Typed Iterators for XML

Giuseppe Castagna\*  Kim Nguyễn♦

ICFP 2008, Victoria, BC, Canada

\* PPS (CNRS), Université Paris 7, Paris, France
♦ LRI, Université Paris-Sud 11, Orsay, France
Dealing with XML

XML ?
Dealing with XML

XML?

⇒ Tree organized data
⇒ Pervasive (XHTML, Ajax, Web Services, . . . )
Dealing with XML

XML?

⇒ Tree organized data
⇒ Pervasive (XHTML, Ajax, Web Services, . . . )

Types?
Dealing with XML

XML ?

⇒ Tree organized data
⇒ Pervasive (XHTML, Ajax, Web Services, ...)

Types ?

⇒ Regular tree grammars (a.k.a. regular types)
⇒ Describe sets of documents very precisely
Dealing with XML

XML ?

⇒ Tree organized data
⇒ Pervasive (XHTML, Ajax, Web Services, ...)

Types ?

⇒ Regular tree grammars (a.k.a. regular types)
⇒ Describe sets of documents very precisely

Programs ? (iterators)
Dealing with XML

XML ?
⇒ Tree organized data
⇒ Pervasive (XHTML, Ajax, Web Services, ...)

Types ?
⇒ Regular tree grammars (a.k.a. regular types)
⇒ Describe sets of documents very precisely

Programs ? (iterators)
⇒ “remove every <a> element occurring in the input”
⇒ “convert an xhtml document from transitional to strict”
Dealing with XML

XML?

⇒ Tree organized data
⇒ Pervasive (XHTML, Ajax, Web Services, ...)

Types?

⇒ Regular tree grammars (a.k.a. regular types)
⇒ Describe sets of documents very precisely

Programs? (iterators)

⇒ “remove every <a> element occurring in the input”
⇒ “convert an xhtml document from transitional to strict”
Being both polymorphic and precise

“remove every <a> element occurring in the input”
Being both polymorphic and precise

“remove every <a> element occurring in the input”

⇒ May be applied to any type of document = polymorphism
Being both polymorphic and precise

“remove every <a> element occurring in the input”

⇒ May be applied to any type of document = polymorphism
⇒ The output type remains precise

<foo>[ a* b* ] \sim \rightarrow \langle foo\rangle[ b* ]
<bar>[ b a* b* ] \sim \rightarrow \langle bar\rangle[ b+ ]
<baz>[ c* b? ] \sim \rightarrow \langle baz\rangle[ c* b? ]
Being both polymorphic and precise

“remove every <a> element occurring in the input”

⇒ May be applied to any type of document = polymorphism
⇒ The output type remains precise

\[
\begin{align*}
<\text{foo}>&[ a* b* ] \quad \sim \quad <\text{foo}>&[ b* ] \\
<\text{bar}>&[ b \ a* b* ] \quad \sim \quad <\text{bar}>&[ b+ ] \\
<\text{baz}>&[ c* b? ] \quad \sim \quad <\text{baz}>&[ c* b? ]
\end{align*}
\]

Neither parametric polymorphism (à la ML) nor regular expression types (à la XDuce/CDuce) are up to the task
Example: List concatenation

<table>
<thead>
<tr>
<th>type of ( \ell_1 )</th>
<th>type of ( \ell_2 )</th>
<th>type of ( \text{concat } \ell_1 \ell_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Int* ]</td>
<td>[ Bool+ ]</td>
<td>[ Int* Bool+ ]</td>
</tr>
<tr>
<td>[ Int ]</td>
<td>[ Int* Char? ]</td>
<td>[ Int+ Char? ]</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>
Example: List concatenation

<table>
<thead>
<tr>
<th>type of $\ell_1$</th>
<th>type of $\ell_2$</th>
<th>type of $(\text{concat } \ell_1 \ell_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Int* ]</td>
<td>[ Bool+ ]</td>
<td>[ Int* Bool+ ]</td>
</tr>
<tr>
<td>[ Int ]</td>
<td>[ Int* Char? ]</td>
<td>[ Int+ Char? ]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

val concat : ...

let $\ell = \text{concat } \ell_1 \ell_2$ in ...
### Example: List concatenation

<table>
<thead>
<tr>
<th>type of $\ell_1$</th>
<th>type of $\ell_2$</th>
<th>type of $(\text{concat } \ell_1 \ell_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Int* ]</td>
<td>[ Bool+ ]</td>
<td>[ Int* Bool+ ]</td>
</tr>
<tr>
<td>[ Int ]</td>
<td>[ Int* Char? ]</td>
<td>[ Int+ Char? ]</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

val concat : $\alpha$ list $\rightarrow$ $\alpha$ list $\rightarrow$ $\alpha$ list

let $\ell = \text{concat } \ell_1 \ell_2$ in ...
### Example: List concatenation

<table>
<thead>
<tr>
<th>type of $\ell_1$</th>
<th>type of $\ell_2$</th>
<th>type of $(\text{concat } \ell_1 \ell_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Int* ]</td>
<td>[ Bool+ ]</td>
<td>[ Int* Bool+ ]</td>
</tr>
<tr>
<td>[ Int ]</td>
<td>[ Int* Char? ]</td>
<td>[ Int+ Char? ]</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

val concat : $\alpha$ list $\rightarrow$ $\alpha$ list $\rightarrow$ $\alpha$ list

let $\ell = \text{concat } \ell_1 \ell_2$ in ...

