Type-Based Specialization of XML Transformations

Kazutaka Matsuda*,
Zhenjiang Hu**, Masato Takeichi*

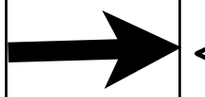
*The University of Tokyo
**National Institute of Informatics (Japan)

January 19th, 2009　PEPM'09

XML

- XML is widely-used.
- Trans. between XMLs are very important!

```
<members>
 <member>
   <name>Kztk</name>
   <email>kztk@...</email>
   <tel>+81-90...</tel>
 </member>
 ...
</members>
```



```
<html>...
<body>
<h1>Member List</h1>
<h2>Kztk</h2>
<address>kztk@...</address>
<address>+81-90...</address>
...
</body>
</html>
```

Question

Why specialization is required?

A Simple XML Trans.

convC(<email>*e*</email>) = <mailto>*e*</mailto>
convC(<phone>*t*</phone>) = <tel>*t*</tel>

Input XML (Elem.) → Output XML (Elem.)

<email>kztk@ipl...</email> → <mailto>kztk@ipl...</mailto>

<phone>+81-90-...</phone> → <tel>+81-90-...</tel>

A Simple XML Trans.

convC(<email>(*e*)) = <mailto>(*e*)
convC(<phone>(*t*)) = <tel>(*t*)

Input XML (Elem.)

<email>(kztk@ipl...)

<phone>(+81-90-...)

Output XML (Elem.)

<mailto>(kztk@ipl...)

<tel>(+81-90-...)

Type (Schema)

```
convC(<email>(e::String)) = <mailto>(e)
convC(<phone>(t::String)) = <tel>(t)
```

Src. Type

```
<email>(String)
| <phone>(String)
```

Ψ

<email>(kztk@ipl...)

<phone>(+81-90-...)

Tgt. Type

```
<mailto>(String)
| <tel>(String)
| <fax>(String)
```

Ψ

<mailto>(kztk@ipl...)

<tel>(+81-90-...)

Type Checking with Subtyping

```
convC(<email>(e::String)) = <mailto>(e)
convC(<phone>(t::String)) = <tel>(t)
```

convC::

```
<email>(String)
| <phone>(String)
```

→

```
<mailto>(String)
| <tel>(String)
```

Src. Type $\subseteq$?

```
<email>(String)
```

? $\subseteq$ Tgt. Type

```
<mailto>(String)
| <tel>(String)
| <fax>(String)
```

Problem

```
convC(<email>(e::String)) = <mailto>(e)
convC(<phone>(t::String)) = <tel>(t)
```

convC:: <email>(String) | <phone>(String) → <mailto>(String) | <tel>(String)

? Src. Type

<email>(String)

? Tgt. Type

<mailto>(String)

Use Trans. Def. for Preciseness

```
convC(<email>(e)) = <mailto>(e)
convC(<phone>(t)) = <tel>(t)
```

```
data S = <email>(String)
convC_onS(x::S) = convC(x)
```

Type of "x"

do Specialization

```
convC_S(<email>(e::String)) = <mailto>(e)
```

Sp. ver. of *convC* w.r.t. *S*

```
convC(<email>(e)) = <mailto>(e)
convC(<phone>(t)) = <tel>(t)
```

```
data S = <email>(String)
convC_onS(x::S) = convC_S(x)
```

convC$_S$:: <email>(*String*) → <mailto>(*String*)

Src. Type $\preceq$?

<email>(*String*)

? $\preceq$ Tgt. Type

<mailto>(*String*)

Short Summary

$convC_s$(<email>(e::*String*)) = <mailto>(e)

convC:: <email>(*String*) | <phone>(*String*) → <mailto>(*String*) | <tel>(*String*)

Specialize

$convC_s$:: <email>(*String*) → <mailto>(*String*)

Src. Type $\subseteq$?

<email>(*String*)

? $\subseteq$ Tgt. Type

<mailto>(*String*)

Question

Why specialization is required?

Answer

For precise analyses of programs.

Dream

- We hope to specialize
  - Types as expressive as XML schemata
    - Recursively defined types

- Transformation on XML trees
  - Recursively defined transformations

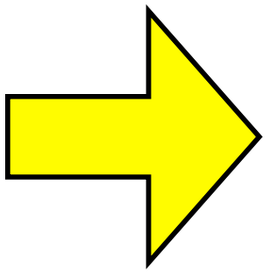
Difficulty

- Infinite specialization

```
ab2cd($)          = $
ab2cd(<a/>.r)     = <c/>.ab2cd(r)
ab2cd(<b/>.r)     = <d/>.ab2cd(r)
```

Context-Free
$\langle a/\rangle^n\langle b/\rangle^n$

```
data S = <a/>.S.<b/> | $
ab2cd_onS(x::S) = ab2cd(r)
```



```
ab2cd_S(<a/>.r::SB) = <c/>.ab2cd_SB(r)
ab2cd_SB(<a/>.r::SBB) = <c/>.ab2cd_SBB(r)
...
```

```
data SB   = S.<b/>
data SBB  = SB.<b/>
...
```

Infinite!

