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 ?

$\Rightarrow$ Tree organized data

$\Rightarrow$ Pervasive (XHTML, Ajax, Web Services, . . .)

Types ?

$\Rightarrow$ Regular tree grammars (a.k.a. regular types)

$\Rightarrow$ 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 ocurring 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 ocurring in the input”

⇒ “convert an xhtml document from *transitional* to *strict*”

Being both polymorphic and precise

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

Being both polymorphic and precise

"remove every <a> element ocurring in the input"

$\Rightarrow$ May be applied to any type of document = polymorphism

Being both polymorphic and precise

"remove every <a> element ocurring in the input"

$\Rightarrow$ May be applied to any type of document = polymorphism

$\Rightarrow$ The output type remains precise

```
<foo>[ a* b* ]   ⇝ <foo>[ b* ]

<bar>[ b a* b* ] ⇝ <bar>[ b+ ]

<baz>[ c* b?  ]  ⇝ <baz>[ c* b?  ]
```

Being both polymorphic and precise

"remove every <a> element ocurring in the input"

⇒ May be applied to any type of document = polymorphism

⇒ The output type remains precise

```
<foo>[ a* b* ]   ⇝ <foo>[ b* ]

<bar>[ b a* b* ] ⇝ <bar>[ b+ ]

<baz>[ c* b?  ]  ⇝ <baz>[ c* b?  ]
```

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

Example: List concatenation

type of ℓ_1	type of ℓ_2	type of (concat ℓ_1 ℓ_2)
[Int*]	[Bool+]	[Int* Bool+]
[Int]	[Int* Char?]	[Int+ Char?]
⋮	⋮	⋮

Example: List concatenation

type of ℓ_1	type of ℓ_2	type of (concat ℓ_1 ℓ_2)
`[ Int* ]`	`[ Bool+ ]`	`[ Int* Bool+ ]`
`[ Int ]`	`[ Int* Char? ]`	`[ Int+ Char? ]`
⋮	⋮	⋮

```
val concat : ...
let ℓ = concat ℓ1  ℓ2 in ...
```

Example: List concatenation

type of ℓ_1	type of ℓ_2	type of (concat ℓ_1 ℓ_2)
[Int*]	[Bool+]	[Int* Bool+]
[Int]	[Int* Char?]	[Int+ Char?]
$\vdots$	$\vdots$	$\vdots$

val concat : α list $\rightarrow$ α list $\rightarrow$ α list

let ℓ = concat ℓ_1 ℓ_2 in ...

Example: List concatenation

type of ℓ_1	type of ℓ_2	type of (concat ℓ_1 ℓ_2)
[Int*]	[Bool+]	[Int* Bool+]
[Int]	[Int* Char?]	[Int+ Char?]
$\vdots$	$\vdots$	$\vdots$

val concat : α list $\to$ α list $\to$ α list

let ℓ = concat $\underbrace{\ell_1 \quad \ell_2}$ in ...

must have the same type

Example: List concatenation

type of ℓ_1	type of ℓ_2	type of (concat ℓ_1 ℓ_2)
[Int*]	[Bool+]	[Int* Bool+]
[Int]	[Int* Char?]	[Int+ Char?]
⋮	⋮	⋮

```
val concat : [ Any* ] → [ Any* ] → [ Any* ]
let ℓ = concat ℓ1  ℓ2 in ...
```

Example: List concatenation

type of ℓ_1	type of ℓ_2	type of (concat ℓ_1 ℓ_2)
[Int*]	[Bool+]	[Int* Bool+]
[Int]	[Int* Char?]	[Int+ Char?]
⋮	⋮	⋮

```
val concat : [ Any* ] → [ Any* ] → [ Any* ]
let ℓ = concat ℓ1 ℓ2 in ...
```

ℓ has type [Any*]

Example: List concatenation

type of ℓ_1	type of ℓ_2	type of (concat ℓ_1 ℓ_2)
[Int*]	[Bool+]	[Int* Bool+]
[Int]	[Int* Char?]	[Int+ Char?]
⋮	⋮	⋮

```
val concat : “no type”
let ℓ = concat ℓ1  ℓ2 in ...
```

