Types and Patterns for Querying XML

Giuseppe Castagna

CNRS
École Normale Supérieure de Paris

Trondheim, 28th of August, 2005
Working on XML data requires at least two kinds of primitives:

- Deconstruction/extraction primitives: pinpoint and capture subparts of the XML data.
- Iteration primitives: iterate over XML trees the process of extraction and transformation of data.
Working on XML data requires at least two kinds of primitives:

1. **deconstruction/extraction primitives**: pinpoint and capture subparts of the XML data
   Two solutions stem from practice:
   - Path expressions
   - Regular expression patterns

2. **iteration primitives**: iterate over XML trees, the process of extraction and transformation of data.
   No emerging solution: FLWR (XQuery), select-from-where (Cω, CQL), select-where (Lorel, loto-ql), filter (XDuce), xtransform (CDuce), in the language semantics (XSLT), ...
Working on XML data requires at least two kinds of primitives:

1. **deconstruction/extraction primitives**: pinpoint and capture subparts of the XML data
   Two solutions stem from practice:
   - Path expressions
   - Regular expression patterns

2. **iteration primitives**: iterate over XML trees the process of extraction and transformation of data.
   No emerging solution: FLWR (XQuery), select-from-where (Cω, CQL), select-where (Lorel, loto-ql), filter (XDuce), xtransform (CDuce), in the language semantics (XSLT), ...
Language Primitives

Working on XML data requires at least two kinds of primitives:

1. **deconstruction/extraction primitives**: pinpoint and capture subparts of the XML data
   Two solutions stem from practice:
   - Path expressions
   - Regular expression patterns

2. **iteration primitives**: iterate over XML trees the process of extraction and transformation of data.
   No emerging solution: FLWR (XQuery), select-from-where (Cω, CQL), select-where (Lorel, loto-ql), filter (XDuce), xtransform (CDuce), in the language semantics (XSLT), ...
Working on XML data requires at least two kinds of primitives:

1. **deconstruction/extraction primitives**: pinpoint and capture subparts of the XML data
   
   Two solutions stem from *practice*:
   - Path expressions
   - Regular expression patterns

2. **iteration primitives**: iterate over XML trees the process of extraction and transformation of data.
   
   No emerging solution: FLWR (XQuery), select-from-where ($C\omega$, $CQL$), select-where (Lorel, loto-ql), filter (XDuce), xtransform (CDuce), in the language semantics (XSLT), …
In running query/programming languages:

- Paths: "vertical" exploration of data, capture elements that may be at different depths (unary queries).
  Usually XPath paths, but also the "dot" navigations (Cω, Lorel, TQL) or caterpillar expressions.

- Regular expression patterns: "horizontal" exploration of data, perform finer grained decomposition on sequences of elements.
  Proposed by Hosoya & Pierce for XDuce and then adopted by CDuce/CQL, Xtratic, Scala, XHaskell.

The two primitives are not antagonist: they are orthogonal and complementary.
Deconstructors/extractors

In running query/programming languages:

- **Paths:** “vertical” exploration of data, capture elements that may be at different depths (unary queries)

  Usually XPath paths, but also the “dot” navigations ($C\omega$, Lorel, TQL) or caterpillar expressions.

- **Regular expression patterns:** “horizontal” exploration of data, perform finer grained decomposition on sequences of elements

  Proposed by Hosoya&Pierce for XDuce and then adopted by CDuce/CQL, Xtatic, Scala, XHaskell.

The two primitives are not antagonist: they are orthogonal and complementary.
Deconstructors/extractors

In running query/programming languages:

- **Paths:** “vertical” exploration of data, capture elements that may be at different depths (unary queries)
  Usually XPath paths, but also the “dot” navigations ($C\omega$, Lorel, TQL) or caterpillar expressions.

- **Regular expression patterns:** “horizontal” exploration of data, perform finer grained decomposition on sequences of elements
  Proposed by Hosoya&Pierce for XDuce and then adopted by CDue/CQL, Xtatic, Scala, XHaskell, ...

The two primitives are not antagonist: they are orthogonal and complementary.
Deconstructors/extractors

In running query/programming languages:

- **Paths:** “vertical” exploration of data, capture elements that may be at different depths (unary queries)
  Usually XPath paths, but also the “dot” navigations (Cω, Lorel, TQL) or caterpillar expressions.

- **Regular expression patterns:** “horizontal” exploration of data, perform finer grained decomposition on sequences of elements
  Proposed by Hosoya&Pierce for XDuce and then adopted by CDuce/CQL, Xtatic, Scala, XHaskell,…

The two primitives are not antagonist:
they are orthogonal and complementary.
Deconstructors/extractors

In running query/programming languages:

- **Paths**: “vertical” exploration of data, capture elements that may be at different depths (unary queries)
  Usually XPath paths, but also the “dot” navigations (Cω, Lorel, TQL) or caterpillar expressions.

- **Regular expression patterns**: “horizontal” exploration of data, perform finer grained decomposition on sequences of elements
  Proposed by Hosoya&Pierce for XDuce and then adopted by CDuce/CQL, Xtatic, Scala, XHaskell,…

The two primitives are not antagonist:
they are orthogonal and complementary.
Deconstructors/extractors

In running query/programming languages:

- **Paths**: “vertical” exploration of data, capture elements that may be at different depths (unary queries)
  - Usually XPath paths, but also the “dot” navigations (Cω, Lorel, TQL) or caterpillar expressions.

- **Regular expression patterns**: “horizontal” exploration of data, perform finer grained decomposition on sequences of elements
  - Proposed by Hosoya&Pierce for XDuce and then adopted by CDuce/CQL, Xtatic, Scala, XHaskell,…

**The two primitives are not antagonist:**
- they are orthogonal and complementary.
Deconstructors/extractors

In running query/programming languages:

- **Paths**: “vertical” exploration of data, capture elements that may be at different depths (unary queries)
  Usually XPath paths, but also the “dot” navigations (Cω, Lorel, TQL) or caterpillar expressions.

- **Regular expression patterns**: “horizontal” exploration of data, perform finer grained decomposition on sequences of elements
  Proposed by Hosoya&Pierce for XDuce and then adopted by CDuce/CQL, Xtatic, Scala, XHaskell,…

The two primitives are not antagonist:
they are orthogonal and complementary.

It seems natural to integrate both of them into a query/programming language for XML.
Mixing horizontal and vertical selectors

Several theoretical works from *different areas* about integrating vertical and horizontal exploration:

1. *Unranked tree logics*: e.g. Neven&Schwentick’s ETL.
2. *Spatial modal logics*: e.g. Cardelli&Ghelli’s TQL.
3. *Query languages*: e.g. Papakonstantinou&Vianu’s Loto-ql

But in running languages I am aware of just two examples:

- CQL (i.e. CDuce Query Language)
- Xtatic (an extension of C#)

Paths and Regexp Patterns “coexist” but they are not integrated.
Mixing horizontal and vertical selectors

Several theoretical works from different areas about integrating vertical and horizontal exploration:

1. Unranked tree logics: e.g. Neven&Schwentick’s ETL.
2. Spatial modal logics: e.g. Cardelli&Ghelli’s TQL.
3. Query languages: e.g. Papakonstantinou&Vianu’s Loto-ql

But in running languages I am aware of just two examples:

1. CQL (i.e. CDuce Query Language)
2. Xtatic (an extension of C#)

Paths and Regexp Patterns “coexist” but they are not integrated.
Mixing horizontal and vertical selectors

Several theoretical works from different areas about integrating vertical and horizontal exploration:

1. *Unranked tree logics*: e.g. Neven & Schwentick’s ETL.

2. *Spatial modal logics*: e.g. Cardelli & Ghelli’s TQL.

3. *Query languages*: e.g. Papakonstantinou & Vianu’s Loto-ql

But in running languages I am aware of just two examples:

1. **CQL** (i.e. CDuce Query Language)

2. **Xtatic** (an extension of C#)

Paths and Regexp Patterns “coexist” but they are not integrated.
Mixing horizontal and vertical selectors

Several theoretical works from different areas about integrating vertical and horizontal exploration:

1. Unranked tree logics: e.g. Neven&Schwentick’s ETL.
2. Spatial modal logics: e.g. Cardelli&Ghelli’s TQL.
3. Query languages: e.g. Papakonstantinou&Vianu’s Loto-ql

But in running languages I am aware of just two examples:

1. CQL (i.e. CDuce Query Language)
2. Xtatic (an extension of C#)

Paths and Regexp Patterns “coexist” but they are not integrated.
Mixing horizontal and vertical selectors

Several theoretical works from different areas about integrating vertical and horizontal exploration:

1. *Unranked tree logics*: e.g. Neven&Schwentick’s ETL.
2. *Spatial modal logics*: e.g. Cardelli&Ghelli’s TQL.
3. *Query languages*: e.g. Papakonstantinou&Vianu’s Loto-ql

But in running languages I am aware of just two examples:

1. CQL (i.e. CDuce Query Language)
2. Xtatic (an extension of C#)

Paths and Regexp Patterns “coexist” but they are not integrated.
Mixing horizontal and vertical selectors

Several theoretical works from different areas about integrating vertical and horizontal exploration:

1. Unranked tree logics: e.g. Neven&Schwentick’s ETL.
2. Spatial modal logics: e.g. Cardelli&Ghelli’s TQL.
3. Query languages: e.g. Papakonstantinou&Vianu’s Loto-ql

But in running languages I am aware of just two examples:

1. CQL (i.e. CDuce Query Language)
2. Xtatic (an extension of C#)

Paths and Regexp Patterns “coexist” but they are not integrated.

Opportunity of collaboration between the database and the programming languages communities
Outline of the talk

1. An overview of regexp types/patterns
   - Patterns in functional languages
   - Patterns as types with variables
   - Regexp Patterns and types for XML

2. Eight reasons to consider regexp types/patterns
   - Classic usages of type systems
   - Efficient and type precise main memory execution
   - Secondary memory optimization

3. Conclusion.
Outline of the talk

1. An overview of regexp types/patterns
   - Patterns in functional languages
   - Patterns as types with variables
   - Regexp Patterns and types for XML

2. Eight reasons to consider regexp types/patterns
   - Classic usages of type systems
   - Efficient and type precise main memory execution
   - Secondary memory optimization

3. Conclusion.
Outline of the talk

1. An overview of regexp types/patterns
   - Patterns in functional languages
   - Patterns as types with variables
   - Regexp Patterns and types for XML

2. Eight reasons to consider regexp types/patterns
   - Classic usages of type systems
   - Efficient and type precise main memory execution
   - Secondary memory optimization

3. Conclusion.
Regular expression
Types and Patterns for XML
**Types & patterns: the functional languages perspective**

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of:

```plaintext
let x = fst(e) in
let y = snd(e) in (y,x)
```

with pattern one can write:

```plaintext
let (x,y) = e in (y,x)
```

which syntactic sugar for:

```plaintext
match e with (x,y) -> (y,x)
```

“match” is more interesting than “let”, since it can test several “|”-separated patterns.
Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of

```plaintext
let x = fst(e) in
let y = snd(e) in (y,x)
```

with pattern one can write

```plaintext
let (x,y) = e in (y,x)
```

which syntactic sugar for

```plaintext
match e with (x,y) -> (y,x)
```

"match" is more interesting than "let", since it can test several "|"-separated patterns.
Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of

\[
\begin{align*}
\text{let } x &= \text{fst}(e) \text{ in } \\
\text{let } y &= \text{snd}(e) \text{ in } (y,x)
\end{align*}
\]

with pattern one can write

\[
\text{let } (x,y) = e \text{ in } (y,x)
\]

which syntactic sugar for

\[
\text{match } e \text{ with } (x,y) \rightarrow (y,x)
\]

