of the locations the binder is associated to.

5. Algorithms on Patterns

We define a number of algorithms for type checking and type inference for patterns. Each of these algorithms is specified in an abstract way, by defining a relation over a finite domain using inductive definitions. Actually implementing them is a constraint solving issue. Standard techniques can be used, such as search pruning (when an assertion is either obviously true or obviously false), memoization (so as not to perform the same computation several times), and lazy computation (in order not to compute unused parts of the relation).

The size of the finite domain provides a bound on the complexity of the algorithm. We don't study the precise complexity of these algorithms, as we believe this would not be really meaningful. In particular, the complexity of all these algorithms is polynomial in the sizes of the automata associated to the patterns it operates on, but these sizes can be exponential in the size of the patterns. Our experience on the subject leads us to believe that the algorithms should perform well in practice.

$$\textsc{Root}\quad [\pi_1^0] \smile [\pi_2^0]$$

$$\textsc{Dec-Left}\quad \frac{\sigma_1 \smile \sigma_2 \qquad \sigma_1 \rightsquigarrow \sigma_1'}{\sigma_1' \smile \sigma_2} \qquad \textsc{Dec-Right}\quad \frac{\sigma_1 \smile \sigma_2 \qquad \sigma_2 \rightsquigarrow \sigma_2'}{\sigma_1 \smile \sigma_2'}$$

$$\textsc{Enter}\quad \frac{\begin{array}{c}\sigma_1 \smile \sigma_2 \\ L_1 \bowtie L_2 \qquad \sigma_1 \xrightarrow{\delta} L_1,\sigma_1',\sigma_1'' \\ \sigma_1'' \leftrightarrow \sigma_2'' \qquad \sigma_2 \xrightarrow{\delta} L_2,\sigma_2',\sigma_2''\end{array}}{\sigma_1' \smile \sigma_2'}$$

$$\textsc{Shift}\quad \frac{\begin{array}{c}\sigma_1 \smile \sigma_2 \\ L_1 \bowtie L_2 \qquad \sigma_1 \xrightarrow{\delta} L_1,\sigma_1',\sigma_1'' \\ \sigma_1' \leftrightarrow \sigma_2' \qquad \sigma_2 \xrightarrow{\delta} L_2,\sigma_2',\sigma_2''\end{array}}{\sigma_1'' \smile \sigma_2''}$$

Figure 14. Type Propagation $\sigma \smile \sigma$

5.1 Exhaustiveness

The input of the algorithm is the input type $T = (R, \psi, \pi, \emptyset)$ and the different patterns $P_i = (\Pi, \phi, \pi_i, \mathcal{B}_i)$ of the pattern construction. We define the union of the patterns P_i by $P = (\Pi \cup \{\star\}, \phi', \star, \emptyset)$ where the location $\star$ is assumed not to be in Π and the mapping ϕ' is such that $\phi'(\star) = \pi_1 \cup \ldots \cup \pi_n$ (the union of all root locations) and $\phi'(\pi) = \phi(\pi)$ for $\pi \in \Pi$ One can easily prove that the semantics of the pattern P is the union of the semantics of the patterns P_i. Then, the pattern is exhaustive if and only if $T <: P$.

Note that the union construction above can be applied to any finite set of patterns sharing a common mapping ϕ. This construction is also used for type inference (section 5.6).

5.2 Type Propagation

This algorithm propagates type information in a pattern. It is used both for checking whether a pattern is allowed (section 5.4) and for type inference (section 5.6). The input of the algorithm is composed of two patterns $P_1 = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$ and $P_2 = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2)$ and a relation $\sigma_1 \leftrightarrow \sigma_2$ (where the sequences σ_1 and σ_2 range over the states of the automata associated respectively to P_1 and P_2). The relation controls when the type information is propagated across an element. The algorithm is defined in figure 14 as a relation $\sigma_1 \smile \sigma_2$. The roots of the two patterns are related (rule ROOT). The relation is preserved by epsilon transition (rules DEC-LEFT and DEC-RIGHT). The rules ENTER and SHIFT specify how the relation is propagated to the contents of an element and aside an element.

Though the rules are symmetric, the algorithm is used in an asymmetric way. One of the pattern is actually always a type and the algorithm can be read as propagating type information derived from this type into the other pattern. Besides, we are not interested in computing the whole relation $\sigma_1 \smile \sigma_2$. Rather, for some given sequences σ_1, the set of pattern sequences σ_2 such that $\sigma_1 \smile \sigma_2$ must be computed.

As the algorithm is defined as a binary relation $\sigma_1 \smile \sigma_2$ over the states of the automata associated to the patterns P_1 and P_2, it is quadratic in the size of these automata.

Pattern

$$\frac{\phi(\pi) = L[\pi'] \qquad \lhd L \qquad \lhd \pi'}{\lhd \pi} \qquad \frac{\phi(\pi) = \epsilon}{\lhd \pi}$$

$$\frac{\phi(\pi) = \pi_1, \pi_2 \qquad \lhd \pi_1 \qquad \lhd \pi_2}{\lhd \pi}$$

$$\frac{\phi(\pi) = \pi_1 \cup \ldots \cup \pi_n \qquad \lhd \pi_i}{\lhd \pi} \qquad \frac{\phi(\pi) = \pi'*}{\lhd \pi}$$

$$\frac{\phi(\pi) = \Box}{\lhd \pi} \qquad \frac{\phi(\pi) = X}{\lhd \pi}$$

Pattern sequence

$$\lhd [\,] \qquad \lhd [*\pi] \qquad \lhd [\diamond] \qquad \frac{\lhd \pi}{\lhd [\pi]} \qquad \frac{\lhd \sigma_1 \qquad \lhd \sigma_2}{\lhd \sigma_1; \sigma_2}$$

Figure 15. Non-Emptiness $\lhd \pi$ and $\lhd \sigma$

5.3 Type Non-Emptiness

In order to check whether a pattern is allowed (section 5.4), it turns out that we need an algorithm to decide whether, given a pattern $P = (\Pi, \phi, \pi_0, \mathcal{B})$, the semantics of a pattern location π or a pattern sequence σ (belonging to the set of states of the automaton associated to pattern P) is empty, that is, whether there exists a value v such that $v \lhd \pi$ or $v \lhd \sigma$. These algorithms are defined in figure 15 as two relations $\lhd \pi$ and $\lhd \sigma$. Their properties can be stated as follows.

LEMMA 3. *Let π be a location in pattern P and σ be a state of the automaton associated to pattern P. If there exists a value v such that $v \lhd \pi$, then $\lhd \pi$. Likewise, if there exists a value v such that $v \lhd \sigma$, then $\lhd \sigma$. The converse holds in any typing environment such that for all type variables X there exists at least a value v such that $v \lhd X$.*

The proof of the lemma is straightforward. The reason for the restriction in the converse case can be seen on the last rule in figure 15: if $\phi(\pi) = X$, then we have $\lhd \pi$. We thus need to ensure that there exists a value v such that $v \lhd X$.

The inference rules define a relation $\lhd \pi$ over the finite set of pattern locations Π in pattern P. Each rule can be implemented in constant time. Hence, computing the relation for all locations in a pattern can be done in linear time in the size of the pattern P. Likewise, the relation $\lhd \sigma$ can be computed in linear time in the size of the automaton associated to the pattern P.

5.4 Disallowed Pattern

The algorithm checking whether a pattern $P = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2)$ is allowed with respect to an input type $T = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$ is based on an instance of the type propagation algorithm (section 5.2) applied to the type T and the pattern P. For this instance, we take $\sigma_1 \leftrightarrow \sigma_2$ iff $\lhd \sigma_1$. Intuitively, if we have a type `L[T1],T2` and a pattern `L'[P1],P2` such that the sets `L` and `L'` are not disjoint, the type information `T1` should be propagated in the pattern `P1`, but only if there is indeed a value of type `L[T1],T2`, thus in particular only if the semantics of type `T2` is not empty. On the other hand, as the implementation of the automaton may try to match a value against the subpattern `P1` before considering the subpattern `P2`, nothing should be assumed about `P2`. The algorithm relies on the following theorem.

$$\frac{\sigma_1' \bowtie \sigma_2 \qquad \sigma_1 \overset{\mathsf{l}}{\rightsquigarrow} \sigma_1'}{\sigma_1 \bowtie \sigma_2} \qquad \frac{\sigma_1 \bowtie \sigma_2' \qquad \sigma_2 \overset{\mathsf{l}}{\rightsquigarrow} \sigma_2'}{\sigma_1 \bowtie \sigma_2}$$

$$\frac{\begin{array}{ll}\sigma_1' \bowtie \sigma_2' & \sigma_1 \xrightarrow{\mathsf{l}} L_1, \sigma_1', \sigma_1'' \\ \sigma_1'' \bowtie \sigma_2'' & \sigma_2 \xrightarrow{\mathsf{l}} L_2, \sigma_2', \sigma_2'' \\ \multicolumn{2}{c}{L_1 \bowtie L_2}\end{array}}{\sigma_1 \bowtie \sigma_2}$$

$$[\,] \bowtie [\,] \qquad \frac{\phi(\pi) = X}{[\pi] \bowtie [\diamond]}$$

Figure 16. Non-Disjointness $\sigma \bowtie \sigma$

THEOREM 4. *If the pattern P is disallowed with respect to the type T, then there exists two locations $\pi_1 \in \Pi_1$ and $\pi_2 \in \Pi_2$ such that $\phi(\pi_1)$ is a type variable X, the location π_2 is a test (as defined in section 4.7), and $[\pi_1] \smile [\pi_2]$. The converse holds in any typing environment such that for all type variables X there exists a foreign value e such that $e \lhd X$.*

The restriction in the converse case ensures that the relation $\lhd \sigma$ really coincide with the non-emptiness of the sequence σ. It also ensures that if $\phi(\pi_1) = X$ and π_2 is a test, then there exists a foreign value e such that $e \not\lhd \pi_2$. From this and $[\pi_1] \smile [\pi_2]$, one can then show that the whole pattern is disallowed.

A naïve implementation of the algorithm would compute all pairs of locations π_1 and π_2 such that $[\pi_1] \smile [\pi_2]$ and then checks whether there exists a pair for which both $\phi(\pi_1)$ is a type variable X and the location π_2 is a test. This implementation would have the same complexity as the type propagation algorithm. An immediate optimization is to stop propagating type information whenever a type location π_1 with no type variable below it is reached.

5.5 Non-Disjointness

The type inference algorithm relies on an algorithm that, given a type

$$T = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$$

and an erased pattern

$$P = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2),$$

decides the non-disjointness of the semantics of two pattern sequences σ_1 and σ_2 belonging to the states of the automata associated respectively to the type T and the pattern P. The pattern P is assumed to be allowed with respect to type T. The algorithm is defined in figure 16 as a relation $\sigma_1 \bowtie \sigma_2$. It is based on a standard algorithm for checking non-disjointness of tree automata, with a special case for type variables X. Note that only transitions with tag l are used. The relation would remain unchanged if this restriction was removed.

The intended semantics of the algorithm is that $\sigma_1 \bowtie \sigma_2$ if and only if there exists a value v such that $v \lhd \sigma_1$ and $v \lhd \sigma_2$. However this does not hold for arbitrary sequences σ_1 and σ_2. The completeness of the algorithm can be stated as follows.

LEMMA 5 (Completeness). *Let σ_1 and σ_2 be two states of the automata associated respectively to the type T and the pattern P. We assume that:*

- *the pattern P is allowed with respect to type T;*
- *$\sigma_1 \smile \sigma_2$ (where this relation is the instance defined in section 5.4 for checking for disallowed patterns);*

- *the typing environment is such that for all type variables X there exists at least a value v such that $v \lhd X$.*

Then, if $\sigma_1 \bowtie \sigma_2$, there exists a value v such that $v \lhd \sigma_1$ and $v \lhd \sigma_2$.

The condition $\sigma_1 \smile \sigma_2$ together with the allowed pattern condition ensures that, for instance, one never considers the type sequence $[\pi]$ with $\phi(\pi) = X$ against the pattern sequence $[\diamond; \diamond]$ for which the algorithm may give a wrong answer: one has $[\pi] \bowtie [\diamond; \diamond]$ as the sequence $[\diamond; \diamond]$ reduces by epsilon transition to the sequence $[\diamond]$, but there is not reason for the sequences $[\pi]$ and $[\diamond; \diamond]$ to share a common value. The constraint on typing environments can be easily understood by looking at the rule concerning type variables.

The soundness property is hard to state. Rather than defining precisely when it holds, which would involve defining an additional complex relation, we use the following somewhat imprecise statement.

LEMMA 6 (Soundness). *Let σ_1 and σ_2 be two sequences of the automaton respectively associated to the input type and the input pattern. In any position where the relation $\sigma_1 \bowtie \sigma_2$ is used in the type inference algorithm below (section 5.6), if there exists a value v such that $v \lhd \sigma_1$ and $v \lhd \sigma_2$, then $\sigma_1 \bowtie \sigma_2$.*

The algorithm is quadratic in the size of the automata associated to the type T and the pattern P.

5.6 Type Inference

The input of the type inference algorithm is a type

$$T = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$$

and an erased pattern

$$P = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2).$$

The pattern P is assumed to be allowed with respect to the type T. The algorithm is based on an instance of the type propagation algorithm (section 5.2) applied to the type T and the pattern P. For this instance, we take $\sigma_1 \leftrightarrow \sigma_2$ iff $\sigma_1 \bowtie \sigma_2$. Intuitively, if we have a type `L[T1],T2` and a pattern `L'[P1],P2`, the type information `T1` should be propagated in the pattern `P1`, but only if there is indeed a value of the whole type matched by the whole pattern. In particular, there should be a value shared by the type `T2` and the pattern `P2`. We rely on the following result.

THEOREM 7. *If a value v is included in the semantics of a location π_2 of the pattern, then there exists a sequence σ_1 such that $\sigma_1 \smile [\pi_2]$. The converse holds in any typing environment such that for all type variables X there exists at least a value v such that $v \lhd X$.*

The type inferred for a subpattern π_2 should thus corresponds to the union of the sequences σ_1 such that $\sigma_1 \smile [\pi_2]$. One can show that for any such sequence σ_1, as it belongs to the set of states of an automaton associated to a type, one can build a type T_1 with the same semantics. Then, the type inferred for the subpattern π_2 can be build by taking the union of these types, as defined in section 5.1.

The algorithm is complete only when all type variables have a non-empty semantics. It may be possible to get a stronger result by extending our type system with *conditional types* [1]. But we believe this would unnecessarily complicate the type system.

An interesting feature of this algorithm is that it works directly on the syntax of patterns and types. In particular, the inferred type is build from the input type using only simple operations (concatenation and union).

As in the case of the "disallowed pattern" check (section 5.4), a naïve implementation which would compute the whole relation $\sigma_1 \smile \sigma_2$ can be improved by stopping the propagation of type information whenever one reach a pattern sequence σ_2 with no location whose type needs to be inferred below it.

5.7 Preservation of the Semantics

The input of the type inference algorithm is a type

$$T = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$$

and a pattern

$$P = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2).$$

The algorithm is as follows. For each location π in the pattern P corresponding to a type variable X (that is, $\phi_1(\pi) = X$), a type T' is computed using the type inference algorithm on the erasure of pattern P. We then compare this type with the part of the pattern P corresponding to location π, that is, the pattern $P' = (\Pi_2, \phi_2, \pi, \mathcal{B}_2)$. The semantics is preserved if for all such locations π we have $T' <: P'$. The soundness of the algorithm relies on the following theorem.

THEOREM 8. *The semantics of the pattern is preserved by erasure if and only if for any pattern location π_2 such that $\phi(\pi_2) = X$ (for any type variable X) and any type location σ_1 such that $\sigma_1 \smile \pi_2$ (by type inference on the erasure of pattern P), we have $\sigma_1 <: [\pi_2]$.*

6. Related Works

As mentioned in the introduction, we presented in a previous paper [21] a calculus dealing with polymorphism for regular tree types. Values are binary trees rather than sequences of elements. It is straightforward to translate sequences into binary trees by representing an element contents as a node whose first child is its contents and second child is its right sibling. A similar translation can be defined to some extent for patterns. But there is a number of restrictions. In particular, wildcards and binders should only occur in tail position. The present paper deals directly with sequences, which makes it possible to avoid these restrictions. Additionally, we specify patterns in a more precise way: we believe very few extensions to patterns, besides support for XML attributes, would be necessary for a realistic implementation.

Hosoya, Frisch and Castagna have also proposed an extension of XDuce with polymorphism [12], now implemented in the latest release. In their work, type variables range over sets of basic XDuce values rather than over sets of arbitrary values. This results in design decisions which are drastically different from ours. For instance, they consider that pattern matching on values whose type is a type variable is possible (as the structure of all such values can be explored by pattern matching), while we consider that this would break abstraction. They can deal with bounded quantification. On the other hand, it is not clear how to extend their work to deal with foreign types and higher-order functions.

Sulzmann and Lu propose to use a structured representation of XDuce values [19] and interpret subtyping as a runtime coercion. As types reflect the structure of values, they do not have the issue of concatenating foreign values: the values of type `A,B` are pairs, rather than concatenations of values of type `A` and type `B`. However, they may need to use algorithms similar to ours in order to ensure that pattern matching interact well with polymorphism.

Several type inference algorithms have been proposed for regular expression types. The first one [13], by Hosoya and Pierce, is precise (assuming a first-match policy) but can infer a type only for binders in tail position in the pattern. Hosoya later proposed a simpler design [11], corresponding to a non-deterministic semantics for patterns, where this restriction was removed. Both algorithms use a translation of types and patterns into tree automata. Several algorithms for precise type inference for different match policies

have also been presented by Vansummeren [20]. They work directly on the syntax of patterns but require complex operations on types such as intersection and difference.

CDuce [4] has some extensive support for importing functions from OCaml. Contrary to what we propose in section 3, their extension relies on a runtime translation of ML values into CDuce values according to their types.

References

[1] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1994. ACM Press.

[2] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[3] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[4] CDuce Development Team. *CDuce Programming Language User's Manual*. Available from `http://www.cduce.org/documentation.html`.

[5] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1rst 2002.

[6] J. H. Conway. *Regular Algebra and Finite Machines*. William Clowes and Sons, 1971.

[7] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *17th IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[8] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.

[9] V. Gapeyev and B. C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[10] M. W. Hopkins. Converting regular expressions to non-deterministic finite automata, May 1992. Newsgroup message on comp.theory.

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

[12] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 50–62. ACM Press, 2005.

[13] H. Hosoya and B. C. Pierce. Regular expression pattern matching. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004.

[14] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[15] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[16] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1-2):95–130, 1986.

[17] M. Murata. Hedge automata: a formal model for XML schemata. `http://www.xml.gr.jp/relax/hedge_nice.html`, 2000.

[18] G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. *SIGPLAN Not.*, 39(1):26–38, 2004.

[19] M. Sulzmann and K. Z. M. Lu. A type-safe embedding of xduce into ml. In *ACM SIGPLAN Workshop on ML*, informal proceedings, Sept. 2005.

[20] S. Vansummeren. Type inference for unique pattern matching. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2003. To appear.

[21] J. Vouillon. Polymorphic regular tree types and patterns. In *Proceedings of the 33th ACM Conference on Principles of Programming Languages*, Charleston, USA, Jan. 2006. To appear. Available from `http://www.pps.jussieu.fr/~vouillon/publi/`.

Composing Monadic Queries in Trees

Emmanuel Filiot
INRIA Futurs, Lille

Joachim Niehren
INRIA Futurs, Lille

Jean-Marc Talbot
LIFL, Lille

Sophie Tison
LIFL, Lille

Abstract

Node selection in trees is a fundamental operation to XML databases, programming languages, and information extraction. We propose a new class of querying languages to define *n*-ary node selection queries as compositions of monadic queries. The choice of the underlying monadic querying language is parametric. We show that compositions of monadic MSO-definable queries capture *n*-ary MSO-definable queries, and distinguish an MSO-complete *n*-ary query language that enjoys an efficient query answering algorithm.

1 Introduction

Node selection in trees [12] is a fundamental operation to XML databases, programming languages, and information extraction. Node selecting captures the matching aspect of tree transformations. Iterated node selection can be used to navigate through input trees while producing output data structures.

From the *database perspective*, node selecting is usually viewed as a querying problem [13, 15, 10]. The W3C standard querying language XPath provides descriptions of *monadic queries*, i.e. queries that define sets of nodes in trees. XPath queries are used by the W3C standard languages XQuery and XSLT for defining XML document transformations.

Modern *programming languages* support node selection in trees via *pattern matching*, for instance all functional programming languages of the ML family (Caml, SML, Haskell). Tree pattern with *n* capture variables define *n-ary queries*, i.e. queries that select sets of *n*-tuples of nodes. The XML programming languages XDuce [11] and CDuce [4, 2] support more expressive *recursive tree pattern*.

Information extraction tasks for the Web can frequently be reduced to defining *n*-ary queries in HTML or XML trees. Gottlob et. al. [9] proposed monadic Datalog as querying language for this purpose, and show that it captures monadic MSO-definable queries [8]. Their Lixto system [6, 1] provides a visual interface by which to specify monadic queries in monadic Datalog, and to compose them into *n*-ary queries. Composed monadic queries are defined in Elog, a binary Datalog language.

We propose a new class of querying languages to define *n*-ary node selection queries as compositions of monadic queries. The choice of the underlying monadic querying language is parametric. We show that compositions of monadic MSO-definable queries capture *n*-ary MSO-definable queries, and distinguish an MSO-complete *n*-ary query language that enjoys efficient query answering algorithms. Moreover, our language allow to compose different monadic query languages, for instance a monadic query defined by a Datalog program with another monadic query defined by an XPath node formula.

Compositions of monadic MSO-definable queries are relevant to information extraction. They might be useful to approach an open question in the context of the Lixto system [6], of how to enhance such system by machine learning techniques. Given a composition formula, and examples for n-tuples that are to be selected, one can use existing learning algorithms for monadic MSO-definable queries [3], in order to infer n-ary MSO definable queries.

The paper is organized as follows. We recall the definitions of *n*-ary MSO definable queries in tree in *Section 2*, introduce languages of compositions of monadic queries in *Section 3*, and discuss some instances in *Section 4*. We study their expressiveness in *Section 5* and their algorithmic complexity in *Section 6*, including algorithms for the model-checking and the query answering problems, and prove the satisfiability problem to be NP-hard. Finally in *Section 7* we propose a fragment of it, study its expressiveness and give an efficient algorithm for the query answering problem.

2 Node Selection Queries in Trees

We recall the definition of *n*-ary MSO-definable queries in trees. We develop our theory for binary trees. This will be sufficient to deal with unranked trees, since unranked trees can be viewed as binary trees via a `firstchild − nextsibling` encoding [12].

We consider binary trees as acyclic digraphs with labeled nodes and ordered children. We start with a finite set Σ of node labels. A binary Σ-tree $t \in T_\Sigma$ is a finite rooted acyclic directed graph, whose nodes are labeled in Σ. Every node is connected to the root by a

unique path. All nodes of binary trees either have 0 or 2 children. Nodes without children are called *leaves*. All other nodes have a distinguished first and second child.

We write $\mathtt{root}(t)$ for the root of tree t and $\mathtt{nodes}(t)$ for the set of nodes of tree t and $\mathtt{edges}(t) \subseteq \mathtt{nodes}(t)^2$ for the set of edges of t. For all labels $a \in \Sigma$, we write $\mathtt{lab}_a(t) \subseteq \mathtt{nodes}(t)$ for the subset of nodes of t labeled by a. Given two nodes $v_1, v_2 \in \mathtt{nodes}(t)$ we call v_2 a *child* of v_1 and write $v_1 \lhd v_2$ iff there exists an edge from v_1 to v_2, i.e., if $(v_1, v_2) \in \mathtt{edges}(t)$.

The *descendant* relation $\lhd^*$ on nodes is the reflexive transitive closure of the child relation $\lhd$.