Example: List concatenation

type of ℓ_1	type of ℓ_2	type of (concat ℓ_1 ℓ_2)
`[ Int* ]`	`[ Bool+ ]`	`[ Int* Bool+ ]`
`[ Int ]`	`[ Int* Char? ]`	`[ Int+ Char? ]`
⋮	⋮	⋮

```
val concat : “no type”
let ℓ = concat ℓ1 ℓ2 in ...
```

$\underbrace{\texttt{concat}\ \ell_1\ \ \ell_2}_{\text{type (concat } \ell_1\ \ \ell_2)}$

Example: List concatenation

type of ℓ_1	type of ℓ_2	type of (concat ℓ_1 ℓ_2)
[Int*]	[Bool+]	[Int* Bool+]
[Int]	[Int* Char?]	[Int+ Char?]
⋮	⋮	⋮

```
val concat : “no type”
let ℓ = concat ℓ1 ℓ2 in ...
```

type (concat ℓ_1 ℓ_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

```
type t = [] | [ a t b ]
```

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

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

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

```
type t = [] | [ a t b ]
flatten t ⇝ { [ aⁿ bⁿ ] | 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

```
type t = [] | [ a t b ]
flatten t ⇝ { [ aⁿ bⁿ ] | n ≥ 0}
```

$\texttt{flatten t} \rightsquigarrow \{ \texttt{[a}^n \texttt{ b}^n \texttt{]} \mid n \geq 0 \}$

- In general there isn't a best regular approximation

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

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

```
type t = [] | [ a t b ]
flatten t ⇝ { [ aⁿ bⁿ ] | n ≥ 0}
```

- In general there isn't a best regular approximation

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

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

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

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

- In general there isn't a best regular approximation

```
[ (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 :

$$
\begin{array}{rcll}
f & ::= & e & \text{(expression of the host language)} \\
& | & p \rightarrow f & \text{(pattern)} \\
& | & (f, f) & \text{(product)} \\
& | & f | f & \text{(union)} \\
& | & f ; f & \text{(composition)}
\end{array}
$$

$$f(v) \leadsto r$$

with some restrictions

Examples

$$id = x \to x$$

Examples

$$id \;=\; x \rightarrow x$$

$$id(\texttt{"foo"}) \rightsquigarrow$$

$$id(\texttt{"foo"})$$

Examples

$$id \;=\; x \rightarrow x$$

$$id(\texttt{"foo"}) \rightsquigarrow$$

$$(x \rightarrow x)(\texttt{"foo"})$$

Examples

$id \;=\; x \to x$

$id(\texttt{"foo"}) \leadsto$

x

x	"foo"

Examples

$$id \;=\; x \rightarrow x$$

$id($`"foo"`$) \leadsto$

`"foo"`

x	"foo"

Examples

$$id = x \to x$$

$$\begin{aligned} bump &= [\,] \to [\,] \\ &\mid (x \to x + 1, bump) \end{aligned}$$

Examples

$$id = x \to x$$

$$\begin{aligned} bump &= [\,] \to [\,] \\ &\mid (x \to x + 1, bump) \end{aligned}$$

$bump(\texttt{[1 2 3]}) \leadsto$

$\quad bump(\texttt{(1,(2,(3,[])))})$

Examples

$$id = x \to x$$

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

$bump(\texttt{[1 2 3]}) \rightsquigarrow$

$\texttt{(2,(}bump\texttt{(2,(3,[]))))}$

x	1

Examples

$$
\begin{aligned}
id &= x \to x \\[2em]
bump &= [\,] \to [\,] \\
&\mid\; (x \to x + 1, bump)
\end{aligned}
$$

$bump([1\ 2\ 3]) \rightsquigarrow$

$(2,(3,(bump(3,[]))))$

x	2

Examples

$$id = x \to x$$

$$\begin{aligned} bump &= [\,] \to [\,] \\ &\mid \ (x \to x + 1, bump) \end{aligned}$$

$bump($`[1 2 3]`$) \leadsto$

`(2,(3,(4,(`$bump$` []))))`

x	3

Examples

$$id = x \to x$$

$$\begin{aligned} bump &= [\,] \to [\,] \\ &\mid (x \to x + 1, bump) \end{aligned}$$