must have the same type
## Example: List concatenation

<table>
<thead>
<tr>
<th>type of $\ell_1$</th>
<th>type of $\ell_2$</th>
<th>type of $(\text{concat } \ell_1 \ell_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>[ Int* ]</code></td>
<td><code>[ Bool+ ]</code></td>
<td><code>[ Int* Bool+ ]</code></td>
</tr>
<tr>
<td><code>[ Int ]</code></td>
<td><code>[ Int* Char? ]</code></td>
<td><code>[ Int+ Char? ]</code></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

val concat : `[ Any* ]$ → `[ Any* ]$ → `[ Any* ]$

let $\ell = \text{concat } \ell_1 \ell_2$ in ...
Example: List concatenation

<table>
<thead>
<tr>
<th>type of $\ell_1$</th>
<th>type of $\ell_2$</th>
<th>type of (concat $\ell_1 , \ell_2$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Int* ]</td>
<td>[ Bool+ ]</td>
<td>[ Int* Bool+ ]</td>
</tr>
<tr>
<td>[ Int ]</td>
<td>[ Int* Char? ]</td>
<td>[ Int+ Char? ]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

val concat : [ Any* ] → [ Any* ] → [ Any* ]

let $\ell$ = concat $\ell_1 \, \ell_2$ in ...

$\ell$ has type [ Any* ]
Example: List concatenation

<table>
<thead>
<tr>
<th>type of $\ell_1$</th>
<th>type of $\ell_2$</th>
<th>type of $(\text{concat } \ell_1 \ \ell_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Int* ]</td>
<td>[ Bool+ ]</td>
<td>[ Int* Bool+ ]</td>
</tr>
<tr>
<td>[ Int ]</td>
<td>[ Int* Char? ]</td>
<td>[ Int+ Char? ]</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

val concat : “no type”

let $\ell = \text{concat } \ell_1 \ \ell_2$ in ...
Example: List concatenation

<table>
<thead>
<tr>
<th>type of $\ell_1$</th>
<th>type of $\ell_2$</th>
<th>type of $(\text{concat } \ell_1 \ell_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Int* ]</td>
<td>[ Bool+ ]</td>
<td>[ Int* Bool+ ]</td>
</tr>
<tr>
<td>[ Int ]</td>
<td>[ Int* Char? ]</td>
<td>[ Int+ Char? ]</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

val concat : “no type”

let $\ell = \text{concat } \ell_1 \ell_2$ in ...

\[
\text{type (concat } \ell_1 \ell_2)\]
**Example: List concatenation**

<table>
<thead>
<tr>
<th>type of $\ell_1$</th>
<th>type of $\ell_2$</th>
<th>type of $(\text{concat } \ell_1 \ell_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ Int* ]</td>
<td>[ Bool+ ]</td>
<td>[ Int* Bool+ ]</td>
</tr>
<tr>
<td>[ Int ]</td>
<td>[ Int* Char? ]</td>
<td>[ Int+ Char? ]</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

val concat : “no type”

let $\ell = \text{concat } \ell_1 \ell_2$ in ...

```
type (concat $\ell_1 \ell_2$)
```

“Execute the transformation at the type level”
Computation at the type level?

- Ensure terminations of iterators $\Rightarrow$ not Turing complete
Computation at the type level?

- Ensure terminations of iterators $\Rightarrow$ not Turing complete
- Turing completeness is useful for non-XML computations
Computation at the type level?

- Ensure terminations of iterators $\Rightarrow$ not Turing complete
- Turing completeness is useful for non-XML computations
- If the language is too expressive, it escapes regular types

```plaintext
type t = [] | [a t b]
```
Computation at the type level?

- Ensure terminations of iterators ⇒ not Turing complete
- Turing completeness is useful for non-XML computations
- If the language is too expressive, it escapes regular types

```haskell
type t = [] | [ a t b ]
flattened t
```
Computation at the type level?

- Ensure terminations of iterators ⇒ not Turing complete
- Turing completeness is useful for non-XML computations
- If the language is too expressive, it escapes regular types

```plaintext
type t = [] | [ a t b ]
flatten t ↝ { [ a^n b^n ] | n ≥ 0}
```
Computation at the type level?

- Ensure terminations of iterators $\Rightarrow$ not Turing complete
- Turing completeness is useful for non-XML computations
- If the language is too expressive, it escapes regular types

```plaintext
type t = [] | [ a t b ]
flatten t $\sim$ \{ [ a^n b^n ] | n $\geq$ 0 \}
```

- In general there isn’t a best regular approximation

```plaintext
[ (a | b)* ]
[ a* b* ]
[] | [ a+ b+ ] ...
```
Computation at the type level?

- Ensure terminations of iterators ⇒ not Turing complete
- Turing completeness is useful for non-XML computations
- If the language is too expressive, it escapes regular types

\[
\text{type } t = \text{[]} \mid [a\ t\ b] \\
\text{flatten } t \rightsquigarrow \{ [a^n\ b^n] \mid n \geq 0 \}
\]

- In general there isn’t a best regular approximation

\[
\text{[ (a\ |\ b)^\ast ]} \\
\text{[ a^\ast\ b^\ast ]} \\
\text{[\ ]} \mid [a^+\ b^+] \ldots
\]

- The language must be expressive enough to express flattening, reversal, XPath, \ldots
Computation at the type level?