The *subtree of tree t rooted by node* $v \in \mathtt{nodes}(t)$ is the tree denoted by $t|_v$ that satisfies:

$$\begin{array}{rcll} \mathtt{nodes}(t|_v) & = & \{v' \in \mathtt{nodes}(t) \mid v \lhd^* v'\} & \\ \mathtt{edges}(t|_v) & = & \mathtt{edges}(t) \cap \mathtt{nodes}(t|_v)^2 & \\ \mathtt{root}(t|_v) & = & v & \\ \mathtt{lab}_a(t|_v) & = & \mathtt{lab}_a(t) \cap \mathtt{nodes}(t|_v) & \forall a \in \Sigma \end{array}$$

Definition 1. *Let $n \in \mathbb{N}$. An n-*ary query in binary trees over Σ *is a function q that maps trees $t \in T_\Sigma$ to set of n-tuples of nodes, such that $\forall t \in T_\Sigma : q(t) \subseteq \mathtt{nodes}(t)^n$. Moreover, we require q to be closed under tree-isomorphism, i.e. $h(q(t)) = q(h(t))$ for a tree isomorphism h.*

Simple examples for monadic queries in binary trees over Σ are the functions $\mathtt{lab}_a$ that map trees t to the sets of nodes of t that are labeled by a for $a \in \Sigma$. The binary query $\mathtt{descendant}$ relates nodes v to their descendants, i.e. $\mathtt{descendant}(t) = \{(v, v') \in \mathtt{nodes}(t)^2 \mid v \lhd^* v'\}$.

Definition 2. *A* query language L *over alphabet Σ is a pair $L = (N, [\![.]\!])$ where N is a set of names and $[\![.]\!]$ an interpretation function mapping names $c \in N$ to queries $[\![c]\!]$ in Σ-trees.*

The monadic second-order logic (MSO) in trees is a query language that is widely accepted as the yardstick for comparing the expressiveness of XML-query languages [9, 15]. This is because of the close correspondence between MSO, tree automata, and regular tree languages [17]. Every MSO formula with n free node variables defines an n-ary query.

In MSO, binary trees $t \in T_\Sigma$ are seen as *logical structures* with domain $\mathtt{nodes}(t)$. The signature $\mathbb{T}$ of this structure contains symbols for the binary relations $\mathtt{child}_1$ and $\mathtt{child}_2$ and the unary relations $\mathtt{lab}_a$ for all $a \in \Sigma$.

Let x, y, z range over a countable set of first-order variables and X over a countable set of monadic second-order variables. Formulas ϕ of MSO have the following abstract syntax, where $a \in \Sigma$:

$$\begin{array}{rcl} \phi & ::= & p(x) \mid \mathtt{child}_1(x,y) \mid \mathtt{child}_2(x,y) \\ & & \mid \mathtt{lab}_a(x) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \forall x \phi \mid \forall X \phi \end{array}$$

A variable assignment α into a tree t maps first-order variables $\mathtt{nodes}(t)$ and second-order variables to subsets of $\mathtt{nodes}(t)$. We define the validity of formulas ϕ in trees t under variable assignments α in the usual Tarskian manner, and write $t, \alpha \models \phi$ in this case. The *first-order logic* FO is obtained from MSO by omitting the set quantification. Actually the notations FO and MSO stands for FO[$\mathbb{T}$] and MSO[$\mathbb{T}$] respectively, i.e. formulae over the vocabulary $\mathbb{T}$.

We view MSO as a query language. The names of n-ary queries are MSO formulas $\phi(x_1, \ldots, x_n)$ with n free first-order variables $x_1, ..., x_n$. These define the following queries:

$$[\![\phi(x_1, ..., x_n)]\!](t) = \{(\alpha(x_1), ..., \alpha(x_n)) \mid t, \alpha \models \phi\}$$

Definition 3. *An n-ary query is* MSO definable *if it is equal to some query $[\![\phi(x_1, \ldots, x_n)]\!]$.*

Unfortunately [5] shows that the satisfiability problem is not fixed-parameter tractable, i.e. there exists no polynomial p and elementary function f such that we can decide in time $O(f(|\phi|)\, p(|t|))$ whether an monadic MSO-formula ϕ is satisfiable in the tree t. However, there exists query languages that can express all MSO-definable queries, which have polynomial-time combined complexity: e.g. monadic queries defined by successful runs of tree automata have exactly the power of MSO in defining monadic queries but deciding the non-emptyness of a monadic query is in polynomial-time w.r.t. combined complexity [14].

Let us define some algorithmic tasks for query languages $(N, [\![.]\!])$ that are common to database theory:

- *model-checking*: given a query name c, a tree t and an n-tuple $(v_1, \ldots, v_k) \in \mathtt{nodes}(t)^k$, does $(v_1, \ldots, v_k) \in [\![c]\!](t)$ hold ?
- *query answering*: given a query name c and a tree t, return $[\![c]\!](t)$. An expected complexity might be polynomial in the number of solutions.
- *satisfiability (over a fixed tree)*: given a query name c and a tree t, does $[\![c]\!](t) \neq \emptyset$ hold ?

Unranked trees are like binary trees, except that all nodes may have arbitrarily many ordered children. The next-sibling of a node is the successor of the same parent in the sibling ordering.

Unranked trees can be encoded as binary trees by only using edges for the first-child and next-sibling relations. Fig. 1 gives a DTD, an unranked tree matching this DTD and its first-child next-sibling encoding t. A simple binary query on that tree is to select all pairs of name and title of the same book. It can be expressed with respect to the binary encoding by the following MSO formula with two free variables y, z:

$$\exists x\, (\mathtt{lab}_{\mathtt{author}}(x) \wedge \mathtt{child}_1(x,y) \wedge \mathtt{child}_2(x,z))$$

3 Composing Monadic Queries

Query languages for monadic queries in trees have been widely studied by the database community in the last few years. See [12] for a comprehensive overview. Languages for n-ary queries are less frequent but have started to arise with the XML programming languages

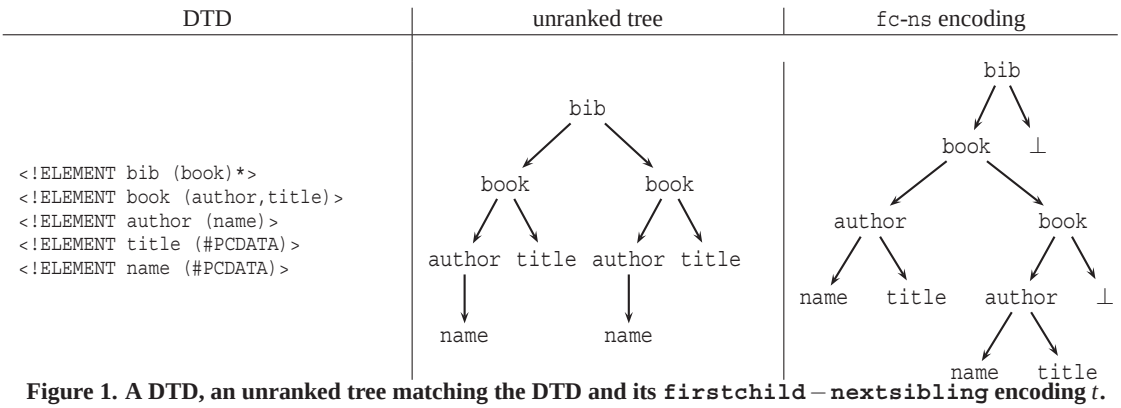


Figure 1. A DTD, an unranked tree matching the DTD and its `firstchild` − `nextsibling` encoding t.

XDuce and CDuce [11, 2, 4] as well as with information extraction tools such as Lixto [8, 6].

In this paper, we propose a new class of languages for defining *n*-ary queries by composition of monadic queries. We leave the choice of the underlying monadic querying language parametric, so the reader may choose his prefered monadic querying language, and extend it to an *n*-ary query language by query composition. The composition operator is motivated by Lixto's way of defining *n*-ary queries [6].

The principle of composition is quite simple: a composition of two monadic queries first selects a node answering the first sub-query and then launches the second sub-query at that node. All nodes seen meanwhile can be memoized and returned in an output tuple.

We start from a language L of monadic queries c and an infinite set $x, y, z \in \mathtt{Var}$ of variables. We then define compositions of monadic queries on basis of the composition operator that we write as the dot '.' . Informally a composition query $c_1(x_1).c_2(x_2)$ on a tree t will first bind x_1 nondeterministically to some node $v_1 \in [\![c_1]\!](t)$, and then launch query c_2 in the subtree $t|_{v_1}$ rooted at v_1 in order to bind x_2 to some node $v_2 \in [\![c_2]\!](t|_{v_1})$.

For expressivity reasons – that is to capture MSO as soon as the monadic query language capture MSO – we add conjunction, disjunction and projection to our composition language [1].

Given a monadic query language $L = (\mathtt{N}, [\![.]\!])$, *composition formulae* $\phi \in \mathcal{C}(L)$ are defined by the following abstract syntax:

ϕ	::=		*composition formula*
		$\top$	
	|	$c(x).\phi$	*composition*, $c \in \mathtt{N}$, $x \in \mathtt{Var}$
	|	$\phi \wedge \phi$	*conjunction*
	|	$\phi \vee \phi$	*disjunction*
	|	$\exists x\, \phi$	*projection*

Given a composition formula ϕ, we denotes by $\mathrm{FV}(\phi)$ the set of free variables of ϕ. We will often write $c(x)$ instead of $c(x).\top$. The set of subformulas of ϕ is denoted by $\mathtt{Sub}(\phi)$. The *composition size* $|\phi|$ of a formula ϕ is inductively as follows:

$$\begin{array}{rclcrcl} |\top| & = & 0, & & |\phi \wedge \phi'| & = & |\phi| + |\phi'| + 1 \\ |c(x).\phi| & = & 1 + |\phi|, & & |\phi \vee \phi'| & = & |\phi| + |\phi'| + 1 \\ |\exists x\, \phi| & = & 1 + |\phi| & & & & \end{array}$$

Note that this definition implies that query names are of size one.

For all trees t, all valuations $\nu : \mathtt{Var} \to \mathtt{nodes}(t)$ ranging over the nodes of t, and all composition formula $\phi \in \mathcal{C}(L)$ we define the satisfaction relation $t, \nu \models \phi$ as follows:

(i)
$\mathtt{range}(\nu) \subseteq \mathtt{nodes}(t)$

(ii)

$$\begin{array}{lll} t, \nu \models \top & & \\ t, \nu[x/u] \models c(x).\phi & \text{iff} & \begin{cases} u \in [\![c]\!](t) & (1) \\ t|_u, \nu \models \phi & (2) \end{cases} \\ t, \nu \models \phi_1 \wedge \phi_2 & \text{iff} & t, \nu \models \phi_1 \text{ and } t, \nu \models \phi_2 \\ t, \nu \models \phi_1 \vee \phi_2 & \text{iff} & t, \nu \models \phi_1 \text{ or } t, \nu \models \phi_2 \\ t, \nu \models \exists x\, \phi & \text{iff} & \text{there exists } u \in \mathtt{nodes}(t), \\ & & \quad \text{s.t. } t, \nu[x/u] \models \phi \end{array}$$

Let us consider the satisfiability of $c(x).\phi$. Condition (1) implies that $[\![c]\!]$ selects the node u in t, condition (2) implies that the interpretation of ϕ is relativized to the subtree of t rooted at u.

Valuations define possible values for free variables in composition formulae. A formula can define an *n*-ary query by sorting its free variables. Formally, a formula $\phi \in \mathcal{C}(L)$ with free variables $\{x_1, \ldots, x_n\} = \mathrm{FV}(\phi)$ defines the *n*-ary query $[\![\phi(x_1, \ldots, x_n)]\!]$ such that for all trees t:

$$[\![\phi(x_1, \ldots, x_n)]\!](t) = \{(\nu(x_1), \ldots, \nu(x_n)) \mid t, \nu \models \phi\}$$

4 Examples of Composition Languages

We now discuss some instances of query languages $\mathcal{C}(L)$ by instantiating the parameter L to some concrete

[1] We proved that conjunctions, disjunctions and projections are required to express all MSO-queries by composition of monadic MSO-definable queries

monadic query language.

As first instance, we let L be the monadic query language containing all monadic MSO formulas. For illustration, we consider XML documents defining collections of books, which satisfy the DTD in Fig. 1. Our target is to select all pairs of author names and titles of the same book by composition.

We define the binary query on `firstchild-nextsibling` encodings. Names and titles of a book are contained in siblings of `author`-labeled nodes. To select the pairs, we first select all `author` nodes by the monadic query $[\![c_1]\!]$ defined by the monadic MSO formula $c_1 = \mathtt{lab}_{\mathtt{author}}(x)$. We then compose it with two independant monadic queries $[\![c_2]\!]$ and $[\![c_3]\!]$, for selecting name by $c_2 = \exists y\ \mathtt{root}(y) \wedge \mathtt{child}_1(y,x)$ and title by $c_3 = \exists y\ \mathtt{root}(y) \wedge \mathtt{child}_2(y,x)$. The modeling composition formula is:

$$\phi = \exists z c_1(z).(c_2(x) \wedge c_3(y))$$

Note that according to the DTD and the semantic of composition, the first query can select nodes labeled by `author`, and then, in each subtrees induced by the previous selected nodes, one can select the node labeled by `name`, and the node labeled by `title`, by two independant monadic queries $c'_2 = \mathtt{lab}_{\mathtt{name}}(x)$ and $c'_3 = \mathtt{lab}_{\mathtt{title}}(x)$ respectively. The modeling composition formula is then:

$$\phi = \exists z c_1(z).(c'_2(x) \wedge c'_3(y))$$

A second instance is obtained by composing monadic Datalog queries [8] which are well known to capture all monadic MSO. Indeed, our idea of compositions is very much inspired by the way in which n-ary queries are defined from monadic Datalog queries by the Lixto system for visual Web information extraction [1, 6].

We illustrate the correspondence at the example of selecting pairs of author names and titles of the same books. Such a query is expressed in Lixto by a Monadic Datalog program P and an additional information about the predicate hierarchy, which we model by a tree. We express this query in the `firstchild`$-$`nextsibling` encodings. The monadic Datalog program P and the predicate hierarchy are given on Figure 4.

We can express the same query by composing the following three monadic Datalog queries by

$$\phi(y,z) = \exists x\, P_1(x).(\ P_2(y)\ \wedge\ P_3(z)\ )$$

.

P_1 : $P_{\mathtt{author}}(x)$:- $\mathtt{lab}_{\mathtt{author}}(x)$ with the goal $P_{\mathtt{author}}$

P_2 : $P_{\mathtt{name}}(x)$:- $\mathtt{root}(y), \mathtt{child}_1(y,x)$ with the goal $P_{\mathtt{name}}$

P_3 : $P_{\mathtt{title}}(x)$:- $\mathtt{root}(y), \mathtt{child}_2(y,x)$ with the goal $P_{\mathtt{title}}$.

Implementation We have implemented a rather naive algorithm for answering compositions of monadic queries, defined either in MSO, XPath, or by tree automata. Further monadic query languages are be easily added by new modules called *query machines*. Each monadic query can be expressed by different formalism within the same composition formula.

Our concrete syntax for expressing composition queries is given in Fig. 3. A typical input consists of an XML document and a composition query. The output is an XML document representing the set of all answers. The implementation is done in OCaml.

5 MSO Completeness

We call an n-ary query language MSO-complete if it can express all MSO-definable n-ary queries. For instance, monadic Datalog is known to be a MSO complete monadic query language. In this paragraph, we study the expressiveness of composition languages over MSO-complete monadic query languages.

We show that the composition operator can be expressed in first-order logic, so that n-ary-compositions of monadic MSO definable queries are MSO-definable too.

Let $L = (\mathtt{N}, [\![.]\!])$ be a monadic query language. For every name $c \in \mathtt{N}$ we introduce a binary predicate symbol B_c that we interpret as a binary relation on $B_c^t \subseteq \mathtt{nodes}(t)^2$

$$B_c^t = \{(v,v') \mid v' \in [\![c]\!](t|_v)\}$$

We now consider the first-order logic over the signature $(B_c)_{c\in\mathtt{N}} \cup \mathbb{T}$.

Proposition 1. *Every composition formula $\phi(\bar{x}) \in \mathcal{C}(L)$ is equivalent to some first-order formula $\gamma(\bar{x})$ over the signature $(B_c)_{c\in N} \cup \mathbb{T} \cup \{\lhd^*\}$.*

Proof. We define a function $\langle . \rangle_x$ encoding composition formulas into first-order formulas over the signature $(B_c)_{c\in\mathtt{N}} \cup \mathbb{T} \cup \{\lhd^*\}$ inductively:

$$\begin{array}{lcl}
\langle \top \rangle_x & = & \top \\
\langle c(y).\phi \rangle_x & = & B_c(x,y) \wedge \langle \phi \rangle_y \\
\langle \phi_1 \wedge \phi_2 \rangle_x & = & \langle \phi_1 \rangle_x \wedge \langle \phi_2 \rangle_x \\
\langle \phi_1 \vee \phi_2 \rangle_x & = & \langle \phi_1 \rangle_x \vee \langle \phi_2 \rangle_x \\
\langle \exists y\ \phi \rangle_x & = & \exists y\ x \lhd^* y \wedge \langle \phi \rangle_x
\end{array}$$

Let $\gamma(x_1,\ldots,x_n) \equiv \exists \mathrm{FV}(\phi)\backslash\{x_1,\ldots,x_n\}\ (\exists y\ \mathtt{root}(y) \wedge \langle \phi \rangle_y)$, where $y \notin \mathrm{FV}(\phi)$. Finally note that $\mathtt{root}(x)$ is FO[$\mathbb{T}$]-definable. □

If the monadic query language captures MSO then the binary predicates B_c are MSO-definable. The first important technical contribution of this paper is that the converse holds too.

Theorem 1. *The class of n-ary queries defined by composition of MSO-definable monadic queries is exactly the class of n-ary MSO definable queries.*

To prove the first direction it suffices to show that each predicate B_c is MSO-definable whenever $[\![c]\!]$ is. The

$P_{\texttt{author}}(x)$:- $\texttt{lab}_{\texttt{author}}(x)$
$P_{\texttt{name}}(x)$:- $P_{\texttt{author}}(y)$, $\texttt{child}_1(y,x)$
$P_{\texttt{title}}(x)$:- $P_{\texttt{author}}(y)$, $\texttt{child}_2(y,x)$

(a) program P

(b) hierarchy

Figure 2. A set of Monadic Datalog rules and its predicates hierarchy

query	::=	**SELECT** *vars* **FROM** *formula*
formula	::=	*atom* | *formula* **AND** *formula* | *formula* **OR** *formula*
atom	::=	*machine***(***var***)**
vars	::=	*var* | *var*,*vars*
var	::=	*identifier*
machine	::=	**XPATH[***xpath_specif***]** | **AUTOMATON[***automaton_specif***]** | **MSO[***mso_specif***]**

Figure 3. Concrete sytnax for composition queries

binary MSO formula $\gamma_{B_c}(x,y)$ defining B_c is exactly the formula $\gamma_c(y)$ defining $[\![c]\!]$ where each quantification is relativized to x.

The rest of this section prove the other direction, i.e. the composition of monadic MSO-definable queries is complete for n-ary MSO-definable queries. The proof is based on the equivalence between MSO-definable queries and node selection automata as defined in [16], which is a consequence of the seminal theorem of Thatcher and Wright [17].

We recall that a *node selection automaton* (NSA) is a pair (A,S) where $A=(\Sigma,Q,F,\Delta)$ is a tree automata and S is a set of selection tuples $\bar{q}$. We write $(A,\bar{q})$ instead of $(A,\{\bar{q}\})$. A *run* of a tree automata A over a tree t is a tree r isomorphic to t via an isomorphism Φ, where each node is labeled in Q, and such that the following holds:

- if $v \in \texttt{nodes}(t)$ is a leaf labeled by $a \in \Sigma$, then $a \rightarrow \texttt{lab}_r(\Phi(v))$ is in Δ,
- if $v \in \texttt{nodes}(t)$ is an inner node labeled by $f \in \Sigma$, and $v_1, v_2 \in \texttt{nodes}(t)$ are its first child and its second child respectively, then the rule $f(\texttt{lab}_r(\Phi(v_1)), \texttt{lab}_r(\Phi(v_2))) \rightarrow \texttt{lab}_r(\Phi(v))$ is in Δ.

A run r of A over t is successful iff its root is labeled by an accepting state from F. A NSA (A,S) selects a tuple of nodes $(v_1,\ldots,v_n)$ of a tree t iff there exists a successful run r over t (isomorphic to t via Φ), and a selection tuple $(q_1,\ldots,q_n) \in S$, such that for each $i \in \{1,\ldots,n\}$, the node $\Phi(v_i)$ is labeled by q_i in r. When it is clear from the context we will omit the isomorphism Φ. Finally, the class of MSO-definable n-ary queries is exactly the class of n-ary queries defined by node selection automata over binary trees [17, 16].

In order to prove Theorem 1 we introduce some notations. Given a set R, an n-tuple $\bar{r}=(r_1,\ldots,r_n) \in R^n$, and a set $J \subseteq \{1,\ldots,n\}$, we denote by $\Pi_J(\bar{r})$ the projection of $\bar{r}$ w.r.t. J, defined by $\Pi_J(\bar{r})=(r_i)_{i\in J}$. In particular, $\Pi_\emptyset(\bar{r})=()$. Given a tree t, an n-tuple of nodes $\bar{v}=(v_1,\ldots,v_n) \in \texttt{nodes}(t)^n$, a NSA $(A,\bar{q})$ with a selection tuple $\bar{q}=(q_1,\ldots,q_n) \in Q^n$, and a state $q \in Q$, a *q-run of $(A,\bar{q})$ over t selecting $\bar{v}$* is a run of A over t such that the root is labeled by q, and v_i is labeled by q_i for each $i \in \{1,\ldots,n\}$. In particular, when $n=0$, a q-run of $(A,())$ over t selecting the empty sequence is a run of A over t labeling the root by q.

Lemma 1. *Let $n \geq 2$ be a natural. Let $t \in T_\Sigma$ be a binary tree. Let $\bar{v}=(v_1,\ldots,v_n)$ be a tuple of length n of nodes from t, such that there exists at least two different nodes. Let v_a be the least common ancestor of $\bar{v}$. Let v_a^1 be the first child of v_a, and v_a^2 its second child. Define I,J,K as follows:*

$$\begin{array}{lcl} I & = & \{i \mid v_a = v_i\} \\ J & = & \{j \mid v_a^1 \lhd^* v_j\} \\ K & = & \{k \mid v_a^2 \lhd^* v_k\} \end{array}$$

Let $(A,\bar{q})$ be a NSA, and q a state, then there exists a q-run r of A over t selecting $\bar{v}$ iff

$\exists q', q'' \in Q$ *s.t.*
- there exists a q-run of $(A,\Pi_I(\bar{q}))$ over t selecting $\Pi_I(\bar{v})$ and labeling v_a^1, v_a^2 by q', q'' respectively
- there exists a q'-run of $(A,\Pi_J(\bar{q}))$ over $t|_{v_a^1}$ selecting $\Pi_J(\bar{v})$
- there exists a q''-run of $(A,\Pi_K(\bar{q}))$ over $t|_{v_a^2}$ selecting $\Pi_K(\bar{v})$

Proof. The proof is not difficult and left to the reader. □

Lemma 2. *Let n be a natural. Given a node selection automaton $(A,\bar{q})$ where $\bar{q}$ is an n-tuple of states, given a state $q \in Q$, there exists a composition formula $\phi_{A,\bar{q},q}(x_1,\ldots,x_n)$ over MSO-definable monadic queries such that for all Σ-tree t, for all $\bar{v} \in \texttt{nodes}(t)^n$, the following are equivalent:*

(i) *there exists a q-run of A over t selecting $\bar{v}$*
(ii) $\bar{v} \in [\![\phi_{A,\bar{q},q}(x_1,\ldots,x_n)]\!](t)$

Proof. We construct the formula inductively on n. The construction mimics the decomposition given by lemma 1.
If $n = 0$ we take $\phi_{A,\bar{q},q} = \exists x\, c_0(x)$ where $[\![c_0]\!](t)$ is equal to $\mathtt{nodes}(t)$ if and only if there exists a q-run of A over t. By Thatcher and Wright's theorem, this monadic query is MSO-definable.
If $n = 1$, then $\bar{q} = (p)$ for some $p \in Q$, and we take $\phi_{A,(p),q}(x) = c_1(x)$ where $[\![c_1]\!]$ is defined by the NSA $(A,(p))$. Again by Thatcher and Wright's theorem, this query is MSO-definable.
If $n > 1$, we consider two cases depending on whether the variables $x_1,\ldots,x_n$ will be instantiated by the same node or not. So $\phi_{A,\bar{q},q}(x_1,\ldots,x_n)$ will be written as a disjunction $\phi^{eq}_{A,\bar{q},q} \vee \phi^{neq}_{A,\bar{q},q}$:

- case 1 (variables will be instantiated by the same node). Let $\gamma_{(A,\bar{q})}(\bar{x})$ be an MSO formula such that for a tree t and an n-tuple $\bar{v}$ of nodes of t, it holds that $\bar{v} \in [\![\gamma_{(A,\bar{q})}]\!](t)$ iff there exists a q-run of $(A,\bar{q})$ over t selecting $\bar{v}$. It is easy to show that this formula exists, by Thatcher and Wright's theorem. Then we take $\phi^{eq}_{A,\bar{q},q}(x_1,\ldots,x_n) = \exists x\ c_1(x).(\bigwedge_i c_r(x_i))$, where $[\![c_1]\!]$ is the query defined by the monadic MSO formula $\exists y_1,\ldots,y_{n-1} \bigwedge_i (y_n = y_i) \wedge \gamma_{(A,\bar{q})}(y_1,\ldots,y_n)$ and $[\![c_r]\!](t)$ selects the root of t, for any tree t.