$bump($`[1 2 3]`$) \leadsto$

`(2,(3,(4,(`$bump$ `[]))))`

Examples

$$id \;=\; x \rightarrow x$$

$$\begin{aligned} bump \;&=\; [\,] \rightarrow [\,] \\ &\mid\; (x \rightarrow x + 1, bump) \end{aligned}$$

$bump($`[1 2 3]`$) \leadsto$

`(2,(3,(4,[])))`

Examples

$$id \;=\; x \to x$$

$$\begin{aligned} bump \;&=\; [\,] \to [\,] \\ &\;\mid\; (x \to x + 1, bump) \end{aligned}$$

$$\begin{aligned} concat \;&=\; (x,y) \to (x;aux) \\ aux \;&=\; [\,] \to y \\ &\;\mid\; (z \to z, aux) \end{aligned}$$

Examples

$$id = x \to x$$

$$\begin{aligned} bump &= [\,] \to [\,] \\ &\mid (x \to x + 1, bump) \end{aligned}$$

$$\begin{aligned} concat &= (x, y) \to (x; aux) \\ aux &= [\,] \to y \\ &\mid (z \to z, aux) \end{aligned}$$

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

$concat((1,(2,(3,[]))),(4,(5,[])))$

Examples

$$id \;=\; x \rightarrow x$$

$$\begin{aligned} bump \;=&\; [\,] \rightarrow [\,] \\ \mid&\; (x \rightarrow x + 1, bump) \end{aligned}$$

$$\begin{aligned} concat \;=&\; (x,y) \rightarrow (x;aux) \\ aux \;=&\; [\,] \rightarrow y \\ \mid&\; (z \rightarrow z, aux) \end{aligned}$$

$concat(\texttt{([1 2 3],[4 5])}) \rightsquigarrow$

x

x	(1,(2,(3,[ ])))
y	(4,(5,[ ]))

Examples

$$id = x \to x$$

$$\begin{array}{lcl} bump & = & [\,] \to [\,] \\ & | & (x \to x + 1, bump) \end{array}$$

$$\begin{array}{lcl} concat & = & (x,y) \to (x;aux) \\ aux & = & [\,] \to y \\ & | & (z \to z, aux) \end{array}$$

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

(1,(2,(3,[])))

x	(1,(2,(3,[ ])))
y	(4,(5,[ ]))

Examples

$$id = x \to x$$

$$\begin{aligned} bump &= [\,] \to [\,] \\ &\mid \ (x \to x + 1, bump) \end{aligned}$$

$$\begin{aligned} concat &= (x,y) \to (x; aux) \\ aux &= [\,] \to y \\ &\mid \ (z \to z, aux) \end{aligned}$$

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

$aux((1,(2,(3,[]))))$

x	(1,(2,(3,[ ])))
y	(4,(5,[ ]))

Examples

$$id = x \to x$$

$$\begin{aligned} bump &= [\,] \to [\,] \\ &\mid (x \to x + 1, bump) \end{aligned}$$

$$\begin{aligned} concat &= (x, y) \to (x; aux) \\ aux &= [\,] \to y \\ &\mid (z \to z, aux) \end{aligned}$$

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

$(1,(aux((2,(3,[])))))$

x	(1,(2,(3,[ ])))
y	(4,(5,[ ]))

Examples

$$id = x \to x$$

$$\begin{aligned} bump &= [\,] \to [\,] \\ &\mid (x \to x + 1, bump) \end{aligned}$$

$$\begin{aligned} concat &= (x,y) \to (x;aux) \\ aux &= [\,] \to y \\ &\mid (z \to z, aux) \end{aligned}$$

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

$(1,(2,(aux((3,[]))))$

x	(1,(2,(3,[ ])))
y	(4,(5,[ ]))

Examples

$$id \;=\; x \to x$$

$$\begin{aligned} bump \;&=\; [\,] \to [\,] \\ &\mid\; (x \to x + 1, bump) \end{aligned}$$

$$\begin{aligned} concat \;&=\; (x,y) \to (x;aux) \\ aux \;&=\; [\,] \to y \\ &\mid\; (z \to z, aux) \end{aligned}$$

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

(1,(2,(3,aux []))))

x	(1,(2,(3,[ ])))
y	(4,(5,[ ]))

