Type-based XML processing in Logic Programming

Jorge Coelho and Mário Florido

LIACC, University of Porto

14 January, 2003

Motivation

- XML: Standard for information exchange.

Motivation

- XML: Standard for information exchange.
- XSLT, DOM and SAX.

Motivation

- XML: Standard for information exchange.
- XSLT, DOM and SAX.
- Static validation.

Motivation

- XML: Standard for information exchange.
- XSLT, DOM and SAX.
- Static validation.
- XDuce and HaXml.

Main Goal

Use of Logic Programming with static validation for XML processing.

Implementation

1. Translator from XML to Prolog.

Implementation

1. Translator from XML to Prolog.

2. Translator from DTDs to Regular Types.

Implementation

1. Translator from XML to Prolog.

2. Translator from DTDs to Regular Types.

3. Type inference.

Implementation

1. Translator from XML to Prolog.

2. Translator from DTDs to Regular Types.

3. Type inference.

4. Translator from Prolog to XML.

Outline

- XML

Outline

- XML
- DTD

Outline

- XML
- DTD
- Translation from XML to Prolog

Outline

- XML
- DTD
- Translation from XML to Prolog
- Regular Types

Outline

- XML
- DTD
- Translation from XML to Prolog
- Regular Types
- Type inference (Zobel 1990)

Outline

- XML
- DTD
- Translation from XML to Prolog
- Regular Types
- Type inference (Zobel 1990)
- Translation from DTDs to Regular Types

Outline

- XML
- DTD
- Translation from XML to Prolog
- Regular Types
- Type inference (Zobel 1990)
- Translation from DTDs to Regular Types
- Type inference for XML processing programs

Outline

- XML
- DTD
- Translation from XML to Prolog
- Regular Types
- Type inference (Zobel 1990)
- Translation from DTDs to Regular Types
- Type inference for XML processing programs
- Conclusions

Outline

- XML
- DTD
- Translation from XML to Prolog
- Regular Types
- Type inference (Zobel 1990)
- Translation from DTDs to Regular Types
- Type inference for XML processing programs
- Conclusions
- Future Work

XML - eXtensible Markup Language

XML - eXtensible Markup Language

Example:

```
<addressbook>
   <name>Jorge</name>
   <address>Porto</address>
   <email>jorge@mailserver.pt</email>
   <name>Mario</name>
   <address>Lisboa</address>
   <address>Portugal</address>
   <phone>
      <home>12457834</home>
   </phone>
</addressbook>
```

Document Type Definition

Document Type Definition

DTDs are grammars that specify the document structure.

Document Type Definition

DTDs are grammars that specify the document structure.

Example:

```
<!ELEMENT addressbook (name,address+,phone?,email?)*>
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT phone (home,mobile*)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT home (#PCDATA)>
<!ELEMENT mobile (#PCDATA)>
```

Translation from XML to Prolog

Translation from XML to Prolog

Element with only character data

```
<!ELEMENT b (#PCDATA)>

<b>Text of element b</b>

b("Text of element b")
```

Translation from XML to Prolog

Element with only character data

```
<!ELEMENT b (#PCDATA)>

<b>Text of element b</b>

b("Text of element b")
```

Empty element

```
<!ELEMENT b EMPTY>

<b/>

b
```

Translation from XML to Prolog

Element with sub elements

```
<!ELEMENT a (b,c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

```
<a>
  <b> Text for element b </b>
  <c> Text for element c </c>
</a>
```

```
a(
  b(" Text for element b "),
  c(" Text for element c "))
```

Translation from XML to Prolog

Element with zero or more occurrences

```
<!ELEMENT a (b)*>
<!ELEMENT b (#PCDATA)>

<a>
 <b> First b </b>
 <b> Second b </b>
 <b> Third b </b>
</a>
```

```
a(
  [b(" First b "),
   b(" Second b "),
   b(" Third b ")])
```

Translation from XML to Prolog

Element with one or more occurrences

```
<!ELEMENT a (b+,c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>

<a>
   <b> Text for b </b>
   <c> Text for c </c>
</a>

a(
  [b(" Text for b ")],
   c(" Text for c "))
```

Translation from XML to Prolog

Optional element