- case 2 (variables will be instantiated by at least two different nodes). Let $\bar{x}$ denotes $(x_1,\ldots,x_n)$ and let $\mathcal{P}_n$ be the sets of partitions (with possibly empty parts) of $\{1,\ldots,n\}$ such that for each partition, there exists at most one empty part. We define $\phi^{neq}_{A,\bar{q},q}(\bar{x})$ by:

$$\bigvee_{\{I,J,K\}\in\mathcal{P}_n} \bigvee_{q',q''\in Q} \exists x \exists y \exists z\ c^{q}_{q',q''}(x).\ (\bigwedge_{i\in I} c_r(x_i) \wedge c_1(y).\phi_{A,\Pi_J(\bar{q}),q'}(\Pi_J(\bar{x})) \wedge c_2(z).\phi_{A,\Pi_K(\bar{q}),q''}(\Pi_K(\bar{x})))$$

  where $[\![c_1]\!](t)$ selects the first child of the root of t, and $[\![c_2]\!](t)$ its second child. For any tree t, the query $[\![c^{q}_{q',q''}]\!](t)$ selects a node $v \in \mathtt{nodes}(t)$ iff there exists a q-run of $(A,\Pi_I(\bar{q}))$ over t selecting $(v,v,\ldots,v)$ (of length $|I|$), such that its first child is labeled by q', and its second child by q''. This query is MSO-definable, again by Thatcher and Wright's theorem. Remark that subformulae $\phi_{A,\Pi_J(\bar{q}),q'}(\Pi_J(\bar{x}))$ and $\phi_{A,\Pi_K(\bar{q}),q'}(\Pi_K(\bar{x}))$ are recursively well defined, since $|K|,|J| < n$.

  The rest of the proof is a direct application of Lemma 1.

□

To conclude the proof of Theorem 1, we state the following corollary:

Corollary 1. *For each MSO formula $\gamma(\bar{x})$, there exists an equivalent composition formula $\phi(\bar{x})$ over MSO-definable monadic queries.*

Proof. By [17, 16], there exists a NSA (A,S) equivalent to γ, and we define ϕ by: $\phi = \bigvee_{\bar{q}\in S} \bigvee_{q\in F} \phi_{A,\bar{q},q}$, where $\phi_{A,\bar{q},q}$ has been defined in the previous lemma. □

6 Algorithmic Complexity

In this paragraph $L = (\mathtt{N}, [\![.]\!])$ is a monadic query language, and we suppose that there exists an algorithm for the model-checking problem in time-complexity $\mathtt{mc}(c,t)$, where $c \in \mathtt{N}$ and $t \in T_\Sigma$, and an algorithm for the query answering problem in time-complexity $\mathtt{qa}(c,t)$.

Fig. 4 represents a simple algorithm for the model-checking problem of a formula ϕ, a tree t and a valuation ν. It is written in a pseudo ML-like code. It runs in time $O(|\phi|M|t|^{\max_{\phi'\in\mathtt{Sub}(\phi)}(|FV(\phi')|)} + |\phi|^2)$ where $M = \max_{c(x)\in\mathtt{Sub}(\phi)} \mathtt{mc}(c,t)$.

This gives a naive algorithm for the query answering problem: generate all the valuations of free variables of a formula ϕ in a tree t, and apply the model-checking algorithm on them. This leads to an exponential grow up, but it is not clear how to avoid it since the satisfiability problem of monadic query composition is NP-hard.

Proposition 2. *Let $\Sigma = \{0,1,\circ\}$ be an alphabet and $L = (\{c_0,c_1\},[\![.]\!])$ a monadic query language over Σ where $[\![c_b]\!]$ selects all the nodes labeled by $b \in \{0,1\}$ in Σ-trees. Let t the binary whose roots is labeled by $\circ$, its first child by 0, and its second child by 1. Given a composition formula ϕ over L, the satisfiability problem of ϕ over t is NP-hard.*

Proof. To prove that it is NP-hard we give a polynomial reduction of CNF satisfiability into our problem. The idea is to associate with a given CNF formula $\Psi = \bigwedge_{1\le i\le p} C_i$ a composition formula $\phi = \bigwedge_{1\le i\le p} \phi_i$ over L. Each ϕ_i is a composition formula associated to the i-th clause C_i. It is defined by associating to each litteral x_j the atomic formula $c_1(x_j)$ and to $\neg x_j$ the formula $c_0(x_j)$, and to a disjunction of litterals a disjunction of atomic formulae. For example, if we consider $\Psi = (x_1 \vee \neg x_2) \wedge (x_2 \vee \neg x_3)$, then $\phi = (c_1(x_1) \vee c_0(x_2)) \wedge (c_1(x_2) \vee c_0(x_3))$. □

Composition and conjunctive queries Conjunctive queries over finite relational structures have been widely studied by the database community since it is the most common database query in practice. The particular case of conjunctive queries over unranked trees have been studied in [7] over particular binary XPath axis $\mathcal{A} = \{$ Child, Child$^+$, Child*, NextSibling,

$$
\begin{array}{lcl}
\textbf{let } \texttt{check}(\phi,t,\nu) & = & \textbf{match } \phi \textbf{ with} \\
& & \mid \top \rightarrow \text{true} \\
& & \mid \psi(x).\phi' \rightarrow \nu(x) \in [\![\psi]\!](t) \ \wedge\ \texttt{check}(\phi', t|_{\nu(x)}, \nu) \ \wedge \forall y \in \mathrm{FV}(\phi')\ \nu(x) \lhd^* \nu(y) \\
& & \mid \phi' \vee \phi'' \rightarrow \texttt{check}(\phi', t, \nu) \ \vee \texttt{check}(\phi'', t, \nu) \\
& & \mid \phi' \wedge \phi'' \rightarrow \texttt{check}(\phi', t, \nu) \ \wedge \texttt{check}(\phi'', t, \nu) \\
& & \mid \exists x\, \phi' \rightarrow \bigvee_{u \in \texttt{nodes}(t)} \texttt{check}(\phi', t, \nu[x/u])
\end{array}
$$

Figure 4. Model-checking algorithm for a formula $\phi \in \mathcal{C}(L)$, a tree t and a valuation ν

NextSibling$^+$, NextSibling*, Following $\}$. Surprisingly the complexity of these queries quickly fall into NP-hardness. Since each conjunctive queries over these axis in an unranked tree is expressible by a composition query over a particular monadic query language in binary trees, all the complexity lower bounds from [7] apply to our formalism. For example, the satisfiability problem of a composition query over the monadic query languge $(\{c_{1^*}, c_2\}, [\![.]\!])$ is NP-hard w.r.t. combined complexity, where $[\![c_{1^*}]\!](t) = \{v \mid \text{Child}_1^*(\texttt{root}(t), v)\}$ and $[\![c_2]\!](t) = \{v \mid \text{Child}_2(\texttt{root}(t), v)\}$.

In the next section we propose a composition fragment for which the satisfiability problem is in PTIME whenever this holds for the underlying monadic query language, and give an efficient algorithm for query answering. In addition we prove that this fragment can express all MSO-definable *n*-ary queries whenever the underlying monadic query language captures MSO.

7 An MSO-Complete and Tractable Fragment

In this section, we introduce a "tractable" syntactic fragment of composition formulae $\mathcal{E}(L)$, that leads to an *n*-ary MSO-complete query language (as soon as the monadic query language L is), while enjoying efficient query answering algorithms.

Let L be a language of MSO-definable monadic queries. In this fragment, variable sharing between conjunctions and composition are not permitted, more precisely, if $\phi \wedge \phi'$ and $c(x).\phi''$ are $\mathcal{E}(L)$-formula, then $\mathrm{FV}(\phi) \cap \mathrm{FV}(\phi') = \emptyset$, and $x \notin \mathrm{FV}(\phi'')$. CDuce patterns for instance are built under this restriction for conjunctions [4].

If the satisfiability problem for the underlying query language is PTIME, then it holds for the composition fragment too. The algorithm is based on dynamic programming – a satisfiability table defined inductively is computed with memoization –. Then the query answering algorithm processes the formula inductively under the assumption that it is satisfied in the current tree.

7.1 MSO-completeness

We start by a theorem on expressiveness of the fragment $\mathcal{E}(L)$, over MSO-definable monadic queries.

Theorem 2. *Let L be a language of MSO-definable monadic queries. The class of n-ary queries defined by $\mathcal{E}(L)$-formulae is exactly the class of n-ary MSO-definable queries.*

Proof. The proof is the same than those of Theorem 1. It suffices to remark that the constrution of an equivalent composition formula given in Theorem 1 respects the required restrictions on variable sharing. □

7.2 Answering algorithm

In this section we give an algorithm for answering a composition query q on a tree t, so that the complexity may depend on the size of the output. Since the answering complexity depends on the maximal number of free variables of the subformulae of the formula defining the query, we first show that each composition formula ϕ is equivalent to a composition formula where there is a most 1 free variable different from the free variables of ϕ in its subformulae (wlog we assume that the quantified variables of ϕ are different from the free variables of ϕ). Moreover, in order to avoid the problem of non-valued variables – for example in the formula $c(x) \vee c(y)$ –, we complete each formula so that each part of disjunctions has the same free variable sets. For instance the formula $c(x) \vee c(y)$ is rewriting into the equivalent formula $(c(x) \wedge \text{true}(y)) \vee (\text{true}(x) \wedge c(y))$. The size of the output formula can be at most quadratic in the size of the input formula.

Let $L = (\mathtt{N}, [\![.]\!])$ be a monadic query language. Let $t \in T_\Sigma$ be a tree and let $\phi \in \mathcal{E}(L)$ a composition formula. We suppose to have an algorithm to answer monadic queries. The query answering algorithm processes in four steps:

1. rewrite ϕ into an equivalent formula ϕ' in which there is at most one free variable different from the free variables of ϕ', in its subformulae, and such that for each $\gamma \vee \gamma' \in \texttt{Sub}(\phi')$, $\mathrm{FV}(\gamma) = \mathrm{FV}(\gamma')$;
2. compute two data structures $Q_a : \mathtt{N} \times \texttt{nodes}(t) \rightarrow \texttt{nodes}(t)$ and $Q_c : \mathtt{N} \times \texttt{nodes}(t) \times \texttt{nodes}(t) \rightarrow \{0,1\}$ such that given a query name $c \in \mathtt{N}$ appearing in ϕ', and two nodes $v, v' \in \texttt{nodes}(t)$, $Q_a(c,v)$ returns the set $\{v' : v' \in [\![c]\!](t|_v)\}$ in linear time in the size of the output, and $Q_c(c,v,v')$ checks in constant time whether $v' \in [\![c]\!](t|_v)$;
3. compute a data structure $\texttt{Sat} : \texttt{Sub}(\phi') \times \texttt{nodes}(t) \rightarrow \{0,1\}$ such that $\texttt{Sat}(\phi'', v)$ checks in

constant time whether a formula $\phi'' \in \mathtt{Sub}(\phi')$ is satisfied in $t|_v$;

4. answer the query by processing the formula ϕ' recursively with satisfiability tests, doubles elimination, and memoization.

Step 1 Let ϕ be a composition formula. Wlog assume that quantified variables of ϕ are different from its free variables. We define the width $\mathtt{w}(\phi)$ of ϕ as the maximal number, over the subformulae of ϕ, of free variables different from the free variables of ϕ. More formally $\mathtt{w}(\phi) = \max_{\phi' \in \mathtt{Sub}(\phi)} |\mathrm{FV}(\phi') \backslash \mathrm{FV}(\phi)|$. As we said we transform ϕ into an equivalent formula ϕ' with $\mathtt{w}(\phi') \leq 1$. The transformation is simple by pushing down the quantifiers. We sum up it in the following lemma:

Lemma 3. *Each query q defined by a composition formula $\phi \in \mathcal{E}(L)$ is equal to some query defined by a composition formula $\phi' \in \mathcal{E}(L)$ such that $w(\phi') \leq 1$.*

Proof. We define the translation of ϕ into ϕ' by the following rewriting rules:

$$\begin{array}{lcl} \exists x\ (\gamma \vee \gamma') & \rightarrow & (\exists x\ \gamma) \vee (\exists x\ \gamma') \\ \exists x\ (\gamma \wedge \gamma') & \rightarrow & (\exists x\ \gamma)\ \wedge\ (\exists x\ \gamma') \\ \exists x\ c(y).\phi & \rightarrow & c(y).(\exists x\ \phi) \text{ with } y \neq x \\ \exists x\ \gamma & \rightarrow & \gamma \text{ if } x \notin \mathrm{FV}(\gamma) \end{array}$$

We can show this rewriting system to terminate, and to be confluent. The normal form is a formula where each occurence of a quantified variable in an atomic formula $c(x)$ is preceded by an existential quantification $\exists x\ c(x)$. Hence, normal forms are of width at most 1. Now we show that the normal form ϕ' of a formula ϕ is equivalent to ϕ. The only difficulties come from $\exists x\ (\gamma \wedge \gamma') \rightarrow (\exists x\ \gamma)\ \wedge\ (\exists x\ \gamma')$ and $(\exists x\ c(y).\phi) \rightarrow (c(y).(\exists y\ \phi)$. The first case holds since $\mathrm{FV}(\gamma) \cap \mathrm{FV}(\gamma') = \emptyset$, and the following proves the second case:
$t, \nu[y/_{\nu'}] \models (\exists x\ c(y).\phi)$
iff there exists $v \in \mathtt{nodes}(t)$ s.t. $t, \nu[y/_{\nu'}][x/_{\nu}] \models c(y).\phi$
iff there exists $v \in \mathtt{nodes}(t|_{\nu'})$, $v' \in [\![c]\!](t)$ and $t|_{\nu'}, \nu[x/_{\nu}] \models \phi$
iff $t, \nu[y/_{\nu'}] \models c(y).(\exists x\ \phi)$.
We conclude by induction on the reduction length. □

Remark that the size of the resulting formula is linear – multiply by two – in the size of the input formula, since each occurence of free variable is preceded by its quantification. Then we transform ϕ' so that each part of a disjunction shares the same free variable sets, and such that each quantified variable is different from each free variable of ϕ'.

Step 2 It is quite obvious, by using hash tables.

Step 3 We compute – using memoization – a table $\mathtt{Sat}[.,.]$ defined inductively by:

$$\begin{array}{rcll} \mathtt{Sat}[\top, u] & = & 1 & (1) \\ \mathtt{Sat}[c(x).\phi, u] & = & \bigvee_{u' \in Q_a(c,u)} \mathtt{Sat}[\phi, u'] & (2) \\ \mathtt{Sat}[\phi \wedge \phi', u] & = & \mathtt{Sat}[\phi, u] \wedge \mathtt{Sat}[\phi', u] & (3) \\ \mathtt{Sat}[\phi \vee \phi', u] & = & \mathtt{Sat}[\phi, u] \vee \mathtt{Sat}[\phi', u] & (4) \\ \mathtt{Sat}[\exists x\ \phi, u] & = & \mathtt{Sat}[\phi, u] & (5) \end{array}$$

Step 4 The last phase is given on Fig. 5. Moreover, we use memoization to avoid exponential grow-up. Valuations are represented by sequences of pairs (variable, node). We assume union and projection operations to eliminate doubles, so that their time complexities are linear in the input sets. This can be done by storing tuples in hash tables.

7.3 Answering Complexity

In this section we study the complexity of the previous algorithm. Let $L = (\mathtt{N}, [\![.]\!])$ be a monadic query language. Inputs of the algorithm are a tree t and a composition formula $\phi \in \mathcal{E}(L)$. Moreover, we suppose to have of an algorithm to answer $[\![c]\!]$ on a tree t, for each $c \in \mathtt{N}$, in time complexity $\mathtt{qa}(n,t)$. We write $M(\phi,t)$ for $\max_{v \in \mathtt{nodes}(t), c(x) \in \mathtt{Sub}(\phi)} \mathtt{qa}(c, t|_v)$. We sum-up the complexity by the following proposition:

Proposition 3. *Answering a query q defined by a composition formula $\phi \in \mathcal{E}(L)$ is in time $O(M(\phi,t)|t||\phi| + |\phi|^2|t|^2|\phi(t)|)$, where $|\phi(t)|$ is the output size.*

Proof. The first step produces a formula ϕ' such that $|\phi'| = O(|\phi|^2)$. The second step is in time $O(M(\phi,t)|t||\phi|)$, and the computation of the satisfiability table is in time $O(|\phi'||t|^2) = O(|\phi|^2|t|^2)$.
It remains to show the time complexity of algorithm depicted in figure 5 to be $O(|\phi'||t|^2 nK)$, where K is the number of solutions and n the arity of the query – we consider that $|\phi(t)| = Kn$ –. We are going to show that each recursive call returns at most $|t|K$ valuations, and performs at most $O(|t| + nK|t|)$ operations. Each call to ans begins by a satisfiability test, so that the following property holds: if $\mathtt{ans}(\gamma, t, v)$ is a recursive call occuring during the processing of ϕ', then the projection of each valuation returned by $\mathtt{ans}(\gamma, t, v)$ on the variables from $\mathrm{FV}(\phi')$ can be extended to a valuation ν such that $t, \nu, \mathtt{root}(v) \models \phi'$. Hence, the number of valuations returned by $\mathtt{ans}(\gamma, t, v)$ is at most $|t|^{|\mathrm{FV}(\gamma) \backslash \mathrm{FV}(\phi')|} K$. Moreover, since $\mathtt{w}(\phi') = 1$, we get $|\mathrm{FV}(\gamma) \backslash \mathrm{FV}(\phi')| \leq 1$.
It is clear that for conjunctions, disjunctions, and projections, each recursive call performs at most $O(|t|nK)$ operations. If γ is of the form $c(x).\gamma'$, then $\mathrm{FV}(\gamma') = \mathrm{FV}(\gamma') \cap \mathrm{FV}(\phi')$, since $\mathtt{w}(\gamma) = 1$. Hence, any recursive call to $\mathtt{ans}(\gamma', t, v')$ for $v' \in [\![c]\!](t)$ returns at most K valuations. Moreover, there are at most $|t|$ nodes satisfying $[\![c]\!](t)$, so that the recursive call $\mathtt{ans}(\gamma, t, v)$ performs at most $|t| + nK|t|$ operations.
Finally, since we use memoization, there are at most $|t||\phi'|$ recursive call to ans, so that the whole complexity of ans on input, ϕ', t and $\mathtt{root}(t)$ is $O(|\phi'||t|^2 nK))$. □

8 Conclusion

8.1 Summary.

We proposed and investigated an n-ary query language $\mathcal{C}(L)$ in which queries are specified as composition of monadic queries. The choice of the underlying monadic query language L is parametric, so that we can express a wide variety of n-ary query specification languages, for

```
1  let ans(φ,t,u)   =   if Sat[φ,u] then
2                        match φ with
3                          | ⊤ → {ε}
4                          | c(x).φ' → ⋃_{u'∈Q_a(c,u)} {(x,u')·ν | ν ∈ ans(φ',t,u')}
5                          | φ' ∧ φ'' → ans(φ',t,u) × ans(φ'',t,u)
6                          | φ' ∨ φ'' → ans(φ',t,u) ∪ ans(φ'',t,u)
7                          | ∃xφ → {ν : dom(ν) = dom(ν')\x, ν = ν'|_{dom(ν)\x}, ν' ∈ ans(φ,t,u)}
8                        else ∅
9  in
10 ans(φ,t,root(t))
```

Figure 5. Answering algorithm with implicit memoization

instance composition of XPath formula, Monadic Datalog programs or node selection automata. We proved our language to capture MSO as soon as the underlying monadic query language capture MSO too. We proved the satisfiability problem to be NP-hard and proposed an efficient fragment $\mathcal{L}(L)$ of the composition language which remains MSO-complete as soon as L captures MSO. We gave an algorithm for the query answering problem in time $O(M(\phi,t)|t||\phi| + |\phi|^2|t|^2|\phi(t)|)$, where $|\phi(t)|$ is the output size and $M(\phi,t)$ is the maximal complexity of the query answering problem over subtrees of t, of the monadic queries appearing in ϕ.

8.2 Future Work.

A more practical aspect is the extension of the existing implementation of query composition to the algorithms in *Section 7* and the comparison of their query answering efficiencies with other querying languages, such as implementations of XQuery, and programming languages such as $\mathbb{C}$Duce .

We would like to investigate the correspondence – mentioned in *Section 4* between the underlying query formalism of Lixto and our query composition language over Monadic Datalog programs. In particular, we think that there exists a systematic translation between the two formalisms.

Finally, in some cases it seems to be more efficient to have the possibility to navigate everywhere in the tree, without restriction on subtrees. The binary query example given in *Section 3*, on the tree of figure 1 seems to be more natural when one first selects a node labeled by `name`, and then its sibling. In this way it is interesting to investigate the more general problem of binary query composition.

We would like to thank Manuel Loth who worked on the implementation of monadic query composition.

9 References

[1] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *28th International Conference on Very Large Data Bases*, pages 119–128, 2001.

[2] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an XML-centric general-purpose language. *ACM SIGPLAN Notices*, 38(9):51–63, 2003.

[3] Julien Carme, Aurlien Lemay, and Joachim Niehren. Learning node selecting tree transducer from completely annotated examples. In *7th International Colloquium on Grammatical Inference*, volume 3264 of *Lecture Notes in Artificial Intelligence*, pages 91–102. Springer Verlag, 2004.

[4] Giuseppe Castagna. Patterns and types for querying XML. In *10th International Symposium on Database Programming Languages*, Lecture Notes in Computer Science. Springer Verlag, August 2005.

[5] Markus Frick and Martin Grohe. The complexity of first-order and monadic second-order logic revisited. In *Proc. LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 215–224, Washington, DC, USA, 2002. IEEE Computer Society.

[6] G. Gottlob, C. Koch, R. Baumgartner, M. Herzog, and S. Flesca. The Lixto data extraction project - back and forth between theory and practice. In *23rd ACM SIGPLAN-SIGACT Symposium on Principles of Database Systems*, pages 1–12. ACM-Press, 2004.

[7] G. Gottlob, C. Koch, and K. Schulz. Conjunctive queries over trees, 2004.

[8] Georg Gottlob and Christoph Koch. Monadic datalog and the expressive power of languages for web information extraction. In *21rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 17–28. ACM-Press, 2002.

[9] Georg Gottlob and Christoph Koch. Monadic queries over tree-structured data. In *17th Annual IEEE Symposium on Logic in Computer Science*, pages 189–202, Copenhagen, 2002.

[10] Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient algorithms for processing xpath queries. *ACM Transactions on Database Systems*, 30(2):444–491, 2005.

[11] Haruo Hosoya and Benjamin Pierce. Regular expression pattern matching for XML. *Journal of Functional Programming*, 6(13):961–1004, 2003.

[12] Leonid Libkin. Logics over unranked trees: an

overview. In *Automata, Languages and Programming: 32nd International Colloquium*, number 3580 in Lecture Notes in Computer Science, pages 35–50. Springer Verlag, 2005.

[13] Frank Neven and Jan Van Den Bussche. Expressiveness of structured document query languages based on attribute grammars. *Journal of the ACM*, 49(1):56–100, 2002.

[14] Frank Neven and Thomas Schwentick. Query automata. In *Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems*, pages 205–214, 1999.

[15] Frank Neven and Thomas Schwentick. Query automata over finite trees. *Theoretical Computer Science*, 275(1-2):633–674, 2002.

[16] Joachim Niehren, Laurent Planque, Jean-Marc Talbot, and Sophie Tison. N-ary queries by tree automata. In *10th International Symposium on Database Programming Languages*, volume 3774 of *Lecture Notes in Computer Science*, pages 217–231. Springer Verlag, September 2005.

[17] J. W. Thatcher and J. B. Wright. Generalized finite automata with an application to a decision problem of second-order logic. *Mathematical System Theory*, 2:57–82, 1968.

