A Typeful and Tagless Representation for XML Documents[⋆]

Dengping Zhu and Hongwei Xi

Computer Science Department
Boston University

{zhudp, hwxi}@cs.bu.edu

Abstract. When constructing programs to process XML documents, we immediately face the question as to how XML documents should be represented internally in the programming language we use. Currently, most representations for XML documents are typeless in the sense that the type information of an XML document cannot be reflected in the type of the representation of the document (if the representation is assumed to be typed). Though convenient to construct, a typeless representation for XML documents often makes use of a large number of representation tags, which not only require some significant amount of space to store but may also incur numerous run-time tag checks when the represented documents are processed. Moreover, with a typeless representation for XML documents, it becomes difficult or even impossible to statically capture program invariants that are related to the type information of XML documents. Building upon our recent work on guarded recursive datatypes, we present an approach to representing XML documents in this paper that not only allows the type information of an XML document to be reflected in the type of the representation of the document but also significantly reduces the need for representation tags that are required in typeless representations. With this approach, we become able to process XML documents in a typeful manner, thus reaping various well-known software engineering benefits from the presence of types.

1 Introduction

XML (eXtensible Markup Language) [13] is a simple and flexible text format derived from SGML. Originally designed to support large-scale electronic publishing, XML is now also playing an increasingly important role in the exchange of a wide variety of data on the Web and elsewhere. As a markup language, XML adds structural information around the data in a document. For instance, an XML document in text format is presented in the left part of Figure 1. We use the name *start tag* (*end tag*) for what is denoted by the syntax <*tagname*> (</*tagname*>) in XML. Also, we use the name *data* loosely for the contents between tags. A tagged element, or element for short, consists of a start tag and an

⋆ Partially supported by NSF grants no. CCR-0224244 and no. CCR-0229480

```
<addrbook>
  <person>                             <!DOCTYPE addrbook [
    <name>Dengping Zhu</name>            <!ELEMENT addrbook (person*)>
    <email>zhudp@cs.bu.edu</email>       <!ELEMENT person (name, (email)?)>
  </person>                              <!ELEMENT name (#PCDATA)>
  <person>                               <!ELEMENT email (#PCDATA)>
    <name>Hongwei Xi</name>            ] >
  </person>
</addrbook>
```

Fig. 1. An XML document and a DTD

end tag, which may enclose data or any sequence of other elements. A well-formed XML document consists of a single root element that contains other properly nested elements. For instance, in the XML document in Figure 1, `<name>` is a start tag, `</name>` is an end tag, and `Dengping Zhu` is a piece of data, and the start and end tags for the root element are `<addrbook>` and `</addrbook>`, respectively.

Currently, there have been many proposed type systems for XML (e.g., XML Schema [10, 14, 15], RELAX NG [8], etc.), and most of these type systems extend as well as refine the notion of DTD (Document Type Definition), which itself is a simple type system designed for describing the structure of XML documents [13]. As an example, we present a DTD in the right part of Figure 1, to which the XML document in the left part conforms. Essentially, the DTD defines a document type containing a root element *addrbook* and three other elements *person*, *name* and *email*; an *addrbook* element contains a (possibly empty) sequence of *person* elements, and a *person* element contains a *name* element followed by an *email* element, and a *name* element contains some character data, and an *email* element contains some character data as well.

```
datatype element = E of string * content list
     and content = CE of element | CT of string

(* The following is a generic representation of the XML
   document in Figure 1 *)

E ("addrbook", [CE (E ("person", [CE (E ("name",  CT "Denping Zhu")),
                CE (E ("email", CT "zhudp@cs.bu.edu"))])),
                CE (E ("person", [CE (E ("name",  CT "Hongwei Xi"))]))])
```

Fig. 2. A generic representation for XML

```
datatype addrbook = Addrbook of person list
     and person = Person of name * email option
     and name = Name of string
     and email = Email of string

(* The following is a specific representation of the XML
   document in Figure 1 *)

Addrbook [Person (Name "Dengping Zhu", SOME (Email "zhudp@cs.bu.edu")),
          Person (Name "Hongwei Xi", NONE) ]
```

Fig. 3. A specific representation for XML

In order to construct programs for processing XML documents, we need to form some data structure to represent the documents internally in the programming language we use. In Figure 2, we declare a datatype *element* in Standard ML (SML) [6] for this purpose. Evidently, each XML document[1] can be readily represented as a value of type *element*. However, with this approach, the type information of an XML document is completely lost in the type of its representation. In the following presentation, we refer to this representation as a *generic representation* for XML documents. This is essentially the first of the two approaches presented in [12] with which various generic combinators can be implemented to facilitate the construction of programs for processing XML documents. Though viable, this approach makes it impossible to verify through static type-checking whether the representation of an XML document is valid, that is, it is well-formed and conforms to a specified DTD. Also, a great number of constructors (*E*, *CE* and *CT*) are often needed to represent XML documents, potentially resulting in less efficient processing of XML documents.

We can also declare a datatype *addrbook* in Figure 3 to represent the particular XML document in Figure 1. In the following representation, we refer to this representation as a *specific representation* for XML documents. This is essentially the second of the two approaches presented in [12], where a translation from DTDs to types in Haskell is given. While this approach does allow us to use types to differentiate XML documents conforming to distinct DTDs, a troubling problem occurs: It now seems difficult or even impossible to construct programs for processing XML documents that are polymorphic on DTDs.

Clearly, there is a serious contention between the first and the second approaches above. In [4], it is proposed that the specific representation (in the second approach) be used to represent XML documents and generic programming as is supported in Generic Haskell [2] be adopted to implement generic combinators to facilitate the construction of programs for processing XML documents. In particular, it is shown in [4] that a function can be constructed in Generic Haskell to compress XML documents conforming to distinct DTDs. We, however, take an alternative in this paper. We are to present a representation for

[1] We ignore attribute lists in XML documents at this moment.