"match" is more interesting than "let", since it can test several "|"-separated patterns.
Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of

```ml
let x = fst(e) in
let y = snd(e) in (y,x)
```

with pattern one can write

```ml
let (x,y) = e in (y,x)
```

which syntactic sugar for

```ml
match e with (x,y) -> (y,x)
```

"match" is more interesting than "let", since it can test several "|"-separated patterns.
Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**

Instead of

```plaintext
let x = fst(e) in
let y = snd(e) in (y,x)
```

with pattern one can write

```plaintext
let (x,y) = e in (y,x)
```

which syntactic sugar for

```plaintext
match e with (x,y) -> (y,x)
```

“**match**” is more interesting than “**let**”, since it can test several “**|**”-separated patterns.
Example: tail-recursive version of length for lists:

```plaintext
type List = (Any,List) | 'nil

fun length (x:(List,Int)):Int =
  match x with
  | ('nil , n) -> n
  | ((_ , t), n) -> length(t,n+1)
```

So patterns are values with

But if we:

1. use for types the same constructors as for values
2. use values to denote singleton types
3. consider the wildcard "_" as synonym of Any
Example: tail-recursive version of length for lists:

```haskell
type List = (Any, List) | 'nil

fun length (x:(List, Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_, t), n) -> length(t, n+1)
```

So patterns are values with

But if we:

1. use for types the same constructors as for values (e.g. \((s, t)\) instead of \(s \times t\))
2. use values to denote singleton types (e.g. ‘nil in the list type);
3. consider the wildcard "\_" as synonym of Any

G. Castagna 8/28
Example: tail-recursive version of length for lists:

```haskell
import 

type List = (Any, List) | 'nil

fun length (x:(List, Int)): Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t, n+1)
```

So patterns are values with

But if we:

1. use for types the same constructors as for values
2. use values to denote singleton types
3. consider the wildcard "_" as synonym of Any
Example: tail-recursive version of `length` for lists:

```haskell
type List = (Any,List) | 'nil

fun length (x:(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | (_,t), n) -> length(t,n+1)
```

So patterns are values with

But if we:

1. use for types the same constructors as for values
2. use values to denote singleton types
3. consider the wildcard “?” as synonym of `Any`
Example: tail-recursive version of length for lists:

```haskell
type List = (Any,List) | ‘nil

fun length (x:(List,Int)) : Int =
    match x with
    | (‘nil , n) -> n
    | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables,

But if we:

1. use for types the same constructors as for values
2. use values to denote singleton types
3. consider the wildcard "_" as synonym of Any
Example: tail-recursive version of length for lists:

```
type List = (Any,List) | 'nil

fun length (x:(List,Int)):Int =
    match x with
    | ('nil , n) -> n
    | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards,

But if we:

1. use for types the same constructors as for values
2. use values to denote singleton types
3. consider the wildcard "_" as synonym of Any
Example: tail-recursive version of length for lists:

```haskell
type List = (Any,List) | 'nil

fun length (x:(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

- use for types the same constructors as for values (e.g. `(s,t)` instead of `s × t`)
- use values to denote singleton types
- consider the wildcard "underscore" as synonym of Any
Example: tail-recursive version of `length` for lists:

```haskell
type List = (Any,List) | 'nil

fun length (x:(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_ , t), n) -> length(t,n+1)
```

So patterns are values with **capture variables, wildcards, constants**.

**But if we:**

1. use for types the same constructors as for values (e.g. `(s,t)` instead of `s × t`)
2. use values to denote singleton types (e.g. `'nil` in the list type);
3. consider the wildcard "_" as synonym of `Any`
Example: tail-recursive version of length for lists:

```haskell
type List = (Any, List) | 'nil

fun length (x:(List, Int)): Int =
  match x with
  | ('nil , n) -> n
  | ((_ , t), n) -> length(t, n+1)
```

So patterns are values with capture variables, wildcards, constants.

**But if we:**

1. **use for types the same constructors as for values** (e.g. \(s, t\) instead of \(s \times t\))
2. **use values to denote singleton types** (e.g. ‘nil in the list type);
3. **consider the wildcard ‘_’ as synonym of Any**
Example: tail-recursive version of length for lists:

```haskell
type List = (Any, List) | 'nil

fun length (x: (List, Int)): Int =
    match x with
    | ('nil, n) -> n
    | (_, t), n) -> length(t, n+1)
```

So patterns are values with capture variables, wildcards, constants.

But if we:

1. use for types the same constructors as for values (e.g. (s, t) instead of s × t)
2. use values to denote singleton types (e.g. 'nil in the list type);
3. consider the wildcard "_" as synonym of Any
Example: tail-recursive version of `length` for lists:

```haskell
type List = (Any,List) | 'nil

fun length (x:(List,Int)):Int =
  match x with
  | ('nil , n) -> n
  | ((_ ,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

**But if we:**

1. **use for types the same constructors as for values**  
   (e.g. `(s,t)` instead of `s \times t`)

2. **use values to denote singleton types**  
   (e.g. `'nil` in the list type);

3. **consider the wildcard “_” as synonym of Any**
Example: tail-recursive version of length for lists:

```haskell
type List = (Any,List) | 'nil

fun length (x:(List,Int)):Int =
    match x with
    | ('nil , n) -> n
    | ((_,t), n) -> length(t,n+1)
```

So patterns are values with **capture variables, wildcards, constants**.

**But if we:**

1. **use for types the same constructors as for values**
   (e.g. \((s,t)\) instead of \(s \times t\))

2. **use values to denote singleton types**
   (e.g. ‘nil in the list type);

3. **consider the wildcard “_” as synonym of Any**
Example: tail-recursive version of length for lists:

```
type List = (Any,List) | 'nil

fun length (x:(List,Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

So patterns are values with **capture variables, wildcards, constants.**

**But if we:**

1. **use for types the same constructors as for values**
   (e.g. \((s,t)\) instead of \(s \times t\))

2. **use values to denote singleton types**
   (e.g. ‘nil in the list type);

3. **consider the wildcard “\(\_\)” as synonym of Any**
Example: tail-recursive version of \texttt{length} for lists:

\begin{verbatim}
type List = (\textbf{Any},\text{List}) | 'nil

fun length (x:(\text{List},\text{Int})):\text{Int} =
  match x with
  | ('nil , n) -> n
  | ((\_,t), n) -> length(t,n+1)
\end{verbatim}

So patterns are values with capture variables, wildcards, constants.

But if we:

1. use for types the same constructors as for values (e.g. $(s,t)$ instead of $s \times t$)
2. use values to denote singleton types (e.g. ‘\texttt{nil} in the list type);
3. consider the wildcard “\_” as synonym of \textbf{Any}
Example: tail-recursive version of length for lists:

type List = (Any,List) | ‘nil

fun length (x:(List,Int)):Int =
  match x with
  | (‘nil , n) -> n
  | ((_ , t), n) -> length(t,n+1)

So patterns are values with capture variables, wildcards, constants.

Key idea behind regular patterns

Patterns are types with capture variables
Example: tail-recursive version of length for lists:

```haskell
type List = (Any,List) | ‘nil

fun length (x:(List,Int)):Int =
  match x with
  | (‘nil , n) -> n
  | ((_,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

**Key idea behind regular patterns**

Patterns are types with capture variables
Example: tail-recursive version of `length` for lists:

```haskell
type List = (Any,List) | 'nil

fun length (x:(List,Int)) : Int = 
  match x with
  | ('nil , n) -> n
  | ((__,t), n) -> length(t,n+1)
```

So patterns are values with capture variables, wildcards, constants.

### Key idea behind regular patterns

Patterns are types with capture variables

Define types: patterns come for free.
Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\):

```plaintext
type List = (Any,List) | 'nil

fun length (x :(List,Int)):Int =
    match x with
    | ('nil', n) -> n
    | ((_,t), n) -> length(t,n+1)
```
Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (\(|\)\)), intersections (\(\&\)) and differences (\(\backslash\)):

type List = (Any,List) | ‘nil

fun length (x :(List,Int)) : Int =
    match x with
    | (‘nil , n) -> n
    | ((_,t), n) -> length(t,n+1)
Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\):

type List = (Any,List) | ‘nil

fun length (x :(List,Int)): Int =
    match x with
    | (‘nil , n) -> n
    | ((_,t), n) -> length(t,n+1)
Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\): 

\[
\text{type List} = (\text{Any,} \text{List}) \mid \text{‘nil}
\]

\[
\text{fun length (x :(List,Int)) : Int} = \\
\text{match x with} \\
\mid (\text{‘nil} , n) \rightarrow n \\
\mid ((_,t), n) \rightarrow \text{length}(t,n+1)
\]

To type this function we need basic types products, singletons,…

\[
t ::= \text{Int} \mid \nu \mid (t,t) \mid
\]
Which types should we start from?

Patterns are tightly connected to boolean type constructors, that is unions (|), intersections (&) and differences (\):

\[
type \text{List} = (\text{Any},\text{List}) | '\text{nil}
\]

\[
\text{fun length } (x : (\text{List},\text{Int})) : \text{Int} =
\text{match } x \text{ with}
| ('\text{nil} , n) \rightarrow n
| ((\_ , t), n) \rightarrow \text{length}(t,n+1)
\]

To type this function we need basic types products, singletons,…

\[
t ::= \text{Int} | v | (t, t) | t|t | t&t | t\setminus t | \text{Empty} | \text{Any}
\]

but also boolean type constructors.
Which types should we start from?

type List = (Any, List) | 'nil

fun length (x :(List, Int)) : Int =
  match x with
  | ('nil , n) -> n
  | ((_,t), n) -> length(t, n+1)

To type this function we need basic types products, singletons,…

t ::= Int | v | (t, t) | t|t | t&t | t\t | Empty | Any

but also boolean type constructors.