- Ensure terminations of iterators ⇒ not Turing complete
- Turing completeness is useful for non-XML computations
- If the language is too expressive, it escapes regular types

```haskell
type t = [] | [ a t b ]
flatten t ↦ { [ a^n b^n ] | n ≥ 0 }```

- In general there isn’t a best regular approximation

```haskell
[ (a | b)* ]
[ a* b* ]
[ ] | [ a+ b+ ] ...
```

- The language must be expressive enough to express flattening, reversal, XPath,...
Contributions: Filters

Small sub-language of combinators:

- grafted into an host language
  - are used to define XML transformations
  - the host language is used for non-XML stuff
  - implementation with CDuce as host
- can express XPath and XSLT-like transformations
- is precisely typed
- relies on some type annotations for “non-regular cases”
  - annotations are sparse and well-localized
  - completeness result up-to annotations
Host language

Filters: iterate expressions of the host language over a data-structure (list, tree, XML document, ...)

Requirements for the host language:

- type algebra with a product constructor
Filters

Definition (filter)

A filter is a regular (possibly infinite) production of:

\[
f ::= e \quad \text{(expression of the host language)}
\]

\[
| \quad p \rightarrow f \quad \text{(pattern)}
\]

\[
| \quad (f,f) \quad \text{(product)}
\]

\[
| \quad f|f \quad \text{(union)}
\]

\[
| \quad f;f \quad \text{(composition)}
\]

\[
f(v) \rightsquigarrow r
\]

with some restrictions
Examples

\[ id = x \rightarrow x \]
Examples

\[ id = x \rightarrow x \]

\[ id("foo") \sim \]

\[ id("foo") \]
Examples

\[ id = x \rightarrow x \]

\[ id(\text{"foo"}) \sim (x \rightarrow x)(\text{"foo"}) \]
Examples

\[ id = x \rightarrow x \]

\[ id("foo") \sim \]

\[ x \rightarrow "foo" \]
Examples

\[ id = x \rightarrow x \]

\[ id("foo") \rightarrow "foo" \]
Examples

\[ id = x \rightarrow x \]

\[ bump = [ ] \rightarrow [ ] \]
\[ \quad \mid (x \rightarrow x + 1, bump) \]
Examples

\[ id = x \rightarrow x \]

\[ \textit{bump} = [ ] \rightarrow [ ] \]
\[ \quad | \quad (x \rightarrow x + 1, \textit{bump}) \]

\[ \textit{bump}([1\ 2\ 3]) \rightarrow \]
\[ \quad \textit{bump}((1,(2,(3,[])))) \]
Examples

\[ id = x \rightarrow x \]

\[ bump = [\ ] \rightarrow [\ ] \]
\[ \mid (x \rightarrow x + 1, bump) \]

\[ \text{bump}([1 \ 2 \ 3]) \leadsto \]
\[ (2, (\text{bump}(2, (3, [])))) \]
\[ \begin{array}{c|c}
  x & 1 \\
\end{array} \]
Examples

\[ id = x \rightarrow x \]

\[ bump = [ ] \rightarrow [ ] \]
\[ | (x \rightarrow x + 1, bump) \]

\[ bump([1 \ 2 \ 3]) \sim \]
\[ (2, (3, (bump(3, [])))) \]

\[ x \ 2 \]
Examples

\(id = x \rightarrow x\)

\(bump = \[
\] \rightarrow \[
\]

| (\(x \rightarrow x + 1, bump\)) |

\(bump([1 \ 2 \ 3]) \sim\)

\((2, (3, (4, (bump \ [ ]))))\) \[x\ 3\]
Examples

\[ id = x \rightarrow x \]

\[ bump = \left[ \right] \rightarrow \left[ \right] \]
\[ \mid \left( x \rightarrow x + 1, bump \right) \]

\[ bump([1 \ 2 \ 3]) \leadsto \]
\[ (2,(3,(4,(bump \ \left[ \right]))))) \]
Examples

\[ id \ = \ x \rightarrow x \]

\[ bump \ = \ [ ] \rightarrow [ ] \]
\[ \quad \mid (x \rightarrow x + 1, bump) \]

\[ bump([1 \ 2 \ 3]) \rightsquigarrow \]
\[ (2, (3, (4, []))) \]
Examples

\[ id = x \rightarrow x \]

\[ \text{bump} = \begin{array}{l}
[ ] \rightarrow [ ] \\
( x \rightarrow x + 1, \text{bump} )
\end{array} \]

\[ \text{concat} = ( x, y ) \rightarrow ( x; \text{aux} ) \]

\[ \text{aux} = [ ] \rightarrow y \]
\[ ( z \rightarrow z, \text{aux} ) \]
Examples

\[ id = x \rightarrow x \]

\[ \textbf{bump} = \[ \] \rightarrow \[ \] \]
\[ | \quad (x \rightarrow x + 1, \text{bump}) \]

\[ \textbf{concat} = (x,y) \rightarrow (x;\text{aux}) \]
\[ \textbf{aux} = \[ \] \rightarrow y \]
\[ | \quad (z \rightarrow z,\text{aux}) \]

\[ \text{concat}((\[1\ 2\ 3\],\[4\ 5\])) \leadsto \]
\[ \text{concat}((1,(2,(3,[]))), (4,(5,[]))) \]
Examples

\[ id = x \mapsto x \]

\[ \text{bump} = \[] \mapsto \[] \
\quad \mid (x \mapsto x + 1, \text{bump}) \]

\[ \text{concat} = (x,y) \mapsto (x; \text{aux}) \]

\[ \text{aux} = \[] \mapsto y \
\quad \mid (z \mapsto z, \text{aux}) \]

\[ \text{concat}(([1 \ 2 \ 3], [4 \ 5])) \leadsto \]

\[
\begin{array}{|c|}
\hline
x & (1, (2, (3, []))) \\
\hline
y & (4, (5, [])) \\
\hline
\end{array}
\]
Examples

\[ id = x \rightarrow x \]

\[ bump = \left[ \right] \rightarrow \left[ \right] \]