XML documents that not only allows the type information of an XML document to be reflected in the type of its representation but also significantly (though, not completely if the document conforms to a cyclic DTD) reduces the need for numerous representation tags.[2] This representation is based upon a recently invented notion of guarded recursive datatypes [17]. General speaking, we are to introduce a type constructor *XML* that takes a type index i to form a type $\mathit{XML}(i)$ for XML documents conforming to the DTD denoted by i; a value of type $\mathit{XML}(i)$ consists of a *type* part and a *data* part such that the former represents the structure of an XML document while the latter represents the data of the document. In this paper we will skip the generation of this representation because it can be readily done by a validator.

We organize the rest of the paper as follows. In Section 2, we introduce a type index language $\mathcal{L}_{\mathrm{dtd}}$ in which type index expressions can be formed to represent DTDs. We then present a typeful and tagless representation for XML documents in Section 3 and use some examples to show how such a representation can effectively support XML document processing. In Section 4, we report some measurements gathered from processing an XML document and give a brief explanation about the measurements. We mention some related work in Section 5 and then conclude.

2 The Type Index Language $\mathcal{L}_{\mathrm{dtd}}$

In Dependent ML [18, 16], a language schema is presented for extending ML with a restricted form of dependent types where type index expressions are required to be drawn from a given constraint domain or a type index language as we now call it. We use DML($\mathcal{L}$) for such an extension, where $\mathcal{L}$ denotes the given type index language.

In this section, we present a type index language $\mathcal{L}_{\mathrm{dtd}}$ based on DTDs. To ease the presentation, we do not treat the feature of attribute lists in DTDs. However, we emphasize that there is no obstacle in theory that prevents us from handling attribute lists. The language $\mathcal{L}_{\mathrm{dtd}}$ is typed, and we use the name *sort* for a type in $\mathcal{L}_{\mathrm{dtd}}$ so as to avoid potential confusion. The syntax of $\mathcal{L}_{\mathrm{dtd}}$ is given as follows.

$$\begin{array}{lll}
\text{sorts} & \gamma ::= & \mathit{tag} \mid \mathit{doc} \\
\text{index exp.} & i \ ::= & a \mid t \mid b \mid \underline{\mathit{elemdoc}}(t) \mid \underline{\mathit{altdoc}}(i_1, i_2) \mid \\
 & & \underline{\mathit{seqdoc}}(i_1, i_2) \mid \underline{\mathit{stardoc}}(i) \mid \underline{\mathit{plusdoc}}(i) \mid \underline{\mathit{optdoc}}(i) \\
\text{index var. ctx.} & \phi ::= & \emptyset \mid \phi, a : \gamma \mid \phi, i_1 \equiv i_2 \\
\text{index subst.} & \theta ::= & [] \mid \theta[a \mapsto i]
\end{array}$$

For each tag <*tagname*> in XML, we assume there exists a corresponding type index constant t in $\mathcal{L}_{\mathrm{dtd}}$: t is referred to as a *tag index* with the name *tagname*. We assume a base sort *tag* for tag indexes. Also, we assume a base sort *doc* and

[2] We use the name *representation tag* for a tag used in the representation of an XML document, which should be distinguished from a start or end tag in the document.

use the name *document type index* (or simply *doctype index*) for a term of the sort *doc*. Intuitively, a doctype index is used to describe the type of a content in an XML document. We use γ for a sort, which is either *tag* or *doc*, and a for a variable ranging over index expressions. We use b for some base doctype indexes such as $\underline{empdoc}$ for empty content and $\underline{strdoc}$ for string content. We use i for an index expression, which is either a tag index denoted by t or a doctype index denoted by d. Also, we assume the existence of a signature $\mathcal{S}$ that associates each tag index t with a doctype index $\mathcal{S}(t)$ such that $\mathcal{S}(t)$ is the doctype index for contents that can be placed between tags <*tagname*> and </*tagname*>, where *tagname* is the name of t. To facilitate presentation, we may write $\langle t\rangle$, $(d_1 \mid d_2)$, $(d_1; d_2)$, $d\star$, $d+$ and $d?$ for $\underline{elemdoc}(t)$, $\underline{altdoc}(d_1, d_2)$, $\underline{seqdoc}(d_1, d_2)$, $\underline{stardoc}(d)$, $\underline{plusdoc}(d)$ and $\underline{optdoc}(d)$, respectively. We say a content in an XML document is of doctype index d if the structure of the content is described by d. Intuitively, given doctype indexes d_1 and d_2,

– $\langle t\rangle$ is the doctype index for an element that encloses a content of doctype index $\mathcal{S}(t)$ with tags <*tagname*> and </*tagname*>, where *tagname* is the name of the tag index t.
– $(d_1 \mid d_2)$ is the doctype index for a content of doctype index d_1 or d_2, and
– $(d_1; d_2)$ is the doctype index for a content consisting of some content of doctype index d_1 followed by some other content of doctype index d_2, and
– $d\star$ is the doctype index for a possibly empty sequence of contents of doctype index d, and
– $d+$ is the doctype index for a nonempty sequence of contents of doctype index d, and
– $d?$ is the doctype index for a content that is either empty or of doctype index d.