Let us type the function.
Which types should we start from?

\[ t = \{ v \mid v \text{ value of type } t \} \]

type List = (Any,List) | ‘nil

fun length (x :(List,Int)):Int = 
    match x with 
    | (‘nil , n) -> n 
    | ((_,t), n) -> length(t,n+1)
Which types should we start from?

\[ t = \{ v \mid v \text{ value of type } t \}, \quad \text{and} \quad \\{p\} = \{ v \mid v \text{ matches pattern } p \} \]

(type this is a type)

```haskell
type List = (Any,List) | 'nil

fun length (x :(List,Int)):Int =
  match x with
  | ('nil , n) -> n
  | ((_ ,t), n) -> length(t,n+1)
```
Which types should we start from?

\[ t = \{ v \mid v \text{ value of type } t \} \quad \text{and} \quad \{p\} = \{ v \mid v \text{ matches pattern } p \} \]

\[ \text{type List} = (\text{Any}, \text{List}) \mid \text{‘nil} \]

\[ \text{fun length (x : (List, Int)) : Int =} \]
\[ \quad \text{match x with} \]
\[ \quad \mid (\text{‘nil , n}) \rightarrow n \]
\[ \quad \mid ((_, t), n) \rightarrow \text{length}(t, n+1) \]

The first branch is executed only for values and are both in (List, Int) and in \( \{ (\text{‘nil}, n) \} \)
Which types should we start from?

\[ t = \{ v \mid v \text{ value of type } t \} \text{ and } \{ p \} = \{ v \mid v \text{ matches pattern } p \} \]


\[
\text{type List} = (\text{Any, List}) | \text{`nil} \\
\text{fun length (x : (List, Int)): Int} = \\
\quad \text{match x with} \\
\quad \quad | (\text{`nil, n}) \to n \\
\quad \quad | ((_, t), n) \to \text{length(t, n+1)}
\]

The first branch is executed only for values and are both in 

\((\text{List, Int}) \text{ and in } \{ (\text{`nil, n}) = (\text{`nil, Any) }\)
Which types should we start from?

\[
t = \{ v \mid v \text{ value of type } t \} \text{ and } \mathcal{L}_p = \{ v \mid v \text{ matches pattern } p \}
\]

(type List = (Any,List) | ‘nil)

fun length (x :(List,Int)): Int =
match x with
| (‘nil , n) -> n
| (__,t), n) -> length(t,n+1)

The first branch is executed only for values and are both in (List,Int) and in \(\mathcal{L}_p\) in \(\mathcal{L}_p\) of (‘nil,Any)
Which types should we start from?

\[ t = \{ v \mid v \text{ value of type } t \} \quad \text{and} \quad \{ p \} = \{ v \mid v \text{ matches pattern } p \} \]

(type List = (Any,List) | ‘nil

fun length (x : (List,Int)) : Int =
match x with
| (‘nil, n) -> n
| ((_,t), n) -> length(t, n+1)

The first branch is executed only for values and are both in (List,Int) and in \( \{ (\text{‘nil,n) \} \}

\text{(List,Int) and (‘nil,Any) = (‘nil,Int)}
Which types should we start from?

\[ t = \{ v \mid v \text{ value of type } t \} \quad \text{and} \quad \{ p \} = \{ v \mid v \text{ matches pattern } p \} \]

(type List = (Any,List) | ‘nil)

\[
\text{fun length (x :(List,Int)) : Int = match x with }
\]
\[
| ('nil , n) -> n \quad \text{Int}
\]
\[
| (_,t), n) -> length(t,n+1)
\]

The first branch is executed only for values and are both in (List,Int) and in \{('nil,n)\} (List,Int) & ('nil,Any) = ('nil,Int)
Which types should we start from?

\[ t = \{ v \mid v \text{ value of type } t \} \text{ and } \{ p \} = \{ v \mid v \text{ matches pattern } p \} \]

(type List = (Any,List) | ‘nil)

fun length (x :(List,Int)): Int =
    match x with
    | (‘nil, n) -> n
    | ((_,t), n) -> length(t,n+1)

The second branch is executed for values that are in (List,Int) not in \{ (‘nil, n) \} and in \{ ((_,t), n) \}.
Which types should we start from?

\[
t = \{ v \mid v \text{ value of type } t \} \quad \text{and} \quad \{ p \} = \{ v \mid v \text{ matches pattern } p \}
\]

(type List = (Any,List) | ‘nil

fun length (x :(List,Int)): Int =
    match x with
    | (‘nil , n) -> n
    | ((_,t), n) -> length(t,n+1)

The second branch is executed for values that are in
(List,Int) not in \{ ‘nil,n \} and in \{ (_,t),n \}

((List,Int) \ (‘nil,Any))&((Any,Any),Any)
Which types should we start from?

t = \{ v \mid v \text{ value of type } t \} \text{ and } \{p\} = \{ v \mid v \text{ matches pattern } p \}

(type List = (Any,List) \mid ‘nil)

fun length (x :(List,Int)) : Int =
  match x with
  | (‘nil , n) -> n \hspace{1cm} \text{Int}
  | ((_,t), n) -> length(t,n+1)

The second branch is executed for values that are in (List,Int) \textbf{not in } \{\{‘nil,n\}\} \textbf{and in } \{((_,t),n)\}

\{(List,Int)\setminus\{‘nil,Any)\&((Any,Any),Any) = ((Any,List),Int)\}
Which types should we start from?

\[ t = \{ v \mid v \text{ value of type } t \} \quad \text{and} \quad \{ p \} = \{ v \mid v \text{ matches pattern } p \} \]

(type List = (Any,List) | ‘nil)

fun length (x :(List,Int)) : Int =
  match x with
  | (‘nil , n) -> n Int
  | ((_,t), n) -> length(t,n+1) Int

The second branch is executed for values that are in (List,Int) not in \{ (‘nil,n) \} and in \{ ((_,t),n) \}

((List,Int) \ (‘nil,Any)) & ((Any,Any),Any) = ((Any,List),Int)
Which types should we start from?

\[ t = \{v \mid v \text{ value of type } t\} \quad \text{and} \quad \{p\} = \{v \mid v \text{ matches pattern } p\} \]

(type List = (Any,List) | ‘nil

fun length (x :(List,Int)): Int =
    match x with
    | (‘nil , n) -> n Int
    | ((_,t), n) -> length(t,n+1) Int

The match expression has type the union of the possible results
Which types should we start from?

\[ t = \{ v \mid v \text{ value of type } t \} \text{ and } \mathcal{P} = \{ v \mid v \text{ matches pattern } p \} \]

(type List = (Any,List) | ‘nil)

fun length (x : (List,Int)) : Int =
match x with
| (‘nil , n) -> n
| ((_,t), n) -> length(t,n+1)

The match expression has type the union of the possible results

Int | Int
Which types should we start from?

\[ t = \{v \mid \text{value of type } t\} \quad \text{and} \quad \mathcal{L}_p = \{v \mid v \text{ matches pattern } p\} \]

\[
\text{type List} = (\text{Any, List}) \mid \text{`nil}
\]

\[
\text{fun length} \ (x : (\text{List, Int})) : \text{Int} = \\
\quad \text{match } x \text{ with} \\
\quad \quad | (\text{`nil , n}) \rightarrow n \\
\quad \quad | ((\_ , t), n) \rightarrow \text{length}(t, n+1)
\]

The match expression has type the union of the possible results

\[ \text{Int} \mid \text{Int} = \text{Int} \]
Which types should we start from?

\[
t = \{ v \mid v \text{ value of type } t \} \quad \text{and} \quad \mathcal{L}p = \{ v \mid v \text{ matches pattern } p \}
\]

(type List = (Any,List) | ‘nil

fun length (x :(List,Int)):Int =
  match x with
  | (‘nil , n) -> n
  | ((_,t), n) -> length(t,n+1)

The match expression has type the union of the possible results

Int | Int = Int

The function is well-typed
Unions, intersections, differences

- **Boolean operators are needed to type pattern matching:**

  \[
  \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
  \]

  - To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \{ p_1 \} \) (where \( e : t \));
  - To infer the type \( t_2 \) of \( e_2 \) we need \( (t \setminus \{ p_1 \}) \& \{ p_2 \} \);
  - The type of the match is \( t_1 \mid t_2 \).

- **Boolean type constructors are useful for programming:**

  \[
  \text{map catalogue with}
  \]

  \[
  x :: (\text{Car} \& (\text{Guaranteed} \mid (\text{Any} \\setminus \text{Used}))) \rightarrow x
  \]

  Select in `catalogue` all cars that if used then are guaranteed.
Unions, intersections, differences

- **Boolean operators are needed to type pattern matching:**
  
  \[
  \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
  \]

  - To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \{ p_1 \} \) (where \( e : t \));
  
  - To infer the type \( t_2 \) of \( e_2 \) we need \( (t \setminus \{ p_1 \}) \& \{ p_2 \} \);
  
  - The type of the match is \( t_1 \mid t_2 \).

- **Boolean type constructors are useful for programming:**

  \[
  \text{map catalogue with }
  x :: (\text{Car} \& (\text{Guaranteed} \mid (\text{Any} \setminus \text{Used}))) \rightarrow x
  \]

  Select in \( \text{catalogue} \) all cars that if used then are guaranteed.
Unions, intersections, differences

- **Boolean operators are needed to type pattern matching:**

  \[
  \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
  \]

  - To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \emptyset p_1 \emptyset \) (where \( e : t \));
  - To infer the type \( t_2 \) of \( e_2 \) we need \( (t \setminus \emptyset p_1 \emptyset) \& \emptyset p_2 \emptyset \);
  - The type of the match is \( t_1 \mid t_2 \).

- **Boolean type constructors are useful for programming:**

  \[
  \text{map catalogue with } x :: (\text{Car} \& (\text{Guaranteed} \mid (\text{Any} \setminus \text{Used}))) \rightarrow x
  \]

  Select in \textit{catalogue} all cars that if used then are guaranteed.
Boolean operators are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

- To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \{ p_1 \} \) (where \( e : t \));
- To infer the type \( t_2 \) of \( e_2 \) we need \( (t \setminus \{ p_1 \}) \& \{ p_2 \} \);
- The type of the match is \( t_1 \mid t_2 \).

Boolean type constructors are useful for programming:

\[
\text{map catalogue with} \\
\text{ x :: (Car & (Guaranteed|Any\Used)) -> x} \\
\text{ Select in catalogue all cars that if used then are guaranteed.}
\]
1. Introduction
2. XML regexp types/patterns
3. Properties of regexp types/patterns
4. Conclusion

Unions, intersections, differences

- **Boolean operators are needed to type pattern matching:**
  \[
  \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
  \]

  - To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \left\llbracket p_1 \right\rrbracket \) (where \( e : t \));
  - To infer the type \( t_2 \) of \( e_2 \) we need \((t \setminus \left\llbracket p_1 \right\rrbracket) \& \left\llbracket p_2 \right\rrbracket\);
  - The type of the match is \( t_1 \mid t_2 \).

- **Boolean type constructors are useful for programming:**
  \[
  \text{map catalogue with }
  \quad x :: (\text{Car } \& (\text{Guaranteed} \mid (\text{Any} \setminus \text{Used}))) \rightarrow x
  \]

  Select in `catalogue` all cars that if used then are guaranteed.
### Unions, intersections, differences

- **Boolean operators are needed to type pattern matching:**

  ```
  match e with p_1 -> e_1 | p_2 -> e_2
  ```

  - To infer the type $t_1$ of $e_1$ we need $t & \{ p_1 \}$ (where $e : t$);
  - To infer the type $t_2$ of $e_2$ we need $(t \setminus \{ p_1 \}) \& \{ p_2 \}$;
  - The type of the match is $t_1 | t_2$.

- **Boolean type constructors are useful for programming:**

  ```
  map catalogue with
  x :: (Car & (Guaranteed | (Any \ Used))) -> x
  ```

  Select in `catalogue` all cars that if used then are guaranteed.
Unions, intersections, differences

- **Boolean operators are needed to type pattern matching:**
  
  \[
  \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
  \]
  
  - To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \{ p_1 \} \) (where \( e : t \));
  - To infer the type \( t_2 \) of \( e_2 \) we need \( (t \setminus \{ p_1 \}) \& \{ p_2 \} \);
  - The type of the match is \( t_1 \mid t_2 \).

- **Boolean type constructors are useful for programming:**
  
  \[
  \text{map catalogue with}
  \]
  
  \[
  x :: (\text{Car} \& (\text{Guaranteed} \mid (\text{Any} \setminus \text{Used}))) \rightarrow x
  \]
  
  Select in `catalogue` all cars that if used then are guaranteed. 
Boolean operators are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

- To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \{ p_1 \} \) (where \( e : t \));
- To infer the type \( t_2 \) of \( e_2 \) we need \( (t \setminus \{ p_1 \}) \& \{ p_2 \} \);
- The type of the match is \( t_1 \mid t_2 \).