\[ \quad | \quad (x \rightarrow x + 1, bump) \]

\[ concat = (x, y) \rightarrow (x; aux) \]
\[ aux = \left[ \right] \rightarrow y \]
\[ \quad | \quad (z \rightarrow z, aux) \]

\[ concat((\left[1 2 3\right], \left[4 5\right])) \sim \]
\[ (1, (2, (3, []))) \]

<table>
<thead>
<tr>
<th></th>
<th>(x)</th>
<th>(y)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(1, (2, (3, [ ]))))</td>
<td>(4, (5, [ ]))</td>
</tr>
</tbody>
</table>
Examples

\[ id = x \rightarrow x \]

\[ bump = [ ] \rightarrow [ ] \]
\[ \mid (x \rightarrow x + 1, bump) \]

\[ concat = (x,y) \rightarrow (x; aux) \]
\[ aux = [ ] \rightarrow y \]
\[ \mid (z \rightarrow z, aux) \]

\[ concat(([1 \ 2 \ 3], [4 \ 5])) \leadsto aux((1, (2, (3, [])))) \]

<table>
<thead>
<tr>
<th></th>
<th>( x )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( (1, (2, (3, [ ]))) )</td>
<td>( (4, (5, [ ])) )</td>
</tr>
</tbody>
</table>
Examples

\[ id = x \mapsto x \]

\[ bump = [ ] \mapsto [ ] \]
\[ \mid (x \mapsto x + 1, bump) \]

\[ concat = (x,y) \mapsto (x; aux) \]
\[ aux = [ ] \mapsto y \]
\[ \mid (z \mapsto z, aux) \]

\[ concat(([1 \ 2 \ 3],[4 \ 5])) \leadsto \]
\[ (1, (aux((2,(3,[]))))) \]

<table>
<thead>
<tr>
<th></th>
<th>(1,(2,(3,[]))))</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>(1,(2,(3,[]))))</td>
</tr>
<tr>
<td>y</td>
<td>(4,(5,[]))</td>
</tr>
</tbody>
</table>
Examples

\[ id = x \rightarrow x \]

\[ \text{bump} = [] \rightarrow [] \]
\[ \quad | \quad (x \rightarrow x + 1, \text{bump}) \]

\[ \text{concat} = (x,y) \rightarrow (x;\text{aux}) \]
\[ \text{aux} = [] \rightarrow y \]
\[ \quad | \quad (z \rightarrow z, \text{aux}) \]

\[ \text{concat}([ [1 \ 2 \ 3] , [4 \ 5] ]) \sim \]
\[ (1 , (2 , (\text{aux}((3, []))))) \]

<table>
<thead>
<tr>
<th>x</th>
<th>(1, (2, (3, [ ])))</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>(4, (5, [ ]))</td>
</tr>
</tbody>
</table>
Examples

\[ id = x \mapsto x \]

\[ bump = \begin{array}{c} [] \mapsto [] \\ (x \mapsto x + 1, bump) \end{array} \]

\[ concat = (x,y) \mapsto (x; aux) \]

\[ aux = \begin{array}{c} [] \mapsto y \\ (z \mapsto z, aux) \end{array} \]

\[ concat(([1 2 3], [4 5])) \sim (1, (2, (3, aux []))) \]

\[ \begin{array}{|c|}
\hline
x & (1, (2, (3, []))) \\
y & (4, (5, [])) \\
\hline
\end{array} \]
id \ = \ x \rightarrow x

bump \ = \ [ ] \rightarrow [ ]
\ | \ (x \rightarrow x + 1, bump)

concat \ = \ (x,y) \rightarrow (x;aux)
aux \ = \ [ ] \rightarrow y
\ | \ (z \rightarrow z, aux)

concat(([[1 2 3],[4 5]])) \sim (1, (2, (3, y))))

<table>
<thead>
<tr>
<th>x</th>
<th>(1, (2, (3, [ ]))))</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>(4, (5, [ ]))</td>
</tr>
</tbody>
</table>
Examples

\( id = x \rightarrow x \)

\( bump = [ ] \rightarrow [ ] \)
  \( | (x \rightarrow x + 1, bump) \)

\( concat = (x, y) \rightarrow (x; aux) \)

\( aux = [ ] \rightarrow y \)
  \( | (z \rightarrow z, aux) \)

\( concat(([1, 2, 3], [4, 5])) \sim (1, (2, (3, (4, (5, [])))))) \)

<table>
<thead>
<tr>
<th></th>
<th>( x )</th>
<th>( y )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>( (1, (2, (3, [ ])))) )</td>
<td>( (4, (5, [ ])) )</td>
</tr>
</tbody>
</table>
Termination

Well-formedness conditions:
Termination

Well-formedness conditions:

\[
\text{concat} = (x,y) \rightarrow (x;aux)
\]

\[
\text{aux} = [ ] \rightarrow y
\]

\[
\mid (z \rightarrow z, aux)
\]
Termination

Well-formedness conditions:

\[ \text{concat} = (x, y) \rightarrow (x; aux) \]
\[ \text{aux} = [ ] \rightarrow y \]
\[ \mid (z \rightarrow z, aux) \]

- contractivity
Termination

Well-formedness conditions:

\[ \text{concat} = ((x, y) \rightarrow (x; aux)) \]
\[ \text{aux} = [ ] \rightarrow y \]
\[ \quad | (z \rightarrow z, aux) \]

- contractivity
- local recursion for the composition
Termination

Well-formedness conditions:

\[
\begin{align*}
\text{concat} & = (x, y) \rightarrow (x; \text{aux}) \\
\text{aux} & = [\ ] \rightarrow y \\
& \quad | (z \rightarrow z, \text{aux})
\end{align*}
\]

- contractivity
- local recursion for the composition
Termination

Well-formedness conditions:

\[
concat = (x, y) \rightarrow (x; aux)
\]

\[
aux = [\ ] \rightarrow y
\]

\[
| (z \rightarrow z, aux)
\]

- contractivity
- local recursion for the composition

Example:

\[
bad = x \rightarrow (x, x); bad
\]
Termination

Well-formedness conditions:

\[
concat = ((x,y) \rightarrow (x;aux)) \\