For instance, the DTD in Figure 1 introduces four tag indexes *name*, *email*, *person* and *addrbook*, which are associated with the following doctype indexes:

$$\mathcal{S}(name) = \underline{strdoc} \qquad \mathcal{S}(email) = \underline{strdoc}$$
$$\mathcal{S}(person) = (\langle name\rangle; \langle email\rangle?) \qquad \mathcal{S}(addrbook) = \langle person\rangle\star$$

Note that recursion may be involved in the declaration of a DTD. For instance, the following DTD makes use of recursion as the content of a *folder* element may contain other *folder* elements.

```
<!DOCTYPE folder [
  <!ELEMENT folder (record, (record | folder)*)>
  <!ELEMENT record EMPTY>
]>
```

It is soon to become clear that recursion in DTD poses some serious difficulties in forming a typeful representation for XML documents. The above DTD introduces two tag indexes *folder* and *record*, which are associated with the following doctype indexes:

$$\mathcal{S}(folder) = (\langle record\rangle; (\langle record\rangle \mid \langle folder\rangle)\star) \qquad \mathcal{S}(record) = \underline{empdoc}$$

$$\frac{}{\vdash \emptyset\ [\mathit{ictx}]} \qquad \frac{\vdash \phi\ [\mathit{ictx}]}{\vdash \phi, a : \gamma\ [\mathit{ictx}]} \qquad \frac{\vdash \phi\ [\mathit{ictx}] \quad \phi \vdash i_1 : \gamma \quad \phi \vdash i_2 : \gamma}{\vdash \phi, i_1 \equiv i_2\ [\mathit{ictx}]}$$

Fig. 4. The rules for forming ind. var. contexts

$$\frac{\vdash \phi\ [\mathit{ictx}]}{\phi \vdash b : \mathit{doc}} \qquad \frac{\vdash \phi\ [\mathit{ictx}] \quad \phi(a) = \gamma}{\phi \vdash a : \gamma} \qquad \frac{\vdash \phi\ [\mathit{ictx}]}{\phi \vdash t : \mathit{tag}}$$

$$\frac{\phi \vdash t : \mathit{tag}}{\phi \vdash \underline{\mathit{elemdoc}}(t) : \mathit{doc}} \qquad \frac{\phi \vdash i_1 : \mathit{doc} \quad \phi \vdash i_2 : \mathit{doc}}{\phi \vdash \underline{\mathit{altdoc}}(i_1, i_2) : \mathit{doc}} \qquad \frac{\phi \vdash i_1 : \mathit{doc} \quad \phi \vdash i_2 : \mathit{doc}}{\phi \vdash \underline{\mathit{seqdoc}}(i_1, i_2) : \mathit{doc}}$$

$$\frac{\phi \vdash i : \mathit{doc}}{\phi \vdash \underline{\mathit{stardoc}}(i) : \mathit{doc}} \qquad \frac{\phi \vdash i : \mathit{doc}}{\phi \vdash \underline{\mathit{plusdoc}}(i) : \mathit{doc}} \qquad \frac{\phi \vdash i : \mathit{doc}}{\phi \vdash \underline{\mathit{optdoc}}(i) : \mathit{doc}}$$

Fig. 5. Sorting Rules for $\mathcal{L}_{\mathrm{dtd}}$

In general, it should be straightforward to turn a DTD into a list of tags and then associate with each of these tags a properly chosen doctype index, and we here omit the effort to formalize this process.

We use ϕ for index variable contexts. In addition to assigning sorts to index variables, we can also assume equalities on index expressions in an index variable context. We use θ for index substitutions: $[]$ stands for the empty substitution, and $\theta[a \mapsto i]$ extends θ with an extra link from a to i. Given i and θ, we use $i[\theta]$ for the result of applying θ to i, which is defined in a standard manner. We use a judgment of the form $\vdash \phi\ [\mathit{ictx}]$ to mean that ϕ is well-formed. In addition, we use a judgment of the form $\phi \vdash i : \mathit{doc}$ to mean that i can be assigned the sort doc under ϕ. The rules for deriving these judgments are given in Figure 4 and Figure 5.

Given two index expressions i_1 and i_2, we write $i_1 = i_2$ to mean that i_1 and i_2 are syntactically the same. Please notice the difference between $i_1 = i_2$ and $i_1 \equiv i_2$. The following rules are for deriving judgments of the form $\vdash \theta : \phi$, which roughly means that θ matches ϕ.

$$\frac{}{\vdash [] : \cdot} \qquad \frac{\vdash \theta : \phi \quad \vdash i : \gamma}{\vdash \theta[a \mapsto i] : \phi, a : \gamma} \qquad \frac{\vdash \theta : \phi \quad i_1[\theta] = i_2[\theta]}{\vdash \theta : \Delta, i_1 \equiv i_2}$$

We use $\phi \models i_1 \equiv i_2$ for a type index constraint; this constraint is satisfied if we have $\vdash i_1[\theta] = i_2[\theta]$ for every θ such that $\vdash \theta : \phi$ is derivable. As can be expected, we have the following proposition.

Proposition 1.

– *If* $\phi \vdash i : \gamma$ *is derivable, then* $\phi \models i \equiv i$ *holds.*
– *If* $\phi \models i_1 \equiv i_2$ *holds, then* $\phi \models i_2 \equiv i_1$ *also holds.*

$$\frac{\mathit{const} \text{ is not } \mathit{const}'}{\vec{a}:\vec{\gamma}, \mathit{const}(i_1,\ldots,i_n) \equiv \mathit{const}'(i'_1,\ldots,i'_{n'}), \phi \vdash i \equiv i'} \qquad \frac{\vec{a}:\vec{\gamma} \vdash i : \gamma}{\vec{a}:\vec{\gamma} \vdash i \equiv i}$$

$$\frac{\vec{a}:\vec{\gamma}, \phi \vdash i \equiv i'}{\vec{a}:\vec{\gamma}, a \equiv a, \phi \vdash i \equiv i'} \qquad \frac{i_0 \text{ contains a free occurrence of } a \text{ but is not } a}{\vec{a}:\vec{\gamma}, a \equiv i_0, \phi \vdash i \equiv i'}$$

$$\frac{i_0 \text{ contains a free occurrence of } a \text{ but is not } a}{\vec{a}:\vec{\gamma}, i_0 \equiv a, \phi \vdash i \equiv i'}$$

$$\frac{\begin{array}{c} i_0 \text{ contains no free occurrences of } a \\ \vec{a}:\vec{\gamma}, \phi \vdash i[a \mapsto i_0] \equiv i'[a \mapsto i_0] \end{array}}{\vec{a}:\vec{\gamma}, a \equiv i_0, \phi \vdash i \equiv i'} \qquad \frac{\begin{array}{c} i_0 \text{ contains no free occurrences of } a \\ \vec{a}:\vec{\gamma}, \phi \vdash i[a \mapsto i_0] \equiv i'[a \mapsto i_0] \end{array}}{\vec{a}:\vec{\gamma}, i_0 \equiv a, \phi \vdash i \equiv i'}$$

$$\frac{\vec{a}:\vec{\gamma}, i_1 \equiv i'_1, \ldots, i_n \equiv i'_n, \phi \vdash i \equiv i'}{\vec{a}:\vec{\gamma}, \mathit{const}(i_1,\ldots,i_n) \equiv \mathit{const}(i'_1,\ldots,i'_n), \phi \vdash i \equiv i'}$$