Boolean type constructors are useful for programming:

\[
\text{map catalogue with }
\text{x :: (Car & (Guaranteed|(Any\setminus Used))) \rightarrow x}
\]

Select in \textit{catalogue} all cars that if used then are guaranteed.

Roadmap to extend it to XML:
1. Define types for XML documents,
2. Add boolean type constructors,
3. Define patterns as types with capture variables
Unions, intersections, differences

**Boolean operators are needed to type pattern matching:**

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

- To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \{ p_1 \} \) (where \( e : t \));
- To infer the type \( t_2 \) of \( e_2 \) we need \( (t \setminus \{ p_1 \}) \& \{ p_2 \} \);
- The type of the match is \( t_1 \mid t_2 \).

**Boolean type constructors are useful for programming:**

\[
\text{map catalogue with } \quad x :: \text{Car} \& (\text{Guaranteed} \mid (\text{Any} \setminus \text{Used})) \rightarrow x
\]

Select in \textit{catalogue} all cars that if used then are guaranteed.

**Roadmap to extend it to XML:**

1. Define types for XML documents,
2. Add boolean type constructors,
3. Define patterns as types with capture variables
Unions, intersections, differences

- **Boolean operators are needed to type pattern matching:**
  
  \[
  \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
  \]

  - To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \{ p_1 \} \) (where \( e : t \));
  - To infer the type \( t_2 \) of \( e_2 \) we need \( (t \setminus \{ p_1 \}) \& \{ p_2 \} \);
  - The type of the match is \( t_1 | t_2 \).

- **Boolean type constructors are useful for programming:**
  
  \[
  \text{map } \text{catalogue with}
  \]

  \[
  x :: \text{(Car} \& \text{(Guaranteed|Any\setminus\text{Used})} \rightarrow x
  \]

  Select in \text{catalogue} all cars that if used then are guaranteed.

**Roadmap to extend it to XML:**

1. Define types for XML documents,
2. Add boolean type constructors,
3. Define patterns as types with capture variables.
Unions, intersections, differences

- **Boolean operators are needed to type pattern matching:**
  
  \[
  \text{match } e \text{ with } p_1 \rightarrow e_1 | p_2 \rightarrow e_2
  \]

  - To infer the type \( t_1 \) of \( e_1 \) we need \( t \& \{ p_1 \} \) (where \( e : t \));
  - To infer the type \( t_2 \) of \( e_2 \) we need \( (t \setminus \{ p_1 \}) \& \{ p_2 \} \);
  - The type of the match is \( t_1 | t_2 \).

- **Boolean type constructors are useful for programming:**

  \[
  \text{map catalogue with } \quad x :: (\text{Car} & (\text{Guaranteed} | (\text{Any} \setminus \text{Used}))) \rightarrow x
  \]

  Select in `catalogue` all cars that if used then are guaranteed.

**Roadmap to extend it to XML:**

1. Define types for XML documents,
2. Add boolean type constructors,
3. Define patterns as types with capture variables
XML Types Example: A bibliography

```xml
 type Bib = <bib>[Book*]
 type Book = <book year=String>[Title  
                        (Author+ | Editor+ )  
                        Price?  
                        Publisher]
 type Author = <author>[Last First]
 type Editor = <editor>[Last First]
 type Title = <title>[PCDATA]
 type Last = <last>[PCDATA]
 type First = <first>[PCDATA]
 type Publisher = String
 type Price = <price>[PCDATA]
```

This and: singletons, intersections, differences, Empty, and Any.
XML Types Example: A bibliography

type Bib = <bib>[Book*]

type Book = <book year=String>[Title
(Author+ | Editor+ )
Price?
Publisher]

type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Publisher = String
type Price = <price>[PCDATA]
XML Types Example: A bibliography

type Bib = <bib>[Book*]
type Book = <book year=String>[attribute types
   Title
   (Author+ | Editor+ )
   Price?
   Publisher]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Publisher = String
type Price = <price>[PCDATA]

XML Types Example: A bibliography

type Bib = <bib>[Book*]
type Book = <book year=String>[Title
   (Author+ | Editor+ )
   Price?
   Publisher]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Publisher = String
type Price = <price>[PCDATA]
XML Types Example: A bibliography

type Bib = <bib>[Book*]
type Book = <book year=String>[Title
  (Author+ | Editor+ ) unions
  Price?
  Publisher]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Publisher = String
type Price = <price>[PCDATA]

This and: singletons, intersections, differences, Empty, and And.
XML Types Example: A bibliography

type Bib = <bib>[Book*]
type Book = <book year=String>[ Title 
  (Author+ | Editor+ ) 
  Price? optional elems 
  Publisher] 
type Author = <author>[Last First] 
type Editor = <editor>[Last First] 
type Title = <title>[PCDATA] 
type Last = <last>[PCDATA] 
type First = <first>[PCDATA] 
type Publisher = String 
type Price = <price>[PCDATA]
XML Types Example: A bibliography

type Bib = <bib>[Book*]
type Book = <book year=String>[Title
  (Author+ | Editor+ )
  Price?
  Publisher] mixed content

type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Publisher = String
type Price = <price>[PCDATA]

This and: singletons, intersections, differences, Empty, and Any
XML Types Example: A bibliography

```xml
type Bib = <bib>[Book*]
type Book = <book year=String>[Title
    (Author+ | Editor+ )
    Price?
    Publisher]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Publisher = String
type Price = <price>[PCDATA]
```

This and: singletons, intersections, differences, Empty, and Any.
Patterns = Types + Capture variables
Patterns

Patterns = Types + Capture variables

type Bib = <bib>[Book*]
Patterns

Patterns = Types + Capture variables

TYPES

type Bib = <bib>[Book*]

PATTERNS

<bib>[x::Book*]
Patterns = Types + Capture variables

**Types**

\[ \text{type Bib = } <\text{bib}>[\text{Book}*] \]

**Patterns**

\[ <\text{bib}>[x::\text{Book}^*] \]

The pattern binds \( x \) to the \textit{sequence} of all books in the bibliography.
Patterns

Patterns = Types + Capture variables

**TYPES**

```xml
type Bib = <bib>[Book*]
```

**PATTERNS**

```xml
match bibs with
  <bib>[x::Book*] -> x
```
Patterns

Patterns = Types + Capture variables

**TYPES**

type Bib = <bib>[Book*]

**PATTERNS**

match bibs with

  <bib>[x::Book*] -> x

Returns the content of bibs.
Patterns

Patterns = Types + Capture variables

type Bib = <bib>[Book*]

<bib>[\( \begin{array}{c}
  x::<\text{book year="2005"}>_ | y::_ \\
\end{array} \)\]*]
Patterns

Patterns = Types + Capture variables

**TYPES**

type Bib = <bib>[Book*]

**PATTERNS**

<bib>[( x:<book year="2005">_ | y:_ )]*

Binds $x$ to the sequence of all this year’s books, and $y$ to all the other books.
Patterns

Patterns = Types + Capture variables

**TYPES**

```plaintext
type Bib = <bib>[Book*]
```

**PATTERNS**

```plaintext
match bibs with
  <bib>[( x::<book year="2005">_ | y::_ )*] -> x@y
```
Patterns

Patterns = Types + Capture variables

type Bib = <bib>[Book*]

match bibs with

  <bib>[( x::<book year="2005">_ | y::_ )*] -> x@y

Returns the concatenation (i.e., “@”) of the two captured sequences
Patterns

Patterns = Types + Capture variables

**TYPES**

type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]

**PATTERNS**

<bib>[(x::<book year="1990">[ _* Publisher"ACM"] | _)*]
Patterns

Patterns = Types + Capture variables

TYPES

type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]

PATTERNS

<bib>[(x::<book year="1990">[ _* Publisher\"ACM"] | _)*]

Binds x to the sequence of books published in 1990 from publishers others than “ACM” and discards all the others.
Patterns

Patterns = Types + Capture variables

TYPES

type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]

PATTERNS

match bibs with
  <bib>[(x::<book year="1990">[ _* Publisher"ACM" ] | _)*] -> x
Patterns

Patterns = Types + Capture variables

**TYPES**

\[
\text{type Bib} = \langle \text{bib} \rangle [\text{Book}*] \\
\text{type Book} = \langle \text{book year=String} \rangle [\text{Title Author+ Publisher}]
\]

**PATTERNS**

\[
\text{match bibs with} \\
\langle \text{bib} \rangle [(x::\langle \text{book year=\"1990\"}\rangle [ \_\_ \text{Publisher\"ACM\"}] | \_\_\_\_\_)*] \rightarrow x
\]

Returns all the captured books
Patterns

Patterns = Types + Capture variables

**Types**

type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]

**Patterns**

match bibs with
    <bib>[(x::<book year="1990">[* Publisher\"ACM"] | _)*)] -> x

Returns all the captured books

**Exact type inference:**

E.g.: if we match the pattern [(x::Int|_)*] against an expression of type [Int* String Int] the type deduced for x is [Int+]
Patterns

Patterns = Types + Capture variables

Types

type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]

Patterns

match bibs with
   <bib>[(x::<book year="1990">[ _) Publisher"ACM"] | _)*] -> x

Returns all the captured books

Exact type inference:
E.g.: if we match the pattern [(x::Int|_)*] against an expression of type [Int* String Int] the type deduced for x is [Int+]
Select-from-where

Instead of just variables

select $e$ from

$x_1$ in $e_1$

\ldots

$x_n$ in $e_n$

where $c$
Instead of just variables use patterns

\[
\text{select } e \text{ from } \\
p_1 \text{ in } e_1 \\
\vdots \\
p_n \text{ in } e_n \\
\text{where } c
\]
Select-from-where

**Instead of just variables use patterns**

```xml
select e from
  p_1 in e_1
  ...
  p_n in e_n
where c
```

```xml
<bib>[b::Book*]
<book year="1990">[ t::Title _+ <price>"69.99" ]
```

(1) captures in b all the books of a bibliography
(2) captures in t the title of a book if it is of 1990 and costs 69.99

**Biblio** = `<bib>[Book*]`
**Book** = `<book year=String>[Title (Author+|Editor+) Price?]`
Instead of just variables use patterns

```xml
select e from
  p₁ in e₁
  ...
  pₙ in eₙ
where c
```

```xml
select <book>t from
  <bib>[b::Book*] in bibs,
  <book year="1990">[ t::Title + <price>"69.99" ] in b
```

Biblio = <bib>[Book*]
Book = <book year=String>[Title (Author+|Editor+) Price?]
Select-from-where

Instead of just variables use patterns

\[
\text{select } e \text{ from } \\
p_1 \text{ in } e_1 \\
\vdots \\
p_n \text{ in } e_n \\
\text{where } c
\]

\[
\text{select } <\text{book}>t \text{ from } \\
<\text{bib}>[b::\text{Book}^*] \text{ in } \text{bibs}, \\
<\text{book year="1990"}>[ t::\text{Title} \_+ <\text{price}>"69.99" ] \text{ in } b
\]