Examples

$$id = x \to x$$

$$\begin{array}{lcl} bump & = & [\,] \to [\,] \\ & | & (x \to x + 1, bump) \end{array}$$

$$\begin{array}{lcl} concat & = & (x, y) \to (x; aux) \\ aux & = & [\,] \to {\color{red}y} \\ & | & (z \to z, aux) \end{array}$$

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

$(1,(2,(3,\ {\color{red}y}\ ))))$

x	(1,(2,(3,[ ])))
y	(4,(5,[ ]))

Examples

$$id \;=\; x \rightarrow x$$

$$\begin{aligned} bump \;&=\; [\,] \rightarrow [\,] \\ &\;\mid\; (x \rightarrow x + 1, bump) \end{aligned}$$

$$\begin{aligned} concat \;&=\; (x,y) \rightarrow (x;aux) \\ aux \;&=\; [\,] \rightarrow y \\ &\;\mid\; (z \rightarrow z, aux) \end{aligned}$$

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

`(1,(2,(3,(4,(5,[])))))`

x	(1,(2,(3,[ ])))
y	(4,(5,[ ]))

Termination

Well-formedness conditions:

Termination

Well-formedness conditions:

$$
\begin{array}{lcl}
concat & = & \mathbf{(}x\mathbf{,}y\mathbf{)} \rightarrow (x\mathbf{;}aux) \\
aux & = & [\,] \rightarrow y \\
 & \mid & \mathbf{(}z \rightarrow z\mathbf{,}aux\mathbf{)}
\end{array}
$$

Termination

Well-formedness conditions:

$$
\begin{array}{lcl}
concat & = & (x, y) \to (x; aux) \\
aux & = & [\,] \to y \\
 & | & (z \to z, aux)
\end{array}
$$

- contractivity

Termination

Well-formedness conditions:

$$
\begin{array}{lcl}
concat & = & \mathbf{(}x\mathbf{,}y\mathbf{)} \rightarrow (x\mathbf{;}aux) \\
aux & = & [\,] \rightarrow y \\
 & \mid & \mathbf{(}z \rightarrow z\mathbf{,}aux\mathbf{)}
\end{array}
$$

- contractivity
- local recursion for the composition

Termination

Well-formedness conditions:

$$
\begin{array}{lcl}
concat & = & \mathbf{(}x\mathbf{,}y\mathbf{)} \rightarrow (x\mathbf{;}aux) \\
aux & = & [\,] \rightarrow y \\
 & \mid & \mathbf{(}z \rightarrow z\mathbf{,}aux\mathbf{)}
\end{array}
$$

- contractivity
- local recursion for the composition

Termination

Well-formedness conditions:

$$
\begin{aligned}
concat &= (x,y) \to (x;aux) \\
aux &= [\,] \to y \\
&\mid\ (z \to z,aux)
\end{aligned}
$$

- contractivity
- local recursion for the composition

Example:

$$bad = x \to (x,x);bad$$

Termination

Well-formedness conditions:

$$\begin{array}{lcl} concat & = & (x,y) \rightarrow (x;aux) \\ aux & = & [\,] \rightarrow y \\ & | & (z \rightarrow z,aux) \end{array}$$

- contractivity
- local recursion for the composition

Example:

$$bad \ = \ x \rightarrow (x,x);bad \quad bad(0)$$

Termination

Well-formedness conditions:

$$
\begin{array}{lcl}
concat & = & \mathbf{(}x\mathbf{,}y\mathbf{)} \rightarrow (x\mathbf{;}aux) \\
aux & = & [\,] \rightarrow y \\
 & \mid & \mathbf{(}z \rightarrow z\mathbf{,}aux\mathbf{)}
\end{array}
$$

- contractivity
- local recursion for the composition

Example:

$bad \; = \; x \rightarrow (x, x)\mathbf{;}bad \quad bad(\texttt{(0,0)})$

Termination

Well-formedness conditions:

$$
\begin{aligned}
concat &= (x,y) \to (x;aux) \\
aux &= [\,] \to y \\
&\mid\ (z \to z, aux)
\end{aligned}
$$

- contractivity
- local recursion for the composition

Example:

$bad = x \to (x,x);bad \quad bad(((0,0),(0,0)))$

Termination

Well-formedness conditions:

$$
\begin{array}{lcl}
concat & = & (x,y) \to (x;aux) \\
aux & = & [\,] \to y \\
 & | & (z \to z,aux)
\end{array}
$$

- contractivity
- local recursion for the composition

Example:

$bad \;=\; x \to (x,x);bad$ $\;bad$((((0 , 0) , (0 , 0)) , ((0 , 0) , (0 ,

Termination

Well-formedness conditions:

$$\begin{array}{lcl} concat & = & \mathbf{(}x\mathbf{,}y\mathbf{)} \to (x\mathbf{;}aux) \\ aux & = & [\,] \to y \\ & | & \mathbf{(}z \to z\mathbf{,}aux\mathbf{)} \end{array}$$

- contractivity
- local recursion for the composition

Example:

$$bad \;=\; x \to (x, x)\mathbf{;}bad \quad \cdots$$

Termination

Well-formedness conditions:

$$\begin{array}{lcl} concat & = & (x,y) \to (x;aux) \\ aux & = & [\,] \to y \\ & \mid & (z \to z, aux) \end{array}$$

- contractivity
- local recursion for the composition

Example:

$$\sout{bad \;=\; x \to (x, x); bad}$$

Termination

Well-formedness conditions:

$$\begin{array}{lcl} concat & = & \mathbf{(}x\mathbf{,}y\mathbf{)} \rightarrow (x\mathbf{;}aux) \\ aux & = & [\,] \rightarrow y \\ & \mid & \mathbf{(}z \rightarrow z\mathbf{,}aux\mathbf{)} \end{array}$$

- contractivity
- local recursion for the composition

Example:

$$\sout{bad \;=\; x \rightarrow (x,x)\mathbf{;}bad}$$

Theorem

The evaluation of a filter on a finite value terminates.

Typing example (1/2)

$$\begin{array}{lll} \mathit{bump} & = & [\,] \to [\,] \\ & \mid & (x \to x + \mathit{1}, \mathit{bump}) \end{array} \qquad t = \texttt{[]} \,|\, (\texttt{Int}, t) \quad (\equiv \texttt{[Int*]})$$

Typing example (1/2)

$$bump \;=\; [\,] \to [\,] \quad | \quad (x \to x + 1, bump) \qquad t = \texttt{[]} | (\texttt{Int}, t) \;\; (\equiv \texttt{[Int*]})$$

Let us compute $bump(t)$:

$$bump(t) \;=$$

Typing example (1/2)

$$bump \;=\; [\,]\to[\,] \;\;|\;\; (x\to x+1, bump) \qquad t = \texttt{[]}\,|\,(\texttt{Int}, t)\;\;(\equiv \texttt{[Int*]})$$

Let us compute $bump(t)$:

$$bump(t) \;=\; \texttt{[]}$$

Typing example (1/2)

$$bump \;=\; [\,] \to [\,] \qquad\qquad t = \texttt{[]}\,|\,(\texttt{Int},t) \;\; (\equiv \texttt{[Int*]})$$
$$\;\;|\;\; (x \to x + 1, bump)$$

Let us compute $bump(t)$:

$$bump(t) \;=\; \texttt{[]} \;|$$

Typing example (1/2)

$$\begin{array}{lcl} bump & = & [\,] \to [\,] \\ & | & (x \to x + 1, bump) \end{array} \qquad t = \texttt{[]} | (\texttt{Int}, t) \ \ (\equiv \texttt{[Int*]})$$

Let us compute $bump(t)$:

$$bump(t) \ = \ \texttt{[]} \mid (\quad , \quad)$$

Typing example (1/2)

$$\begin{array}{rcl} bump & = & [\,] \to [\,] \\ & | & (x \to x + 1, bump) \end{array} \qquad t = \texttt{[]} | (\texttt{Int}, t) \;\; (\equiv \texttt{[Int*]})$$

Let us compute $bump(t)$:

$$bump(t) \;\; = \;\; \texttt{[]} \mid (\texttt{Int}, \qquad\qquad)$$

Typing example (1/2)

$$\begin{array}{rcl} bump & = & [\,] \to [\,] \\ & \mid & (x \to x + 1, bump) \end{array} \qquad t = \texttt{[]} \,|\, (\texttt{Int}, t) \;\; (\equiv \texttt{[Int*]})$$