Fig. 6. The rules for solving constraints

– *If* $\phi \models i_1 \equiv i_2$ *and* $\phi \models i_2 \equiv i_3$ *hold, then* $\phi \models i_1 \equiv i_3$ *also holds.*

The rules for solving type index constraints are given in Figure 6, where *const* and *const*$'$ range over tag indexes t, $\underline{\mathit{elemdoc}}$, $\underline{\mathit{altdoc}}$, $\underline{\mathit{seqdoc}}$, $\underline{\mathit{stardoc}}$, $\underline{\mathit{plusdoc}}$ and $\underline{\mathit{optdoc}}$. The following proposition justifies both the soundness and completeness of these rules.

Proposition 2. *A type index constraint* $\phi \models i_1 \equiv i_2$ *is satisfied if and only if we can use the rules in Figure 6 to derive* $\phi \vdash i_1 \equiv i_2$*.*

Proof. Assume that $\phi = (\vec{a}, \phi')$ for some ϕ' that does not begin with an index variable. The proof follows from induction on the lexicographic ordering (n_1, n_2), where n_1 is the number of free index variables in ϕ' and n_2 is the size of ϕ'.

With the type index language $\mathcal{L}_{\mathrm{dtd}}$ being well-defined, the language DML($\mathcal{L}_{\mathrm{dtd}}$) is also well-defined according to the DML language schema [16, 18]. In particular, type-checking in DML($\mathcal{L}_{\mathrm{dtd}}$) involves generating and then solving constraints of the form $\phi \vdash i_1 \equiv i_2$.

3 Representing XML Documents

In this section, we present a typeful and tagless representation for XML documents, which not only allows the type information of an XML document to be reflected in the type of the representation of the document but also makes it possible to significantly eliminate the need for representation tags in the representation of the document.

An XML document is to be represented as a pair $(\mathit{rep}, \mathit{dat})$ such that *rep* represents the structure of the document and *dat* represents the data in the

```
datatype ELEM (tag) = {'a,t:tag,d:doc}. ELEM (t) of 'a TAG (t,d) * 'a

and (type) REP (doc) =
  (unit) REPemp (empdoc)
| (string) REPstr (strdoc)
| {'a,t:tag,d:doc}. (ELEM(t)) REPelem (elemdoc(t))
| {'a,t:tag,d:doc}. ('a) REPelem' (elemdoc(t)) of 'a TAG (t, d)
| {'a1,'a2,d1:doc,d2:doc}.
    ('a1+'a2) REPalt (altdoc(d1,d2)) of 'a1 REP (d1) * 'a2 REP (d2)
| {'a1,'a2,d1:doc,d2:doc}.
    ('a1*'a2) REPseq (seqdoc(d1,d2)) of 'a1 REP (d1) * 'a2 REP (d2)
| {'a,d:doc}. ('a list) REPstar (stardoc(d)) of 'a REP(d)
| {'a,d:doc}. ('a * 'a list) REPplus (plusdoc(d)) of 'a REP(d)
| {'a,d:doc}. ('a option) REPopt (optdoc(d)) of 'a REP(d)

datatype XML (doc) = {'a, d:doc}. XML(d) of 'a REP (d) * 'a
```

Fig. 7. Datatypes for representing XML documents

document. This representation makes use of guarded recursive (g.r.) datatypes, which, syntactically, are like dependent datatypes though types may be used as type indexes. Please find more details about g.r. datatypes in [17]. In the following presentation, we also allow type variables as well as equalities between types to be declared in an index variable context ϕ.

$$
\begin{aligned}
\mathcal{T}(\underline{empdoc}) &= \mathbf{1} \\
\mathcal{T}(\underline{strdoc}) &= \mathit{string} \\
\mathcal{T}(\underline{elemdoc}(t)) &= \mathit{ELEM}(t) \\
\mathcal{T}(\underline{altdoc}(d_1, d_2)) &= \mathcal{T}(d_1) + \mathcal{T}(d_2) \\
\mathcal{T}(\underline{seqdoc}(d_1, d_2)) &= \mathcal{T}(d_1) * \mathcal{T}(d_2) \\
\mathcal{T}(\underline{stardoc}(d)) &= (\mathcal{T}(d))\,\mathit{list} \\
\mathcal{T}(\underline{plusdoc}(d)) &= \mathcal{T}(d) * (\mathcal{T}(d))\,\mathit{list} \\
\mathcal{T}(\underline{optdoc}(d)) &= (\mathcal{T}(d))\,\mathit{option}
\end{aligned}
\qquad
\begin{aligned}
\mathcal{R}(\underline{empdoc}) &= \mathit{REPemp} \\
\mathcal{R}(\underline{strdoc}) &= \mathit{REPstr} \\
\mathcal{R}(\underline{elemdoc}(t)) &= \mathit{REPelem} \\
\mathcal{R}(\underline{altdoc}(d_1, d_2)) &= \mathit{REPalt}(\mathcal{R}(d_1), \mathcal{R}(d_2)) \\
\mathcal{R}(\underline{seqdoc}(d_1, d_2)) &= \mathit{REPseq}(\mathcal{R}(d_1), \mathcal{R}(d_2)) \\
\mathcal{R}(\underline{stardoc}(d)) &= \mathit{REPstar}(\mathcal{R}(d)) \\
\mathcal{R}(\underline{plusdoc}(d)) &= \mathit{REPplus}(\mathcal{R}(d)) \\
\mathcal{R}(\underline{optdoc}(d)) &= \mathit{REPopt}(\mathcal{R}(d))
\end{aligned}
$$