Selects from \text{bibs} the titles of all books of 1990 and of price 69.99

\text{Biblio} = <\text{bib}>[\text{Book}^*] \\
\text{Book} = <\text{book year=String}>[\text{Title (Author+|Editor+) Price?}]
Instead of just variables use patterns

\[
\text{select } \ e \ \text{from}
\]
\[
\begin{align*}
& p_1 \ \text{in } e_1 \\
& \quad \vdots \\
& p_n \ \text{in } e_n \\
\text{where } c
\end{align*}
\]

fun getTitles(bibs : Biblio) : [(<book>[Title])\]*

\[
\text{select } <book>t \ \text{from}
\]
\[
\begin{align*}
& <bib>[b::Book*] \ \text{in } \ bibs, \\
& <book \ year="1990">[ t::Title \_+ <price>"69.99" ] \ \text{in } b
\end{align*}
\]

Selects from \( \text{bibs} \) the titles of all books of 1990 and of price 69.99
and has type \( \text{Biblio} \rightarrow [(<book>[Title])\]*\)

\( \text{Biblio} = <bib>[Book*] \)
\( \text{Book} = <book \ year=\text{String}>[Title \ (Author+|Editor+) \ Price?] \)
For instance in CQL (but see Xtatic for a very different encoding):

- All children of e with tag $\text{tag}$
  
  ```
  select x from `<_ ..>`[( x::(<$\text{tag}$ ..>_)|_)*)] in e
  ```

- All attributes labelled by $\text{id}$
  
  ```
  select x from `<_ $\text{id}$=x ..>` in e
  ```

- Notice that regexp patterns can define non-unary queries.
For instance in CQL (...but see Xtatic for a very different encoding):

- All children of e with tag tag
  
  \[
  \text{select x from } <\_ \ldots>[( x::(<\text{tag} \ldots>\_)|\_)*] \text{ in e}
  \]

- All attributes labelled by id
  
  \[
  \text{select x from } <\_ id=x \ldots> \text{ in e}
  \]

- Notice that regexp patterns can define non-unary queries.
XPath encoding

For instance in CQL (...but see Xtatic for a very different encoding):

- All children of $e$ with tag $tag$  
  \[
  \text{select } x \text{ from } < \ldots > [\text{<tag }\ldots>] \text{ in } e
  \]

- All attributes labelled by $id$  
  \[
  \text{select } x \text{ from } < \ldots \text{id=x} \ldots > \text{ in } e
  \]

Notice that regexp patterns can define non-unary queries.
For instance in CQL (…but see Xtatic for a very different encoding):

- All children of $e$ with tag $tag$ 
  $$\text{select } x \text{ from } <\_ \ldots>[ ( x::(<tag \ldots>\_)|\_ )\ast ] \text{ in } e$$

- All attributes labelled by $id$
  $$\text{select } x \text{ from } <\_ id=x \ldots> \text{ in } e$$

- Notice that regexp patterns can define non-unary queries.
For instance in CQL (…but see Xtatic for a very different encoding):

- All children of e with tag tag \((e/tag)\)
  
  \[
  \text{select } x \text{ from } \langle_\ldots>[\langle x::\langle tag \ldots \rangle\rangle|\_\_\_\_\_\_\_] \ast \text{ in } e
  \]

- All attributes labelled by id \((e/@id)\)
  
  \[
  \text{select } x \text{ from } \langle_\_id=x \ldots \rangle \text{ in } e
  \]

- Notice that regexp patterns can define non-unary queries.

Rationale

CQL, Xtatic, add syntactic sugar for XPath …
XPath encoding

For instance in CQL (…but see Xtatic for a very different encoding):

- All children of e with tag tag \((e/tag)\)
  
  select x from <__ ..>[( x::(<tag ..>_)|_ )*] in e

- All attributes labelled by id \((e/@id)\)
  
  select x from <__ id=x ..> in e

- Notice that regexp patterns can define non-unary queries.

Rationale

CQL, Xtatic, add syntactic sugar for XPath … but we need more
... it is all syntactic sugar!

### Types

\[ t ::= \text{Int} \mid \nu \mid (t, t) \mid t \lor t \mid t \land t \mid \neg t \mid \text{Any} \]

### Patterns

\[ p ::= t \mid x \mid (p, p) \mid p \lor p \mid p \land p \]

Example:

\[
\text{type Book} = \langle \text{book} \rangle \text{[Title (Author+|Editor+) Price?]}
\]

encoded as

\[
\text{Book} = (\text{'book}, (\text{Title}, X \lor Y))
\]
\[
X = (\text{Author}, X \lor (\text{Price}, \text{'nil}) \lor \text{'nil})
\]
\[
Y = (\text{Editor}, Y \lor (\text{Price}, \text{'nil}) \lor \text{'nil})
\]
... it is all syntactic sugar!

Types

\[ t ::= \text{Int} \mid \nu \mid (t, t) \mid t \lor t \mid t \land t \mid \neg t \mid \text{Any} \]

Patterns

\[ p ::= t \mid \chi \mid (p, p) \mid p \lor p \mid p \land p \]

Example:

\[
\text{type } \text{Book} = \langle \text{book}\rangle [\text{Title} \ (\text{Author}+|\text{Editor}+) \ \text{Price}?]\n\]

encoded as

\[
\text{Book} = (\text{book}, (\text{Title}, X \lor Y)) \\
X = (\text{Author}, X \lor (\text{Price}, \text{‘nil}) \lor \text{‘nil}) \\
Y = (\text{Editor}, Y \lor (\text{Price}, \text{‘nil}) \lor \text{‘nil})
\]
...it is all syntactic sugar!

### Types

\[ t ::= \text{Int} \mid \nu \mid (t, t) \mid t \lor t \mid t \land t \mid \neg t \mid \text{Any} \]

### Patterns

\[ p ::= t \mid x \mid (p, p) \mid p \lor p \mid p \land p \]

**Example:**

\[
\text{type Book} = \langle \text{book} \rangle [\text{Title (Author+|Editor+)} \text{ Price?}] \\
\]

coded as

\[
\begin{align*}
\text{Book} &= (\text{'book, (Title, X \lor Y)}) \\
X &= (\text{Author, X \lor (Price, 'nil) \lor 'nil}) \\
Y &= (\text{Editor, Y \lor (Price, 'nil) \lor 'nil})
\end{align*}
\]
... it is all syntactic sugar!

### Types

| \( t \) ::= | \( \text{Int} \) | \( \nu \) | \((t, t)\) | \( t \lor t \) | \( t \land t \) | \( \neg t \) | \( \text{Any} \) |
| --- | --- | --- | --- | --- | --- | --- |

### Patterns

<table>
<thead>
<tr>
<th>( p ) ::=</th>
<th>( t )</th>
<th>( \nu )</th>
<th>((p, p))</th>
<th>( p \lor p )</th>
<th>( p \land p )</th>
</tr>
</thead>
</table>

Example:

```plaintext
type Book = <book>[Title (Author+|Editor+) Price?]
```

encoded as

```plaintext
Book = ('book, (Title, X \lor Y))
X = (Author, X \lor (Price, 'nil) \lor 'nil)
Y = (Editor, Y \lor (Price, 'nil) \lor 'nil)
```
Some reasons to consider regular expression types and patterns
Some good reasons to consider regexp patterns/types

- Theoretical reason: very compact

- Eight practical reasons:
  1. Classic usage
  2. Informative error messages
  3. Error mining
  4. Efficient execution
  5. Logical optimisation of pattern-based queries
  6. Pattern matches as building blocks for iterators
  7. Type/pattern-based data pruning for memory usage optimisation
  8. Type based query optimisation
Some good reasons to consider regexp patterns/types

- Theoretical reason: very compact (≠ simple)

  - Classic usage
  - Informative error messages
  - Error mining
  - Efficient execution
  - Logical optimisation of pattern-based queries
  - Pattern matches as building blocks for iterators
  - Type/pattern-based data pruning for memory usage optimisation
  - Type-based query optimisation
Some good reasons to consider regexp patterns/types

- **Theoretical reason:** very compact (≠ simple)

- **Eight practical reasons:**
  1. Classic usage
  2. Informative error messages
  3. Error mining
  4. Efficient execution
  5. Logical optimisation of pattern-based queries
  6. Pattern matches as building blocks for iterators
  7. Type/pattern-based data pruning for memory usage optimisation
  8. Type-based query optimisation
Some good reasons to consider regexp patterns/types

- **Theoretical reason:** very compact (≠ simple)

- **Eight practical reasons:**
  1. Classic usage
  2. Informative error messages
  3. Error mining
  4. Efficient execution
  5. Logical optimisation of pattern-based queries
  6. Pattern matches as building blocks for iterators
  7. Type/pattern-based data pruning for memory usage optimisation
  8. Type-based query optimisation
1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

It is possible to specify constraints such as:

If the attribute a has value x, then e-elements that do not contain f-elements must contain two g-elements.

Types can be composed:

Then <\bib>((Biblio\&ThisYear)\&WithPrice)*\bib> defines a view containing only this year’s books that do not have price element.
1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:
  - If the attribute has value x, then all elements that do not contain f elements must contain two g elements.

- Types can be composed:
  - type WithPrice = ...
  - type ThisYear = ...
  - Then: $\langle (\text{Biblio} & \text{ThisYear}) \text{WithPrice} \rangle$ defines a view containing only this year’s books that do not have price element.
1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:
- It is possible to specify constraints such as:
  
  If the attribute `a` has value `x`, then `e`-elements that do not contain `f`-elements must contain two `g`-elements.

- Types can be composed:

```
type WithPrice = <...
  Price
...
>
type ThisYear = <
  year="2005"
>
```

then `'<Biblio&ThisYear> WithPrice`' defines a view containing only this year’s books that do not have price element.
1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:
  
  \[
  \text{If the attribute } a \text{ has value } x, \text{ then } e\text{-elements that do not contain } f\text{-elements must contain two } g\text{-elements.}
  \]

- Types can be composed:

  ```xml
  type WithPrice = <_ ..>[@* Price _*]
  type ThisYear = <_ year="2005">_
  then filter (ThisYear && WithPrice) defines a view containing only this year’s books that do not have price element.
  ```
1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:

  \textit{If the attribute a has value x, then e-elements that do not contain f-elements must contain two g-elements.}

- Types can be composed:

  \begin{verbatim}
  type WithPrice = <_ ..>[_* Price _*]
  type ThisYear = <_ year="2005">_
  then <bib>((Biblio&ThisYear)\WithPrice)*] defines a view containing only this year’s books that do not have price element.
  \end{verbatim}
1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:

  If the attribute $a$ has value $x$, then $e$-elements that do not contain $f$-elements must contain two $g$-elements.

- Types can be composed:

  type WithPrice = <__ ..>[[* Price _*]]
  type ThisYear = <__ year="2005">_

  then <bib>[(Biblio&ThisYear)\WithPrice)*] defines a view containing only this year’s books that do not have price element.
1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:
  
  *If the attribute a has value x, then e-elements that do not contain f-elements must contain two g-elements.*

- Types can be composed:
  
  ```xml
  type WithPrice = <_ ..>[_* Price _*]
  type ThisYear = <_ year="2005">_
  ```

  then `<bib>[((Biblio&ThisYear)\WithPrice)*]` defines a view containing only this year’s books that do not have price element.
1. Classic usages of types

Use these types as usual: static detection of errors, partial correctness, schema specification

Not much to say here, just notice that:

Singletons, unions, intersections, and differences have set-theoretic semantics on “types as set of values”: they are easy to understand.