Type Checking For Functional XML Programming Without Type Annotation (Extended Abstract)

Akihiko Tozawa
IBM Tokyo Research Lab
1623-14, Shimotsuruma, Yamato-shi,
Kanagawa-ken 242-8502, Japan
atozawa@jp.ibm.com

1 Introduction

We discuss the type checking for XML programming with higher-order functions. Our type checking does not require type annotations on programs. This is beneficial for programmers. In XDuce [HP03] and CDuce [FCB02], programmers always need to figure out, for all functions in the program, what type annotations are necessary. This task sometimes becomes very tedious, in particular, when structures of target XML documents are complex.

To achieve the type checking without type annotation, we use the *tree transducer* type-checking technique. In particular, we employ the *high-level tree transducer*, first introduced by Engelfriet [EV88]. We can enjoy much benefits of functional programming with this transducer, because we can use higher order functions. Given input trees, the high-level tree transducer emits functional values.

Our method has two steps. The first step is a conversion from functional programs to high-level tree transducers. The second step is the inverse type inference, which receives an *output* XML type and a high-level tree transducer and creates an *input* XML type.

- The conversion in the first step is made possible by imposing restrictions on functional programs. These restrictions ensure that (1) a program is not allowed to examine what it creates, (2) a program does not receive more than one input tree, (3) the number of internal states a program can reach is finite. These restrictions are obviously necessary, and are even sufficient for the conversion to tree transducers. We impose these restrictions by using simple types called *sort*s.
- The key idea for the second step is the abstract interpretation of values emitted by transducers. For this interpretation, we start from the finite algebras called *binoids*. Any XML type can be captured by some binoid and homomorphism from XML values to this binoid. Such homomorphism can be extended to functional values. As far as type-checking is concerned, we always consider functional values under abstract interpretation by this homomorphism. Our inverse type inference is done by combining Maneth's algorithm and such abstract interpretation.

Let us outline the rest of the paper. Section 2 discusses the problem we deal with in a ML-style functional language. Section 3 gives a formal discussion and k-level tree transducers. Section 4 gives the type checking algorithm. Section 5 summarizes the related work. Section 6 discusses the future work.

2 The Language and Problem

An ML-like Language for XML Programming We first introduce an ML-like yet simply-typed functional language with higher order functions. This language supports XML programming. In particular, this language manipulates two XML values, *input* XML values and *output* XML values. Input XML values are only processed. We however cannot create input XML values in the language, so that such values are always supplied from the outer world. On the other hand, output XML values, or we can say, *non-observable* XML values, are only constructed. We do not have any method to inspect their structures.

Let us explain the language step by step. As an example, we use the following program representing the identity tree transformation.

$$\begin{array}{l}\texttt{letrec } id^{(i\to o)}\ x := \\ \quad *x[\texttt{if } x \models 1 \texttt{ then } id\ x\cdot 1 \texttt{ else } ()], \\ \quad (\texttt{if } x \models 2 \texttt{ then } id\ x\cdot 2 \texttt{ else } ()) \\ \texttt{in} \\ \quad id\end{array}$$

First, we have sorts. In the program, we see a superscript $i \to o$ appearing on id. This superscript indicates a sort, i.e., simple type, of the function variable id. Let $\mathcal{B} = \{i, o, \mathbb{B}, \mathbb{L}\}$. This $\mathcal{B}$ is the set of base sorts. Sort i corresponds to input XML values. Input XML values indicate some nodes in the input XML document. Sort o corresponds to output XML values. Output XML values are sequences of XML trees. Sorts $\mathbb{B}$ and $\mathbb{L}$ are sorts for boolean values and labels, respectively. We use b to range over base sorts. We extend base sorts $\mathcal{B}$ to sorts $\mathcal{S}(\mathcal{B})$ for functional values as

$$\mathcal{S}(\mathcal{B}) \ni s ::= b \mid s \to s$$

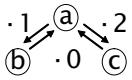
where $b \in \mathcal{B}$. We use s to range over sorts. Here $\to$ associates to the right as usual. Note that in sorts, their constructors $i, o, \mathbb{B}, \mathbb{L}$ and $\to$, are just syntactical objects.

In the rest of the paper, we often make sorts of function variables explicit for readability. In practice, sorts of variables as well as those of expressions can be, though not uniquely, inferred by known unification-based algorithms such as the one in a textbook [Mit96]. Note that sorts are not types in this paper – we will introduce types themselves later.

Next, this language allows the set of specific constant primitives. They operate on input and output XML values of sorts i and o.

XML instances given as inputs, are seen as binary trees which are navigated by using the set of primitives. For input XML values of

sort i, we define operators $\cdot 0$, $\cdot 1$, and $\cdot 2$ as follows.



This figure illustrates an XML instance a[b[]],c[] seen as a binary tree. Assume that a node $x^{(i)}$ is the root of the above XML instance. From x, we reach a node labelled by b using $x{\cdot}1$, and a node labelled by c using $x{\cdot}2$, and from these nodes we can move back to the root node by $x{\cdot}1{\cdot}0$ or $x{\cdot}2{\cdot}0$. Predicates $x \models 0$, $x \models 1$ and $x \models 2$ represent tests whether it is allowed to move to that direction. If there are no nodes in that direction the test fails. We can also obtain the label of the node by $*x$. For instance, we have $*x = \mathtt{a}$ on the root node x.

For construction of output XML values, we have a constant $()^{(o)}$ which represents a null sequence, and two operators; $\_[\_]^{(\mathbb{L}\to o\to o)}$ and $(\_,\_)^{(o\to o\to o)}$. The operator $\_[\_]^{(\mathbb{L}\to o\to o)}$ creates a node $\ell[t]$ from the label ℓ and an output XML value t. The operator $(\_,\_)^{(o\to o\to o)}$ concatenates two output XML values.

Furthermore, the language has the `if`-construct and equality test on the finite set of labels. We also have `letrec` for defining mutually recursive functions.

Finally, we emphasize what this language does *not* have. Although we can convert an XML value x of sort i into the same value of sort o using $id^{(i\to o)}\ x$, the conversion in the reverse direction is not expressed in the language. Namely, the language does not have a primitive constant of sort $o \to i$ representing this reverse conversion. Neither we can define a program performing such a conversion. In general, our language can neither create input XML values, e.g., $x^{(i)} := \mathtt{a}[]$, nor inspect the information of some output XML values, e.g., $*t$ for an output XML value t.

Using Higher-order Functions In XML programming, the use of higher order functions have a number of advantages. Here we look through several use cases of higher order functions through examples. Note that we later translate functional programs into transducers, and we here only discuss functions to which such translation can be applied.

A typical higher order function is the *map* function, which applies a function given as an argument to a set of elements at once. In XML programming, it is particularly useful to have map functions which apply argument functions to nodes selected by a certain criteria, e.g., children, following siblings, etc. The following functions *chilren* and *siblings* take argument functions of sort $i \to o$, and return the concatenation of the results of applications.

$$\begin{array}{l} \mathit{children}^{(i\to(i\to o)\to o)}\ x\ f := \mathtt{if}\ x \models 1\ \mathtt{then}\ \mathit{siblings}\ x{\cdot}1\ f\ \mathtt{else}\ () \\ \mathit{siblings}^{(i\to(i\to o)\to o)}\ x\ f := f\ x, \mathtt{if}\ x \models 2\ \mathtt{then}\ \mathit{siblings}\ x{\cdot}2\ f\ \mathtt{else}\ () \end{array}$$

Example 1. *For example, when applied to the root node x of an input tree* a[b[],c[],d[]], *children x f returns $f\ (x{\cdot}1), f\ (x{\cdot}1{\cdot}2), f\ (x{\cdot}1{\cdot}2{\cdot}2)$.*

Note that functions such as *chilren* and *siblings* are usually supplied as library functions, rather than being a part of the user program. With such library functions, programmers do not have to deal with primitive navigation operators such as $x{\cdot}m$ ($x{\cdot}0$, $x{\cdot}1$ and $x{\cdot}2$).

The function *dept* in Figure 2 implements the transformation in Figure 1. The function recursively applies itself to a set of nodes selected by *children*, *root*, etc. Readers familiar with XSLT should be aware that the program is written in a style similar to XSLT programs.

Not only map functions, but we can also provide library functions for testing the document structure. For example, a function which tests the existence of a child node of $x^{(i)}$ with a certain label $l^{(\mathbb{L})}$ i.e., corresponding to the XPath predicate $[x/l]$, can be written in a manner similar to the function *children*.

More generally, we can even implement a deterministic (binary) tree automaton which tests the substructure of x through the following trick. This function *autom* takes a transition function *trans* and initial state *ini* as arguments.

$$\begin{array}{l} \mathit{autom}^{(i\to\mathbb{L}\to(\mathbb{L}\to\mathbb{L}\to\mathbb{L}\to\mathbb{L})\to\mathbb{L})}\ x\ \mathit{ini}\ \mathit{trans} := \\ \quad \mathit{trans} \\ \quad\quad (*x) \\ \quad\quad (\mathtt{if}\ x \models 1\ \mathtt{then}\ \mathit{autom}\ x{\cdot}1\ \mathit{ini}\ \mathit{trans}\ \mathtt{else}\ \mathit{ini}) \\ \quad\quad (\mathtt{if}\ x \models 2\ \mathtt{then}\ \mathit{autom}\ x{\cdot}2\ \mathit{ini}\ \mathit{trans}\ \mathtt{else}\ \mathit{ini}) \end{array}$$

The transition function $\mathit{trans}\ l\ q_1\ q_2$ takes a label l, two successor states q_1 and q_2, and returns the result of the transition. Here, we assume that we encode the finite set of states of the automaton as a subset of the finite label set of sort $\mathbb{L}$. We do not show the detail but this technique can be also extended to implement pattern match constructs in the style of XDuce based on regular expression patterns [HP03].

Another interesting application of higher order functions is to use them for representing XML values containing holes. Such holes are also called *gap* in the language JWIG [CMS02]. For instance, a value $p^{(o\to o)}$ represents a (first-order) gapped value whose gaps can be filled at once by a value $v^{(o)}$ by the application $(p\ v)^{(o)}$. E.g., $p^{(o\to o)}\ v :=$ dept[v] is a gapped value dept[$\Box$] where $\Box$ is the position of a gap. This gap can be filled by emp[] as p emp[] = dept[emp[]].

We can implement a set of *gap operators* using higher order functions.

$$\begin{array}{lllll} (i) & \mathit{gap}^{(o\to o)}\ v & := & v \\ (ii) & \mathit{nogap}^{(o\to o\to o)}\ v\ w & := & v \\ (iii) & \mathit{concgap}^{((o\to o)\to(o\to o)\to o\to o)}\ p\ q\ v & := & p\ v, q\ v \\ (iv) & \mathit{nodegap}^{(\mathbb{L}\to(o\to o)\to o\to o)}\ l\ p\ v & := & l[p\ v] \\ (v) & \mathit{pluggap}^{((o\to o)\to(o\to o)\to o\to o)}\ p\ q\ v & := & p\ (q\ v) \end{array}$$

The gap operators implement the following operations.

Example 2. *Some examples on the use of the gap operators. (i) gap =* $\Box$. *(ii) nogap* dept[] *is a gapped value* dept[] *with no gaps. (iii) concgap* dept[$\Box$] $\Box$ = dept[$\Box$],$\Box$. *(iv) nodegap* comp dept[$\Box$] = comp[dept[$\Box$]]. *(v) pluggap* dept[$\Box$] dept[$\Box$] = dept[dept[$\Box$]].

The function *dept* in Figure 2 traverses the input tree many times due to the call to the *root* function. Interestingly, a similar function can be written by the function using gapped values, shown in Figure 3.[1] This function computes an answer by the single traversal.

The earlier examples of higher order functions are just useful in

[1] In this program we use sort $\mathbb{B} \to s$ to implement pairs. It is not difficult to extend the language with pairs and projections. We do not do this here for simplicity.

writing concise programs. Their use is however not essential. The last example using gapped values, essentially uses higher order functions. As Engelfriet observed [EV88], by raising the order of sorts for output values, i.e., o, $o \to o$, $(o \to o) \to o \to o$ and so on, we can arbitrarily increase the expressive power of the language.

Type-Checking Problem Types for XML values, i.e., instances of sorts i or o, are described by tree regular expressions, such as $\tau = (\mathtt{a}[\mathtt{b}[]^* \mid c[]])^*$. For example, *id* transforms any XML value into itself, hence a value of type $(\mathtt{a}[\mathtt{b}[]^* \mid c[]])^*$ into the value of the same type. This observation is denoted by $id : (\mathtt{a}[\mathtt{b}[]^* \mid c[]])^* \to (\mathtt{a}[\mathtt{b}[]^* \mid c[]])^*$. For a function with sort $i \to o$, the type checking problem $f_{\mathbf{I}}^{(i \to o)} : \upsilon_{\mathbf{I}} \to \tau_{\mathbf{I}}$ can be stated as follows.

Problem 1. *(Type checking) Given a program $f_{\mathbf{I}}^{(i \to o)}$, an input type $\upsilon_{\mathbf{I}}$ and output type $\tau_{\mathbf{I}}$, the type checking problem $f_{\mathbf{I}}^{(i \to o)} : \upsilon_{\mathbf{I}} \to \tau_{\mathbf{I}}$ is to test whether or not the transformation of any XML value of type $\upsilon_{\mathbf{I}}$ produces an XML value of type $\tau_{\mathbf{I}}$.*

In understanding Problem 1, we need to clarify the case when the transformation does not terminate. For example, the program given in Figure 2 does not terminate if there are occurrences of `dept`-nodes inside `emp`-nodes in the input tree.

```
comp[                      comp[
 dept[],                    dept[
 emp[akihiko[]],   ⇒         emp[akihiko[]],
 emp[dept[]]                emp[
]                            dept[
                               emp[akihiko[]],
                               ···
```

We can use the input type $\upsilon_{\mathbf{I}}$ to guarantee that `dept`-nodes never occurs inside `emp`-nodes, so that the function terminates for any input of type $\upsilon_{\mathbf{I}}$. There is a choice whether or not we include the non-termination in type errors. In this paper, we chose to include it.

Restrictions on the Functional Language We can solve the type checking problem for the language introduced so far, when the program of interest can be translated into a high-level tree transducer. Unfortunately, not all programs can be translated into such tree transducers. We here explain sufficient restrictions on programs which make this translation possible.

- Any functions or variable $f^{(s)}$ declared in `let` or `letrec` as $f^{(s)}x := e$, either has their sort $s = b$ or $s = s_1 \to \cdots \to s_{n-1} \to b$, such that none of $s_2, ..., s_{n-1}, b$ are i. Namely, only the first argument can be of sort i. Note that we do not restrict sorts in the form $i \to s$ to appear other argument positions, e.g., $children^{(i \to (i \to o) \to o)}$.
- Any function of sort $i \to s$ must be declared in the top-level `letrec` of the program. In other words, they must not be defined in a `letrec` within another `letrec`.

These two restrictions correspond to the fact that the tree transducer only have a single input parameter (= first restriction) and a finite set of states (= first and second restrictions). Obviously, we cannot have a function definition of sort $i \to \underline{i} \to s$, because it means that this function has multiple input parameters (we underline the erroneous part). In the translation, functions of sort $i \to s$ are seen as the finite set of states. This is guaranteed only if there are finitely many possibilities for such functions. Assume that there is a function f with sort $(i \to s) \to \underline{i} \to s$ and g with sort $i \to s$. In our language, we can create fg, $f(fg) = f^2 g$, $f(f(fg)) = f^3 g$, and so on. In particular, we can enumerate such functional values up to $f^n g$, where, for example, n is the size of the input to the program. This makes the translation into tree transducers impossible, since the number of states should not be related to the size of any input. The use of nested `let` and `letrec` for functions of sort $i \to s$ also causes the same problem.

Let us give another explanation from a different point of view. As we discussed in the introduction, the decidability results for the tree transducer type checking come from the fact that the *inverse* image $f_{\mathbf{I}}^{-1}(\tau_{\mathbf{I}})$ of a transformation with respect to a regular language $\tau_{\mathbf{I}}$, is always regular. Since the subsumption for regular languages is decidable, the type checking amounts to check whether $\upsilon_{\mathbf{I}}$ is contained by $f_{\mathbf{I}}^{-1}(\tau_{\mathbf{I}})$.

Here, consider the following program which has a sort $i \to \underline{i} \to o$.

```
letrec cmp^(i→i→o) x y :=
  if not(x ⊨ 2)  &&  not(y ⊨ 2) then ok[] else cmp x·2 y·2
in cmp
```

This program checks whether two sequences starting from nodes x and y have the same length (*not* and && here can be defined using `if`). Clearly the inverse image of such a program for $\tau_{\mathbf{I}} = \mathtt{ok}[]$ does not have a regular property. For example, we cannot test by means of tree automata, if a tree $l[t_1], t_2$ has the width of t_1 is equal to the width of t_2. This is the source of difficulty with programs having sorts such as $i \to \underline{i} \to o$.

3 Values, Tree Automata and High-level tree transducers

We introduce XML values, and then tree automata which are the model of XML types. The high-level tree transducer is the model of XML transformations as given, using a functional language, in examples in the last section. We discuss its syntax and semantics in the latter half of this section.

Here are some notations used throughout. We consistently use bold font, e.g., $\boldsymbol{a}$, to emphasize meta-variables denoting words or tuples. We use $\epsilon \in A^*$ for an empty word, and an associative operator $(\cdot)$ for word concatenation. We let $\mathbb{B} = \{\mathrm{true}, \mathrm{false}\}$ be the set of boolean values. This $\mathbb{B}$ appears in the text, so that there should be no confusion with the symbol $\mathbb{B}$ appearing in sorts.

XML Values An XML value is a sequence of unranked ordered trees over the finite set $\mathbb{L}$ of labels. The set of XML values is defined as follows.

$$V \ni t ::= () \mid \ell[t] \mid t, t$$

where $\ell \in \mathbb{L}$. We omit () if it is directly enclosed in $\ell[\_]$. We assume that $\_, \_$ is associative and () is an identity. As explained earlier, an XML value can also be seen as a binary tree, since each t is represented either as $\ell[t_1], t_2$ or (). For each t, its domain $dom(t) \subseteq \{1,2\}^*$ is the set of locations, when seen as a binary tree, of that tree. We define the set of tree nodes U by

$$U = \bigcup_{t \in V} \Big(\{t\} \times dom(t) \Big).$$

That is, U is the set of all nodes in all trees. The label of a node $u \in U$ is denoted by $*u \in \mathbb{L}$. We can move inside XML trees by the

```
comp[                          comp[
  dept[],                        dept[
  emp[akihiko[]],      ⇒           emp[akihiko[]],
  emp[yoshinori[]]               emp[yoshinori[]]
]                              ]
                               ]
```

Figure 1. The *dept*-transformation. Namely, we collect all nodes labelled `emp`, as well as subtrees of such `emp`-nodes, and put them into all `dept`-nodes in the document.

```
letrec
(* libraries *)
  children^(i→(i→o)→o) x f := if x ⊨ 1 then siblings x·1 f else ()
  siblings^(i→(i→o)→o) x f := f x,if x ⊨ 2 then siblings x·2 f else ()
  root^(i→(i→o)→o) x f := if x ⊨ 0 then root x·0 f else f x
(* user program *)
  dept^(i→o) x :=
    if *x = dept then dept[root x emp]
    else if *x = emp then ()
    else *x[children x dept]
  emp^(i→o) x :=
    if *x = emp then *x[children x dept]
    else children x emp
in
dept
```

Figure 2. Function *dept x* first looks at the label of the node *x*. If it is `dept`, the function creates a copy of this `dept`-node, in which it puts the result of call to the function *emp* at the root. If the label is `emp` the function *dept* skips this node, and otherwise it creates the copy of the given node. The function *emp* collects and copies all `emp` nodes inside the tree. This function *emp* has some problem, because it again calls *dept* to copy its substructure. See the Type-Checking Problem paragraph.

```
letrec
...
(* user program *)
  deptgap^(i→𝔹→o→o) x :=
    let n^(𝔹→o→o) b := nogap () in
    let p1^(𝔹→o→o) := if x ⊨ 1 then deptgap x·1 else n in
    let p2^(𝔹→o→o) := if x ⊨ 2 then deptgap x·2 else n in
    let l^(𝕃) := *x in
    let p0^(𝔹→o→o) b :=
      if b then
        if l = dept then
          concgap (nodegap dept gap) (p2 true)
        else if l = emp then
          p2 true
        else
          concgap (nodegap l (p1 true)) (p2 true)
      else
        if l = emp then
          pluggap (concgap (nogap (p1 true ())) gap) (p2 false)
        else
          pluggap (p1 false) (p2 false)
    in p0
  dept^(i→o) x :=
    let p := deptgap x in
    p true (p false ())
in
dept
```

Figure 3. A similar function as the one in Figure 2, using first order gapped values. The function *deptgap* returns $p^{(\mathbb{B}\to o\to o)}$ which represents two gap values, namely *p true* and *p false*. For example, for the left hand side document of Figure 1, *deptgap* returns a pair of gapped values *p true* = `comp [dept[□]]` and *p false* = `emp[akihiko[]],emp[yoshinori[]],□`. They are finally plugged by *p true* (*p false* ()) and create the resulting document on the right hand side of Figure 1.

operator $(\cdot m)$ $(m = 0,1,2)$. For $\boldsymbol{k} \in dom(t)$

$$\begin{array}{ll} (t,\boldsymbol{k})\cdot m = (t,\boldsymbol{k}\cdot m) & \text{if } m = 1,2 \text{ and } \boldsymbol{k}\cdot m \in dom(t) \\ (t,\boldsymbol{k}\cdot k)\cdot m = (t,\boldsymbol{k}) & \text{if } m = 0 \text{ and } k = 1,2 \\ u\cdot m = \bot & \text{otherwise} \end{array}$$

The value $\bot$ here represents a non-existing node such that $\bot \notin U$. Finally, the set of root nodes $\Lambda(U) \subseteq U$ is a set $\{(t,\epsilon) \mid t \in V\}$.

XML Types and Tree Automata We introduce XML types. Each XML type represents a certain set of XML values. We use metavariables τ, υ for ranging over XML types throughout the paper. As a candidate of models of XML types, we have tree regular expressions as defined by Hosoya et al. For $\ell \in \mathbb{L}$, and α ranging over a set of type variables, tree regular expressions use the syntax such as

$$T^{\mathbf{XML}} \ni \tau,\upsilon ::= () \mid \ell[\tau] \mid \tau\,\mathtt{,}\,\tau \mid \tau \cup \tau \mid \tau^* \mid \mathtt{letrec}\ \alpha := \tau; \ldots\ \mathtt{in}\ \tau \mid \alpha$$

Example 3. *This is an example of XML type. By constraining input XML values, this type guarantees the termination of the program dept in Figure 2.*

$$\mathtt{letrec}\ ds := \mathtt{dept}[es]^*;\ es := \mathtt{emp}[]^*\ \mathtt{in}\ \mathtt{comp}[ds]$$

It says that the root node is always comp *and in which we have a sequence of* dept *nodes. Inside* depts*, we have* emp *nodes, and so on.*

In this paper, we do not directly discuss the semantics $[\![\tau]\!] \subseteq V$ of the above syntax. Instead, we introduce tree automata which is well-known as a canonical model of XML types τ. Here we actually introduce three forms of them. The first one is the most standard.

Definition 1. *A (total) non-deterministic tree automaton* $\mathcal{M} = (Q,\mathbb{L},\Delta,F,\bullet)$ *is a tuple where* $\Delta \subseteq \mathbb{L}\times Q\times Q\times Q$ *is a set of transitions,* $F \subseteq Q$ *is a set of final states and* $\bullet \in Q$ *is an initial state. A mapping* $\mu \in U \to Q$ *is called a run of* $\mathcal{M}$ *if* $(*u,\mu(u),\mu(u\cdot 1),\mu(u\cdot 2)) \in \Delta$ *for any* $u \in U$*, where we define* $\mu(\bot) = \bullet$*. An XML value with root node* $u \in \Lambda(U)$ *is accepted if there is a run* μ *such that* $\mu(u) \in F$*.*

We can assume for each XML type τ that we have a tree automaton $\mathcal{M}(\tau)$ which defines the semantics $[\![\tau]\!] = \{t \in V \mid t \text{ accepted by } \mathcal{M}(\tau)\}$. This is a standard assumption in the study of typed XML programming. See Hosoya et al. [HVP00], for this detail.