```
<!ELEMENT a (b?,c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

```
<a>
   <c> Text for c </c>
</a>
```

```
a(
  c(" Text for c "))
```

```
<a>
  <b> Text for b </b>
  <c> Text for c </c>
</a>
```

```
a(
  b(" Text for b "),
  c(" Text for c "))
```

Translation from XML to Prolog

Disjoint elements

```
<!ELEMENT a (b|c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>

<a> <b> Text </b> </a>

<a> <c> Another text </c> </a>

a(b(" Text "))

a(c(" Another text "))
```

Translation guided by DTDs

Using two distinct DTDs to validate the same document can lead to different (valid) terms:

Translation guided by DTDs

Using two distinct DTDs to validate the same document can lead to different (valid) terms:

```
<a>
  <b> First b </b>
  <b> Second b </b>
</a>
```

```
   <!ELEMENT a (b,b)>
   <!ELEMENT b (#PCDATA)>
a(
  b(" First b "),
  b(" Second b "))
```

```
   <!ELEMENT a b*>
   <!ELEMENT b (#PCDATA)>
a(
  [b(" First b "),
   b(" Second b ")])
```

Regular Types

Regular Types

Regular types are defined as the class of types that can be specified by sets of type rules.

Regular Types

Regular types are defined as the class of types that can be specified by sets of type rules.

Type symbol

Regular Types

Regular types are defined as the class of types that can be specified by sets of type rules.

Type symbol $\longrightarrow$

Regular Types

Regular types are defined as the class of types that can be specified by sets of type rules.

$$\text{Type symbol} \longrightarrow \{\text{Types that describe terms}\}$$

Regular Types

Regular types are defined as the class of types that can be specified by sets of type rules.

$$\underbrace{\text{Type symbol} \longrightarrow \{\text{Types that describe terms}\}}_{\text{Type Rule}}$$

Regular Types

Regular types are defined as the class of types that can be specified by sets of type rules.

$$\underbrace{\text{Type symbol} \longrightarrow \{\text{Types that describe terms}\}}_{\text{Type Rule}}$$

Given the following rules:

- $\alpha \longrightarrow \{a\}$
- $\beta \longrightarrow \{nil, .(\alpha, \beta)\}$

Regular Types

Regular types are defined as the class of types that can be specified by sets of type rules.

$$\underbrace{\text{Type symbol} \longrightarrow \{\text{Types that describe terms}\}}_{\text{Type Rule}}$$

Given the following rules:

- $\alpha \longrightarrow \{a\}$
- $\beta \longrightarrow \{nil, .(\alpha, \beta)\}$

Regular Types produced by α:

$$\{a\}$$

Regular Types produced by α:

$$\{a\}$$

Regular Types produced by β:

$$\{nil, .(a, nil), .(a, a, nil), \ldots\}$$

Type inference (Zobel 1990)

We built a type inference system that uses *Regular Types* as an approximation to program types. For example, given the next program:

```
p(0).
p(f(X)):-q(X),X=f(Y).
q(f(0)).
q(g(X)).
q(f(X)):-p(X).
```

Type inference (Zobel 1990)

We built a type inference system that uses *Regular Types* as an approximation to program types. For example, given the next program:

```
p(0).
p(f(X)):-q(X),X=f(Y).
q(f(0)).
q(g(X)).
q(f(X)):-p(X).
```

The system reaches the following types:

- $\alpha_p \longrightarrow \{0, f(f(\alpha_1))\}$
- $\alpha_q \longrightarrow \{g(\mu), f(\alpha_1)\}$
- $\alpha_1 \longrightarrow \{0, \alpha_p\}$

Translating from DTDs to Regular Types

Translating from DTDs to Regular Types

Element with only character data

$$\mathcal{T}(\text{<!ELEMENT } e \text{ (\#PCDATA) >}) \quad = \quad \tau_e \rightarrow \{e(string)\}$$