 aux = [[] \rightarrow y \\
 | (z \rightarrow z, aux)
\]

- contractivity
- local recursion for the composition

Example:

\[
bad = x \rightarrow (x, x); bad \quad bad(0)
\]
Termination

Well-formedness conditions:

\[
\text{concat} = (x, y) \rightarrow (x; aux)
\]
\[
\text{aux} = \[ \] \rightarrow y
\]
\[
\mid (z \rightarrow z, aux)
\]

- contractivity
- local recursion for the composition

Example:

\[
\text{bad} = x \rightarrow (x, x); \text{bad bad}((0,0))
\]
Termination

Well-formedness conditions:

\[
\begin{align*}
\text{concat} &= (x, y) \rightarrow (x; \text{aux}) \\
\text{aux} &= [] \rightarrow y \\
&\quad \mid (z \rightarrow z, \text{aux})
\end{align*}
\]

- contractivity
- local recursion for the composition

Example:

\[
\text{bad} = x \rightarrow (x, x); \text{bad} \quad \text{bad}((0, 0), (0, 0))
\]
Termination

Well-formedness conditions:

\[
\text{concat} = (x, y) \rightarrow (x; aux) \\
\text{aux} = [ ] \rightarrow y \\
\quad \mid (z \rightarrow z, aux)
\]

- contractivity
- local recursion for the composition

Example:

\[
\text{bad} = x \rightarrow (x, x); \text{bad} \quad \text{bad}((((0,0), (0,0)), ((0,0), (0,0))))
\]
Termination

Well-formedness conditions:

\[
\text{concat} = (x, y) \rightarrow (x; \text{aux}) \\
\text{aux} = [\ ] \rightarrow y \\
\big| (z \rightarrow z, \text{aux})
\]

- contractivity
- local recursion for the composition

Example:

\[
\text{bad} = x \rightarrow (x, x); \text{bad} \quad \cdots
\]
Termination

Well-formedness conditions:

\[ \text{concat} = (x, y) \rightarrow (x; aux) \]
\[ \text{aux} = [ ] \rightarrow y \]
\[ \mid (z \rightarrow z, aux) \]

- contractivity
- local recursion for the composition

Example:

\[ \text{bad} = x \rightarrow (x, x); \text{bad} \]
Termination

Well-formedness conditions:

\[ \text{concat} = (x, y) \rightarrow (x; aux) \]
\[ aux = [ ] \rightarrow y \]
\[ | (z \rightarrow z, aux) \]

- contractivity
- local recursion for the composition

Example:

\[ \text{bad} = x \rightarrow (x, x); bad \]

Theorem

The evaluation of a filter on a finite value terminates.
Typing example (1/2)

\[
\text{bump} = [\ ] \rightarrow [\ ] \quad \text{t} = [\ ]\mid (\text{Int}, t) \ (\equiv [\text{Int*}])
\]

| (x \rightarrow x + 1, \text{bump}) |
Typing example (1/2)

\[
bump = [\ ] \rightarrow [\ ] \quad \quad t = [\ ] | (\text{Int}, t) \ (\equiv [\text{Int}^*])
| \quad (x \rightarrow x + 1, bump)
\]

Let us compute \( bump(t) \) :

\[
bump(t) =
\]
Typing example (1/2)

\[
bump = [\ ] \rightarrow [\ ] \quad t = [\ ] | (\text{Int}, t) \ (\equiv [\text{Int*}])
\]

\[
| (x \rightarrow x + 1, \text{bump})
\]

Let us compute \(bump(t)\):

\[
bump(t) = [\ ]
\]
Typing example (1/2)

\[ \text{bump} = [\ ] \rightarrow [\ ] \quad t = [\ ] | (\text{Int}, t) \ (\equiv [\text{Int}^*]) \]

\[
\begin{array}{c}
| (x \rightarrow x + 1, \text{bump})
\end{array}
\]

Let us compute \( \text{bump}(t) \):

\[ \text{bump}(t) = [\ ] \mid \]
Typing example (1/2)

\[
bump = \left[ \cdot \right] \rightarrow \left[ \cdot \right] \quad t = \left[ \cdot \right] \mid (\text{Int}, t) \ (\equiv \left[ \text{Int}^* \right])
\]

Let us compute \( \text{bump}(t) \):

\[
\text{bump}(t) = \left[ \cdot \right] \mid (\quad, \quad)
\]
Typing example (1/2)

\[ \text{bump} = \left[ \right] \rightarrow \left[ \right] \quad t = \left[ \right]|(\text{Int},t) \ (\equiv [\text{Int}^*]) \]
\[ | (x \rightarrow x + 1, \text{bump}) \]

Let us compute \( \text{bump}(t) : \)

\[ \text{bump}(t) = \left[ \right] | (\text{Int}, ) \]
Typing example (1/2)

\[ \text{bump} = [\ [] \rightarrow [\ []] \quad t = [\ [] | (\text{Int}, t) \ (\equiv [\text{Int}*]) \]
\]
\[
| (x \rightarrow x + 1, \text{bump})
\]

Let us compute \( \text{bump}(t) \) :

\[ \text{bump}(t) = [\ [] | (\text{Int}, \text{bump}(t)) \]
Typing example (1/2)

\[
bump \ = \ [ \ ] \rightarrow [ \ ] \quad \quad \quad \quad t = []|(\text{Int},t) \ (\equiv [\text{Int*}])
\]
\[
\mid (x \mapsto x + 1, \text{bump})
\]

Let us compute \( bump(t) \):

\[
bump(t) \ = \ [] \ | (\text{Int}, bump(t)) \\
\equiv \quad [ \text{Int*} ]
\]
Typing example (2/2)

\[ t = \text{[
  \]
  | 
  \]
  [ a t b ] \]

“flattens nested lists”
Typing example (2/2)

\[ t = \text{flatten} \left[ \left[ a \quad t \quad b \right] \right] \]

“flattens nested lists”

Let us compute \( \text{flatten}(t) \):

\[ \text{flatten}(t) = \]
Typing example (2/2)