A natural and powerful specification and constraint language:

- It is possible to specify constraints such as:
  
  \( \text{If the attribute } a \text{ has value } x, \text{ then } e\text{-elements that do not contain } f\text{-elements must contain two } g\text{-elements.} \)

- Types can be composed:
  
  \[
  \text{type WithPrice} = \langle \_ \_ \rangle[* \text{ Price } _*] \\
  \text{type ThisYear} = \langle \_ \_ \text{ year}="2005" \rangle._
  \]

  then \( \langle \text{bib} \rangle[(\text{Biblio} & \text{ThisYear}) \backslash \text{WithPrice})] \) defines a view containing only this year’s books that do not have price element.

Not very innovative but useful properties
2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

List of books of a given year, stripped of the Editors and Price
2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

List of books of a given year, stripped of the Editors and Price
2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

List of books of a given year, stripped of the Editors and Price

fun onlyAuthors (year:Int, books:[Book*]):[Book*] =

type Book = <book year=String>[Title (Author+|Editor+) Price?]
2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

List of books of a given year, stripped of the Editors and Price

fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
select <book year=y>(t@a) from
  <book year=y>[(t::Title | a::Author | _)+] in books
where int_of(y) = year

type Book = <book year=String>[Title (Author+|Editor+) Price?]
2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one.

List of books of a given year, stripped of the Editors and Price

fun onlyAuthors (year:Int,books:[Book*]):[Book*] = 
  select <book year=y>(t@a) from 
  <book year=y>[(t::Title | a::Author | _)+] in books 
  where int_of(y) = year

type Book = <book year=String>[Title (Author+|Editor+) Price?]
2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

List of books of a given year, stripped of the Editors and Price

```haskell
fun onlyAuthors (year:Int, books:[Book*]):[Book*] =
  select <book year=y>(t@a) from
  <book year=y>[(t::Title | a::Author | _)+] in books
  where int_of(y) = year
```

Returns the following error message:

Error at chars 81-83:
  select <book year=y>(t@a) from
This expression should have type:
[ Title (Editor+|Author+) Price? ]
but its inferred type is:
[ Title Author+ | Title ]
which is not a subtype, as shown by the sample:
[ <title>[][ ] ]

type Book = <book year=String>[Title (Author+|Editor+) Price?]
2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

List of books of a given year, stripped of the Editors and Price

```scala
fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
    select <book year=y>(t@a) from
    <book year=y>[(t::Title | a::Author | _)+] in books
    where int_of(y) = year
```

Returns the following error message:

Error at chars 81-83:
    select <book year=y>(t@a) from
This expression should have type:
    [ Title (Editor+|Author+) Price? ]
but its inferred type is:
    [ Title Author+ | Title ]
which is not a subtype, as shown by the sample:
    [ <title>[
    ] ]

```
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```
2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one.

List of books of a given year, stripped of the Editors and Price

```haskell
fun onlyAuthors (year:Int,books:[Book*]):[Book*] =
  select <book year=y>(t@a) from <book year=y>[(t::Title | a::Author | _)+] in books
  where int_of(y) = year
```

Returns the following error message:

Error at chars 81-83:

```
  select <book year=y>(t@a) from
```

This expression should have type:

```
[ Title (Editor+|Author+) Price?  ]
```

but its inferred type is:

```
[ Title Author+  | Title ]
```

which is not a subtype, as shown by the sample:

```
[ <title>[], [] ]
```

```
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```
2. Informative error messages

In case of error return a sample value in the difference of the inferred type and the expected one

List of books of a given year, stripped of the Editors and Price

```plaintext
fun onlyAuthors (year:Int,books:[Book*]):[Book*] = 
sel <book year=y>(t@a) from
  <book year=y>[ t::Title a::Author+ _* ] in books
where int_of(y) = year
```

Returns the following error message:

Error at chars 81-83:
  select <book year=y>(t@a) from
This expression should have type:
[ Title (Editor+|Author+) Price? ]
but its inferred type is:
[ Title Author+ | Title ]
which is not a subtype, as shown by the sample:
[ <title>[ ] ]

```plaintext
type Book = <book year=String>[Title (Author+|Editor+) Price?]
```
Fun extract(x: [Book*]) =
select (z,y) from
<book ..> [ z::Title y::(<author>_|<editor>_)+ _* ] in x

- Despite the typo the function is well-typed:
  - no typing rule is violated
  - the pattern is not useless, it can match authors
- They are not regexp-patterns specific:
- Such errors are not always typos: they can be conceptual errors.
Spot subtle errors that elude current type checking technology

fun extract(x:[Book*]) = 
  select (z,y) from 
  <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x

- Despite the typo the function is well-typed:
  - no typing rule is violated
  - the pattern is not useless, it can match authors
- They are not regexp-patterns specific:
  -bibbs/book/(title|prize)
- Such errors are not always typos; they can be conceptual errors.
3. Error mining

Spot subtle errors that elude current type checking technology

fun extract(x:[Book*]) =  
select (z,y) from  
<book ..> [ z::Title y::(<author>_|<editor>_)+ _* ] in x

- Despite the typo the function is well-typed:
  - no typing rule is violated
  - the pattern is not useless, it can match authors

- They are not regexp-patterns specific:
  - bibs/book/(title|prize)

- Such errors are not always typos; they can be conceptual errors.
3. Error mining

Spot subtle errors that elude current type checking technology

fun extract(x:[Book*]) = 
select (z,y) from
  <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x

- Despite the typo the function is well-typed:
  - no typing rule is violated
  - the pattern is not useless, it can match authors

- They are not regexp-patterns specific:
  
- Such errors are not always typos: they can be conceptual errors.
fun extract(x:[Book*]) = 
  select (z,y) from 
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x

- Despite the typo the function is well-typed:
  - no typing rule is violated
  - the pattern is not useless, it can match authors

- They are not regexp-patterns specific:
  bibs/book/(title|prize)

- Such errors are not always typos: they can be conceptual errors.
3. Error mining

Spot subtle errors that elude current type checking technology

fun extract(x: [Book*]) = 
  select (z,y) from 
  <book ..>[ z::Title y::(<author>_.|<editor>_.)+ _* ] in x

- Despite the typo the function is well-typed:
  - no typing rule is violated
  - the pattern is not useless, it can match authors

- They are not regexp-patterns specific:
  bibs/book/(title|prize)

- Such errors are not always typos: they can be conceptual errors.
3. Error mining

Spot subtle errors that elude current type checking technology

```haskell
fun extract(x:[Book*]) = select (z,y) from
    <book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x
```

- Despite the typo the function is well-typed:
  - no typing rule is violated
  - the pattern is not useless, it can match authors

- They are not regexp-patterns specific:
  ```
  bibs/book/(title|prize)
  ```

- Such errors are not always typos: they can be conceptual errors.
3. Error mining

Spot subtle errors that elude current type checking technology

fun extract(x:[Book*]) =  
select (z,y) from  
<book ..>[ z::Title y::(<author>_|<editor>_)+ _* ] in x

- Despite the typo the function is well-typed:  
  - no typing rule is violated  
  - the pattern is not useless, it can match authors

- They are not regexp-patterns specific:  
  bibs/book/(title|prize)

- Such errors are not always typos: they can be conceptual errors.

Can be formally characterised and statically detected by the types/patterns presented here and integrated in current regexp type-checkers with no overhead
4. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don’t test it.

type A = <a>[A*]
type B = <b>[B*]
4. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don’t test it.

type A = <a>[A*]
type B = <b>[B*]
4. Efficient execution

Use static type information to perform an optimal set of tests

**Idea:** if types tell you that something cannot happen, don’t test it.

\[
\text{type } A = \langle a \rangle [A^*] \\
\text{type } B = \langle b \rangle [B^*]
\]
4. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don’t test it.

type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with A -> 1 | B -> 0
4. Efficient execution

Use static type information to perform an optimal set of tests

**Idea:** if types tell you that something cannot happen, don’t test it.

```latex
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with A -> 1 | B -> 0
```

G. Castagna
4. Efficient execution

Use static type information to perform an optimal set of tests

Idea: if types tell you that something cannot happen, don’t test it.

\[
\begin{align*}
type \ A &= \langle a \rangle [A^*] \\
type \ B &= \langle b \rangle [B^*] \\
\text{fun check}(x: A|B) &= \text{match } x \text{ with } \begin{cases} A &\to 1 \mid B &\to 0 \end{cases} \\
\text{fun check}(x: A|B) &= \text{match } x \text{ with } \begin{cases} \langle a \rangle &\to 1 \mid \_ &\to 0 \end{cases}
\end{align*}
\]
4. Efficient execution

**Idea:** if types tell you that something cannot happen, don’t test it.

```haskell
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with A -> 1 | B -> 0
fun check(x : A|B) = match x with <a>_ -> 1 | _ -> 0
```

- No backtracking.
- Whole parts of the matched data are not checked.
4. Efficient execution

Use static type information to perform an optimal set of tests

**Idea:** if types tell you that something cannot happen, don’t test it.

\[
\text{type } A = \langle a \rangle [A^*] \\
\text{type } B = \langle b \rangle [B^*] \\
\]

\[
\text{fun check}(x : A | B) = \text{match } x \text{ with } A \rightarrow 1 \mid B \rightarrow 0 \\
\text{fun check}(x : A | B) = \text{match } x \text{ with } \langle a \rangle \_ \rightarrow 1 \mid \_ \rightarrow 0 \\
\]

- No backtracking.
- Whole parts of the matched data are not checked.
4. Efficient execution

Use static type information to perform an optimal set of tests

**Idea:** if types tell you that something cannot happen, don’t test it.

```
type A = <a>[A*]
type B = <b>[B*]

fun check(x : A|B) = match x with A -> 1 | B -> 0
fun check(x : A|B) = match x with <a>_ -> 1 | _ -> 0
```

- No backtracking.
- Whole parts of the matched data are not checked

**Computing the optimal solution requires to fully exploit intersections and differences of types**
5. Logical pattern-specific optimisation of queries

Transform the from clauses so as to capture in a single pattern as much information as possible
5. Logical pattern-specific optimisation of queries

Transform the `from` clauses so as to capture in a single pattern as much information as possible

1. merge distinct patterns that work on a common sequence,
2. transform `where` clauses into patterns,
3. transform paths into nested pattern-based selections, then merge.
5. Logical pattern-specific optimisation of queries

Transform the \texttt{from} clauses so as to capture in a single pattern as much information as possible

1. merge distinct patterns that work on a common sequence,
2. transform \texttt{where} clauses into patterns,
3. transform paths into nested pattern-based selections, then merge.
5. Logical pattern-specific optimisation of queries

Transform the from clauses so as to capture in a single pattern as much information as possible

1. merge distinct patterns that work on a common sequence,
2. transform where clauses into patterns,
3. transform paths into nested pattern-based selections, then merge.
5. Logical pattern-specific optimisation of queries

Transform the `from` clauses so as to capture in a single pattern as much information as possible

```xml
select <book year=y>[t] from
  b in bibs/book,
  p in b/price,
  t in b/title,
  y in b/@year
where p = <price>"69.99"
```
5. Logical pattern-specific optimisation of queries

Transform the from clauses so as to capture in a single pattern as much information as possible

```xml
select <book year=y>[t] from
  b in bibs/book,
  p in b/price,
  t in b/title,
  y in b/@year
where p = <price>"69.99"

optimised as

select <book year=y> t from
  <bib>[b::Book*] in bibs,
  <book year=y>[ t::Title _+ <price>"69.99" ] in b
```
5. Logical pattern-specific optimisation of queries

Transform the `from` clauses so as to capture in a single pattern as much information as possible.

```
select <book year=y>[t] from
  b in bibs/book,
  p in b/price,
  t in b/title,
  y in b/@year
where p = <price>'69.99'
```

optimised as

```
select <book year=y> t from
  <bib>[b::Book*] in bibs,
  <book year=y>[ t::Title _+ <price>'69.99' ] in b
```

These optimisations are orthogonal to the classical optimisations: they sum up and bring a further gain of performance.
6. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators
6. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).
6. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).
6. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases.
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).
Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases.
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).