Fig. 8. Two functions on doctypes

We use *TAG* for a type constructor that takes a type τ, a tag index t and a doctype index d to form a type $(\tau)\,\mathit{TAG}(t, d)$. Also, we introduce a language construct **ifEqTag**. Given expressions e_1, e_2, e_3, e_4, the concrete syntax for the expression **ifEqTag**(e_1, e_2, e_3, e_4) is given as follows,

$$\texttt{ifEqTag}\ (\ulcorner e_1 \urcorner, \ulcorner e_2 \urcorner)\ \texttt{then}\ \ulcorner e_3 \urcorner\ \texttt{else}\ \ulcorner e_4 \urcorner$$

```
(REPelem,
 ELEM (TAGaddrbook,
   [ ELEM (TAGperson,
      (ELEM (TAGname, "Dengping Zhu"),
            SOME (ELEM (TAGemail, "zhudp@cs.bu.edu"))))
     ELEM (TAGperson, (ELEM (TAGname, "Hongwei Xi"), NONE))]))
```

Fig. 9. An example of XML representation

```
(REPelem,
 ELEM (TAGfolder,
   (ELEM (TAGrecord, ()),
    [ (inr (ELEM (TAGfolder, (ELEM (TAGrecord, ()), []))))])))
```

Fig. 10. Another example of XML representation

where we assume that for each $1 \leq i \leq 4$, $\ulcorner e_i \urcorner$ is the representation of e_i in concrete syntax. Intuitively, in order to evaluate this expression, we first evaluate e_1 to a tag constant c_1 and then evaluate e_2 to another tag constant c_2; if c_1 equals c_2, we evaluate e_3; otherwise, we evaluate e_4. The rule for typing the construct **ifEqTag** is given as follows:

$$\frac{\begin{array}{c}\phi; \Gamma \vdash e_1 : (\tau_1)\,\mathit{TAG}(t_1, d_1) \quad \phi; \Gamma \vdash e_2 : (\tau_2)\,\mathit{TAG}(t_2, d_2) \\ \phi, \tau_1 \equiv \tau_2, t_1 \equiv t_2, d_1 \equiv d_2; \Gamma \vdash e_3 : \tau \quad \phi; \Gamma \vdash e_4 : \tau\end{array}}{\phi; \Gamma \vdash \mathbf{ifEqTag}(e_1, e_2, e_3, e_4) : \tau}$$

Note that we use $\phi; \Gamma \vdash e : \tau$ for a typing judgment, where Γ is a context for assigning types to free expression variables in e. The essential point in the above typing rules is simple: If e_1 and e_2 are of types $(\tau_1)\,\mathit{TAG}(t_1, d_1)$ and $(\tau_2)\,\mathit{TAG}(t_2, d_2)$, respectively, then we can assume $\tau_1 \equiv \tau_2$, $t_1 \equiv t_2$ and $d_1 \equiv d_2$ when typing e_3 (since e_1 and e_2 must evaluate to the same tag constant in order for e_3 to be evaluated).

As usual, we use value constructors *inl* and *inr* to form values of sum types, which are assigned the following types,

$$\begin{array}{c} \mathit{inl} : \forall\alpha\forall\beta.\alpha \rightarrow \alpha + \beta \\ \mathit{inr} : \forall\alpha\forall\beta.\beta \rightarrow \alpha + \beta \end{array}$$

and employ pattern matching to decompose values of sum types. We declare three datatype constructors *ELEM*, *REP* and *XML* in Figure 7. For instance, the syntax indicates that the value constructor *ELEM* associated with the type constructor *ELEM* is assigned the following type:

$$\Pi t : \mathit{tag}.\Pi d : \mathit{doc}.(\alpha)\,\mathit{TAG}(t, d) * \alpha \rightarrow \mathit{ELEM}(t)$$

and the value constructor *REPseq* associated with the type constructor *REP* is assigned the following type:

$$\Pi d_1 : \mathit{doc}.\Pi d_2 : \mathit{doc}.(\alpha_1)\mathit{REP}(d_1) * (\alpha_2)\mathit{REP}(d_2) \rightarrow (\alpha_1 * \alpha_2)\mathit{REP}(\underline{\mathit{seqdoc}}(d_1, d_2))$$

Intuitively, given a type τ, a tag index t and a doctype index d, $\mathit{ELEM}(t)$ is the type for representing a content of doctype index $\underline{\mathit{elemdoc}}(t)$, and $(\tau)\mathit{REP}(d)$ is the type for what we call a proof term that shows how a content of doctype index d can be represented as a value of type τ, and $\mathit{XML}(d)$ is the type for a value representing a content of doctype index d. We also assume the existence of a function *repOfTag* of type $\forall\alpha.(\alpha)\,\mathit{TAG}(t,d) \to (\alpha)\mathit{REP}(d)$. For instance, such a function can be readily constructed with the following approach. We define two functions $\mathcal{T}(\cdot)$ and $\mathcal{R}(\cdot)$ on doctypes in Figure 8 such that for each doctype d, $\mathcal{T}(d)$ is a type and $\mathcal{R}(d)$ is a value of type $(\mathcal{T}(d))\mathit{REP}(d)$. Note that we use $\mathbf{1}$ for the unit type. For each tag index t, we assign the type $(\mathit{ELEM}(t))\,\mathit{TAG}(t,\mathcal{S}(t))$ to the tag constant c corresponding to t, and then define $\mathit{repOfTag}(c)$ to be $\mathcal{R}(\mathcal{S}(t))$. As an example, the values of the function *repOfTag* on the tag constants *TAGaddrbook*, *TAGperson*, *TAGname* and *TAGemail* are given below,