The second model of XML types has a form of algebra whose domain is finite. This algebra is called *binoid* [PQ68] in the literature, and is similar to *syntactic monoid* [Per90] for word languages. We employ this representation as a canonical model of output XML types in the type inference algorithm in Section 4. As we can see from the definition, this algebra classifies a set V of XML values into a certain set of finite equivalence classes. The equivalence classes are still diverse enough to check whether or not, an arbitrary XML construction creates the result inside $[\![\tau_{\mathbf{I}}]\!]$. In other words, binoids provide the means of *abstract interpretation* of XML values.

Definition 2. *A binoid for* $\tau_{\mathbf{I}}$ *is an algebra* $\mathcal{V}(\tau_{\mathbf{I}}) = (\mathcal{V},\bullet,F,(\_[\_]),(\_\,\mathtt{,}\,\_))$ *such that (1)* $\mathcal{V}$ *is a finite set,* $\bullet \in \mathcal{V}$ *and* $F \subseteq \mathcal{V}$*, and (2)* $(V,(),[\![\tau^{\mathbf{I}}]\!],(\_[\_]),(\_\,\mathtt{,}\,\_))$ *is homomorphic to* $\mathcal{V}(\tau_{\mathbf{I}})$*. That is, we have a mapping* $(\_^\circ) \in V \to \mathcal{V}$ *satisfying (i)* $()^\circ = \bullet$*, (ii)* $v \in [\![\tau^{\mathbf{I}}]\!]$ *iff* $v^\circ \in F$*, (iii)* $(\ell[t])^\circ = \ell[t^\circ]$*, and (iv)* $(t\,\mathtt{,}\,t')^\circ = t^\circ\,\mathtt{,}\,t'^\circ$*.*

An algorithm, given a non-deterministic tree automaton representing $\tau_{\mathbf{I}}$, that constructs one binoid satisfying the above definition is known. For binoids $\mathcal{V}(\tau_{\mathbf{I}})$ with homomorphism $^\circ$, in what follows, we often use $()^\circ$ and $[\![\tau_{\mathbf{I}}]\!]^\circ$ instead of $\bullet$ and F above, respectively.

Example 4. *Consider the XML type* comp[*ds*] *in Example 3. Here, we give one binoid corresponding to this XML type. We here take a certain set of tree regular expressions as the domain of the binoid* $\mathcal{V}(\mathtt{comp}[ds])$*. In the following, assume that* $\tau_{\mathbf{E}}$ *represents a type for values that do not belong to other elements in the domain of* $\mathcal{V}(\mathtt{comp}[ds])$*.*

$$\begin{array}{l} \mathcal{V} = \{(), \mathtt{dept}[es]^+, \mathtt{emp}[]^+, \mathtt{comp}[ds], \tau_{\mathbf{E}}\} \\ ()^\circ = () \\ [\![\mathtt{comp}[ds]]\!]^\circ = \{\mathtt{comp}[ds]\} \end{array}$$

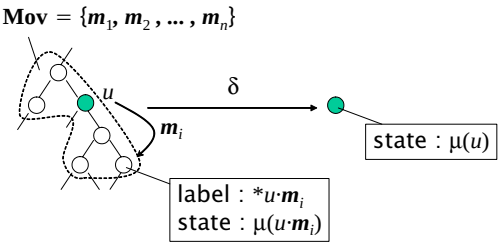
$$\_[\_] = \left\{ \begin{array}{lcl} \mathtt{dept},\mathtt{emp}[]^+ & \mapsto & \mathtt{dept}[es]^+ \\ \mathtt{dept},() & \mapsto & \mathtt{dept}[es]^+ \\ \mathtt{comp},\mathtt{emp}[]^+ & \mapsto & \tau_{\mathbf{E}} \\ \ldots & & \end{array} \right\}$$

$$\_\,\mathtt{,}\,\_ = \left\{ \begin{array}{lcl} (),\mathtt{dept}[es]^+ & \mapsto & \mathtt{dept}[es]^+ \\ \mathtt{dept}[es]^+,\mathtt{dept}[es]^+ & \mapsto & \mathtt{dept}[es]^+ \\ \ldots & & \end{array} \right\}$$

We can confirm that this $\mathcal{V}(\mathtt{comp}[ds])$ *satisfies Definition 2 by using* $(\_^\circ) \in V \to \mathcal{V}(\mathtt{comp}[ds])$ *such that* $t \in [\![t^\circ]\!]$*. For example, take* $t = \mathtt{dept}[]\,\mathtt{,}\,\mathtt{dept}[]$*. We have* $(t\,\mathtt{,}\,t)^\circ = \mathtt{dept}[es]^+ = \mathtt{dept}[es]^+\,\mathtt{,}\,\mathtt{dept}[es]^+ = t^\circ\,\mathtt{,}\,t^\circ$*.*

We lastly give yet another form of tree automaton, which can be efficiently converted into a non-deterministic tree automaton. This automaton provides a trick which will be used at the last step of the type inference algorithm as the model of inferred input XML types. A short explanation of the automaton is that (1) it is a variant of deterministic 2-way tree automaton; (2) it allows cyclic runs; and (3) the transition function can look at a set of locations in the tree bounded by the finite set **Mov**.

We first explain the transition function δ of the look-around tree automaton. This δ takes as an argument, a set of information (the state-label pair) for each node $u\cdot\boldsymbol{m}_i$ at the relative position $\boldsymbol{m}_i$ in $\mathbf{Mov} = \{\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots \boldsymbol{m}_n\}$, and returns the state for the node u.



For each node u, this set of information is given as a *look-around function*, say $h \in \mathbf{Mov} \to (\mathbb{L}\times Q)^\bot$ $(= (\mathbb{L}\times Q) \uplus \{\bot\})$. This h takes an argument $\boldsymbol{m}$ representing a relative position, and returns the pair of the label of, and the state assigned to, the node $u\cdot\boldsymbol{m}$. If there is no node at $\boldsymbol{m}$ i.e., $u\cdot\boldsymbol{m} = \bot$, this h returns $\bot$.

Definition 3. *A look-around tree automaton is* $\mathcal{M} = (Q,\mathbb{L},\mathbf{Mov},\delta,F)$ *such that* $\mathbf{Mov} \subset \{0,1,2\}^*$ *is a finite set of moves,* $\delta \in (\mathbf{Mov} \to (\mathbb{L}\times Q)^\bot) \to Q$ *is a transition function. A mapping* $\mu \in U \to Q$ *is called a run of* $\mathcal{M}$*, if* $\mu(u) = \delta(h)$ *for all* $u \in U$*, where the look-around function* $h \in \mathbf{Mov} \to (\mathbb{L}\times Q)^\bot$ *for this* u*, is defined from* μ *as*

$$\begin{array}{ll} h(\boldsymbol{m}) = (*u\cdot\boldsymbol{m}, \mu(u\cdot\boldsymbol{m})) & \text{if } u\cdot\boldsymbol{m} \in U \\ h(\boldsymbol{m}) = \bot & \text{otherwise} \end{array}$$

$$
\begin{array}{lll}
\mathbf{Term}(C,X) \ni e & ::= & \\
& \mid\ c & (c \in C, \text{constants}) \\
& \mid\ x & (x \in X, \text{variables}) \\
& \mid\ ee & (\text{application}) \\
& \mid\ \texttt{if}\ e\ \texttt{then}\ e\ \texttt{else}\ e & (\text{conditional}) \\
& \mid\ \texttt{letrec}\ f\boldsymbol{x} := e; \cdots\ \texttt{in}\ e & (\text{recursive def.})
\end{array}
$$

$$
\begin{array}{lll}
\mathbf{Con}(\mathbb{L}) \ni c & ::= & \\
& \mid\ \mathit{true}^{(\mathbb{B})}, \mathit{false}^{(\mathbb{B})} & (\text{boolean constants}) \\
& \mid\ \ell^{(\mathbb{L})} & (\ell \in \mathbb{L}, \text{label constants}) \\
& \mid\ (\_ = \_)^{(\mathbb{L}\to\mathbb{L}\to\mathbb{B})} & (\text{label equality}) \\
& \mid\ ()^{(o)} & (\text{empty tree}) \\
& \mid\ (\_[\_])^{(\mathbb{L}\to o\to o)} & (\text{node constructor}) \\
& \mid\ (\_,\_)^{(o\to o\to o)} & (\text{tree concatenation})
\end{array}
$$

Figure 4. Definition of $\mathbf{Term}(C,X)$ and $\mathbf{Con}(\mathbb{L})$.

The automaton accepts u iff $\mu(u) \in F$ for some μ.

Example 5. *A deterministic tree automaton uses a transition function $\delta \in \mathbb{L}\times Q\times Q \to Q$ instead of the transition relation $\Delta \in \mathbb{L}\times Q\times Q\times Q$ of non-deterministic tree automaton. A deterministic tree automaton $(Q,\mathbb{L},\delta,\bullet,F)$ is an instance of look-around automaton. For this, define the transition function δ' of the look-around automaton $(Q,\{\epsilon,1,2\},\mathbb{L},\delta',F)$, as*

$$\delta'(h) = \delta(\mathit{lab}(h(\epsilon)), \mathit{st}(h(1)), \mathit{st}(h(2)))$$

where $\mathit{lab}(\ell,q) = \ell$, $\mathit{st}(\ell,q) = q$ and $\mathit{st}(\bot) = \bullet$. Note here that we always have $h(\epsilon) \neq \bot$ because h is computed for each $u \in U$,

Look-around tree automata can be efficiently converted into non-deterministic tree automata. This is formalized by the following proposition.

Proposition 1. *Look-around tree automata accept exactly regular tree languages. In particular, they can be efficiently converted into non-deterministic tree automata.*

Proof. Given $\mathcal{M} = (Q,\mathbb{L},\mathbf{Mov},\delta,F)$. Without loss of generality, we assume that $\mathbf{Mov}$ is prefix-closed, i.e., $\boldsymbol{m}\cdot m \in \mathbf{Mov} \Rightarrow \boldsymbol{m} \in \mathbf{Mov}$. We create a non-deterministic tree automaton $\mathcal{M}' = ((\mathbf{Mov} \to (\mathbb{L}\times Q) \uplus \bot) \uplus \{\bullet\} (= Q'), \mathbb{L}, \Delta, F', \bullet)$ which accepts the same language as $\mathcal{M}$. We define Δ so that $(\ell,h_0,h_1,h_2) \in \Delta$ iff (i) $h_0(\epsilon) = (\ell,\delta(h_0))$, (ii) $h_0(k) = \bot$ iff $h_k = \bullet$, (iii) $h_k(0) \neq \bot$ if $h_k \neq \bullet$, and (iv)

$$h_0(\boldsymbol{m}\cdot k) = h_k(\boldsymbol{m}) \quad (\boldsymbol{m}\cdot k \in \mathbf{Mov})$$
$$h_k(\boldsymbol{m}\cdot 0) = h_0(\boldsymbol{m}) \quad (\boldsymbol{m}\cdot 0 \in \mathbf{Mov})$$

where $k = 1,2$ and $\bullet(\boldsymbol{m}) = \bot$ for all conditions (i-iv). We define $F' = \{h \mid \mathit{st}(h(\epsilon)) \in F\}$ where $\mathit{st}(\ell,q) = q$ (always $h(\epsilon) \neq \bot$). Note here that if $\mu' \in U \to Q'$ is a run of $\mathcal{M}'$ then $\mu \in U \to Q$ defined by $\mu(u) = \mathit{snd}(\mu'(u)(\epsilon))$ is a run of $\mathcal{M}$. If $\mu \in U \to Q$ is a run of $\mathcal{M}$ then define μ' by $\mu'(u)(\epsilon) = (*u, \mu(u))$ and $\mu'(u)(\boldsymbol{m}) = \mu'(u\cdot\boldsymbol{m})(\epsilon)$. □

High-level Tree Transducer We introduce the high-level tree transducer as a model of XML transformation. Type checking for XML transformations in high-level tree transducers is decidable.

Tree transducers are tree automata with outputs. Recall that tree automata assign states to nodes. Another way to look at this is that tree automata associates state-node pairs with boolean values. That is, a state-node pair (q,u) is associated with the truth value exactly when q is assigned to u. On the other hand, tree transducers associate each such pair with an output value. For example, the identity function *id* given earlier, can be seen as a very simple tree transducer. This tree transducer has one state, say $\underline{id}$, and for each node u in the input tree, $\underline{id}$ is associated with an output tree identical to the subtree of u.

High-level tree transducers provide an extension of tree transducers. The distinction is that each evaluation step of the transducer creates a functional value rather than a tree value. In this sense, high-level tree transducers are closer to functional programs.

A rule of high-level tree transducer is of the form $\underline{f} : \boldsymbol{y} \triangleright e$ where $\underline{f}$ is a state, $\boldsymbol{y}$ is a sequence of parameter variables, and e is called a *term*. Here is an example of the rule, which corresponds to a function given in Section 2.

$$
\begin{array}{l}
\underline{\mathit{autom}} : {}^{(i\to\mathbb{L}\to(\mathbb{L}\to\mathbb{L}\to\mathbb{L}\to\mathbb{L})\to\mathbb{L})}\ \mathit{ini}\ \mathit{trans} \triangleright \\
\quad \mathit{trans} \\
\quad\quad (*)\langle\epsilon\rangle \\
\quad\quad (\texttt{if}\ (\models 1)\langle\epsilon\rangle\ \texttt{then}\ \underline{\mathit{autom}}\langle 1\rangle\ \mathit{ini}\ \mathit{trans}\ \texttt{else}\ \mathit{ini}) \\
\quad\quad (\texttt{if}\ (\models 2)\langle\epsilon\rangle\ \texttt{then}\ \underline{\mathit{autom}}\langle 2\rangle\ \mathit{ini}\ \mathit{trans}\ \texttt{else}\ \mathit{ini})
\end{array}
$$

In this example, $\underline{\mathit{autom}}$ is a state, *ini* and *trans* are parameter variables, and *trans*(... *ini*) is a term. As we can see from above, a term is almost an expression of the functional language. Terms also should be well-sorted, cf., the definition of sorts $\mathcal{S}(\mathcal{B})$ in Section 2. The only difference is that sorts for terms do not have any occurrences of i. Parameter variables $\boldsymbol{y}$ are also the same as those in `let` and `letrec`. They just abbreviate λ-abstractions, i.e., $f : \boldsymbol{y} \triangleright e$ is equivalent to $f : \triangleright \lambda\boldsymbol{y} : e$ or $f : \triangleright\, \texttt{let}\ g\ \boldsymbol{y} := e\ \texttt{in}\ g$.

The meaning of $\underline{\mathit{autom}}\langle 1\rangle$, $(*)\langle\epsilon\rangle$, $(\models 2)\langle\epsilon\rangle$, etc. in terms are supplied by looking into neighbor nodes. For example, the meaning of $\underline{\mathit{autom}}\langle\boldsymbol{m}\rangle$ is supplied by evaluating the state $\underline{\mathit{autom}}$ at relative position $\boldsymbol{m}$. Similarly, the meaning of $(*)\langle\boldsymbol{m}\rangle$ is the label of the node at relative position $\boldsymbol{m}$. And, the meaning of $(\models 2)\langle\boldsymbol{m}\rangle$ is whether or not $\models 2$ holds at relative position $\boldsymbol{m}$. Recall that the meaning of the tree transducer is given at each node $u \in U$. Therefore, when this node u is supplied, such relative positions 1 and ϵ are interpreted by $u\cdot 1$ and $u\cdot\epsilon = u$, respectively.

We call an arbitrary set X whose each element is associated with a sort, as *sorted set*,

- Figure 4 defines the sorted set $\mathbf{Term}(C,X)$ of terms over sorted sets C and X of constants and variables, respectively. We require that each term to be well-sorted in the usual sense for simple types.
- Figure 4 also defines a sorted set of basic constants $\mathbf{Con}(\mathbb{L})$ over a set of labels $\mathbb{L}$.

Let N be a set of states, C be a set of constants which may include $\mathbf{Con}(\mathbb{L})$, and $\mathbf{Mov} \subseteq \{0,1,2\}^*$ be a set of moves. We call $(\models m)$ and $(*)$ *predicates*, whose set is denoted by P. We define $(N \uplus P)\langle\mathbf{Mov}\rangle$ to be a set of pairs in the form $n\langle\boldsymbol{m}\rangle$ such that $n \in N \uplus P$ and $\boldsymbol{m} \in \mathbf{Mov}$. Each term e appearing in the rule $\underline{f} : \boldsymbol{y} \triangleright e$, is an element of $\mathbf{Term}(C, \boldsymbol{y} \uplus (N \uplus P)\langle\mathbf{Mov}\rangle)$.

Let us define the high-level tree transducer. Note that our high-level tree transducers are not exactly equivalent to transducers by Engelfriet [EV88]. An essential difference is that our tree transducer is a tree-walking transducer with upward moves inside the input tree using $(\_\cdot 0)$-operator. Also our transducer allows the recursive inspection of the input tree. For example, the function *autom* in Section 2

cannot be captured by the Engelfriet's definition of the deterministic high-level tree transducer which is a top-down tree transducer. This is comparable to the transducer with regular look-ahead [Eng77], which in our case, is regular look-around.

In the following, for each sorted set X, we denote by $X(s)$, a subpart of X whose elements are associated with sort s.

Definition 4. *A (look-around deterministic) high-level tree-transducer $\mathcal{H}$ over a finite set of labels $\mathbb{L}$ is a tuple $\mathcal{H} = (\mathcal{B}, N, C, P, \mathbf{Var}, \mathbf{Mov}, \underline{f}_{\mathbf{I}}, R)$ where*

- *$\mathcal{B}$ is a set of base sorts. We have $o, \mathbb{L}, \mathbb{B} \in \mathcal{B}$, but not $i \in \mathcal{B}$. In the following, all elements of sorted sets have sorts in $\mathcal{S}(\mathcal{B})$.*
- *N is a sorted set of states.*
- *C is a sorted set of constants. We have $\mathbf{Con}(\mathbb{L}) \subseteq C$.*
- *$P \subseteq \{(\models m) \mid m \in \{0,1,2\}\} \uplus \{(*)\}$ is a set of predicates. Predicates $(\models m)$ and $(*)$ are associated with sort $\mathbb{B}$ and $\mathbb{L}$ respectively.*
- *$\mathbf{Var}$ is a sorted set of variables.*
- *$\mathbf{Mov} \subseteq \{0,1,2\}^*$ is a finite set of moves.*
- *$\underline{f}_{\mathbf{I}} \in N$ is an initial state.*
- *R is a finite set of rules in the form $\underline{f} : \boldsymbol{y} \triangleright e$.*
  – For each $\underline{f} \in N$, we have exactly one rule in R.
  – If $\underline{f} \in N(s_1 \to \cdots \to s_n)$ and $\boldsymbol{y} = y_1, ..., y_{n-1}$, we have (i) $y_j \in \mathbf{Var}(s_j)$ for $j \in 1..n-1$, (ii) $e \in \mathbf{Term}(C, \boldsymbol{y} \uplus (N \uplus P)\langle \mathbf{Mov} \rangle)(s_n)$.

We do not fix the set of base sorts $\mathcal{B}$, so that we can add a new sort. However we always require that such sorts are associated with finite domains. See Figure 6.

Functional programs introduced and satisfying the restriction in Section 2, can be translated into high-level tree transducers. Recall that those programs are already in the similar shape to the transducer, i.e., functions of sort $i \to s$ only occur at top-level `letrec`. Therefore, the translation is straightforward. We here just give ideas. See [Toz05] for the detailed steps.

Essentially, what we need is to remove the occurrence of expressions of sort i, $i \to i$, and $i \to s$. Functional variables of sorts $i \to s$ defined in the top-level `letrec` correspond to the finite set of states N. Their definitions are easily translated into rules of the tree transducer. However, variables of sort $i \to s$ may also occur as parameter variables, e.g., an argument of $\mathit{children}^{(i \to \underline{(i \to o)} \to o)}$. In this case, we interpret such variables as variables of a new base sort $\mathbb{N}$. We then prepare a finite set of constants of sort $\mathbb{N}$, which has one-to-one correspondence to the state set N. We also prepare the equality operator $(\_ = \_)^{(\mathbb{N} \to \mathbb{N} \to \mathbb{B})}$ over $\mathbb{N}$.

As a result, we translate programs into transducers with base sorts $\mathcal{B} = \{o, \mathbb{L}, \mathbb{B}, \mathbb{N}\}$ and constants $C = \mathbf{Con}(\mathbb{L}) \uplus N \uplus \{(\_ = \_)^{(\mathbb{N} \to \mathbb{N} \to \mathbb{B})}\}$. A program given in Figure 2 is translated into the high-level tree transducer $(\mathcal{B}, N, C, P, \mathbf{Var}, \mathbf{Mov}, f_{\mathbf{I}}, R)$ in Figure 5.

$$
\begin{array}{lcl}
\mathcal{B} & = & \{o, \mathbb{L}, \mathbb{B}, \mathbb{N}\} \\
N & = & \left\{ \begin{array}{l} \underline{\mathit{children}}^{(\mathbb{N} \to o)}, \underline{\mathit{siblings}}^{(\mathbb{N} \to o)}, \\ \underline{\mathit{root}}^{(\mathbb{N} \to o)}, \underline{\mathit{dept}}^{(o)}, \underline{\mathit{emp}}^{(o)} \end{array} \right\} \uplus P, \\
P & = & \{(\models 0), (\models 1), (\models 2), (*)\}, \\
C & = & \mathbf{Con}(\mathbb{L}) \uplus N \uplus \{(\_ = \_)^{(\mathbb{N} \to \mathbb{N} \to \mathbb{B})}\} \\
\mathbf{Var} & = & \{f^{(\mathbb{N})}\}, \\
\mathbf{Mov} & = & \{\epsilon, 0, 1, 2\}, \\
f_{\mathbf{I}} & = & \underline{\mathit{dept}} \\
R & = & \left\{ \begin{array}{l}
\underline{\mathit{children}} :^{(\mathbb{N} \to o)} \ f \triangleright \\
\quad \texttt{if} \ (\models 1)\langle\epsilon\rangle \ \texttt{then} \ \underline{\mathit{siblings}}\langle 1\rangle \ f \ \texttt{else} \ () \\
\underline{\mathit{siblings}} :^{(\mathbb{N} \to o)} \ f \triangleright \\
\quad (\texttt{if} \ f = \underline{\mathit{dept}} \ \texttt{then} \ \underline{\mathit{dept}}\langle\epsilon\rangle \ \texttt{else} \ \underline{\mathit{emp}}\langle\epsilon\rangle), \\
\quad \texttt{if} \ (\models 2)\langle\epsilon\rangle \ \texttt{then} \ \underline{\mathit{siblings}}\langle 2\rangle \ f \ \texttt{else} \ () \\
\underline{\mathit{root}} :^{(\mathbb{N} \to o)} \ f \triangleright \\
\quad \texttt{if} \ (\models 0)\langle\epsilon\rangle \ \texttt{then} \ \underline{\mathit{root}}\langle 0\rangle \ f \ \texttt{else} \\
\quad \texttt{if} \ f = \underline{\mathit{dept}} \ \texttt{then} \ \underline{\mathit{dept}}\langle\epsilon\rangle \ \texttt{else} \ \underline{\mathit{emp}}\langle\epsilon\rangle \\
\underline{\mathit{dept}} :^{(o)} \triangleright \\
\quad \texttt{if} \ *\langle\epsilon\rangle = \texttt{emp} \ \texttt{then} \ () \\
\quad \texttt{else if} \ *\langle\epsilon\rangle = \texttt{dept} \ \texttt{then} \ \texttt{dept}[\underline{\mathit{root}}\langle\epsilon\rangle \ \underline{\mathit{emp}}] \\
\quad \texttt{else} \ *\langle\epsilon\rangle[\underline{\mathit{children}}\langle\epsilon\rangle \ \mathit{dept}] \\
\underline{\mathit{emp}} :^{(o)} \triangleright \\
\quad \texttt{if} \ *\langle\epsilon\rangle = \texttt{emp} \ \texttt{then} \ *\langle\epsilon\rangle[\underline{\mathit{children}}\langle\epsilon\rangle \ \mathit{dept}] \\
\quad \texttt{else} \ \underline{\mathit{children}}\langle\epsilon\rangle \ \underline{\mathit{emp}}
\end{array} \right\}
\end{array}
$$

Figure 5. A high-level tree transducer corresponding to the function *dept* in Figure 2

Semantics of High-Level Tree Transducers In the original definition by Engelfriet, the semantics of high-level tree transducers is given by means of the rewrite system, which corresponds to the operational semantics. In this paper, we give a denotational semantics. This gives a clear meaning to functional values emitted by the high-level tree transducer.