Translating from DTDs to Regular Types

Element with only character data

$$\mathcal{T}(\text{<!ELEMENT } e \text{ (\#PCDATA) >}) \quad = \quad \tau_e \rightarrow \{e(string)\}$$

```
<!ELEMENT a (#PCDATA)>
```

Translating from DTDs to Regular Types

Element with only character data

$$\mathcal{T}(<!\text{ELEMENT } e \text{ (\#PCDATA)} >) \quad = \quad \tau_e \longrightarrow \{e(string)\}$$

```
<!ELEMENT a (#PCDATA)>
```

$$\tau \longrightarrow \{a(string)\}$$

Translating from DTDs to Regular Types

Empty element

$$\mathcal{T}(\texttt{<!ELEMENT } e \texttt{ EMPTY >}) \quad = \quad \tau_e \rightarrow \{e\}$$

Translating from DTDs to Regular Types

Empty element

$$\mathcal{T}(<!\text{ELEMENT } e \text{ EMPTY} >) \quad = \quad \tau_e \rightarrow \{e\}$$

```
<!ELEMENT a EMPTY>
```

Translating from DTDs to Regular Types

Empty element

$$\mathcal{T}(\text{<!ELEMENT } e \text{ EMPTY >}) \quad = \quad \tau_e \longrightarrow \{e\}$$

```
<!ELEMENT a EMPTY>
```

$$\tau \longrightarrow \{a\}$$

Translating from DTDs to Regular Types

Element with any contents

$$\mathcal{T}(\text{<!ELEMENT } e \text{ ANY >}) \quad = \quad \tau_e \to \{e(\mu)\}$$

Translating from DTDs to Regular Types

Element with any contents

$$\mathcal{T}(<!\mathsf{ELEMENT}\ e\ \mathsf{ANY}>) \quad = \quad \tau_e \rightarrow \{e(\mu)\}$$

```
<!ELEMENT a ANY>
```

Translating from DTDs to Regular Types

Element with any contents

$$\mathcal{T}(<!\text{ELEMENT}\ e\ \text{ANY}\ >) \quad = \quad \tau_e \longrightarrow \{e(\mu)\}$$

```
<!ELEMENT a ANY>
```

$$\tau \longrightarrow \{a(\mu)\}$$

Translating from DTDs to Regular Types

Element with sub elements

$$\mathcal{T}(\text{<!ELEMENT } e\ (e_1, \ldots, e_n) >\ = \ \tau_e \rightarrow \{e(\tau_{e_1}, \ldots, \tau_{e_n})\},\ \text{where}$$

$$\mathcal{T}(\text{<!ELEMENT } e_i >) \ = \ \tau_{e_i} \rightarrow \Upsilon_{e_i},$$

for $1 \leq i \leq n$

Translating from DTDs to Regular Types

Element with sub elements

$$\mathcal{T}(\text{<!ELEMENT } e\ (e_1, \ldots, e_n) >\ =\ \tau_e \rightarrow \{e(\tau_{e_1}, \ldots, \tau_{e_n})\},\ \text{where}$$

$$\mathcal{T}(\text{<!ELEMENT } e_i >)\ =\ \tau_{e_i} \rightarrow \Upsilon_{e_i},$$

for $1 \leq i \leq n$

```
<!ELEMENT a (b,c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

Translating from DTDs to Regular Types

Element with sub elements

$$\mathcal{T}(\text{<!ELEMENT } e\ (e_1, \ldots, e_n) >\quad = \quad \tau_e \rightarrow \{e(\tau_{e_1}, \ldots, \tau_{e_n})\},$$ where
$$\mathcal{T}(\text{<!ELEMENT } e_i >) \quad = \quad \tau_{e_i} \rightarrow \Upsilon_{e_i},$$
for $1 \leq i \leq n$

```
<!ELEMENT a (b,c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

$$\begin{aligned} \tau_1 &\rightarrow \{a(\tau_2, \tau_3)\} \\ \tau_2 &\rightarrow \{b(string)\} \\ \tau_3 &\rightarrow \{c(string)\} \end{aligned}$$