$$\begin{aligned}
\mathit{repOfTag}(\mathit{TAGaddrbook}) &= \mathit{REPstar}(\mathit{REPelem})\\
\mathit{repOfTag}(\mathit{TAGperson}) &= \mathit{REPseq}(\mathit{REPelem}, \mathit{REPopt}(\mathit{REPelem}))\\
\mathit{repOfTag}(\mathit{TAGname}) &= \mathit{REPstr}\\
\mathit{repOfTag}(\mathit{TAGemail}) &= \mathit{REPstr}
\end{aligned}$$

and the XML document in Figure 1 is represented as a pair in Figure 9. Clearly, fewer representation tags are present at this time than before. As another example, the representation of the following XML document is given in Figure 10,

```
<folder>
  <record/><folder><record/></folder>
</folder>
```

which conforms to the DTD defined on page 5. Note that `<record/>` is a shorthand for `<record></record>`. The values of the function *repOfTag* on tag constants *TAGfolder* and *TAGrecord* are given as follows:

$$\begin{aligned}
\mathit{repOfTag}(\mathit{TAGfolder}) &= \mathit{REPseq}\,(\mathit{REPelem}, \mathit{REPstar}(\mathit{REPalt}(\mathit{REPelem}, \mathit{REPelem})))\\
\mathit{repOfTag}(\mathit{TAGrecord}) &= \mathit{REPemp}
\end{aligned}$$

We next show that more representation tags can be removed.

Given a tag index t, we say $t > t'$ holds if $\underline{\mathit{elemdoc}}(t')$ occurs in $\mathcal{S}(t)$. For instance, both $\mathit{addrbook} > \mathit{person}$ and $\mathit{person} > \mathit{name}$ hold, and $\mathit{folder} > \mathit{folder}$ holds as well. Let $>^*$ be the transitive closure of $>$ and $\not>^*$ be the complement of $>^*$. For instance, we have $\mathit{addrbook} >^* \mathit{name}$ and $\mathit{person} \not>^* \mathit{person}$. We say that a tag index t is recursive if $t >^* t$ holds. We now define two functions $\mathcal{T}'(\cdot)$ and $\mathcal{R}'(\cdot)$ in Figure 11, where $c(t)$ stands for the tag constant corresponding to tag index t. As an example, we have

$$\begin{aligned}
\mathcal{T}'(\mathit{addrbook}) &= (\mathit{string} * (\mathit{string})\mathit{option})\mathit{list}\\
\mathcal{R}'(\mathit{addrbook}) &= \mathit{REPelem}'(\mathit{TAGaddrbook})
\end{aligned}$$

For each tag index t, we assign the type $(\mathcal{T}'(t))\,\mathit{TAG}(t,\mathcal{S}(t))$ to $c(t)$, which stands for the tag constant c corresponding to t, and we define $\mathit{repOfTag}(c(t)) = \mathcal{R}'(\mathcal{S}(t))$. For instance, we now have the following:

$$\begin{aligned}
\mathcal{T}'(\underline{empdoc}) &= \mathbf{1} \\
\mathcal{T}'(\underline{strdoc}) &= string \\
\mathcal{T}'(\underline{elemdoc}(t)) &= ELEM(t) \\
\mathcal{T}'(\underline{elemdoc}(t)) &= \mathcal{T}'(\mathcal{S}(t)) \\
\mathcal{T}'(\underline{altdoc}(d_1,d_2)) &= \mathcal{T}'(d_1) + \mathcal{T}'(d_2) \\
\mathcal{T}'(\underline{seqdoc}(d_1,d_2)) &= \mathcal{T}'(d_1) * \mathcal{T}'(d_2) \\
\mathcal{T}'(\underline{stardoc}(d)) &= (\mathcal{T}'(d))list \\
\mathcal{T}'(\underline{plusdoc}(d)) &= \mathcal{T}'(d) * (\mathcal{T}'(d))list \\
\mathcal{T}'(\underline{optdoc}(d)) &= (\mathcal{T}'(d))option
\end{aligned}$$

$$\begin{aligned}
\mathcal{R}'(\underline{empdoc}) &= REPemp \\
\mathcal{R}'(\underline{strdoc}) &= REPstr \\
\mathcal{R}'(\underline{elemdoc}(t)) &= REPelem \ \text{ if } t >^* t \\
\mathcal{R}'(\underline{elemdoc}(t)) &= REPelem'(c(t)) \ \text{ if } t \not>^* t \\
\mathcal{R}'(\underline{altdoc}(d_1,d_2)) &= REPalt(\mathcal{R}'(d_1), \mathcal{R}'(d_2)) \\
\mathcal{R}'(\underline{seqdoc}(d_1,d_2)) &= REPseq(\mathcal{R}'(d_1), \mathcal{R}'(d_2)) \\
\mathcal{R}'(\underline{stardoc}(d)) &= REPstar(\mathcal{R}'(d)) \\
\mathcal{R}'(\underline{plusdoc}(d)) &= REPplus(\mathcal{R}'(d)) \\
\mathcal{R}'(\underline{optdoc}(d)) &= REPopt(\mathcal{R}'(d))
\end{aligned}$$

Fig. 11. Another two functions on doctypes

$$\begin{aligned}
repOfTag(TAGaddrbook) &= REPstar(REPelem'(TAGperson)) \\
repOfTag(TAGperson) &= REPseq(REPelem'(TAGname), REPopt(REPelem'(TAGemail))) \\
repOfTag(TAGname) &= REPstr \\
repOfTag(TAGemail) &= REPstr \\
repOfTag(TAGfolder) &= \\
&\quad REPseq\,(REPelem'(TAGrecord), REPstar(REPalt(REPemp, REPelem))) \\
repOfTag(TAGrecord) &= REPemp
\end{aligned}$$

The two XML documents represented in Figure 9 and Figure 10 can now be represented as follows,

```
(* the first example *)
(REPelem' (TAGaddrbook),
 [ ("Dengping Zhu", SOME "zhudp@cs.bu.edu"), ("Hongwei Xi", NONE)])

(* the second example *)
(REPelem,
 ELEM (TAGfolder, ((), [inr (ELEM (TAGfolder, ((), []))]))))
```

where far fewer representation tags are involved.