In the denotational semantics, a transducer $\mathcal{H}$ has a meaning on each node u in U, which is an assignment $\rho \in (N \uplus P)(\cdot) \to \mathcal{D}[\![\cdot]\!]$ such that each element in subset $N(s)$ of states, as well as $P(s)$ of predicates, is associated with an element in $\mathcal{D}[\![s]\!]$. Here $\mathcal{D}[\![s]\!]$ is the cpo-based semantic domain given in Figure 6. In other words, $\mathcal{D}[\![s]\!]$ is the set of functional values of sort s. See the end of this paragraph. The above meaning to each node is given by the following function $\mathit{sem}_{\mathcal{H}} : U \to (N \uplus P)(\cdot) \to \mathcal{D}[\![\cdot]\!]$.

Definition 5. *Given $\mathcal{H} = (\mathcal{B}, N, C, P, \mathbf{Var}, \mathbf{Mov}, \underline{f}_{\mathbf{I}}, R)$. The meaning function $\mathit{sem}_{\mathcal{H}} : U \to (N \uplus P)(\cdot) \to \mathcal{D}[\![\cdot]\!]$ is defined as the least solution satisfying the following equations. For any $\underline{f} \in N$ such that $(\underline{f} : \boldsymbol{y} \triangleright e) \in R$, and $(*), (\models m) \in P$,*

$$
\begin{array}{l}
\mathit{sem}_{\mathcal{H}}(u)(*) = *u \\
\mathit{sem}_{\mathcal{H}}(u)(\models m) = (u \cdot m) \in U \\
\mathit{sem}_{\mathcal{H}}(u)(\underline{f}) = \mathcal{D}[\![\lambda \boldsymbol{y} : e]\!][n\langle \boldsymbol{m} \rangle \mapsto \mathit{sem}_{\mathcal{H}}(u \cdot \boldsymbol{m})(n)]_{n \in N \uplus P, \boldsymbol{m} \in \mathbf{Mov}}
\end{array}
$$

where $\mathcal{D}[\![e]\!]\rho$ is a semantics of term e under ρ given in Figure 7, in which $\lambda \boldsymbol{y} : e$ abbreviates `letrec` $g \ \boldsymbol{y} := e$ `in` g. In particular, for the root node $u \in \Lambda(U)$, $\mathit{sem}_{\mathcal{H}}(u)(f_{\mathbf{I}})$ defines the output of the transducer.

The definition of $\mathcal{D}[\![e]\!]\rho$ is the standard cpo semantics of simply-typed call-by-value languages [Mit96].

Let us briefly recall this semantics. A cpo $(X, \sqsubseteq)$ is a poset whose any directed subset has the lub. Starting from flat cpos $\mathcal{D}[\![b]\!]$ for

$$
\begin{array}{lcl}
\mathcal{D}[\![o]\!] & = & V^{\perp} \\
\mathcal{D}[\![b]\!] & = & b^{\perp} \quad \text{where } b \neq o \\
\mathcal{D}[\![s' \to s]\!] & = & (\mathcal{D}[\![s']\!] \to_{\perp} \mathcal{D}[\![s]\!])^{\perp}
\end{array}
$$

Figure 6. Semantic domains

$$
\mathcal{D}[\![\_]\!]\_ \quad \in \quad \mathbf{Term}(C,X)(\cdot) \to (X(\cdot) \to \mathcal{D}[\![\cdot]\!]) \to \mathcal{D}[\![\cdot]\!]
$$

$$
\begin{array}{lcl}
\mathcal{D}[\![x]\!]\rho & = & \rho(x) \\
\mathcal{D}[\![c]\!]\rho & = & c \\
\mathcal{D}[\![ee']\!]\rho & = & \mathcal{D}[\![e]\!]\rho(\mathcal{D}[\![e']\!]\rho) \\
\mathcal{D}[\![\texttt{if } e \texttt{ then } e' \texttt{ else } e'']\!]\rho & = & \begin{cases} \mathcal{D}[\![e']\!]\rho & \text{if } \mathcal{D}[\![e]\!]\rho = \mathit{true} \\ \mathcal{D}[\![e'']\!]\rho & \text{if } \mathcal{D}[\![e]\!]\rho = \mathit{false} \\ \perp & \text{otherwise} \end{cases} \\
\mathcal{D}[\![\texttt{letrec } \theta \texttt{ in } e]\!]\rho & = & \mathcal{D}[\![e]\!]\mathit{lfp}(\zeta_{\theta,\rho})
\end{array}
$$

where

$$
\zeta_{\theta,\rho'} \ (\in (X(\cdot) \to \mathcal{D}[\![\cdot]\!]) \to (X(\cdot) \to \mathcal{D}[\![\cdot]\!])) = \lambda\rho : \\
\rho'[f \mapsto \lambda \boldsymbol{v} \in \mathcal{D}[\![s_1,...,s_{n-1}]\!] : \mathcal{D}[\![e]\!]\rho[\boldsymbol{x} \mapsto \boldsymbol{v}]]_{(f^{(s_1 \to \cdots \to s_n)}\boldsymbol{x} := e^{(s_n)}) \in \theta}
$$

Figure 7. Semantics of terms in $\mathbf{Term}(C,X)$

$$
\begin{array}{lcl}
\mathcal{A}[\![o]\!] & = & \mathcal{V}(\tau_{\mathbf{I}})^{\perp} \\
\mathcal{A}[\![b]\!] & = & \mathcal{D}[\![b]\!] \quad \text{where } b \neq o \\
\mathcal{A}[\![s' \to s]\!] & = & (\mathcal{A}[\![s']\!] \to_{\perp} \mathcal{A}[\![s]\!])^{\perp}
\end{array}
$$

Figure 8. Abstract semantic domain

base sorts, we can obtain cpos for function spaces $\mathcal{D}[\![s' \to s]\!]$. Partial orders for functions are defined as $f \sqsubseteq g$ iff $\forall x : f(x) \sqsubseteq g(x)$. In the case of call-by-value, we use a strict function space $A^{\perp} \to_{\perp} B^{\perp}$ ($\simeq A \to B^{\perp}$) such that $f \in A^{\perp} \to_{\perp} B^{\perp}$ satisfies $f(\perp) = \perp$, i.e., the application of a function to an error value results in an error value, i.e., non-termination. In the cpo-based semantics, the meaning of recursive functions is the least fixpoint of some equations. The above $sem_{\mathcal{H}}$ is indeed such a least fixpoint.

4 Type Checking

So far, we have introduced three tools, namely

- Binoids with homomorphism $^{\circ}$,
- Look-around tree automata, and
- Tree-transducer and its semantics.

Here we connect these tools and derive our type inference algorithm. In particular, the key idea is the extension of the homomorphism $^{\circ}$ for binoids to functional spaces $\mathcal{D}[\![s' \to s]\!]$.

As we discussed, a common technique to the tree transducer type checking is based on the inverse type inference. In the case of tree transducers or macro tree transducers (mtts), the inverse image $f_{\mathbf{I}}^{-1}(\tau_{\mathbf{I}})$ is regular. The expressiveness of high-level tree transducers is the same as k-composition of mtts [EV88], where k is the height of sorts. Therefore the inverse image $f_{\mathbf{I}}^{-1}(\tau_{\mathbf{I}})$ should be a regular language also for high-level tree transducers. However, as far as we know, there is no direct construction algorithm of the inverse image of high-level tree transducers. We give one such construction here.

Maneth [Man04] gave a simple algorithm for inferring regular inverse images for deterministic mtts. His idea was to *run* the automaton, representing $\tau_{\mathbf{I}}$, on the term e of the rule $\underline{f} : \boldsymbol{y} \triangleright e$. In his case, this term defines a tree value, while in our case, it defines a functional value. To interpret e in our case, we extend the homomorphism $^{\circ}$ between the set of XML values V and the binoid $\mathcal{V}(\tau_{\mathbf{I}})$.

Extending Homomorphism $^{\circ}$ to Functional Space Given a type $\tau_{\mathbf{I}}$, we can obtain a finite binoid $\mathcal{V}(\tau_{\mathbf{I}})$ with the homomorphism $^{\circ}$ from V to $\mathcal{V}(\tau_{\mathbf{I}})$. This homomorphism is seen as an abstraction function from infinite values to finite elements.

Here let us extend this definition of $^{\circ}$ to domains $\mathcal{D}[\![s]\!]$ where s is other than o. We define the domain of images of $^{\circ}$ as in Figure 8, so that for $v \in \mathcal{D}[\![s]\!]$ we have $v^{\circ} \in \mathcal{A}[\![s]\!]$. Then, the idea is to define $^{\circ} \in \mathcal{D}[\![\cdot]\!] \to \mathcal{A}[\![\cdot]\!]$ so that it further satisfies

$$v^{\circ}(v'^{\circ}) = (v(v'))^{\circ}$$

for $v \in \mathcal{D}[\![s' \to s]\!]$ and $v' \in \mathcal{D}[\![s']\!]$.

Example 6. *We assume the binoid $\mathcal{V}(\texttt{comp}[ds])$ in Example 4. Let us consider gapped values of sort $o \to o$. For example, $p = \texttt{comp}[\texttt{dept}[\square]]$ ($\in \mathcal{D}[\![o \to o]\!]$). What we need here is to define $p^{\circ} \in (\mathcal{V}(\texttt{comp}[ds])^{\perp} \to_{\perp} \mathcal{V}(\texttt{comp}[ds])^{\perp})^{\perp}$ ($= \mathcal{A}[\![o \to o]\!]$) as a function.*

$$
p^{\circ}(a) = \begin{cases} \texttt{comp}[ds] & \text{if } a = \texttt{emp}[]^{+} \text{ or } a = () \\ \tau_{\mathbf{E}} & \text{otherwise, } a \neq \perp \\ \perp & a = \perp \end{cases}
$$

This function indeed satisfies $p^{\circ}(v'^{\circ}) = (p(v'))^{\circ}$ for $v' \in V$. For example, assume $v' = \texttt{dept}[]$. We have $p^{\circ}(v'^{\circ}) = p^{\circ}(\texttt{dept}[es]^{+}) = \tau_{\mathbf{E}}$, and $(p(v'))^{\circ} = (\texttt{comp}[\texttt{dept}[\texttt{dept}[]]])^{\circ} = \tau_{\mathbf{E}}$.

Note that the homomorphic images $\mathcal{A}[\![s]\!]$ of $\mathcal{D}[\![s]\!]$ are finite sets. The function $^{\circ}$ gives a way to interpret each values as abstract values. Such abstract values are suitable for analysis, because they are finitely enumerable.

Definition 6. *The abstraction function $^{\circ} \in \mathcal{D}[\![\cdot]\!] \to \mathcal{A}[\![\cdot]\!]$ is a partial function defined as follows. We use the induction of the size of s in extending $^{\circ}$ to $\mathcal{D}[\![s]\!] \to \mathcal{A}[\![s]\!]$.*

- *$v^{\circ} = v$ ($\in \mathcal{A}[\![b]\!]$), if $v \in \mathcal{D}[\![b]\!]$ for $b \in \mathcal{B} \setminus \{o\}$.*
- *A value $v \in \mathcal{D}[\![s' \to s]\!]$ is in the domain of $^{\circ}$ written $v \in \mathit{Dom}(^{\circ})$, if for any v' and v'' ($\in \mathit{Dom}(^{\circ}) \cap \mathcal{D}[\![s']\!]$) such that $v'^{\circ} = v''^{\circ}$, this v satisfies $(v(v'))^{\circ} = (v(v''))^{\circ}$.*
- *For $v \in \mathit{Dom}(^{\circ})$, we define v° ($\in \mathcal{A}[\![s' \to s]\!]$) to be the function defined as $v^{\circ}(v'^{\circ}) = (v(v'))^{\circ}$.*

The above definition, however, seems incomplete, since it just says that we ignore values $v \notin \mathit{Dom}(^{\circ})$. That is, values do not satisfy the desired property. What is more interesting is the following result.

Lemma 1. *[Toz05] For any $^{\circ}$, all outputs v ($\in \mathcal{D}[\![s]\!]$) of transducers are in $\mathit{Dom}(^{\circ})$.*

From this lemma, we can show that any output of transducers of sort $o \to \mathbb{B}$ is a constant function. We take a singleton set $\mathcal{V} = \{\bullet\}$ as the homomorphic image of V. Then for any t, t', we have $t^{\circ} = t'^{\circ} = \bullet$, so that $f(t) = (f(t))^{\circ} = (f(t'))^{\circ} = f(t')$. We belive that this is the meaning of *non-observability* of output values of sort o.

Negative Inverse Type Inference The remaining steps of the type inference algorithm are as follows.

- Interpreting the meaning function $sem_{\mathcal{H}}$ by $^\circ$, and obtain $sem^\circ_{\mathcal{H}}$.
- Defining the look-around tree automaton $\mathcal{M}$ capturing $sem^\circ_{\mathcal{H}}$. This $\mathcal{M}$ gives the result of type inference.

Accurately speaking, what the above $\mathcal{M}$ represents, is a *negative* inverse image $f_{\mathbf{I}}^{-1}(V \setminus \tau_{\mathbf{I}})$. This is fortunate. After we inferred such an image, what we need is the emptiness check, known to be efficient.

$$[\![\upsilon_{\mathbf{I}}]\!] \cap [\![f_{\mathbf{I}}^{-1}(V \setminus \tau_{\mathbf{I}})]\!] = \emptyset$$

If this holds, the type checking succeeds. If $\mathcal{M}$ was $f_{\mathbf{I}}^{-1}(\tau_{\mathbf{I}})$, the above emptiness check turns to the containment test, which is not always efficient. We later explain why our construction creates an automaton for such a negative image.

First, we *interpret* the semantic function $sem_{\mathcal{H}}$ by means of $^\circ$ just introduced. Indeed, this can be done. This gives a function $sem^\circ_{\mathcal{H}}$ $(\in U \to (N \uplus P)(\cdot) \to \mathcal{A}[\![\cdot]\!])$ which satisfies, for all $u \in U$ and $n \in N \uplus P$

$$(sem_{\mathcal{H}}(u)(n))^\circ = sem^\circ_{\mathcal{H}}(u)(n)$$

cf., Lemma 2(a). The definition of $sem^\circ_{\mathcal{H}}$ in Figure 9 is exactly the copy of Definition 5 while it uses operators on $\mathcal{V}(\tau_{\mathbf{I}})$.

What remains is to define the look-around automaton that captures this $sem^\circ_{\mathcal{H}}$. The resulting automaton is given in Figure 10. This automaton has its state set $Q = (N \uplus P)(\cdot) \to \mathcal{A}[\![\cdot]\!]$. This set Q classifies the nodes of the input XML tree according to the (abstract) output value of the transducer at its each state. A run of $\mathcal{M}$ gives one such classification of nodes in the input XML tree. Now, recall that the same information was given by $sem^\circ_{\mathcal{H}}$, which defines the abstract semantics of the transducer for each state-node pair. Indeed, this automaton captures $sem^\circ_{\mathcal{H}}$ in the sense that $sem^\circ_{\mathcal{H}}$ is always a run of $\mathcal{M}$. Confirm that the run of automaton $\mu \in U \to Q$ and $sem^\circ_{\mathcal{H}}$ $(\in U \to (N \uplus P)(\cdot) \to \mathcal{A}[\![\cdot]\!])$ has the same signature. Also notice the similarity between the transition function δ of $\mathcal{M}$ and the definition of $sem^\circ_{\mathcal{H}}$. This δ is defined so that it simulates $sem^\circ_{\mathcal{H}}$.

As readers may expect, this $\mathcal{M}$ exactly defines the negative inverse image we want.

Lemma 2. *[Toz05] (a) For all $u \in U$ and $n \in N \uplus P$, we have $(sem_{\mathcal{H}}(u)(n))^\circ = sem^\circ_{\mathcal{H}}(u)(n)$. (b) $sem^\circ_{\mathcal{H}}$ is the least run of $\mathcal{M}$. (c) The automaton $\mathcal{M}$ in Definition 8 accepts u iff $sem_{\mathcal{H}}(u)(f_{\mathbf{I}}) \notin [\![\tau_{\mathbf{I}}]\!]$.*

The detailed proof is omitted here. Here we just note why we need to infer the *negative* inverse image. This is related to our treatment of *non-termination as error*, cf., Section 2.

In Definition 8, we define the final states F of $\mathcal{M}$ negatively, i.e., acceptance means type error. Note that if the program is correct, i.e., $sem_{\mathcal{H}}(u)(f_{\mathbf{I}}) \in [\![\tau_{\mathbf{I}}]\!]$, then $sem^\circ_{\mathcal{H}}$ should also give the correct result in $[\![\tau_{\mathbf{I}}]\!]^\circ$. From the above lemma (b), if $sem^\circ_{\mathcal{H}}$ gives the correct result, i.e., is a *non-accepting* run of $\mathcal{M}$, then "any run" is also non accepting. This shows the *only-if* direction of Lemma (c) (the other direction is easy).

Now, assume that we include $\perp$ (non-termination) to the correct result. In this case, we cannot say more than "some run is correct" from the fact that $sem^\circ_{\mathcal{H}}$ gives the correct result. In fact, in this case, we must have defined the set of final states of $\mathcal{M}$ *positively*.

Definition 7. *The abstract meaning function $sem^\circ_{\mathcal{H}} : U \to (N \uplus P)(\cdot) \to \mathcal{A}[\![\cdot]\!]$ is the least solution of the following equations. For any $\underline{f} \in N$ such that $(\underline{f} : \boldsymbol{y} \triangleright e) \in R$, and $(*), (\models m) \in P$,*

$$sem^\circ_{\mathcal{H}}(u)(*) = *u$$
$$sem^\circ_{\mathcal{H}}(u)(\models m) = (u \cdot m) \in U$$
$$sem^\circ_{\mathcal{H}}(u)(\underline{f}) = \mathcal{A}[\![\lambda \boldsymbol{y} : e]\!][n\langle \boldsymbol{m}\rangle \mapsto sem^\circ_{\mathcal{H}}(u \cdot \boldsymbol{m})(n)]_{n \in N \uplus P, \boldsymbol{m} \in \mathbf{Mov}}$$

where $\mathcal{A}[\![e]\!]\rho$ is an abstract semantics of term e under ρ given similarly to Figure 7, except that it uses operators on abstract semantic domains.

Figure 9. The abstract meaning function $sem^\circ_{\mathcal{H}}$

Definition 8. *Given a transducer $\mathcal{H}$, and a binoid $\mathcal{V}(\tau_{\mathbf{I}})$, we define a look-around automaton $\mathcal{M} = (Q, \mathbb{L}, \mathbf{Mov}, \delta, F)$ as follows.*

- $\mathbf{Mov}, \mathbb{L}$ *are the same as* $\mathcal{H}$,
- $Q = (N \uplus P)(\cdot) \to \mathcal{A}[\![\cdot]\!]$,
- $\delta \in (\mathbf{Mov} \to (\mathbb{L} \times Q)^\perp) \to Q$ *is defined as*
  $\delta(h)(\models m) = (h(m) \neq \perp)$
  $\delta(h)(*) = lab(h(\epsilon))$
  $\delta(h)(\underline{f}) = \mathcal{A}[\![\lambda \boldsymbol{y} : e]\!][n\langle \boldsymbol{m}\rangle \mapsto st(h(\boldsymbol{m}))(n)]_{n \in N \uplus P, \boldsymbol{m} \in \mathbf{Mov}}$ *for all* $\underline{f} : \boldsymbol{y} \triangleright e \in R$
  where $lab(\ell, q) = \ell$, $st(\ell, q) = q$ *and* $lab(\perp) = st(\perp) = \perp$.
- $F = \{\rho \in Q \mid \rho(f_{\mathbf{I}}) \notin [\![\tau_{\mathbf{I}}]\!]^\circ\}$.

Figure 10. The definition of the inferred automaton $\mathcal{M}$

Running Example We apply the algorithm explained so far to a small example. We use the following type checking problem $id : \upsilon_{\mathbf{I}} \to \tau_{\mathbf{I}}$. Let $\mathbb{L} = \{\mathtt{a}, \mathtt{b}\}$.

- The function id is translated into the following transducer $\mathcal{H} = (\mathcal{B}, N, C, P, \mathbf{Var}, \mathbf{Mov}, f_{\mathbf{I}}, R)$ with one rule

  $$\underline{id}^{(\mathbb{N} \to o)} : \triangleright$$
  $$((*)\langle\epsilon\rangle[\mathtt{if}\ (\models 1)\langle\epsilon\rangle\ \mathtt{then}\ \underline{id}\langle 1\rangle\ \mathtt{else}\ ()),$$
  $$\mathtt{if}\ (\models 2)\langle\epsilon\rangle\ \mathtt{then}\ \underline{id}\langle 2\rangle\ \mathtt{else}\ ()$$

  We have $N = \{\underline{id}\}$, $P = \{(*), (\models 1), (\models 2)\}$, $\mathbf{Mov} = \{\epsilon, 1, 2\}$ and $\underline{f_{\mathbf{I}}} = \underline{id}$.
- $\upsilon_{\mathbf{I}} = \mathtt{b}[\mathtt{a}[]]$, and
- $\tau_{\mathbf{I}} = \mathtt{letrec}\ \alpha := \mathtt{a}[\alpha]^*\ \mathtt{in}\ \alpha$.

We have $\mathcal{V}(\tau_{\mathbf{I}}) = \{(), \mathtt{a}[\alpha]^+, \tau_{\mathbf{E}}\}$.

For this problem, we compute the look-around automaton in Definition 8 whose transition function δ is shown below. The state set Q of $\mathcal{M}$ is a set of mappings $(N \uplus P)(\cdot) \to \mathcal{A}[\![\cdot]\!]$, so that the transition function $\delta \in (\{\epsilon, 1, 2\} \to (\mathbb{L} \times Q)^\perp) \to Q$, given $h \in \{\epsilon, 1, 2\} \to (\mathbb{L} \times Q)^\perp$ again returns functions.

$$\delta(h)(\models 1) = (h(1) \neq \perp)$$
$$\delta(h)(\models 2) = (h(2) \neq \perp)$$
$$\delta(h)(*) = lab(h(\epsilon))$$
$$\delta(h)(\underline{id}) = \begin{cases} \mathtt{a}[\alpha]^+ & st(h(\epsilon))(*) = \mathtt{a} \text{ and } st(h(k))(\underline{id}) \in \{\perp, (), \mathtt{a}[\alpha]^+\}\ (k = 1, 2) \\ \tau_{\mathbf{E}} & \text{otherwise} \end{cases}$$

We can convert this automaton $\mathcal{M}$ to a non-deterministic tree automaton using the construction in Proposition 1. We here instead just test the input type $\upsilon_{\mathbf{I}}$ using $\mathcal{M}$.

In this case, since $\upsilon_{\mathbf{I}}$ just defines a single tree with two nodes, the type checking problem amounts to check whether or not $\mathcal{M}$ accepts the tree $\mathsf{b}[\mathsf{a}[]]$ with two nodes, $u_0 = (\mathsf{b}[\mathsf{a}[]], \epsilon)$ and $u_1 = (\mathsf{b}[\mathsf{a}[]], 1)$. In this example, we only have one run μ shown below.

$$\begin{array}{ll} \mu(u_0)(\models 1) = \mathit{true} & \mu(u_1)(\models 1) = \mathit{false} \\ \mu(u_0)(\models 2) = \mathit{false} & \mu(u_1)(\models 2) = \mathit{false} \\ \mu(u_0)(*) = \mathsf{b} & \mu(u_1)(*) = \mathsf{a} \\ \mu(u_0)(\underline{id}) = \tau_{\mathbf{E}} & \mu(u_1)(\underline{id}) = \mathsf{a}[\alpha]^+ \end{array}$$

Now we can see that $\mu(u_0)(\underline{id}) \notin [\![\tau_{\mathbf{I}}]\!]^\circ$. So this run is an accepting run of $\mathcal{M}$. Thus the type checking $id : \upsilon_{\mathbf{I}} \to \tau_{\mathbf{I}}$ in this case is not successful.

5 Related Work

Milo et al. [MSV00] first propose a solution, based on *inverse type inference*, to the type checking for XML programming modeled by tree transducers. Milo et al. solve this problem for k-pebble transducers. The k-pebble transducers are in theory $k+1$-fold composition of mtts [Man04], and it is comparable to high-level tree transducers, which is also represented by k-composition of mtts where k is the height of sorts [EV88]. Similar approaches have been studied for different kinds of tree transducers [Toz01][AMN+01][MN02][MBPS05].