Let us compute $bump(t)$:

$$bump(t) \;=\; \texttt{[]} \mid (\texttt{Int}, bump(t))$$

Typing example (1/2)

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

Let us compute $bump(t)$:

$$\begin{aligned} bump(t) &= \texttt{[]} \mid (\texttt{Int}, bump(t)) \\ &\equiv \texttt{[Int*]} \end{aligned}$$

Typing example (2/2)

flatten $t = \texttt{[]} \mid \texttt{[}\ a\ t\ b\ \texttt{]}$

"flattens nested lists"

Typing example (2/2)

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

"flattens nested lists"

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

$$\mathit{flatten}(t) \ =$$

Typing example (2/2)

flatten $t = \texttt{[]} \mid \texttt{[}\ a\ t\ b\ \texttt{]}$

"flattens nested lists"

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

$$\mathit{flatten}(t) = \texttt{[]}$$

Typing example (2/2)

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

"flattens nested lists"

Let us compute *flatten*(t):

$$\begin{aligned} flatten(t) &= [] \\ &\mid [\ \mathtt{a}\ \mathtt{b}\ ] \end{aligned}$$

Typing example (2/2)

flatten $t = \texttt{[]} \mid \texttt{[} \ a \ t \ b \ \texttt{]}$

"flattens nested lists"

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

$$\begin{aligned} \mathit{flatten}(t) &= \texttt{[]} \\ &\mid \texttt{[a b]} \\ &\mid \texttt{[a a b b]} \end{aligned}$$

Typing example (2/2)

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

"flattens nested lists"

Let us compute *flatten*(t):

$$
\begin{array}{rcl}
\mathit{flatten}(t) & = & \texttt{[]} \\
& | & \texttt{[a b]} \\
& | & \texttt{[a a b b]} \\
& | & \texttt{[a a a b b b]}
\end{array}
$$

Typing example (2/2)

flatten $t = \texttt{[]} \mid \texttt{[}\ a\ t\ b\ \texttt{]}$

"flattens nested lists"

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

$$\begin{aligned}\mathit{flatten}(t) &= \texttt{[]}\\ &\mid \texttt{[a b]}\\ &\mid \texttt{[a a b b]}\\ &\mid \texttt{[a a a b b b]}\\ &\mid \ldots\end{aligned}$$

Typing example (2/2)

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

"flattens nested lists"

Let us compute $flatten(t)$:

$$
\begin{array}{rcl}
flatten(t) & = & \texttt{[]} \\
 & | & \texttt{[a b]} \\
 & | & \texttt{[a a b b]} \\
 & | & \texttt{[a a a b b b]} \\
 & | & \ldots \\
 & \leq & \texttt{[(a|b)*]}
\end{array}
$$

Typing example (2/2)

flatten $\quad t = \texttt{[]} \mid \texttt{[}\ a\ t\ b\ \texttt{]}$

"flattens nested lists"

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

$$\begin{aligned}
\mathit{flatten}(t) &= \texttt{[]} \\
&\mid \texttt{[a b]} \\
&\mid \texttt{[a a b b]} \\
&\mid \texttt{[a a a b b b]} \\
&\mid \ \ldots \\
&\leq \texttt{[a* b*]}
\end{aligned}$$

Typing example (2/2)

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

"flattens nested lists"

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

$$
\begin{array}{rcl}
\mathit{flatten}(t) & = & \texttt{[]} \\
 & | & \texttt{[a b]} \\
 & | & \texttt{[a a b b]} \\
 & | & \texttt{[a a a b b b]} \\
 & | & \ldots \\
 & \leq & \texttt{[ ] | [a+ b+]}
\end{array}
$$

Typing example (2/2)

flatten $t = \;$ `[]` | `[` $a\ t\ b$ `]`

"flattens nested lists"

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

$$
\begin{array}{rcl}
\mathit{flatten}(t) & = & \texttt{[]} \\
 & | & \texttt{[a b]} \\
 & | & \texttt{[a a b b]} \\
 & | & \texttt{[a a a b b b]} \\
 & | & \ldots \\
 & \leq & \texttt{[ ] | [a+ b+]}
\end{array}
$$

Theorem

Subject reduction for filters

Typing algorithm

- Possible to erase the subsumption rule "almost everywhere"
  - ⇒ The subsumption is only necessary to type the left-hand side of a "**;**": this is where we put the annotation.