With this approach, an XML document is represented as a pair (rep, dat), and *there is no need for representation tags in dat if the XML document conforming to a DTD that involves no recursion* (i.e., there is no recursive tag indexes in the doctype corresponding to the DTD).

When processing XML documents, we often encounter situations where we need to select certain parts out of a given XML content according to some criterion. In Figure 12, we implement a function *select* in DML($\mathcal{L}_{\text{dtd}}$) for selecting out of a given XML content the first element satisfying some chosen criterion. Note that the type assigned to *select* is formally written as follows,

$$\begin{aligned}
&\Pi d_1 : doc.\Pi t : tag.\Pi d_2 : doc. \\
&\quad XML(d_1) \to (\alpha)\,TAG(t, d_2) \to (ELEM(t) \to bool) \to (ELEM(t))option
\end{aligned}$$

```
fun select xml tag pred =
  let
      fun aux REPelem =
          (fn dat => let
              val ELEM (tag', dat') = dat
           in
              ifEqTag (tag, tag') then
                if pred dat then SOME dat else aux (repOfTag tag') dat'
              else aux (repOfTag tag') dat'
           end)
        | aux (REPelem tag') = let
             val f = aux (repOfTag tag')
          in
             fn dat =>
               ifEqTag (tag, tag')
                 if pred (ELEM (tag, dat)) then SOME (ELEM (tag, dat) else f dat
               else f dat
          end
        | aux (REPalt (pf1, pf2)) = let
             val f1 = aux pf1 and f2 = aux pf2
          in fn inl (dat) => f1 dat | inr (dat) => f2 dat end
        | aux (REPseq (pf1, pf2)) = let
             val f1 = aux pf1 and f2 = aux pf2
          in
             fn (dat1, dat2) =>
               case f1 dat1 of NONE => f2 dat2 | res as SOME _ => res
          end
        | aux (REPstar pf) = auxList pf
        | aux (REPplus pf) = let
             val f = auxList pf
          in fn (dat, dats) => f (dat :: dats) end
        | aux (REPopt pf) = let
             val f = aux pf
          in fn NONE => NONE | SOME (dat) => f dat end
      withtype {'a,d:doc}. 'a REP(d) -> 'a -> (ELEM(t)) option

      and auxList pf = let
         val f = aux pf
         fun fList [] = NONE
           | fList (dat :: dats) =
             (case f dat of NONE => fList dats | res as SOME _ => res)
         withtype 'a list -> (ELEM(t)) option
      in fList end
      withtype {'a,d:doc}. 'a REP(d) -> 'a list -> (ELEM(t)) option
  in
      let val XML(pf, dat) = xml in aux pf dat end
  end
withtype
  {'a,t:tag,d1:doc,d2:doc}.
    XML(d1) -> 'a TAG (t,d2) -> (ELEM(t) -> bool) -> (ELEM(t)) option
```

Fig. 12. Another selection function on XML elements

which clearly implies that *select* can be applied to any pair of XML document and tag constant. Note that the auxiliary function *aux* in the implementation of *select* takes a value of type $(\tau)REP(d)$ to produce a specialized function that can only be applied to a value of type τ representing some content of doctype index d. This means that we have an opportunity to stage the function *aux* so that it can generate efficient specialized code (at run-time) to search XML contents.

4 Measurements

We report some measurements gathered from an experiment processing an XML document *mybib* containing about 800 entries in bibtex style. The main purpose of the experiment is to provide a proof of concept: The proposed typeful and tagless representation for XML documents can indeed be used to implement functions processing XML documents. However, the software engineering benefits from using a typeful representation for XML documents in implementing such functions are difficult to measure directly.

When the generic representation in Section 1 is used to represent *mybib*, we notice that about 15% of space is spent on storing the representation tags. On the other hand, when the typeful and tagless representation in Section 3 is used, only less than 0.5% of the space is spent on representing the structure of the document and there is *no* need for the representation tags required in the generic representation (as *mybib* conforms to a DTD involving no recursion)

When processing an XML document, we often want to collect all elements enclosed between tags <*tagname*> and </*tagname*> for some *tagname*. For instance, we may want to collect all the titles (of books, articles, technical reports, etc.) in *mybib*. For this purpose, we need a function *selectAll* similar to the function *select* defined previously. Instead of selecting the first of the elements satisfying a chosen criterion, the function *selectAll* selects all of them. We have implemented three versions of *selectAll*.

1. The first one is a select function based on the generic representation presented in Figure 2, and its implementation is similar to *select* in Figure 12.
2. The second one is a select function based on the the typeful and tagless representation in Section 3, and its implementation is similar to *select* in Figure 12.
3. The third one is a staged version of the second one in which we make use of run-time code generation.

The following table lists the various measurements on times spent on collecting all the title elements in *mybib*. The time unit is second.

No.	Sys. time	GC time	Real time
1	0.41	0.02	0.43
2	0.10	0.00	0.10
3	0.05	0.00	0.05

We have also measured times spent on collecting other elements and the measurements are similar in nature. Note that the second and the third implementations of *selectAll* are approximately 4 and 8 times faster than the first one, respectively. This is not surprising as these implementations can tell whether an element is potentially to occur in a content by only inspecting the structure of the content while the first implementation needs to inspect the entire content. This intuition is verified by the following expriment in which the structure of the searched XML document *mybib* is made available to the function *selectAll* while the generic representation is used to represent *mybib*. This time, the measurements are similar to those gathered using the second implementation above.