XDuce is a pioneering work [HVP00, HP03] on typed functional XML programming, which employs type checking with type-annotation. XDuce is a first order language. Its approach has also been employed in a number of typed XML processing languages, including an industrial language such as XQuery. Frisch et al. [FCB02] extended tree regular expression types in XDuce to higher order functional types. Their language is called CDuce.

XDuce and CDuce require type annotations. In general, they cannot solve the type checking problem such as $id : \mathsf{a}[\mathsf{b}[]] \to \mathsf{a}[\mathsf{b}[]]$ as it is, by the following reasons.

- When using XDuce, we can annotate *id* only by trivial types, e.g., $\mathit{Any} \to \mathit{Any}$. For example, when we type-check *id* against $\mathsf{a}[\mathsf{b}[]] \to \mathsf{a}[\mathsf{b}[]]$, we have to check *id* also against $\mathsf{b}[] \to \mathsf{b}[]$. This is not possible in XDuce which associates a single arrow type with each recursive function.
- CDuce has intersection types. By giving a type annotation $\mathsf{a}[\mathsf{b}[]] \to \mathsf{a}[\mathsf{b}[]] \cap \mathsf{b}[] \to \mathsf{b}[]$, the function *id* passes the type check. It is even possible to *prove* that $id : \mathsf{a}[\mathsf{b}[]] \to \mathsf{a}[\mathsf{b}[]]$ holds. This is based on their subtyping algorithm.

  $$\mathsf{a}[\mathsf{b}[]] \to \mathsf{a}[\mathsf{b}[]] \cap \mathsf{b}[] \to \mathsf{b}[] <: \mathsf{a}[\mathsf{b}[]] \to \mathsf{a}[\mathsf{b}[]]$$

  However this process is still not automatic. Users need to figure out what type annotation is necessary in beforehand.

6 Future Work

As a concluding remark, we note several future directions of this work.

- Practical use with XML programming

  We implemented a prototype type-checker, and tried several experiments. Our implementation works well for simple programs using small sorts, such as $i \to o$. Unfortunately, for programs with larger sorts, the initial result was not promising. This reflects the time complexity of the algorithm, which is k-exponential to the height of sorts. However, in practical programming, it is not so usual to use functions whose order is more than second. So it is too early to conclude that the approach is infeasible. Our implementation naively implements the enumeration of states of automata $\mathcal{M}$ in Section 4. We are currently seeking a different algorithm for the practical use, i.e., in XML programming languages.
- Connection to the type theory

  Type-checking is the central issue of functional programming. There are many approaches to type-check programs based on *type systems*. However, as far as we know, there are no such type systems which capture the tree transducer type checking as shown here. In particular, the restrictions as we gave in Section 2, do not seem to be natural assumptions in the study of type systems. We are seeking their meaning.

7 Acknowledgment

I thank to anonymous referees for detailed reading and suggestive comments to the earlier draft of this paper. I also thank to Makoto Murata for proof-reading this version of the paper.

8 References

[AMN+01] N. Alon, T. Milo, F. Neven, D. Suciu, and V. Vianu. XML with data values: Typechecking revisited. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 138–149, 2001.

[CMS02] Aske Simon Christensen, Anders Möller, and Michael I. Schwartzbach. Static analysis for dynamic XML. In *Proceedings of 1st Workshop on Programming Languages Technology for XML (PLAN-X 2002)*, 2002.

[Eng77] Joost Engelfriet. Top-down tree transducer with regular look-ahead. *Mathematical Systems Theory*, 9(3):289–303, 1977.

[EV88] Joost Engelfriet and Heiko Vogler. High level tree transducers and iterated pushdown tree transducers. *Acta Informatica*, 26(2):131–192, 1988.

[FCB02] Alain Frisch, Giuseppe Castagna, and Veronique Benzaken. Semantic Subtyping. In *Proceedings, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[HP03] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. *J. Funct. Program.*, 13(6):961–1004, 2003.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 11–22, Sep., 2000.

[Man04] Sebastian Maneth. *Models of Tree Translation*. PhD thesis, Proefschrift Universiteit Leiden, 2004.

[MBPS05] Sebastian Maneth, Alexandru Berlea, Thomas Perst, and Helmut Seidl. Xml type checking with macro tree transducers. In *PODS 2005, to appear*, 2005.

[Mit96] John C. Mitchell. *Foundations of programming languages*. MIT Press, 1996.

[MN02] Wim Martens and Frank Neven. Typechecking top-down uniform unranked tree transducers. In *ICDT 2002*, pages 64–78, 2002.

[MSV00] Tova Milo, Dan Suciu, and Victor Vianu. Type-checking for XML transformers. In *Proceedings of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 11–22, 2000.

[Per90] Dominique Perrin. Finite automata. In *Handbook of Theoretical Computer Science*, volume B, pages 1–57. 1990.

[PQ68] C. Pair and A. Quere. Definition et etude des bilangages reguliers. *Information and Control*, (6):565–593, Dec 1968.

[Toz01] Akihiko Tozawa. Towards static type checking for XSLT. In *Proceedings of the 1st ACM Symposium on Document Engineering*. ACM Press, 2001.

[Toz05] Akihiko Tozawa. Type checking for functional XML programming using high-level tree transducer, 2005. full paper, in prepation, `http://www.trl.ibm.com/people/akihiko/pub/curry-full.pdf`.

Accelerating XPath Evaluation against XML Streams

Dan Olteanu
Database Group, Saarland University, Germany
olteanu@cs.uni-sb.de

Streams are an emerging technology for data dissemination in cases where the data throughput or size make it unfeasible to rely on the conventional approach based on storing the data before processing it. Querying XML streams without storing and without decreasing considerably the data throughput is especially challenging because XML streams can convey tree structured data with unbounded size and depth. We demonstrate a novel compile-time optimization of SPEX [1], an XML stream query processor with polynomial combined complexity. This optimization is achieved by *stream filters* that exploit the structural relationships between XML fragments encountered along the stream at various processing states in order to skip large stream fragments irrelevant to the query answer. The efficiency of optimized SPEX is positively confirmed by experiments.

Querying XML Streams with SPEX. SPEX compiles XPath queries into networks of deterministic transducers, after rewriting them to forward equivalents. A network for a given forward query consists of two connected parts. The upper part has the shape of the query, i.e., it is a sequence if the query is a simple path, a tree if the query has predicates, and a directed acyclic graph, if the query has set operators. Each step in the query induces a corresponding transducer, and each predicate induces a begin-scope transducer in the network. The upper part is extended with a stream-delivering in transducer at its beginning, and with an answer transducer after the transducer corresponding to the last step outside the query predicates. The lower part is an answer-collecting *funnel*, i.e., a subnetwork of auxiliary transducers serving to collect the potential answers. This funnel mirrors in in out transducers, and begin-scope in end-scope transducers while preserving their nesting.

Processing an XML stream corresponds to a depth-first, left-to-right, preorder traversal of its (implicit) tree. Exploiting the affinity between preorder traversal and stack management, the transducers use their stacks for remembering the depth of the nodes in the tree. This way, binary relations expressed as axes, e.g., child and descendant, can be computed in a single pass. The transducer network processes the stream annotated by its first transducer in, and generates progressively the output stream conveying the answers to the original query. The other transducers in the network process stepwise the received annotated stream and send it with changed annotations to their successor transducers. E.g., a transducer child moves the annotation of each node to all children of that node. The answers computed by a transducer network are among the nodes annotated by the answer transducer. These nodes are *potential* answers, as they may depend on a downstream satisfaction of predicates. The information on predicate satisfaction is conveyed by annotations to the stream. Until the predicate satisfaction is decided, the potential answers are buffered by the out transducer.

Structural Filters. We exemplify structural filters on a (DBLP-like) stream containing information about articles possibly followed only at the very end of the stream by information about books. Consider the SPEX network for a query asking for authors of books with given prices and publishers. In case the transducer instructed to find books-nodes, say the books-transducer, encounters such a node, then it sends it further to its successors, with an additional non-empty annotation signaling a match. In case it encounters other nodes, e.g., article-nodes, then it still sends it further, but with an empty annotation, signaling a non-match. Either way, all nodes from the stream reach all transducers from the network, although this is not necessary. We can reduce the stream traffic between transducers in (at least) two ways.

1. Because all transducers following the books-transducer in the network look always for nodes in the stream following the books-nodes, the query evaluation is not altered, if the books-transducer sends further only the nodes starting with the first books-node and ending together with the stream, and the other transducers do the same for the nodes they are instructed to find relative to nodes found by their previous transducers.

2. Assume the transducers receiving (directly or indirectly) nodes from the books-transducer look for nodes to be found only inside the fragments corresponding to books-nodes (like their descendants, or siblings of their descendants). Then, the books-transducer can safely send further only such stream fragments corresponding to books-nodes.

Both aforementioned approaches to stream traffic minimization can be easily supported by SPEX extended with special-purpose push-down transducers, called structural filters. For example, in the second case above, a *vertical* filter, placed immediately after the books-transducer, sends further only stream fragments corresponding to books-nodes. Also, in the first case above, this filter can be a *diagonal* filter and send further only stream fragments starting with an opening tag books. Diagonal filters are not always superseded by vertical filters, as for the above examples. It is enough to consider a slightly modified example, where the query asks for Web links *following* books, thus a following-transducer gets the stream processed by the books-transducer. Furthermore, if the query constrains the Web links to appear at the same depth with books in the tree conveyed by the stream, then the filter, here a *horizontal* one, would send further only the stream fragments corresponding to the following siblings of the books-nodes.

1 References

[1] F. Bry, F. Coskun, S. Durmaz, T. Furche, D. Olteanu, and M. Spannagel. The XML stream query processor SPEX. In *Proc. of Int. Conf. on Data Engineering (ICDE)*, 2005.

An XQuery-based imperative XML programming language with a database optimizer

Anguel Novoselsky
Zhen Liu
Daniela Florescu
Oracle Corporation

XML data programming will become an increasing important problem of the years to come. The currently proposed solutions fall in one of the two major categories: extensions of existing major programming languages with native XML type and native processing capabilities (e.g. Xlinq) or extensions of existing XML processing languages like XQuery (e.g. XL). The language we propose (temporarily called XScript) is another variant of an XML scripting language based on XQuery.

Two major avenues were investigated in the past to extend XQuery to a full programming language. The first approach added the notion of statements to XQuery, and duplicating the iterators (FLWOR expressions) and conditionals both as expression constructors as well as statement constructors. The second approach is a pure compositional approach: the side-effect operators become normal expressions, and are composed with the rest of the language.

We investigate here a third stylistic approach. We added the notion of statements to XQuery and kept the expressions side-effect free. Statements include update operations, variable assignments and error handling. They also include the iteration and conditionals. However, the iteration and conditions are eliminated the expression part of the language. The main goal in the design of this particular approach is simplicity and ease of use for a large number of developers who might not be familiar or comfortable with a declarative XQuery style of programming. The demo will show that despite of the fact that the expression part of the language is less rich then XQuery the language has the same opportunities for intelligent optimizations, provided that one admit to pay the price of a more complex optimizer.

The demo will show the compiler of such a language, and its virtual machine, and will exemplify the optimization opportunities on a couple of application programs. The virtual machine is common to all three languages: XSLT 2.0, XQuery and XScript and uses extensively Oracle's XML infrastructure (e.g. parsers, type system, runtime). The compiler uses extensive data flow analysis to recover from the imperative style of programming the opportunities for rewriting and optimization traditional in declarative languages. Due to this type of optimization Xscript programs can effectively scale to manipulate large volumes of data, similar to the way databases scale to process large amounts of data.

We will show four different execution scenarios for XScript programs:

(a) standalone execution in the middle-tier
(b) standalone execution in the database server
(c) execution in the middle tier with query shipping to the database server
(d) execution in the database server exploiting the exiting relational optimizer and runtime

The goal of this work is to create a bridge between the imperative style of programming, natural to programmers, and the performance advantages of a declarative compiler and optimizer, hence obtaining the best of both worlds.

Xcerpt and visXcerpt: Integrating Web Querying

Sacha Berger François Bry Tim Furche
University of Munich, Institute for Informatics, http://www.ifi.lmu.de/

Xcerpt [2] and visXcerpt [1], cf. http://xcerpt.org/, are Web query languages related to each other in an unusual way: Xcerpt is a *textual* query language, visXcerpt is a *visual* query language obtained by rendering Xcerpt query programs. Furthermore, Xcerpt and visXcerpt, short (vis)Xcerpt, have been conceived for querying both standard Web data such as XML and HTML and Semantic Web data such as RDF and Topic Maps.

This paper describes a demonstration focusing on three aspects of (vis)Xcerpt. First its core features, especially the pattern-oriented queries and answer-constructors, its rules or views, and its specific language constructs for incomplete specifications. Incomplete specifications are essential for retrieving semi-structured data. Second, the integrated querying of standard Web and Semantic Web data to ease the accessing of the two kinds of data in a same query program. Third, the complementary and integrated nature of the two languages.

Setting of the Demonstration. In the demonstration, prototypes of both, the textual query language Xcerpt and its visual rendering visXcerpt are demonstrated in parallel on the same examples. Both prototypes rely on the same run time system for evaluating queries, but differ in rendering: visXcerpt provides a two-dimensional *rendering* of textual Xcerpt programs implemented using mostly HTML and CSS. Additionally, the visual prototype provides an interactive environment for editing visXcerpt queries, as well as for data, query, and answer browsing.

Excerpts from DBLP[1], and from a computer science taxonomy form the base for the scenario considered in the demonstration. DBLP is a collection of bibliographic entries for articles, books, etc. in the field of Computer Science. DBLP data are representatives for standard Web data using a mixture of rather regular XML content combined with free form, HTML-like information. A small Computer Science taxonomy has been built for the purpose of this demonstration. Very much in the spirit of SKOS [3], this is a lightweight ontology based on RDF and RDFS. Combining such an ontology as metadata with the XML data of DBLP is a foundation for applications such as community based classification and analysis of bibliographic information using interrelations between researchers and research fields. Realizing such applications is eased by using the integrated Web and semantic Web query language (vis)Xcerpt that also allows reasoning using rules.

Technical Content of the Demonstration. The use of query and construction patterns in (vis)Xcerpt is presented, both for binding variables in query terms and for reassembling the variables in so-called construct terms. The variable binding paradigm is that of Datalog, i.e. the programmer specifies patterns (or terms) including variables. Special interactive behavior of variables in visXcerpt highlights the relation between variables in query and construct terms. Arguably, pattern based querying and constructing together with the variable binding paradigm make complex queries easier to specify and read. This is demonstrated by online query authoring and refactoring.

To cope with the semistructured nature of Web data, (vis)Xcerpt query patterns use a notion of incomplete term specifications with optional or unordered content specification. This feature distinguishes (vis)Xcerpt from query languages like Datalog and query interfaces like "Query By Example" [4]. Simple, yet powerful textual and visual constructs of incompleteness are presented in the demonstration.

An important characteristic of (vis)Xcerpt is its rule-based nature: (vis)Xcerpt provides rules very similar to SQL views. Arguably, rules or views are convenient for a logical structuring of complex queries. Thus, in specifying a complex query, it might ease the programming and improve the program readability to specify (abstract) rules as intermediate steps—very much like procedures in conventional programming. Another aspect of rules is the ability, to solve simple reasoning tasks. Both aspects of rules are needed for the demonstration scenario.

Referential transparency and answer closedness are essential properties of Xcerpt and visXcerpt, surfacing in various parts of the demonstration. They are two precisely defined traits of the rather vague notion of "declarativity". Referential transparency means that within a definition scope, all occurrences of an expression have the same value, i.e., denote the same data. Answer-closedness means that replacing a sub-query in a compound query by a possible single answer always yields a syntactically valid query. Referentially transparent and answer-closed programs are easy to understand (and therefore easy to develop and to maintain), as the unavoidable shift in syntax from the data sought for to the query specifying this data is minimized.

A novelty of the visual language visXcerpt is how it has been derived from the textual language: as a rendering without changing the language constructs and the runtime system for query evaluation. This rendering is mainly achieved via CSS styling of the constructs of the textual language Xcerpt. The authors believe that this approach to twin textual and visual languages is promising, as it makes those languages easy to learn—and easy to develop. The first advantages is highlighted in the demonstration by presenting both languages side-by-side.

[1] http://www.informatik.uni-trier.de/~ley/db/

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. http://www.rewerse.net/).

References.

[1] S. Berger, F. Bry, S. Schaffert, and C. Wieser. Xcerpt and visXcerpt: From Pattern-Based to Visual Querying of XML and Semistructured Data. In *29th Intl. Conf. on Very Large Data Bases*, 2003.

[2] S. Schaffert and F. Bry. Querying the Web Reconsidered: A Practical Introduction to Xcerpt. In *Extreme Markup Languages*, 2004.

[3] W3C. *Simple Knowledge Organisation System (SKOS)*, 2004.

[4] Moshé M. Zloof. Query-by-Example: A Data Base Language. *IBM Systems Journal*, 16(4):324–343, 1977.

XJ: Integration of XML Processing into Java™

Rajesh Bordawekar Michael Burke Igor Peshansky Mukund Raghavachari

IBM T.J. Watson Research Center
{bordaw, mgburke, igorp, raghavac}@us.ibm.com

1 Introduction

XML has emerged as the de facto standard for data interchange. One reason for its popularity is that it defines a standard mechanism for structuring data as ordered, labeled trees. The utility of XML as an application integration mechanism is enhanced when interacting applications agree on the structure and vocabulary of labels of the XML data interchanged. This requirement has led to the development of the XML Schemastandard — an XML Schema specifies a set of XML documents whose vocabulary and structure satisfy constraints in the XML Schema.

Despite the increased importance of XML, the available facilities for processing XML in current programming languages are primitive. Programmers often use runtime APIs such as DOM [6], which builds an in-memory tree from an XML document, or SAX [5], where an XML document parser raises events that are handled by an application. None of the benefits associated with high-level programming languages, such as static type checking of operations on XML data are available. The responsibility of ensuring that operations on XML data respect the XML Schema associated with it falls entirely on the programmer.

The alternative approach to using standard interfaces to process XML data is to embed support for XML within the programming language. Support for query languages such as XPath in the programming language provides a natural, succinct and flexible construct for accessing XML data. Extending current programming languages with awareness of XML, XML Schema, and XPath through a careful integration of the XML Schema type system and XPath expression syntax can simplify programming and enables useful services such as static type checking and compiler optimizations.

The subject of this demonstration is XJ, a research language that integrates XML as a first-class construct into Java. The design goals of XJ distinguish it from other projects that integrate XML into programming languages. The goal of introducing XML as a type into an object-oriented imperative language is not new — Cω [1], Xtatic [3], Xact [4] have studied the integration of XML into C♯ and Java. What sets XJ apart from these and other languages is its consistency with XML standards such as XML Schema and XPath, and its support for in-place updates of XML data, thereby keeping with the imperative nature of general-purpose languages like Java.

2 Demonstration Overview

This demonstration will introduce XJ and its language features using an Eclipse-integrated development environment [2], and demonstrate how the XJ compiler converts XJ code into Java code that uses DOM[6] to perform accesses to XML data. We will also discuss optimizations, such as common sub-expression eliminations, which are applicable broadly to any XML processing language, including XQuery.

References

[1] G. Bierman, E. Meijer, and W. Schulte. The essence of data access in Cω. In *Proceedings of the European Conference on Object-Oriented Programming*, 2005.

[2] Eclipse project. XML schema infoset model. http://www.eclipse.org/xsd/.

[3] V. Gapeyev and B. C. Pierce. Regular object types. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 151–175, 2003.

[4] C. Kirkegaard, A. Møller, and M. I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

[5] Simple API for XML. http://www.saxproject.org.

[6] World Wide Web Consortium. *Document Object Model Level 2 Core*, November 2000.

XML Support In Visual Basic 9

Erik Meijer* Brian Beckman†

XML Programming Using DOM

Programming against XML using the DOM API today is a bitch. The accidental complexity of working with the DOM is so high that many programmers are giving up on using XML altogether, cursing the hype that XML makes dealing with data simple, which no one who has actually written DOM code could claim. The W3C DOM was not designed with ease of programming in mind, but rather evolved as a design by committee from the existing DHTML object model originally created by Netscape.

The DOM implementation as surfaced in the .NET frameworks as the `System.Xml.XmlDocument` API is extremely imperative, irregular, and complex. Nodes are not first class citizens and have to be created and exist in the context of a given document. The access patterns for attributes and elements are gratuitously different, and the handling of namespaces is confusing at best. Finally even pretty-printing an XML document takes several lines of arcane and complex code since the `.ToString()` method is not properly overridden.

XML Programming Using XLinq

To adress the complexity of working with XML, we designed XLinq, a new modern lightweight XML API that is designed from the ground up with simplicity and ease of programming in mind. Moreover Xlinq integrates smoothly with the language integrated queries of the LINQ framework. The XLinq object model contains a handful of types. The abstract class `XNode` is the base for element nodes; the abstract class `XContainer` is the base for element nodes that have children. The `XElement` class represents proper XML elements, and the `XAttribute` class represents attributes and is stand-alone; it does not derive from `XNode`. The `XName` class represents fully expanded XML names.

In XLinq nodes are truly first class citizens that can be passed around freely independent of an enclosing document context. Nested elements are constructed in an expression-oriented fashion, but XLinq also supports imperative updates in case programmers need them. Elements and attributes are accessed uniformly using familiar XPath axis-style methods, while namespace handling is simplified using the notion of universal names throughout the API. Last but not least, `.ToString()` actually works, so it is trivial to pretty print XML documents using a single method call.

*Erik.Meijer@microsoft.com
†Brian.Beckman@microsoft.com

XML Programming Using VB

On top of the base XLinq API, Visual Basic adds XML literals with full namespace support, and late bound axis member for attribute, child, and descendant access. Programming against XML now actually is easy, as it was originally intended.

With XML literals, we can directly embed XML fragments in a Visual Basic program. Inside XML literals we can leave holes for attributes, attribute names, or attribute values, for element names by using `(expression)`, or for child elements using the ASP.Net style syntax `<%= expression %>`, or `<% statement %>` for blocks. The Visual Basic compiler takes XML literals and translates them into constructor calls of to the underlying XLinq API. As a result, XML produced by Visual Basic can be freely passed to any other component that accepts XLinq values, and similarly, Visual Basic code can accept XLinq XML produced by external components.

Visual Basic's XML literals also simplify handling of namespaces. We support normal namespace declarations, default namespace declarations, and no namespace declarations, as well as qualified names for elements and attributes. The compiler generates the correct XLinq calls to ensure that prefixes are preserved when the XML is serialized.

Whereas XML literals make constructing XML easy in Visual Basic, the concept of axis members makes accessing XML easy. The essence of the idea is to delay the binding of identifiers to actual XML attributes and elements until run time. When the compiler cannot find a binding for a variable, it emits code to call a helper function at run time. This tactic will be familiar to many under the rubric "late binding", and, indeed, it is a form of ordinary Visual Basic late binding. But it has the advantage that the names of element tags and attributes can be used directly in Visual Basic code without quoting. As such, it relieves the programmer of the significant cognitive burden of switching between object space and XML-data space. The programmer can treat the spaces the same: as hierarchies accessed through ".".

More Information

More information on LINQ, XLinq and Visual Basic 9 can be found on `http://msdn.microsoft.com/netframework/future/linq/`

XACT

XML Transformations in Java

Christian Kirkegaard and Anders Møller
BRICS
Department of Computer Science
University of Aarhus, Denmark
{ck,amoeller}@brics.dk

Introduction

XACT is a framework for programming XML transformations in Java. Among the key features of this approach are

- a notion of immutable *XML templates* for manipulating XML fragments, using XPath for navigation; and
- static guarantees of validity of the generated XML data based on data-flow analysis of XACT programs using a lattice structure of *summary graphs*.

An early version of the language design and the program analysis is described in [3]. In [1], we present an efficient runtime representation. The paper [2] shows how the analysis technique can be extended to support XML Schema as type formalism and permit optional type annotations for improving modularity of the type checking.

Demonstration

We demonstrate the capabilities of XACT by stepping through an example, showing how the program analyzer works "under the hood". This involves

1. desugaring special syntactic constructs to Java code;
2. construction of summary graphs from XML templates and schemas;
3. data-flow analysis (based on Soot), including transfer functions for XML operations; and
4. validation of summary graphs.

Specifically, we focus on the novel features: the support for XML Schema and optional type annotations.

Schemas are converted, without loss of precision (ignoring keys and references), to a convenient subset of RELAX NG, and then further to summary graphs, which are then used in the data-flow analysis. When this analysis reaches a fixed point (which represents a conservative approximation of the XML values that may appear at runtime), the resulting summary graphs are validated relative to the schema annotations.