Typing algorithm

$$f_{\{\texttt{[a* b*]}\}}\mathbf{;}g$$

- Possible to erase the subsumption rule "almost everywhere"
  - ⇒ The subsumption is only necessary to type the left-hand side of a "**;**": this is where we put the annotation.

Typing algorithm

$$f_{\{\texttt{[a* b*]}\}}\boldsymbol{;}g$$

- Possible to erase the subsumption rule "almost everywhere"
  - ⇒ 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

$$f_{\{\texttt{[a* b*]}\}};g$$

- Possible to erase the subsumption rule "almost everywhere"
  - ⇒ 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
  - ⇒ "for every valid derivation in the system, I can annotate the filter so that the algorithm find the exact same type"

Typing algorithm

$$f_{\{\texttt{[a* b*]}\}};g$$

- Possible to erase the subsumption rule "almost everywhere"
  - ⇒ 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
  - ⇒ "for every valid derivation in the system, I can annotate the filter so that the algorithm find the exact same type"

Implementation and examples

Added the filters as a sublanguage of $\mathbb{C}$Duce:

Definition (Concrete syntax)

$$
\begin{array}{rcll}
f & ::= & e \mid p\ \texttt{->}f \mid f\ \texttt{;}f \mid \texttt{(}f\texttt{,}f\texttt{)} \mid f\texttt{|}f & \text{unchanged} \\
 & \mid & \texttt{<}f\ f\texttt{>}f & \text{xml} \\
 & \mid & \texttt{let filter}\ \underline{x}\texttt{=}f\ [\ \texttt{and}\ \underline{x}\texttt{=}f \ldots] & \text{binding} \\
 & \mid & \underline{x} & \text{variable} \\
e & ::= & \ldots \mid \texttt{apply}\ f\ \texttt{to}\ e\ [\ \texttt{where}\ a\ ] & \text{application} \\
a & ::= & \underline{x}\texttt{=}\{t_1\texttt{,} \ldots \texttt{,}t_n\}\ [\ \texttt{and}\ a\ ] & \text{annotation}
\end{array}
$$

Examples (1)

Pattern matching:

```
match e with
 | p1 -> e1
     ...
 | pn -> en
```

Examples (1)

Pattern matching:

```
apply (p1 -> e1) | ... | (pn -> en) 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)

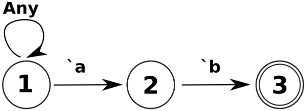
```
let filter flatten =  [] -> []
        | ([Any*] -> flatten, flatten);concat
        | (x->x, flatten)

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>`”