This may explain why there is less consensus on iterators than on extractors.
6. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

In XML processing it is important to allow the programmer to define her/his own iterators.

- XML complex structure makes virtually impossible for a language to provide a set of iterators covering all possible cases
- Iterators programmed in the language are far less precisely typed than built-in operators (require massive usage of casting).

This may explain why there is less consensus on iterators than on extractors.

How to define new iterators?
6. Pattern matches as building blocks for iterators

**Build regexp of “pattern matches” for user-defined iterators**

**Hosoya’s smart idea:** Define regular expression over pattern-matches “\(p \rightarrow e\)” (rather than over patterns).

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.
6. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya’s smart idea: Define regular expression over pattern-matches “p → e” (rather than over patterns).

\[
\text{select } e \text{ from } p \text{ in } e'
\]

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.
6. Pattern matches as building blocks for iterators

**Build regexp of “pattern matches” for user-defined iterators**

**Hosoya’s smart idea:** Define regular expression over pattern-matches “\( p \rightarrow e \)” (rather than over patterns).

\[
\text{select } e \text{ from } p \text{ in } e' = \text{filter}[(p\rightarrow e|\_\rightarrow[])^*](e')
\]

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.
6. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya’s smart idea: Define regular expression over pattern-matches “p -> e” (rather than over patterns).

\[
\text{select } e \text{ from } p \text{ in } e' = \text{filter}[(p\rightarrow e | \_\rightarrow [[]])*](e')
\]

\[
\text{map } e \text{ with } p_1\rightarrow e_1 | \ldots | p_n\rightarrow e_n
\]

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.
6. Pattern matches as building blocks for iterators

**Build regexp of “pattern matches” for user-defined iterators**

**Hosoya’s smart idea:** Define regular expression over pattern-matches “p -> e” (rather than over patterns).

\[
\text{select } e \text{ from } p \text{ in } e' = \text{filter}[(p\rightarrow e|\_\rightarrow [])*](e')
\]

\[
\text{map } e \text{ with } p_1\rightarrow e_1|...|p_n\rightarrow e_n = \text{filter}[(p_1\rightarrow e_1|...|p_n\rightarrow e_n)*](e)
\]

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.
6. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

Hosoya’s smart idea: Define regular expression over pattern-matches “$p \rightarrow e$” (rather than over patterns).

\[
\begin{align*}
\text{select } e \text{ from } p \text{ in } e' &= \text{filter}[(p\rightarrow e \mid \_\rightarrow [])]* (e') \\
\text{map } e \text{ with } p_1\rightarrow e_1 | ... | p_n\rightarrow e_n &= \text{filter}[(p_1\rightarrow e_1 | ... | p_n\rightarrow e_n)*] (e) \\
\text{match } e \text{ with } p_1\rightarrow e_1 | ... | p_n\rightarrow e_n
\end{align*}
\]

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.
6. Pattern matches as building blocks for iterators

**Hosoya’s smart idea:** Define regular expression over pattern-matches “\(p \rightarrow e\)” (rather than over patterns).

\[
\begin{align*}
\text{select } e \text{ from } p \text{ in } e' & = \text{filter}[(p\rightarrow e|\_\rightarrow[])^*](e') \\
\text{map } e \text{ with } p_1\rightarrow e_1|...|p_n\rightarrow e_n & = \text{filter}[(p_1\rightarrow e_1|...|p_n\rightarrow e_n)^*](e) \\
\text{match } e \text{ with } p_1\rightarrow e_1|...|p_n\rightarrow e_n & = \text{filter}[p_1\rightarrow e_1|...|p_n\rightarrow e_n][[e]]
\end{align*}
\]

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible
to define more powerful iterators.
6. Pattern matches as building blocks for iterators

Build regexp of “pattern matches” for user-defined iterators

**Hosoya’s smart idea:** Define regular expression over pattern-matches “\(p \rightarrow e\)” (rather than over patterns).

\[
\begin{align*}
\text{select } e \text{ from } p \text{ in } e' &= \text{filter}[(p \rightarrow e|_->[])\ast](e') \\
\text{map } e \text{ with } p_1 \rightarrow e_1 | ... | p_n \rightarrow e_n &= \text{filter}[(p_1 \rightarrow e_1 | ... | p_n \rightarrow e_n)\ast](e) \\
\text{match } e \text{ with } p_1 \rightarrow e_1 | ... | p_n \rightarrow e_n &= \text{filter}[p_1 \rightarrow e_1 | ... | p_n \rightarrow e_n](\{e\})
\end{align*}
\]

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.
6. Pattern matches as building blocks for iterators

Hosoya’s smart idea: Define regular expression over pattern-matches “$p \rightarrow e$” (rather than over patterns).

- \texttt{select e from p in e'} = \texttt{filter[\((p\rightarrow e|\_\rightarrow [])\)*]}(e')
- \texttt{map e with \(p_1\rightarrow e_1 \mid \ldots \mid p_n\rightarrow e_n\) = filter[\((p_1\rightarrow e_1 \mid \ldots \mid p_n\rightarrow e_n)\)*]}(e)
- \texttt{match e with \(p_1\rightarrow e_1 \mid \ldots \mid p_n\rightarrow e_n\) = filter[\(p_1\rightarrow e_1 \mid \ldots \mid p_n\rightarrow e_n\)]([e])}

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.
6. Pattern matches as building blocks for iterators

**Build regexp of “pattern matches” for user-defined iterators**

**Hosoya’s smart idea:** Define regular expression over pattern-matches “\(p \rightarrow e\)” (rather than over patterns).

\[
\text{select } e \text{ from } p \text{ in } e' = \text{filter}[(p\rightarrow e|\_\rightarrow [[]])^*](e')
\]

\[
\text{map } e \text{ with } p_1 \rightarrow e_1 | ... | p_n \rightarrow e_n = \text{filter}[(p_1 \rightarrow e_1 | ... | p_n \rightarrow e_n)^*](e)
\]

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 | ... | p_n \rightarrow e_n = \text{filter}[p_1 \rightarrow e_1 | ... | p_n \rightarrow e_n][[e]]
\]

- In-depth iterators are obtained by recursive filters
- If instead of regexp we use the core-algebra, then it is possible to define more powerful iterators.

**Type precision obtained by specific typing, as for patterns.**
7. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory.

Given a query $q$ execute it on documents in which parts not necessary to evaluate $q$ are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al.].

We can start from the optimal compilation of patterns. Compile patterns in order to have as many "\*" wildcards as possible:

```fun check(x : A|B) = match x with A -> 1 | B -> 0```

compiled as

```fun check(x : A|B) = match x with <a> -> 1 | <a> -> 0```
7. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query $q$ execute it on documents in which parts not necessary to evaluate $q$ are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al.].

We can start from the optimal compilation of patterns: Compile patterns in order to have as many "*" wildcards as possible:

```haskell
fun check(x : A|B) = match x with A -> 1 | B -> 0
```

compiled as

```haskell
fun check(x : A|B) = match x with <a> -> 1 | 
```

For details, see [G. Castagna] (Types and Patterns for Querying XML).
7. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query \( q \) execute it on documents in which parts not necessary to evaluate \( q \) are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al.].

We can start from the optimal compilation of patterns: Compile patterns in order to have as many "_" wildcards as possible. For example, the pattern \( ^{A}[^{B}]_{A}[^{B}] \) is compiled as

\[
\text{fun check}(x : A|B) = \text{match} x \text{ with } A \rightarrow 1 \mid B \rightarrow 0
\]
7. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory.

Given a query $q$ execute it on documents in which parts not necessary to evaluate $q$ are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al.].

We can start from the optimal compilation of patterns: Compile patterns in order to have as many “_” wildcards as possible.

```haskell
fun check(x : A|B) = match x with A -> 1 | B -> 0
```

compiled as

```haskell
fun check(x : A|B) = match x with <a>__-> 1 | __ -> 0
```
7. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory.

Given a query $q$ execute it on documents in which parts not necessary to evaluate $q$ are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al.].

We can start from the optimal compilation of patterns: **Compile patterns in order to have as many “_” wildcards as possible**

```plaintext
fun check(x : A|B) = match x with A -> 1 | B -> 0
```

compiled as

```plaintext
fun check(x : A|B) = match x with <a>_→ 1 | _ → 0
```
7. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query $q$ execute it on documents in which parts not necessary to evaluate $q$ are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al.].

We can start from the optimal compilation of patterns: Compile patterns in order to have as many “_” wildcards as possible

```
fun check(x : A|B) = match x with A -> 1 | B -> 0
```
compiled as

```
fun check(x : A|B) = match x with <a>_--> 1 | _ --> 0
```
7. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query \( q \) execute it on documents in which parts not necessary to evaluate \( q \) are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al.].

We can start from the optimal compilation of patterns: Compile patterns in order to have as many “_” wildcards as possible

\[
\text{fun check}(x : A|B) = \text{match } x \text{ with } A \rightarrow 1 \mid B \rightarrow 0
\]

compiled as

\[
\text{fun check}(x : A|B) = \text{match } x \text{ with } \langle\text{a}\rangle_\text{--} \rightarrow 1 \mid \_ \rightarrow 0
\]
7. Type/pattern-based pruning to optimise memory usage

Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query $q$ execute it on documents in which parts not necessary to evaluate $q$ are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al.].

We can start from the optimal compilation of patterns: **Compile patterns in order to have as many “_” wildcards as possible**

```haskell
fun check(x : A|B) = match x with A -> 1 | B -> 0
```

declared as

```haskell
fun check(x : A|B) = match x with <a>_ -> 1 | _ -> 0
```

Data matched by wildcards “_” not in the scope of a capture variable are not necessary to the evaluation.
Use type analysis to determine which parts of an XML data need not to be loaded in main memory

Given a query $q$ execute it on documents in which parts not necessary to evaluate $q$ are pruned. Recently adopted in main memory XML query engines, e.g. [Marian-Siméon], [Bressan et al.].

We can start from the optimal compilation of patterns: **Compile patterns in order to have as many “_” wildcards as possible**

```haskell
fun check(x : A|B) = match x with A -> 1 | B -> 0
```

compiled as

```haskell
fun check(x : A|B) = match x with <a>_--> 1 | _ --> 0
```

Data matched by wildcards “_” not in the scope of a capture variable are not necessary to the evaluation. Use boolean type constructors to determine the program data-need.
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- Data description is more precise:
  E.g. in IMDB there are constraints such as:
  \[
  \text{If a show-element contains season-elements, then its type-attribute is "TV Series".}
  \]

- Transformation description is more precise:
  DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision of regexp types to optimise secondary memory queries.
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**
  
  E.g. in IMDB there are constraints such as:
  
  *If a show-element contains season-elements, then its type-attribute is "TV Series".*

- **Transformation description is more precise:**

  DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision of regexp types to optimise secondary memory queries.
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**
  E.g. in IMDB there are constraints such as:
  
  *If a show-element contains season-elements, then its type-attribute is "TV Series".*

- **Transformation description is more precise:**

  DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision of regexp types to optimise secondary memory queries.
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**
  E.g. in IMDB there are constraints such as:
  
  *If a show-element contains season-elements, then its type-attribute is "TV Series".*

- **Transformation description is more precise:**
  - Exact type inference for pattern variables.
  - Finer type inference for queries:
    - Infer type `[Title+|Author+|Editor+]*` rather than `[Title|Author|Editor]*`.

- DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision of regexp types to optimise secondary memory queries.
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**
  E.g. in IMDB there are constraints such as:
  
  If a show-element contains season-elements, then its type-attribute is "TV Series".

- **Transformation description is more precise:**
  - Exact type inference for pattern variables.
  - Finer type inference for queries:
    
    for bibs/book/((title|author|editor)*)
    infer type [(Title (Author+|Editor+))*]
    rather than [(Title|Author|Editor)*]
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**
  E.g. in IMDB there are constraints such as:
  
  *If a show-element contains season-elements, then its type-attribute is "TV Series".*

- **Transformation description is more precise:**
  - Exact type inference for pattern variables.
  - Finer type inference for queries:
    
    ```
    for bibs/book/(title|author|editor) 
    infer type [(Title (Author+|Editor+))*] 
    rather than [(Title|Author|Editor)*]
    ```

- DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision of regexp types to optimise secondary memory queries.
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**
  E.g. in IMDB there are constraints such as:

  *If a show-element contains season-elements, then its type-attribute is "TV Series".*

- **Transformation description is more precise:**
  - Exact type inference for pattern variables.
  - Finer type inference for queries:
    ```
    for bibs/book/(title|author|editor)
    infer type [(Title (Author+|Editor+))*]
    rather than [(Title|Author|Editor)*]
    ```

DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision of the type system in query optimisation.
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**
  E.g. in IMDB there are constraints such as:
  
  \[\text{If a show-element contains season-elements,} \]
  
  \[\text{then its type-attribute is "TV Series".} \]

- **Transformation description is more precise:**
  - Exact type inference for pattern variables.
  - Finer type inference for queries:
    
    for \[\text{bibs/book/(title|author|editor)}\]
    
    infer type \[\text{[(Title (Author+|Editor+))*]}\]
    
    rather than \[\text{[(Title|Author|Editor)*]}\]

- DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision

\[\text{bibs : <bib>[(<book year=String>[Title (Author+|Editor+)|Price?])]*} \]
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- Data description is more precise:
  - E.g. in IMDB there are constraints such as:
    
    *If a show-element contains season-elements, then its type-attribute is "TV Series".*

- Transformation description is more precise:
  - Exact type inference for pattern variables.
  - Finer type inference for queries:
    
    for bibs/book/(title|author|editor)
    infer type [(Title (Author+|Editor+))]*
    rather than [(Title|Author|Editor)]*

- DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision

bibs : <bib>[
  (<book year=String>[Title (Author+|Editor+) Price?])]*
]
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**
  E.g. in IMDB there are constraints such as:
  
  *If a show-element contains season-elements, then its type-attribute is "TV Series".*

- **Transformation description is more precise:**
  
  - Exact type inference for pattern variables.
  - Finer type inference for queries:
    
    ```
    for bibs/book/(title|author|editor)
    infer type [(Title (Author+|Editor+))*]
    rather than [(Title|Author|Editor)*]
    ```

- **DTD/Schema already used to optimise access to XML data on disk.**
  It should be possible to use also the precision of regexp types to optimise secondary memory queries.
8. Type-based query optimisation

Use the precision of the type system in query optimisation

- **Data description is more precise:**
  E.g. in IMDB there are constraints such as:
  
  *If a show-element contains season-elements, then its type-attribute is "TV Series".*

- **Transformation description is more precise:**
  - Exact type inference for pattern variables.
  - Finer type inference for queries:
    
    ```
    for bibs/book/(title|author|editor) 
    infer type [(Title (Author+|Editor+))*]  
    rather than [(Title|Author|Editor)*]
    ```

- **DTD/Schema already used to optimise access to XML data on disk. It should be possible to use also the precision of regexp types to optimise secondary memory queries.**
Conclusion
**Conclusion**

- **Regexp patterns start from two simple ideas:**
  - Use the same constructors for types and value
  - Define patterns as types with capture variables

- **Tightly connected with boolean combinators,**
  make several aspects programmer-friendly:
  - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
  - Concepts are easier for the programmer (e.g., subtyping)
  - Informative error messages.
  - Precise and powerful specification language.

- **Several benefits:**
  - Types yield highly efficient runtimes in main memory if outperforms efficiency-oriented XQuery processors such as Qizx and Qexo.
  - High precision in typing queries, iterators, complex transformations.
  - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications...).
Conclusion

- **Regexp patterns start from two simple ideas:**
  - Use the same constructors for types and value
  - Define patterns as types with capture variables

- Tightly connected with boolean combinators, make several aspects programmer-friendly:
  - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
  - Concepts are easier for the programmer (e.g., subtyping)
  - Informative error messages
  - Precise and powerful specification language

- Several benefits:
  - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo
  - High precision in typing queries, iterators, complex transformations
  - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications, ...)

G. Castagna
Types and Patterns for Querying XML
Conclusion

- **Regexp patterns start from two simple ideas:**
  - Use the same constructors for types and value
  - Define patterns as types with capture variables

- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
  - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
  - Concepts are easier for the programmer (e.g. subtyping)
  - Informative error messages.
  - Precise and powerful specification language

- **Several benefits:**
  - Types yield highly efficient runtimes in main memory if outperforms efficiency-oriented XQuery processors such as Qizx and Qexo
  - High precision in typing queries, iterators, complex transformations
  - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications...)

G. Castagna

Types and Patterns for Querying XML
Conclusion

- **Regexp patterns start from two simple ideas:**
  - Use the same constructors for types and value
  - Define patterns as types with capture variables

- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
  - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
  - Concepts are easier for the programmer (e.g. subtyping)
  - Informative error messages.
  - Precise and powerful specification language

- **Several benefits:**
  - Types yield highly efficient runtimes. In main memory it outperforms efficiency-oriented XQuery processors such as OX and Qizx. In main and XQuery the Case benchmark.
  - High precision in typing queries, iterators, complex transformations.
  - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimizations, constraint specifications...)

G. Castagna  Types and Patterns for Querying XML  27/28
**Conclusion**

- **Regexp patterns start from two simple ideas:**
  - Use the same constructors for types and value
  - Define patterns as types with capture variables

- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
  - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
  - Concepts are easier for the programmer (e.g. subtyping)
  - Informative error messages.
  - Precise and powerful specification language

- **Several benefits:**
  - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [XMark and XQuery Use Cases benchmarks]
  - High precision in typing queries, iterators, complex transformations
  - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimizations, constraint specifications...)

G. Castagna

Types and Patterns for Querying XML
Conclusion

- **Regexp patterns start from two simple ideas:**
  - Use the same constructors for types and value
  - Define patterns as types with capture variables

- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
  - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
  - Concepts are easier for the programmer (e.g. subtyping)
  - Informative error messages.
  - Precise and powerful specification language

- **Several benefits:**
  - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [XMark and XQuery Use Cases benchmarks].
  - High precision in typing queries, iterators, complex transformations.
  - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications, . . .)
Conclusion

- **Regexp patterns start from two simple ideas:**
  - Use the same constructors for types and value
  - Define patterns as types with capture variables

- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
  - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
  - Concepts are easier for the programmer (e.g. subtyping)
  - Informative error messages.
  - Precise and powerful specification language

- **Several benefits:**
  - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [XMark and XQuery Use Cases benchmarks].
  - High precision in typing queries, iterators, complex transformations.
  - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications, . . .)
Regexp patterns start from two simple ideas:
- Use the same constructors for types and value
- Define patterns as types with capture variables

Tightly connected with boolean combinators, make several aspects programmer-friendly:
- Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
- Concepts are easier for the programmer (e.g. subtyping)
- Informative error messages.
- Precise and powerful specification language

Several benefits:
- Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [XMark and XQuery Use Cases benchmarks].
- High precision in typing queries, iterators, complex transformations.
- Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications, . . .)
Conclusion

- **Regexp patterns start from two simple ideas:**
  - Use the same constructors for types and value
  - Define patterns as types with capture variables

- **Tightly connected with boolean combinators, make several aspects programmer-friendly:**
  - Semantic types are set of values, and unions, intersections, and differences behave set-theoretically.
  - Concepts are easier for the programmer (e.g. subtyping)
  - Informative error messages.
  - Precise and powerful specification language

- **Several benefits:**
  - Types yield highly efficient runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [XMark and XQuery Use Cases benchmarks].
  - High precision in typing queries, iterators, complex transformations.
  - Multiple usages without the need of introducing new specific formalisms (error mining, data pruning, logical optimisations, constraint specifications, . . . )
... but that’s not enough

- Regexp are good for horizontal exploration but not for vertical one. Should be integrated with path-like primitives, extended to iterators, endowed with more friendly QBE-like interfaces, ...

- I tried to give an idea about the kind of research that is pursued on XML in the programming language community but much other research goes on (security, distribution, integration in mainstream languages, streaming, ...)

- I hope that this talk convinced some of you that it may be worth to have a look to this kind of research.

- A good place to start from is PLAN-X, ACM SIGPLAN Workshop on Programming Languages Technologies for XML
Conclusion

... but that’s not enough

- Regexp are good for horizontal exploration but not for vertical one. Should be integrated with path-like primitives, extended to iterators, endowed with more friendly QBE-like interfaces,

- I tried to give an idea about the kind of research that is pursued on XML in the programming language community but much other research goes on (security, distribution, integration in mainstream languages, streaming, ...)

- I hope that this talk convinced some of you that it may be worth to have a look to this kind of research.

- A good place to start from is PLAN-X, ACM SIGPLAN Workshop on Programming Languages Technologies for XML
Conclusion

... but that’s not enough

- Regexp are good for horizontal exploration but not for vertical one. Should be integrated with path-like primitives, extended to iterators, endowed with more friendly QBE-like interfaces,

- I tried to give an idea about the kind of research that is pursued on XML in the programming language community but much other research goes on (security, distribution, integration in mainstream languages, streaming, ...)

- I hope that this talk convinced some of you that it may be worth to have a look to this kind of research.

- A good place to start from is PLAN-X, ACM SIGPLAN Workshop on Programming Languages Technologies for XML
... but that’s not enough

- Regexp are good for horizontal exploration but not for vertical one. Should be integrated with path-like primitives, extended to iterators, endowed with more friendly QBE-like interfaces,

- I tried to give an idea about the kind of research that is pursued on XML in the programming language community but much other research goes on (security, distribution, integration in mainstream languages, streaming, ...)

- I hope that this talk convinced some of you that it may be worth to have a look to this kind of research.

- A good place to start from is PLAN-X, ACM SIGPLAN Workshop on Programming Languages Technologies for XML


... but that's not enough

- Regexp are good for horizontal exploration but not for vertical one. Should be integrated with path-like primitives, extended to iterators, endowed with more friendly QBE-like interfaces, ...

- I tried to give an idea about the kind of research that is pursued on XML in the programming language community but much other research goes on (security, distribution, integration in mainstream languages, streaming, ...) 

- I hope that this talk convinced some of you that it may be worth to have a look to this kind of research.

- A good place to start from is PLAN-X, ACM SIGPLAN Workshop on Programming Languages Technologies for XML