By allowing type annotations, XACT permits a modular validity analysis where components can be analyzed individually. At the same time, type annotations are optional – they can be omitted for intermediate results that do not conform to named schema constructs, thereby supporting a flexible style of programming.

Implementation

Our implementation of the XACT analyzer and runtime system is available at

`http://www.brics.dk/Xact/`

References

[1] Christian Kirkegaard, Aske Simon Christensen, and Anders Møller. A runtime system for XML transformations in Java. In *Proc. Second International XML Database Symposium, XSym '04*, volume 3186 of *LNCS*. Springer-Verlag, August 2004.

[2] Christian Kirkegaard and Anders Møller. Type checking with XML Schema in Xact. Technical Report RS-05-31, BRICS, 2005. Presented at Programming Language Technologies for XML, PLAN-X '06.

[3] Christian Kirkegaard, Anders Møller, and Michael I. Schwartzbach. Static analysis of XML transformations in Java. *IEEE Transactions on Software Engineering*, 30(3):181–192, March 2004.

XTATIC

PLAN-X 2006 Demo

Vladimir Gapeyev Michael Levin* Benjamin Pierce Alan Schmitt†

University of Pennsylvania

XTATIC integrates with a mainstream object-oriented language, C♯, the key features of statically typed XML processing previously developed in XDUCE, a domain-specific XML processing language. These features include XML trees as built-in values, a type system based on *regular types* (closely related to schema languages such as DTD and their successors) for static typechecking of computations involving XML, and a powerful form of pattern matching called *regular patterns*.

By being an extension of C♯, XTATIC receives, for free, abstraction, modularization, and control flow mechanisms of an established programming language, as well as access to its extensive libraries. The extension made by XTATIC to the core of C♯ is minimal: it consists of enriching the universe of C♯ values and types by constructs for trees and sequences that generalize those of XDUCE, and adding the pattern matching primitive for their processing. The key observation for the integration is that the semantics of trees in XDUCE easily generalizes to permit using, in place of XML tags, other kinds of values and types as tree labels—for example, objects and classes of C♯. Then the integration of trees with the object-oriented data model of C♯ is accomplished by grafting the subtyping relation of the so generalized XDUCE regular types into the C♯ class hierarchy under a special class `Xtatic.Seq`, therefore making all regular types be subtypes of `seq`. This allows trees and sequences to be passed to generic library facilities such as collection classes, stored in fields of objects, etc. Finally, this general extension encodes XML by trees that use objects from a special class `Xtatic.Tag` as tree labels. This approach is similar to the way arrays—which, like trees, are a form of structural types—are integrated in C♯ as subtypes of the special class `System.Array`.

Subtyping in XTATIC subsumes both the declarative object-oriented subclass relation and the richer extensionally defined subtyping relation of regular types: It turns out that the traditional definition of subclassing can be reformulated—without changing the relation itself—to mimic the XDUCE's definition of subtyping as inclusion between sets of values inhabiting the types. Likewise, XTATIC's pattern matching incorporates a natural form of type-based pattern matching on objects. This provides a safe alternative to casts as a mechanism for determination of an object's run-time type.

XTATIC does not support any form of destructive update of the sequence and tree structure of existing values. Instead, the language promotes a declarative style of processing, in which values and subtrees are extracted from existing trees and used to construct entirely new trees. This approach agrees with the treatment of trees in XSLT and XQUERY, and has a precedent in C♯ provided by strings, which are also decomposable, but immutable, values.

Due to the lightweight extension approach to the design of XTATIC, the feel of XML programming in the resulting language fits between programming with XML APIs and programming in high-level XML-specific languages. On one hand, XTATIC offers—as the high-level languages do—native and concise XML processing primitives and types instead of untyped low-level API manipulations. On the other hand, these primitives are used within the control flow and abstractions framework of an object-oriented language, which is more familiar to the majority of programmers than the more esoteric frameworks of XSLT and XQUERY. Psychological and educational considerations aside, this poses XTATIC as an attractive alternative to API-based programming in applications where efficiency is of immediate concern. Currently, such projects tend to avoid using XSLT, XQUERY, or even XPATH, due to uncertainty over presence of optimization for high-level control flows in a given implementation of these languages, as well as lack of control over decisions of the optimizer. This control (indeed, full responsibility for implementing the high-level control flow) is in the hands of an XTATIC programmer to the same degree as for an API programmer.

These benefits are shared by XTATIC with other current proposals for integrating XML processing into object-oriented languages, e.g., XOBE, XJ, XACT, and Cω. XTATIC differs from these in other respects: more flexible integration of trees into the object-oriented data model and use of regular patterns, rather than paths, as the main XML inspection mechanism. Used in conjunction with regular types, patterns support the full spectrum of processing styles, from dynamic investigation of documents of unknown or partially known types to fully checked processing of documents for which complete type information is known—all without changing the underlying data representation.

XTATIC is implemented as a translator into pure C♯ code, which can be compiled into .NET CLR and executed in conjunction with a small library that implements tree sequences and elementary operations on them.

* Currently at Microsoft.
† Currently at INRIA Rhône-Alpes.

OCamlDuce

Alain Frisch
INRIA Rocquencourt
Alain.Frisch@inria.fr

Context. Over the last few years, the programming language research community has identified issues raised by the support of XML documents in applications and has proposed new linguistic features to deal with them. The work by Hosoya, Pierce and Vouillon on the XDuce project has had a big influence. Amongst its main contributions are the design of *regular expression types* (to express structural constraints on documents) and *regular expression patterns* (to express complex information extraction from documents) which together contribute to a sound and expressive language for developping XML-oriented applications such as transformations.

XDuce encouraged the vision of XML manipulation as a value-based process in the spirit of functional languages. As a matter a fact, XDuce has striking similarities with the family of ML languages. Since XDuce and ML languages are good for different but related kind of problems and because of their apparent similarity, it is natural to try to combine them.

However, despite the similarity, XDuce is missing important features from ML languages such as first-class functions, polymorphism, automatic type reconstruction, and support for programming in the large. There are two natural responses to address this lack of features: either extend XDuce underlying theory to deal with them, or integrate XDuce features in an existing full-blown ML language. Examples of the former include existing extensions of XDuce with first-class functions or with parametric polymorphism. However, it is not clear how these extensions could be combined, and a lot of work is still necessary to integrate other missing features. Also, it seems pointless to design and implement a full-blown language *only* to add support for XML. The idea of integrating XDuce features into an existing full-blown general-purpose language has been explored for instance in the Xtatic project, which adds XDuce types and patterns into the C# programming language. The part of Xtatic programs that deals with XML inherits the functional flavor from XDuce. This might indicate that a functional language could be a very good target for integrating XDuce.

OCamlDuce. OCamlDuce is an experimental merger between the Objective Caml (OCaml) and CDuce languages. The language was designed so as to make it easier to develop possibly large applications which need to deal with XML document without necessarily being focused primarily on XML (unlike, say, pure XML-to-XML transformation). Typical use cases would be to add support for custom XML configuration files, for XHTML report generators, or for web-service interfaces, ... to an existing OCaml application.

OCaml is a powerful general-purpose multi-paradigm/functional-oriented programming language from the ML family with a robust, efficient and popular implementation. CDuce is a small programming language adapted to the development of safe and efficient XML-oriented applications. CDuce supports XML literals, Unicode, XML Namespaces, XML types, XML pattern matching together with a precise type inference and an efficient automata-based and type-driven compilation strategy, XML iterators. Part of the theory behind CDuce relies on the one developped in the XDuce project.

From the programmer point of view, OCamlDuce comes as drop-in replacements for the OCaml tools: bytecode and native compilers, toplevel. All OCaml features are available, and it is possible to reuse standard and third-party OCaml libraries without even recompiling them. OCamlDuce also integrates all of the features from CDuce except overloaded functions. It is thus mostly straightforward to translate CDuce programs to OCamlDuce.

Integrating OCaml and CDuce. OCamlDuce has been implemented by merging together the OCaml and CDuce source trees and adding a relatively small piece of glue code. The only technically challenging part of the OCaml / CDuce integration was the combination of two radically different type systems: OCaml relies on Hindley-Milner-like type inference, and CDuce relies on forward propagation and on tree-automata techniques. The theoretical foundation of the type system is described in a paper to be presented in PLAN-X 2006. The key idea to obtain a clean and simple type system was to keep the XML values and types self-contained: they can appear within regular OCaml values and types, but the converse is not possible. However, bridges between the worlds of XML and ML values are provided in OCamlDuce. They rely on an automatic structural translation of ML types into XML types, which allows to move values between the two worlds.

Because of the way CDuce types and values are dealt with in OCamlDuce, it is not possible e.g. to have first-class functions or arbitrary OCaml values within XML values. We don't see it as a problem because CDuce values are intended to represent XML fragments in OCamlDuce, not arbitrary data containers which OCaml supports already pretty well. More problematic is the lack of interaction between OCaml parametric polymorphism and XML types (OCaml type variables cannot appear within CDuce types). We leave this challenging point for future work.

The demonstration. The demonstration will illustrate how the features added to OCaml can be used to write idiomatic, expressive and safe code that manipulates complex XML structures. The examples will be taken from a medium-sized application developped in OCamlDuce, which parses an XML-Schema definition into an OCaml graph-like data structure, extracts some informations from it, and produces an XHTML report.

LAUNCHPADS: A System for Processing Ad Hoc Data

Mark Daly
Princeton University
mdaly@Princeton.EDU

Mary Fernández
Kathleen Fisher
AT&T Labs Research
mff,kfisher@research.att.com

Yitzhak Mandelbaum
David Walker
Princeton University
yitzhakm,dpw@CS.Princeton.EDU

An Introduction to PADS. Ideally, any data we ever encounter will be presented to us in standardized formats, such as XML. Why? Because for formats like XML, there are a whole host of software libraries, query engines, visualization tools and even programming languages specially designed to help users process their data. However, we do not live in an ideal world, and in reality, vast amounts of data is produced and communicated in *ad hoc formats*, those formats for which no data processing tools are readily available. Figure 1 presents a small selection of ad hoc data sources. As one can see, ad hoc data exists in a very wide variety of fields and the users range from network administrators to computational biologists and genomics researchers to physicists, financial analysts and everyday programmers.

Programmers often deal with this data by whipping up one-time Perl scripts or C programs to parse and analyze their data. Unfortunately, this strategy is slow and tedious, and often produces code that is difficult to understand, lacks adequate error checking, and is brittle to format change over time. To expedite and improve this process, we developed the PADS data description language and system [2, 3]. Using the PADS language, one may write a declarative description of the structure of almost any ad hoc data source. The descriptions take the form of types, drawn from a dependent type theory. For instance, PADS base types describe simple objects including strings, integers, floating-point numbers, dates, times, and ip addresses. Records and arrays specify sequences of elements in a data source, and unions, switched unions and enums specify alternatives. Any of these structured types may be parameterized and users may write arbitrary semantic constraints over their data as well.

Once a programmer has written a description in the PADS language, the PADS compiler can generate a collection of format-specific libraries in C, including a parser, printer, and verifier. In addition, the compiler can compose these libraries with generic templates to create value-added tools such as an ad hoc-to-XML format conversion tool, a histogram generator, and a statistical analysis and error summary tool. Finally, PADS has been composed with the GALAX query engine [6, 4, 5] for XQuery to create PADX [1], a new system that allows users to query and transform any ad hoc data source as if it was XML, without incurring the performance penalty that usually results when one converts ad hoc data into a much more verbose XML representation.

While the PADS language provides an extremely versatile means of creating tools for processing ad hoc data, it is nevertheless a *new* language and learning a new language is time-consuming for anyone, especially for computational biologists or other scientists for whom programming is not their primary area of expertise. To ease the way for novice PADS users, we developed LAUNCHPADS, a new tool that provides access to the PADS system without requiring foreknowledge of the PADS language itself. Hence, LAUNCHPADS graphic interface will also help more experienced PADS users to shorten their development cycle and provides a convenient way for experts to quickly create any of the data processing tools they need.

Name : Use	Representation
Web server logs (CLF): Measure web workloads	Fixed-column ASCII records
CoMon data: Monitor PlanetLab Machines	ASCII records
Call detail: Fraud detection	Fixed-width binary records
AT&T billing data: Monitor billing process	Various Cobol data formats
Netflow: Monitor network performance	Data-dependent number of fixed-width binary records
Newick: Immune system response simulation	Fixed-width ASCII records in tree-shaped hierarchy
Gene Ontology: Gene-gene correlations	Variable-width ASCII records in DAG-shaped hierarchy
CPT codes: Medical diagnoses	Floating point numbers

Figure 1. Selected ad hoc data sources.

LaunchPads. LAUNCHPADS combines mechanisms for graphically defining structure and semantic properties of ad hoc data, for translation of this definition into PADS code, and for compilation/execution of the generic tools that operate over ad hoc data. More specifically, LAUNCHPADS breaks definition of an ad hoc data format and generation of data processing tools into the following steps. Figure 2 presents a screenshot of LAUNCHPADS being used to construct a data description for a web-server log format.

1. **Selection of sample data from which to build the description.** Creation of a definition within LAUNCHPADS begins when a user loads sample data into the graphical interface. In Figure 2, web log data (beginning with the IP address `207.136.97.49 ...`) appears in the top right hand corner of the picture. A user then selects a row of data to work on in the LAUNCHPADS *gridview* immediately below.
2. **Iterative refinement in the gridview.** Once in the gridview, users may specify descriptions for regions of text using a highlighting scheme. The color assigned to a region represents the description class (base or composite) and region boundaries. Structure within a definition is represented through a series of refinement steps: composite regions are broken down and level after level, thereby allowing for nested elements (Figure 2 shows four nesting levels). The refinement process bottoms out when one reaches an atomic description such as a character string, IP address or date. Once all regions have been given a base type in the gridview, LAUNCHPADS will generate a *tree-view* of the definition for further processing.

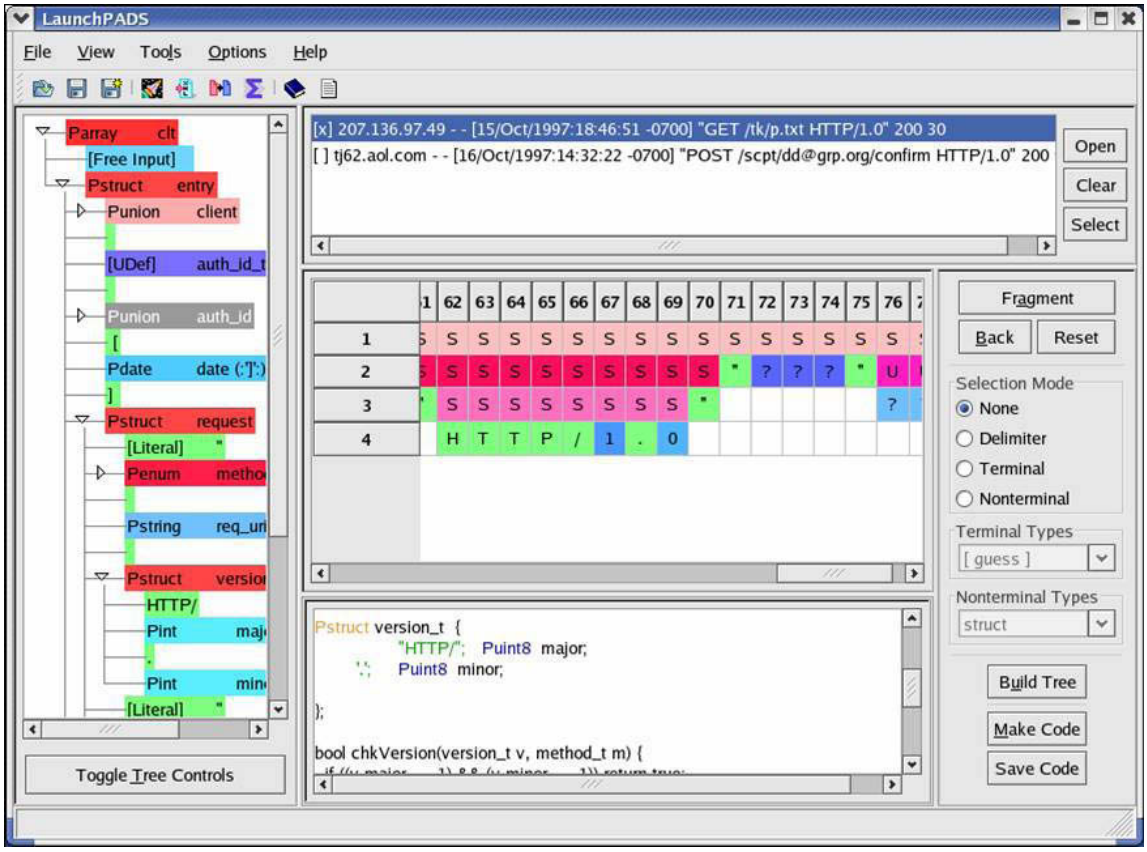


Figure 2. LaunchPads Interface.

3. **Customization in the treeview.** The treeview is a graphical representation of the abstract syntax of a PADS description. In this view, programmers can manipulate definitions with a high degree of precision: definition elements may be created, destroyed, and renamed; type associations for existing elements may be changed (within limitations); element ordering may be altered; user defined types may be added to the definition and applied to elements; content-aware error constraints may be imposed. Indeed, from within the tree view it is possible to access the "expert" functions of PADS directly if one so chooses, or to completely avoid them in lieu of a simpler definition and/or faster development time.
4. **PADS code generation, tool compilation and use.** When the user is satisfied with their PADS definition in the treeview, they may generate PADS code. Any such generated code is guaranteed to be syntactically correct so the user need not worry about fussing with concrete PADS syntax if they do not want to. Figure 2 shows the generated code in the window at the bottom of the interface. By using the pulldown menus at the top and a set of "wizards," the user may now issue commands to compile the generated code and create data processing tools including the XML converter and statistical analyzer. As development of LAUNCHPADS continues, we will add further tools and corresponding wizards to the interface.

.

Conclusions In summary, in this demonstration, we will explain the many challenges that ad hoc data pose and how the PADS language is structured to meet these challenges. In addition, we will explain how LAUNCHPADS provides further support for processing ad hoc data by demonstrating both features for helping users construct data descriptions and features for creating and invoking tools that operate over data. We believe that both expert programmers and novices alike can benefit from this simple system for manipulating ad hoc data.

References

[1] M. Fernández, K. Fisher, and Y. Mandelbaum. PADX: Querying large-scale ad hoc data with XQuery. Submitted to PLAN-X 2006.

[2] K. Fisher and R. Gruber. PADS: A domain-specific language for processing ad hoc data. In *Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, June 2005.

[3] K. Fisher, Y. Mandelbaum, and D. Walker. The next 700 data description languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Jan. 2006. To appear.

[4] Galax user manual. `http://www.galaxquery.org/doc.html#manual`.

[5] C. Ré, J. Siméon, and M. Fernández. A complete and efficient algebraic compiler for XQuery. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, April 2006.

[6] J. Siméon and M. F. Fernández. Build your own XQuery processor. EDBT Summer School, Tutorial on Galax architecture, Sept 2004. `http://www.galaxquery.org/slides/edbt-summer-school2004.pdf`.

XHaskell

Martin Sulzmann and Kenny Zhuo Ming Lu
School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{sulzmann,luzm}@comp.nus.edu.sg

We demonstrate the current programming capabilities of XHaskell – an extension of Haskell with XDuce style regular expression types and regular expression pattern matching. For example, the following classic XDuce program to extract telephone entries out of an address book

```
regtype P = P[N,T?,E*] -- Person
regtype N = N[String]  -- Name
regtype T = T[String]  -- Tel
regtype E = E[String]  -- Email
regtype En = En[N,T]   -- Phonebook Entry
addrbook :: P* -> En*
addrbook (P[n as N, t as T, E*], xs as P*)
         = (En[n,t], (addrbook xs))
addrbook (P[N,E*], xs as P*) = addrbook xs
addrbook () = ()
```

can be rewritten in XHaskell as follows.

```
module Addrbook where
data P = P N (T?) ((E)*) -- Person
data N = N [Char]        -- Name
data T = T [Char]        -- Tel
data E = E [Char]        -- Email
data En = En N T         -- Entry
addrbook :: ((P)*) -> ((En)*)
addrbook (x :: ((P)*)) =
    (map for_each_p) x               -- (1)
for_each_p :: P -> (En?)
for_each_p (P (n :: N) (t :: (T?))
             (es :: ((E)*)))
     = for_each_p2 (n,(t,es))
for_each_p2 :: (N,((T?),((E)*))) -> (En?)
for_each_p2 ((n :: N),((t :: T),
             (es :: ((E)*))))
     = En n t
for_each_p2 ((n :: N),(es :: ((E)*)))
      = ()
```

The interesting point to note is that in XHaskell we can call Haskell Prelude functions such as `map::(a->b)->[a]->[b]` (see location (1)). Thus, we only need to define the transformation from `Person` to `Entry`. Our current implementation does not support regular hedges. Therefore, we need the auxiliary function `for_each_p2`.

XHaskell is compiled to Haskell. Hence, we can easily take advantage of existing XML tools written in Haskell. E.g., we can use the *DtdtoHaskell* command provided by the HaXML tool to generate the `AddrbookDTD` module which describes the DTD structure of the address book example in terms of some Haskell data types. Currently, the XML document representation provided by HaXML is slightly different from XHaskell. Hence, the programmer must provide an extra interface module `HaXMLInterface` for marshalling values between the two representations. Though, this intermediate step could be easily automated.

Here is the code integrating XHaskell with HaXML.

```
module App where
import Addrbook ( addrbook )
import AddrbookDTD
import HaXMLInterface
       ( haxml2xhaskell, xhaskell2haxml )
main =
  fix2Args >>= \(infile,outfile)->
  do value <- fReadXml infile
     let result = xhaskell2haxml
        (addrbook (haxml2xhaskell value))
     fWriteXml result outfile
```

The `main` function parses an XML document specified by argument `infile`, and applies function `addrbook` to the parsed value. Note that `addrbook` has type `[P]->[En]` in the translation to Haskell. Finally it prints the result into the output file specified by argument `outfile`.

The implementation and further background material can found here:
`http://www.comp.nus.edu.sg/~luzm/xhaskell/`

Recent BRICS Notes Series Publications

NS-05-6 **Giuseppe Castagna and Mukund Raghavachari, editors. *PLAN-X 2006 Informal Proceedings,* (Charleston, South Carolina, January 14, 2006), December 2005. ii+92.**

NS-05-5 **Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Maurice Herlihy, Kim G. Larsen, and Martin Raußen, editors. *Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency, GETCO '05,* (San Francisco, California, USA, August 21, 2005), August 2005. vi+44.**

NS-05-4 **Scott A. Smolka and Jiří Srba, editors. *Preliminary Proceedings of the 7th International Workshop on Verification of Infinite-State Systems, INFINITY '05,* (San Francisco, USA, August 27, 2005), June 2005. vi+64 pp.**

NS-05-3 **Luca Aceto and Andrew D. Gordon, editors. *Short Contributions from the Workshop on Algebraic Process Calculi: The First Twenty Five Years and Beyond, PA '05,* (Bertinoro, Forlì, Italy, August 1–5, 2005), June 2005. vi+239 pp.**

NS-05-2 **Luca Aceto and Willem Jan Fokkink. *The Quest for Equational Axiomatizations of Parallel Composition: Status and Open Problems*. May 2005. 7 pp. To appear in a volume of the BRICS Notes Series devoted to the workshop "Algebraic Process Calculi: The First Twenty Five Years and Beyond", August 1–5, 2005, University of Bologna Residential Center Bertinoro (Forlì), Italy.**

NS-05-1 **Luca Aceto, Magnus Mar Halldorsson, and Anna Ingólfsdóttir. *What is Theoretical Computer Science?* April 2005. 13 pp.**

NS-04-2 **Patrick Cousot, Lisbeth Fajstrup, Eric Goubault, Maurice Herlihy, Martin Raußen, and Vladimiro Sassone, editors. *Preliminary Proceedings of the Workshop on Geometry and Topology in Concurrency and Distributed Computing, GETCO '04,* (Amsterdam, The Netherlands, October 4, 2004), September 2004. vi+80.**

NS-04-1 **Luca Aceto, Willem Jan Fokkink, and Irek Ulidowski, editors. *Preliminary Proceedings of the Workshop on Structural Operational Semantics, SOS '04,* (London, United Kingdom, August 30, 2004), August 2004. vi+56.**