Translating from DTDs to Regular Types

Element with zero or more occurrences

$$\mathcal{T}(<!\text{ELEMENT } e\ e_1* >) \quad = \quad \tau_e \rightarrow \{nil, .(\tau_{e_1}, \tau_e)\}, \text{where}$$

$$\mathcal{T}(<!\text{ELEMENT } e_1 >) \quad = \quad \tau_{e_1} \rightarrow \Upsilon_{e_1}$$

Translating from DTDs to Regular Types

Element with zero or more occurrences

$$\mathcal{T}(<!\text{ELEMENT } e \; e_1* >) \quad = \quad \tau_e \rightarrow \{nil, .(\tau_{e_1}, \tau_e)\}, \text{where}$$
$$\mathcal{T}(<!\text{ELEMENT } e_1 >) \quad = \quad \tau_{e_1} \rightarrow \Upsilon_{e_1}$$

```
<!ELEMENT a b*>
<!ELEMENT b (#PCDATA)>
```

Translating from DTDs to Regular Types

Element with zero or more occurrences

$$\mathcal{T}(<!\text{ELEMENT } e\ e_1* >) \quad = \quad \tau_e \rightarrow \{nil, .(\tau_{e_1}, \tau_e)\}, \text{where}$$

$$\mathcal{T}(<!\text{ELEMENT } e_1 >) \quad = \quad \tau_{e_1} \rightarrow \Upsilon_{e_1}$$

```
<!ELEMENT a b*>
<!ELEMENT b (#PCDATA)>
```

$$\begin{aligned} \tau_1 &\rightarrow \{a(\tau_2)\} \\ \tau_2 &\rightarrow \{nil, .(\tau_3, \tau_2)\} \\ \tau_3 &\rightarrow \{b(string)\} \end{aligned}$$

Translating from DTDs to Regular Types

Element with one or more occurrences

$$\mathcal{T}(<!\mathsf{ELEMENT}\ e\ e_1+\ >) \;=\; \tau_e \longrightarrow \{.(\tau_{e_1}, nil), .(\tau_{e_1}, \tau_e)\}, \text{ where}$$

$$\mathcal{T}(<!\mathsf{ELEMENT}\ e_1\ >) \;=\; \tau_{e_1} \longrightarrow \Upsilon_{e_1}$$

Translating from DTDs to Regular Types

Element with one or more occurrences

$$\mathcal{T}(\text{<!ELEMENT } e\ e_1+ \text{ >}) \quad = \quad \tau_e \longrightarrow \{.(\tau_{e_1}, nil), .(\tau_{e_1}, \tau_e)\}, \text{ where}$$
$$\mathcal{T}(\text{<!ELEMENT } e_1 \text{ >}) \quad = \quad \tau_{e_1} \longrightarrow \Upsilon_{e_1}$$

```
<!ELEMENT a b+>
<!ELEMENT b (#PCDATA)>
```

Translating from DTDs to Regular Types

Element with one or more occurrences

$$\mathcal{T}(\text{<!ELEMENT } e\ e_1+ \text{>}) \quad = \quad \tau_e \longrightarrow \{.(\tau_{e_1}, nil), .(\tau_{e_1}, \tau_e)\}, \text{where}$$

$$\mathcal{T}(\text{<!ELEMENT } e_1 \text{>}) \quad = \quad \tau_{e_1} \longrightarrow \Upsilon_{e_1}$$

```
<!ELEMENT a b+>
<!ELEMENT b (#PCDATA)>
```

$$\tau_1 \quad \longrightarrow \quad \{a(\tau_2)\}$$

$$\tau_2 \quad \longrightarrow \quad \{.(\tau_3, nil), .(\tau_3, \tau_2)\}$$

$$\tau_3 \quad \longrightarrow \quad \{b(string)\}$$

Translating from DTDs to Regular Types

Disjoint elements

$$\mathcal{T}(<!\text{ELEMENT}\ e\ (e_1|\cdots|e_n)\ >) \quad = \quad \tau_e \longrightarrow \{\tau_{e_1},\ldots,\tau_{e_n}\}, \text{where}$$

$$\mathcal{T}(<!\text{ELEMENT}\ e_i\ >) \quad = \quad \tau_{e_i} \longrightarrow \Upsilon_{e_i},$$

for $1 \leq i \leq n$

```
<!ELEMENT a (b|c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