3 important things

1. Types
2. Transformations (functions)
3. Method to specialize

1

```
data S = <email>(String)
convC_onS(x::S) = convC(
```

2

```
convC(<email>(e::String)) = <mailto>(e)
convC(<tel>(t::String))   = <callto>(t)
```

3

```
convC_S(<email>(e::String)) = <mailto>(e)
```

3 important things

1. Types
2. Transformations (functions)
3. Method to specialize

1

```
data S = <email>(String)
convC_onS(x::S) = convC(
```

2

```
convC(<email>(e::String)) = <mailto>(e)
convC(<tel>(t::String))   = <callto>(t)
```

Simple Unfolding

```
email>(e::String)) = <mailto>(e)
```

Our Work

1. Types
   - Regular hedge grammar
2. Transformation (functions)
   - Designed to avoid Context-Freeness
3. Method to specialize
   - Simple unfolding of
     - Types/Transformations

(Regular Hedge) Type

- Regular hedge grammars
  - A model of schemata [Murata et al. 05]
    - DTD, XML Schema, Relax NG
  - ≒Tree automata [Comon et al. 97]
  - Types in XDuce/CDuce
    [Hosoya&Pierce 03, Benzaken et al. 03]

```
data Members = <members>(Member*)
data Member  = <member>(Name . Contact*)
data Name    = <name>(String)
data Contact = <email>(String) | <tel>(String)
```

Transformations

- Treeless [Wadler 90]
  - Arguments of funcs. must be variables.
- + Type-based XML-sequence splitting
  - A subset of patterns in XDuce

```
data As = ( <a>(String) )*
data Bs = ( <b>(String) )*
splitAB(x::As . y::Bs) = <as>(x).<bs>(y)
```

<a>1<a>23 → <as><a>1<a>2</as><bs>3</bs>

```
rev($)              = $
rev(x::Tree . r) = rev(r).x
```

rev(<a/><c/>) → <c/><a/>

Core Syntax of Lang.

- Whole (core) syntax

```
program ::= t_decl f_decl

 t_decl ::= data T1 = t1; ... data Tn = tn;
      t ::= $ | σ(T1).T2

f_decl ::= f1(p1) = e1; ... fm(pm) = em;
      p ::= $ | σ(p) | p1.p2 | x::T
      e ::= $ | σ(e) | e1.e2 | x | f(x)
```

Specialization Method

Sp. `rev(x::Tree . r)=rev(r).x`

```
data ABs
 = $ | <a/>.<b/>.ABs
```

1. Specialize pattern

`x::Tree . r` `ABs` → `x::<a/> . r::BABs`

where `data BABs = <b/>.ABs`

2. Generate new rule(s)

rev_{ABs}`(x::<a/> . r::BABs)=`rev_{BABs}`(r).x`

3. Recursively apply specialization

Sp. `rev($)=$`

`rev(x::Tree . r)=rev(r).x`

`BABs`

Termination

Sp. `rev(x::Tree . r)=rev(r).x`

```
data ABs
 = $ | <a/>.<b/>.ABs
```

1. Specialize pattern

`x::Tree . r` `ABs` → `x::<a/> . r::BABs`

W … `<b/>.ABs`

Always Regular

2. Generate new rule

rev_{ABs}`(x::<a/> . r::BABs)=`rev_{BABS}`(r).x`

3. Recurs…

Sp. `r…`

`rev(x…`

#(introduced types) is finite

⇒ Always terminating

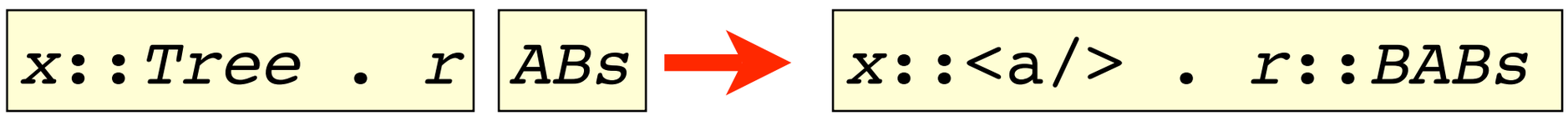
Related Work

- Exact type checking of XML Trans.
  [Maneth et al. 07]
  - Specialization to handle input XML type
  - (Affine) Treeless + Acc.
    - Precisely, (Linear) stay MTT [Fischer 68]

- Main Difference:
  Type-based XML-sequence splitting

Related Work

- Type inference in XDuce/CDuce
  [Hosoya&Pierce 03, Benzaken et al. 03]
  - Related to our pattern specialization



- Ours: precise (exact) but restricted

Conclusion

- Type-based specialization of XML trans.
  - Language:
    - Regular hedge grammars
    - Treeless + Type-based XML-seq. splitting
  - Property:
    - Correct & terminating
  - Applications:
    (Forward) type inference, inversion, ...

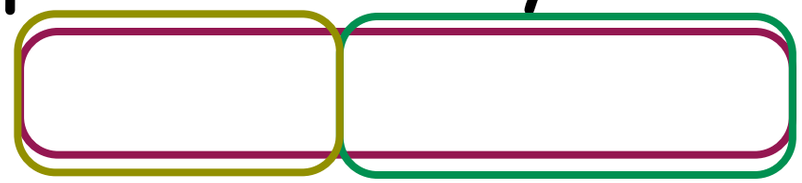
Prototype Implementation:
http://www.ipl.t.u-tokyo.ac.jp/~kztk/sp/

Thank you

Difficulty of Seq. Decomp.

- Without XML-seq. decomposition
  - A pattern only decompose the head

- With XML-seq. decomposition
  - A pattern may extract preceding seq.



- c.f. Type structure

$$\textbf{data}\ T = \sigma(T_1).T_2$$

Properties of Sp.

- Correct
  - $f_S(x) = y \Leftrightarrow x \in S \wedge f(x) = y$
- Always terminating
  - Thanks to regularity
- But, slow
  - #(introduced types) = $O(2^{2^{(2^{(\#types)})^2}})$
  - #(generated rules for a rule) < #(types)$^{\#(vars)}$
  - #(output funcs) <
    #(funcs) * #(introduced types)

Image of Finiteness