No.	Sys. time	GC time	Real time
$1'$	0.12	0.01	0.13

Clearly, the measurements imply that the most significant factor here is whether the structure of an XML content can be made available to a search function on the content. Regardless whether a representation for XML is typeful or typeless, we think that some information on the structure of an XML document should be made available in the representation of the document so as to facilitate efficient search on the document. Also, the measurements indicate that run-time code generation can be employed to significantly speed search on XML documents.

5 Related Work and Conclusion

There have been various attempts to construct functional programs for processing XML documents. In [12], two approaches are presented for processing XML documents in Haskell [9]. The first approach uses a generic representation for XML documents that makes no use of any type information of these documents, making it impossible to use types to differentiate XML documents conforming to distinct DTDs. With this approach, generic combinators can be implemented to facilitate the construction of programs for processing XML documents. The second approach represents each XML document as a value of a special datatype (automatically) generated according to the DTD to which the XML document conforms. With this approach, XML documents conforming to distinct DTDs can be differentiated with type, but it becomes difficult or even impossible to implement programs for processing XML documents that are polymorphic on DTDs. An approach to addressing the issue is through generic programming as is proposed in Generic Haskell [2], and an XML compressor implemented in Generic Haskell is given a detailed description in [1].

In XDuce [3], a functional language is proposed where types based on regular expressions can be formed to represent DTDs. The development of the type index langugage $\mathcal{L}_{\text{dtd}}$ in Section 2 bears some resemblance to the regular expression types. However, it is currently unclear how the type system in XDuce can be extended to effectively handle polymorphism (beyond what subtyping is able to achieve). Another typed functional language XMλ is presented in [5] to facilitate

XML document processing. In XMλ, DTDs can be encoded as types and row polymorphism is used to support generic XML processing functions. We are currently working along a different research line: Instead of designing a new language to process XML documents, we are primarily interested in a typeful embedding of XML in a general purpose language (e.g., DML extended with guarded recursive datatypes in this case). A typeful representation for XML documents was proposed in [11]. There, an XML document can be represented as a program in a family of domain specific languages guaranteeing that the generated documents are well-formed and valid (to a certain extent), and each language in the family of languages is implemented as a combinator library. However, this approach seems, at least currently if not inherently, incapable of handling destruction of XML documents (in a typeful manner), which is of the concern when transformation on XML documents needs to be implemented.

There are also a number of proposed typechecking algorithms for various XML processing languages in the database community. For instance, K-pebble tree transducer [7] is a general framework for XML transformers, where the type system is based on tree automaton. These works are largely of a different nature from ours and are usually not concerned with the actual implementation of XML processing functions in a given programming language.

In this paper, we have presented an approach to representing XML documents that not only allows the type information of an XML document to be reflected in the type of the representation of the document and but also obviates the need for representation tags that are otherwise required in a typeless representation. With this approach, we become able to process XML documents in a typeful manner, thus reaping various well-known software engineering benefits from the presence of types. This work also yields evidence in support of the use of DML-style dependent types in practical programming. In future, we seek to extend our approach to capture more type information (e.g. based on XML schema) of XML documents. Also, we plan to use the presented typeful and tagless XML representation to implement languages such as XQuery and XSLT.

6 Acknowledgments

We thank Chiyan Chen, Sa Cui, Allyn Dimock, Joe Hallett, Assaf Kfoury, Rui Shi and Jin Ye for providing us with their valuable comments on the paper.

References

1. F. Atanassow, D. Clarke, and J. Jeuring. Scripting XML with Generic Haskell. In *Proceedings of Simposio Brasileiro de Linguagens de Programacao (SBLP '03)*, Ouro Preto, Brazil, May 2003.
2. D. Clarke, R. Hinze, J. Jeuring, A. Löh, L. Meertens, and D. Swierstra. Generic Haskell. Available at: `http://www.generic-haskell.org/`.
3. H. Hosoya and B. C. Pierce. "XDuce: A Typed XML Processing Language". In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

4. J. Jeuring and P. Hagg. Generic Programming for XML Tools. Technical Report UU-CS-2002-023, Utrecht University, 2002.
5. E. Meijer and M. Shields. XMLambda: A functional language for constructing and manipulating XML documents, 1999.
6. R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML (Revised).* MIT Press, Cambridge, Massachusetts, 1997.
7. T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 11–22. ACM, 2000.
8. OASIS Technical Committee. RELAX NG Specification, December 2001. Available at: http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.
9. S. Peyton Jones et al. Haskell 98 – A non-strict, purely functional language. Available at: `http://www.haskell.org/onlinereport/`, Feb. 1999.
10. J. Siméon and P. Wadler. The Essence of XML. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 1–13, New Orleans, January 2003.
11. P. Thiemann. A typed representation for HTML and XML documents in Haskell. *Journal of functional programming*, 12((4&5)):435–468, 2002.
12. M. Wallace and C. Runciman. Haskell and XML: Generic Combinators or Type-Based Translation? In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming (ICFP'99)*, volume 34–9, pages 148–159, N.Y., 27–29 1999. ACM Press.
13. World Wide Web Consortium. Extensible Markup Language (XML). Version 1.0 (Second Edition). W3C Recommendation 6 October 2002. http://www.w3.org/TR/REC-xml.
14. World Wide Web Consortium. XML Schema Part 1: Structures. W3C Recommendation 2 May 2001. Available at `http://www.w3.org/TR/xmlschema-1`.
15. World Wide Web Consortium. XML Schema Part 2: Datatypes. W3C Recommendation 2 May 2001. Available at `http://www.w3.org/TR/xmlschema-2`.
16. H. Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998. pp. viii+189. Available as `http://www.cs.cmu.edu/~hwxi/DML/thesis.ps`.
17. H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 224–235, New Orleans, January 2003.
18. H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 214–227, San Antonio, Texas, January 1999.