```
<!ELEMENT a (b|c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

$$\begin{array}{lcl} \tau_1 & \longrightarrow & \{a(\tau_2)\} \\ \tau_2 & \longrightarrow & \{\tau_3, \tau_4\} \\ \tau_3 & \longrightarrow & \{b(string)\} \\ \tau_4 & \longrightarrow & \{c(string)\} \end{array}$$

Translating from DTDs to Regular Types

Optional element

$$\mathcal{T}(\text{<!ELEMENT } e\ (e_1, \ldots, e_i?, \ldots, e_n)) = \tau_e \longrightarrow \{e(\tau_{e_1}, \ldots, \tau_{e_{i-1}}, \tau_{e_{i+1}}, \ldots, \tau_{e_n}), e(\tau_{e_1}, \ldots, \tau_{e_{i-1}}, \tau_{e_i}, \tau_{e_{i+1}}, \ldots, \tau_{e_n})\},$$

where

$$\mathcal{T}(\text{<!ELEMENT } e_i >) = \tau_{e_i} \longrightarrow \Upsilon_{e_i},$$

for $1 \leq i \leq n$

Translating from DTDs to Regular Types

Optional element

```
<!ELEMENT a (b,c?,d?)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
<!ELEMENT d (#PCDATA)>
```

Translating from DTDs to Regular Types

Optional element

```
<!ELEMENT a (b,c?,d?)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
<!ELEMENT d (#PCDATA)>
```

$$
\begin{aligned}
\tau_1 &\longrightarrow \{a(\tau_2), a(\tau_2, \tau_3), \\
&\qquad a(\tau_2, \tau_4), a(\tau_2, \tau_3, \tau_4)\} \\
\tau_2 &\longrightarrow \{b(string)\} \\
\tau_3 &\longrightarrow \{c(string)\} \\
\tau_4 &\longrightarrow \{d(string)\}
\end{aligned}
$$

Type inference for XML processing programs

- DTDs used as type declarations.

Type inference for XML processing programs

- DTDs used as type declarations.
- Use of standard type checking for validation.

Type inference for XML processing programs

$$p(a(X,Y),d(X,Y)).$$

Input DTD:

```
<!ELEMENT a (b,c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

Type inference for XML processing programs

$$p(a(X,Y), d(X,Y)).$$

Input DTD:

```
<!ELEMENT a (b,c)>
<!ELEMENT b (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

Regular types:

$$\begin{array}{lcl} \tau_a & \longrightarrow & \{a(\tau_b, \tau_c)\} \\ \tau_b & \longrightarrow & \{b(string)\} \\ \tau_c & \longrightarrow & \{c(string)\} \end{array}$$

Type inference for XML processing programs

$$p(a(X,Y), d(X,Y)).$$

Output DTD:

```
<!ELEMENT d (e,c)>
<!ELEMENT e (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

Type inference for XML processing programs

$$p(a(X,Y),d(X,Y)).$$

Output DTD:

```
<!ELEMENT d (e,c)>
<!ELEMENT e (#PCDATA)>
<!ELEMENT c (#PCDATA)>
```

Regular types:

$$\begin{array}{lcl} \tau_d & \longrightarrow & \{d(\tau_e,\tau_c)\} \\ \tau_e & \longrightarrow & \{e(string)\} \\ \tau_c & \longrightarrow & \{c(string)\} \end{array}$$

Type inference for XML processing programs

$$p(a(X,Y),d(X,Y)) ::< \tau_a, \tau_d >$$

$$\begin{aligned}
\tau_a &\rightarrow \{a(\tau_b, \tau_c)\} \\
\tau_b &\rightarrow \{b(string)\} \\
\tau_c &\rightarrow \{c(string)\}
\end{aligned}$$

Type inference for XML processing programs

$$p(a(X,Y), d(X,Y)) ::< \tau_a, \tau_d >$$

$$\begin{aligned} \tau_a &\rightarrow \{a(\tau_b, \tau_c)\} \\ \tau_b &\rightarrow \{b(string)\} \\ \tau_c &\rightarrow \{c(string)\} \end{aligned}$$

$$\begin{aligned} \tau_d &\rightarrow \{d(\tau_e, \tau_c)\} \\ \tau_e &\rightarrow \{e(string)\} \\ \tau_c &\rightarrow \{c(string)\} \end{aligned}$$

Type inference for XML processing programs

$$p(a(X, Y), d(X, Y)) ::< \tau_a, \tau_d >$$

$$\begin{array}{lll} \tau_a & \rightarrow & \{a(\tau_b, \tau_c)\} \\ \tau_b & \rightarrow & \{b(string)\} \\ \tau_c & \rightarrow & \{c(string)\} \end{array}$$

$$\begin{array}{lll} \tau_d & \rightarrow & \{d(\tau_e, \tau_c)\} \\ \tau_e & \rightarrow & \{e(string)\} \\ \tau_c & \rightarrow & \{c(string)\} \end{array}$$

Type inference for XML processing programs

$$p(a(X, Y), d(X, Y)) ::< \tau_a, \tau_d >$$

$$\begin{aligned} \tau_a &\longrightarrow \{a(\tau_b, \tau_c)\} \\ \tau_b &\longrightarrow \{b(string)\} \\ \tau_c &\longrightarrow \{c(string)\} \end{aligned}$$

$$\begin{aligned} \tau_d &\longrightarrow \{d(\tau_e, \tau_c)\} \\ \tau_e &\longrightarrow \{e(string)\} \\ \tau_c &\longrightarrow \{c(string)\} \end{aligned}$$

Type inference for XML processing programs

$$p(a(X, Y), d(X, Y)) ::< \tau_a, \tau_d >$$

$$\begin{aligned} \tau_a &\longrightarrow \{a(\tau_b, \tau_c)\} \\ \tau_b &\longrightarrow \{b(string)\} \\ \tau_c &\longrightarrow \{c(string)\} \end{aligned}$$

$$\begin{aligned} \tau_d &\longrightarrow \{d(\tau_e, \tau_c)\} \\ \tau_e &\longrightarrow \{e(string)\} \\ \tau_c &\longrightarrow \{c(string)\} \end{aligned}$$

$\tau_b \cap \tau_e = \emptyset \Rightarrow$ TYPE ERROR

Type inference for XML processing programs

Example:

If we want to translate the next document:

```
<catalogue>
   <book>
       <title> The Art of Computer Programming - Volume 1</title>
       <author> D. Knuth </author>
       <year> 1997 </year>
       <publisher> Addison-Wesley </publisher>
   </book>
   ...
</catalogue>
```

Type inference for XML processing programs

Example:

Validated by the DTD:

```
<!ELEMENT catalogue (book)+>
<!ELEMENT book (title,author,year,publisher)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT publisher (#PCDATA)>
```

Type inference for XML processing programs

Example:

To this new document:

```
<catalogue>
   <book>
       <title> The Art of Computer Programming - Volume 1</title>
       <year> 1997 </year>
   </book>
   ...
</catalogue>
```

Type inference for XML processing programs

Example:

Validated by the DTD:

```
<!ELEMENT catalogue (book)+>
<!ELEMENT book (title,year)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT year (#PCDATA)>
```

Type inference for XML processing programs

Example:

The next (simple) program is enough:

```
process(catalogue(L1),catalogue(L2)):-
    conversion(L1,L2).

conversion([book(A,_,Z,_)],[book(A,Z)]).

conversion([book(A,_,Z,_)|R1],[book(A,Z)|R2]):-
    conversion(R1,R2).
```

Conclusions

- Relation between Regular Types and DTDs.

Conclusions

- Relation between Regular Types and DTDs.
- Translating XML documents to Prolog terms.

Conclusions

- Relation between Regular Types and DTDs.
- Translating XML documents to Prolog terms.
- Type checking leads to correct processing of XML.

Future work

- Improve the efficiency of the type inference algorithm.

Future work

- Improve the efficiency of the type inference algorithm.
- "Real-world" applications.

END