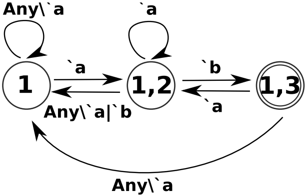
XPath encoding

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



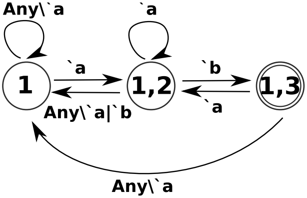
XPath encoding

//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 =
   [] -> []
 | (f1,I_f1);concat
```

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 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

Conclusion

Filters :

- provide a way to define
  - ⇒ expressive ($\mathbb{C}$Duce iterators, XSLT, XPath,. . .)
  - ⇒ precisely typed (esp. typing products, see paper)
  - ⇒ modular

  transformations within an host language
- implementation
  - ⇒ integrated with $\mathbb{C}$Duce
  - ⇒ 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

$$\frac{\texttt{type}(\Gamma, e) = s}{\Gamma \vdash e(t) = s} \qquad \frac{\Gamma \cup t/p \vdash f(t) = s \qquad t \leq \wr p \int \mathbin{\&} \wr f \int}{\Gamma \vdash (p \to f)(t) = s}$$

$$\frac{\pi(t) = \{(t_1^1, t_2^1), \ldots, (t_1^n, t_2^n)\} \qquad \Gamma \vdash f_1(t_1^i) = s_1^i \qquad \Gamma \vdash f_2(t_2^i) = s_2^i}{\Gamma \vdash (f_1, f_2)(t) = \bigvee_{i \in 1..n} (s_1^i, s_2^i)}$$

$$\frac{\begin{array}{l} t \leq \wr f_1 \int \mathbin{|} \wr f_2 \int \\ t_1 = t \mathbin{\&} \wr f_1 \int \\ t_2 = t \setminus \wr f_1 \int \qquad \Gamma \vdash f_1(t_1) = s_1 \qquad \Gamma \vdash f_2(t_2) = s_2 \end{array}}{\Gamma \vdash (f_1 | f_2)(t) = \bigvee_{\{i | t_i \neq \mathsf{Empty}\}} s_i}$$

$$\frac{\begin{array}{l} t \leq \wr f_1 \int \\ s_1 \leq \wr f_2 \int \qquad \Gamma \vdash f_1(t) = s_1 \qquad \Gamma \vdash f_2(s_1) = s_2 \end{array}}{\Gamma \vdash (f_1 ; f_2)(t) = s_2}$$

$$\frac{\Gamma \vdash e(t) = s' \quad s' \leq s}{}$$

Typing the union

$$bump \;=\; [\,] \rightarrow [\,] \;\mid\; \left(x \rightarrow x + 1, bump\right) \qquad t \;=\; \texttt{[Int]} \,\big|\, \left(\texttt{Int}, t\right) \quad (\equiv \texttt{[Int+]}\,)$$

Typing the union

$$
\begin{array}{rcl}
bump & = & [\,] \to [\,] \\
& | & (x \to x + 1, bump)[\texttt{Int+}]\,)
\end{array}
\qquad
t \;=\; [\texttt{Int}] | (\texttt{Int}, t) \qquad (\equiv
$$

$$
\dfrac{\dfrac{\varnothing \vdash [\,]([\,]) = [\,]}{\varnothing \vdash [\,] \to [\,]([\,]) = [\,]} \quad \dfrac{\dfrac{\{x : \texttt{Int}\} \vdash x(\texttt{Int}) = \texttt{Int}}{\varnothing \vdash x \to x + 1(\texttt{Int}) = \texttt{Int}} \quad \dfrac{\vdots}{\varnothing \vdash bump(t) =}}{\varnothing \vdash (x \to x + 1, bump)((\texttt{Int}, t)) = (\texttt{Int}, s}}{\varnothing \vdash bump(t) = s}
$$

With the simple typing rule:

$$s \;=\; [\texttt{Int*}]$$

With the precise typing rule:

$$s \;=\; [\texttt{Int+}]$$

Product decomposition

In general, if $t \leq$ (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.

Product decomposition

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

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

The decomposition affects the properties of the type-system.

Consider:

$$f_1 = 0..3 \to \texttt{A} | 4..7 \to \texttt{B} \qquad f_2 = 0..4 \to \texttt{C} | 0..6 \to \texttt{D} \qquad f = (f_1,f_2)$$

and the types t and s:

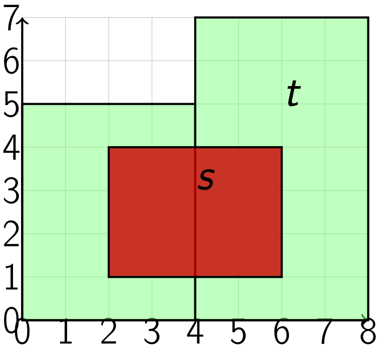
$$t = (0..3,0..4)|(4..7,0..6) \qquad s = (2..5,1..3)$$

We can prove that:

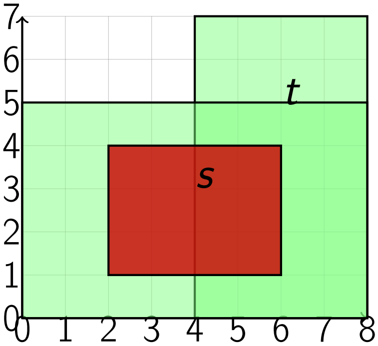
$$\varnothing \vdash f(t) = (\texttt{A},\texttt{C})|(\texttt{B},\texttt{D})$$

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**)**.