```
flatten  t = [[]] | [[a t b]]
"

“flattens nested lists”

Let us compute `flatten(t)`: 

```
flatten(t) = []
```
Typing example (2/2)

`flatten` \[ t = [] \mid [ a \ t \ b ] \]
"flattens nested lists"

Let us compute `flatten(t)`:  

\[
\begin{align*}
\text{flatten}(t) &= [] \\
                   & \mid [ a \ b ]
\end{align*}
\]
Typing example (2/2)

`flatten`  \[ t = [] \mid [ a \ t \ b ] \]

“flattens nested lists”

Let us compute `flatten(t)`:

\[
\begin{align*}
    flatten(t) &= [] \\
               &\quad | [ a \ b ] \\
               &\quad \mid [ a \ a \ b \ b ] \\
\end{align*}
\]
Typing example (2/2)

flatten  \hspace{1cm} t = [\[] \mid [a \ t \ b ]

“flattens nested lists”

Let us compute \textit{flatten}(t):

\[
\text{flatten}(t) \ = \ []
\]
\[
\hspace{2cm} \mid [a \ b ]
\]
\[
\hspace{2cm} \mid [a \ a \ b \ b]
\]
\[
\hspace{2cm} \mid [a \ a \ a \ b \ b \ b]
\]
Typing example (2/2)

Let us compute $\text{flatten}(t)$:

\[
\text{flatten}(t) = \begin{bmatrix}
\end{bmatrix}
\begin{bmatrix}
\begin{bmatrix}
& a & b
\end{bmatrix}
\begin{bmatrix}
& a & a & b & b
\end{bmatrix}
\begin{bmatrix}
& a & a & a & b & b & b
\end{bmatrix}
\cdots
\end{bmatrix}
\]
Typing example (2/2)

\[ t = [] \mid [ a \ t \ b ] \]

“flattens nested lists”

Let us compute \( \text{flatten}(t) \):

\[
\begin{align*}
\text{flatten}(t) &= [] \\
& \quad | [ a \ b ] \\
& \quad | [ a \ a \ b \ b ] \\
& \quad | [ a \ a \ a \ b \ b \ b ] \\
& \quad | \ldots \\
& \leq [ (a|b)^* ]
\end{align*}
\]
Typing example (2/2)

Let us compute $\text{flatten}(t)$:

$$
\text{flatten}(t) = [] |
[ a b ] |
[ a a b b ] |
[ a a a b b b ] |
\cdots |
\leq [ a^* b^* ]
$$
Typing example (2/2)

```
flatten

\[ t = [] \mid [ a \ t \ b ] \]

"flattens nested lists"

Let us compute \( flatten(t) \):

\[
\begin{array}{c}
\text{flatten}(t) = [] \\
| [ a \ b ] \\
| [ a \ a \ b \ b ] \\
| [ a \ a \ a \ b \ b \ b ] \\
| \ldots \\
\leq [ ] \mid [ a+ \ b+ ]
\end{array}
\]
Typing example (2/2)

\(\textit{flatten} \quad t = \text{[[]} \mid \text{[} \text{a t b} \text{]}\)

“flattens nested lists”

Let us compute \(\textit{flatten}(t)\):

\[
\text{\textit{flatten}(t)} = \text{[} \mid \text{[a b]} \mid \text{[a a b b]} \mid \text{[a a a b b b]} \mid \ldots \leq \text{[} \mid \text{[a+ b+]} \]
\]

Theorem

Subject reduction for filters
Typing algorithm

- Possible to erase the subsumption rule “almost everywhere”
  \[ \Rightarrow \text{The subsumption is only necessary to type the left-hand side of a “;”: this is where we put the annotation.} \]
Typing algorithm

\[ f\{[a\ast \ b\ast]\}; g \]

- Possible to erase the subsumption rule “almost everywhere”
  - \( \Rightarrow \) The subsumption is only necessary to type the left-hand side of a “;”: this is where we put the annotation.
Typing algorithm

\[ f\{[a^* \ b^*]\}; g \]

- Possible to erase the subsumption rule “almost everywhere”
  \[ \Rightarrow \text{The subsumption is only necessary to type the left-hand side of a “;”: this is where we put the annotation.} \]

- Algorithm is sound w.r.t. the type system
Typing algorithm

\[ f\{[a^* b^*]\}; g \]

- Possible to erase the subsumption rule “almost everywhere”
  \( \Rightarrow \) The subsumption is only necessary to type the left-hand side of a “;”: this is where we put the annotation.

- Algorithm is sound w.r.t. the type system
- Algorithm is complete up-to annotations
  \( \Rightarrow \) “for every valid derivation in the system, I can annotate the filter so that the algorithm find the exact same type”
Typing algorithm

\[ f\{[a^* b^*]\}; g \]

- Possible to erase the subsumption rule “almost everywhere”
  \[ \Rightarrow \text{The subsumption is only necessary to type the left-hand side of a “;;”: this is where we put the annotation.} \]

- Algorithm is sound w.r.t. the type system

- Algorithm is complete up-to annotations
  \[ \Rightarrow \text{“for every valid derivation in the system, I can annotate the filter so that the algorithm find the exact same type”} \]
Added the filters as a sublanguage of C\textsc{Duce}:

**Definition (Concrete syntax)**

\[
\begin{align*}
  f & ::= e \mid \text{p} \rightarrow f \mid f ; f \mid (f,f) \mid f | f & \quad \text{unchanged} \\
  & \mid <f,f>f & \quad \text{xml} \\
  & \mid \text{let filter } x=f \ [ \ \text{and } x=f \ ... \] & \quad \text{binding} \\
  & \mid x & \quad \text{variable} \\
  e & ::= \ldots \mid \text{apply } f \ \text{to } e \ [ \ \text{where } a \] & \quad \text{application} \\
  a & ::= x=\{t_1,\ldots,t_n\} \ [ \ \text{and } a \] & \quad \text{annotation}
\end{align*}
\]
Examples (1)

Pattern matching:

```plaintext
match e with
 | p1  -> e1
 |     ...
 | pn  -> en
```
Examples (1)

Pattern matching:

\[
\text{apply } (p_1 \rightarrow e_1) \mid \ldots \mid (p_n \rightarrow e_n) \text{ to } e
\]
Examples (1)

Pattern matching:
apply (p1 -> e1) | ... | (pn -> en) to e

Tree mapping:

let filter up = < ( ‘section -> ‘chapter
| ‘subsection -> ‘section
| ‘paragraph -> ‘subsection
| x -> x ) >uplist
| x -> x

and filter uplist = [] -> [] | (up,uplist)
Examples (1)

Pattern matching:
apply (p1 -> e1) | ... | (pn -> en) to e

Tree mapping:

let filter up = < (‘section’ -> ‘chapter’
| ‘subsection’ -> ‘section’
| ‘paragraph’ -> ‘subsection’
| x -> x ) > uplist
| x -> x

and filter uplist = [] -> [] | (up, uplist)

If e : <doc>[ <section>[ (<subsection>[Char+] | Char)* ]+] then:
apply up to e
has type: <doc>[ <chapter>[ (<section>[Char+] | Char)* ]+]
Examples (2)

let filter flatten = [] -> []
  | ([Any*] -> flatten, flatten);concat
  | (x->x, flatten)

type t = [ ‘a t ‘b ] | []
type s = [ ‘c s ‘d ] | [‘c ‘d ]

let u : t = ...
let v : s = ...

apply flatten to u where { | flatten = { [ (‘a|‘b)* ] } | }
apply flatten to v where { | flatten = { [ (‘c|‘d)+ ] } | }
XPath encoding

//a/b : “returns exactly all <b>s which are under an <a>”
**XPath encoding**

//a/b : “returns exactly all <b>s which are under an <a>”

![Diagram](attachment:diagram.png)
XPath encoding

//a/b : “returns exactly all <b>s which are under an <a>”
XPath encoding

//a/b : "returns exactly all <b>s which are under an <a>"

let filter f1 =
    ((‘a -> ‘a) I_f12; <_>x -> x
    | (( x -> x )) I_f1; <_>x -> x
    | _  -> []

and filter f12 =
    ((‘a -> ‘a) I_f12; <_> x -> x
    | res -> ((‘b -> ‘b)>I_f13; <_> x -> (res,x)
    | (x -> x)> I_f1; <_> x -> x
    | _  -> []

and filter f13 =
    ((‘a -> ‘a) I_f12; <_> x -> x
    | (x -> x)> I_f1; <_> x -> x
    | _  -> []

and filter I_f1 =
    [] -> []
**XPath encoding and typing**

- **XPath encoding:**
  - `self`, `child` and `descendant-or-self` axes
  - handles some predicates by rewriting to patterns
  - respects XPath semantics (document order, no duplicates, ...)

- **XPath typing:**
  - Only need one annotation
  - Use of an ad-hoc algorithm to compute the annotation
  - Automatic type inference for a non-trivial subset of XPath
XPath encoding and typing

- **XPath encoding:**
  - self, child and descendant-or-self axes
  - handles some predicates by rewriting to patterns
  - respects XPath semantics (document order, no duplicates, ...)

- **XPath typing:**
  - Only need one annotation
  - Use of an ad-hoc algorithm to compute the annotation

⇒ Automatic type inference for a non-trivial subset of XPath
Conclusion

Filters:
• provide a way to define expressive (CDuce iterators, XSLT, XPath,...)
• precisely typed (esp. typing products, see paper)
• modular transformations within a host language

implementation
• integrated with CDuce
• encoding and automatic typing of an XPath fragment
• other syntactical constructs (“parametrized filters”, ...)
Conclusion

Filters:

- provide a way to define
  - expressive (CDuce iterators, XSLT, XPath, ...)
  - precisely typed (esp. typing products, see paper)
  - modular

  transformations within an host language

- implementation
  - integrated with CDuce
  - encoding and automatic typing of an XPath fragment
  - other syntactical constructs ("parametrized filters", ...)

Future work

- How to infer annotations in the general case?
Future work

- How to infer annotations in the general case?
- Efficient compilation
Future work

- How to infer annotations in the general case?
- Efficient compilation
- Integration with other languages
Future work

• How to infer annotations in the general case?
• Efficient compilation
• Integration with other languages
Future work

• How to infer annotations in the general case?
• Efficient compilation
• Integration with other languages
Real typing rules

\[
\begin{align*}
type(\Gamma, e) &= s & \Gamma \cup t/p \vdash f(t) &= s & t \leq \{ p \cup \{ f \} \} \\
\Gamma \vdash e(t) &= s & \Gamma \vdash (p \rightarrow f)(t) &= s \\
\pi(t) &= \{ (t^1_1, t^1_2), \ldots, (t^n_1, t^n_2) \} & \Gamma \vdash f_1(t^i_1) &= s^i_1 & \Gamma \vdash f_2(t^i_2) &= s^i_2 \\
\Gamma \vdash (f_1, f_2)(t) &= \bigvee_{i \in 1..n} (s^i_1, s^i_2) \\
t &\leq \{ f_1 \cup \{ f \} \} \\
t_1 &\leq t \& \{ f_1 \} \\
t_2 &\leq t \setminus \{ f_1 \} & \Gamma \vdash f_1(t_1) &= s_1 & \Gamma \vdash f_2(t_2) &= s_2 \\
\Gamma \vdash (f_1 | f_2)(t) &= \bigvee \{ i \mid t_i \neq \text{Empty} \} \ s_i \\
t &\leq \{ f_1 \} \\
s_1 &\leq \{ f_2 \} & \Gamma \vdash f_1(t) &= s_1 & \Gamma \vdash f_2(s_1) &= s_2 \\
\Gamma \vdash (f_1;f_2)(t) &= s_2 \\
\Gamma \vdash e(t) &= s' & s' &\leq s
\end{align*}
\]
Typing the union

\[ bump = [\ ] \rightarrow [\ ] \quad t = [\text{Int}]|((\text{Int},t)) \quad (\equiv \]
\[
| (x \rightarrow x + 1, bump)[\text{Int}+]\) \]
Typing the union

\[
bump = [\ ] \rightarrow [\ ] \quad t = [\text{Int}]((\text{Int}, t)) \quad (\equiv \\
\mid (x \rightarrow x + 1, bump)[\text{Int+}])
\]

\[
\emptyset \vdash [\ ]([\ ])([\ ]) = [\ ]
\]

\[
\emptyset \vdash [\ ] \rightarrow [\ ]([\ ])([\ ]) = [\ ]
\]

\[
\emptyset \vdash [\ ] \rightarrow [\ ]([\ ])([\ ]) = [\ ]
\]

\[
\emptyset \vdash \{(x : \text{Int}) \vdash x(\text{Int}) = \text{Int} \quad : \\
\emptyset \vdash x \rightarrow x + 1(\text{Int}) = \text{Int} \quad \emptyset \vdash bump(t) = s
\]

\[
\emptyset \vdash (x \rightarrow x + 1, bump)((\text{Int}, t)) = (\text{Int}, s)
\]

\[
\emptyset \vdash bump(t) = s
\]

With the simple typing rule:

\[
s = [\text{Int}^*]
\]

With the precise typing rule:

\[
s = [\text{Int}^+]\]
Product decomposition

In general, if \( t \leq (\text{Any, Any}) \), \( t = (t_1^1, t_1^2)\| \ldots \| (t_n^1, t_n^2) \) for some \( n \).

Problem: there is more than one way to decompose \( t \).

The decomposition affects the properties of the type-system.
Product decomposition

In general, if $t \leq (\text{Any, Any})$, $t = (t_1^1, t_2^1)\ldots|(t_1^n, t_2^n)$ for some $n$.

Problem: there is more than one way to decompose $t$. The decomposition affects the properties of the type-system.

Consider:

$f_1 = 0..3 \rightarrow A | 4..7 \rightarrow B \quad f_2 = 0..4 \rightarrow C | 0..6 \rightarrow D \quad f = (f_1, f_2)$

and the types $t$ and $s$:

$t = (0..3, 0..4)|(4..7, 0..6) \quad s = (2..5, 1..3)$

We can prove that:

$\emptyset \vdash f(t) = (A, C)|(B, D)$

but also:
Maximal product decomposition

Two disjoint components: 
(0..3, 0..4) and (4..7, 0..6). 
$s$ overlaps both.

Two non-disjoint components: 
(0..7, 0..4) and (4..7, 0..6). 
$s$ is included in (0..7, 0..4).