Types and Patterns for Querying XML

Giuseppe Castagna

CNRS
Laboratoire Preuves, Programmes et Systèmes
Université Paris Diderot - Paris 7

2008/2009

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
2. **iteration primitives**: iterate over XML trees the process of extraction and transformation of data.

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), ...

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 ($\mathrm{C}\omega$, 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), . . .

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), . . .

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.

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 $\mathbb{C}$Duce/$\mathbb{C}$QL, 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:

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.

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 (① ② ③)
    - Efficient and type precise main memory execution (④ ⑤ ⑥)
    - Secondary memory optimization (⑦ ⑧)
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 (①②③)
   - Efficient and type precise main memory execution (④⑤⑥)
   - Secondary memory optimization (⑦⑧)
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 (① ② ③)
    - Efficient and type precise main memory execution (④ ⑤ ⑥)
    - Secondary memory optimization (⑦ ⑧)
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

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

with pattern one can write

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

which syntactic sugar for

```
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

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

with pattern one can write

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

which syntactic sugar for

```
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

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

with pattern one can write

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

which syntactic sugar for

```
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

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

with pattern one can write

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

which syntactic sugar for

```
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

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

with pattern one can write

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

which syntactic sugar for

```
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:

```
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:

- use for types the same constructors as for values
- use values to denote singleton types
- 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

But if we:

- use for types the same constructors as for values
- use values to denote singleton types
- 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

But if we:

- use for types the same constructors as for values
- use values to denote singleton types
- 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

But if we:

- use for types the same constructors as for values
- use values to denote singleton types
- 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,

But if we:

- use for types the same constructors as for values
- use values to denote singleton types
- 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:

```
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
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 `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 `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 `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 `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 `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 `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 `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:

```
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:

```
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 (\):

```
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 (\\):

```
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 (\\):

```
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 (\):

```
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,. . .

$$t ::= \texttt{Int} \mid v \mid (t,t) \mid$$

Which types should we start from?

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

```
type 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,. . .

$$t \; ::= \; \texttt{Int} \mid v \mid (t,t) \mid t\,\texttt{|}\,t \mid t\texttt{\&}t \mid t\backslash t \mid \texttt{Empty} \mid \texttt{Any}$$

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,. . .

$$t \;::=\; \texttt{Int} \mid v \mid (t,t) \mid t\,|\,t \mid t\&t \mid t\backslash t \mid \texttt{Empty} \mid \texttt{Any}$$

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?

$t \;=\{v \mid v \;\text{ value of type } t\}$ **and** $\llparenthesis p \rrparenthesis = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

```
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?

$t \;= \{v \mid v \;\text{ value of type } t\}$ **and** $\llparenthesis p \rrparenthesis = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

```
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 $\llparenthesis$`(‘nil,n)`$\rrparenthesis$

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ **and** $\lbag p \rbag = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

```
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 $\lbag$`(`nil,n)`$\rbag$ = `(`nil,Any)`

Which types should we start from?

$t \ = \{v \mid v \ \text{ value of type } t\}$ **and** $\lbag p \rbag = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

```
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 $\lbag$`(`nil,n)`$\rbag$

```
(List,Int) & (`nil,Any)
```

Which types should we start from?

$t = \{v \mid v$ value of type $t\}$ **and** $\llparenthesis p \rrparenthesis = \{v \mid v$ matches pattern $p\}$

(this is a type)

```
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 $\llparenthesis$`(‘nil,n)`$\rrparenthesis$

`(List,Int) & (‘nil,Any)` $=$ `(‘nil,Int)`

Which types should we start from?

$t \ = \{ v \mid v \ \text{ value of type } t \}$ **and** $\wr p \int = \{ v \mid v \text{ matches pattern } p \}$

(this is a type)

```
type List = (Any,List) | ‘nil

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

The first branch is executed only for values and are both in `(List,Int)` **and** in $\wr$`(‘nil,n)`$\int$

$$\texttt{(List,Int) \& (‘nil,Any)} = \texttt{(‘nil,Int)}$$

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ **and** $\wr p \int = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

```
type List = (Any,List) | ‘nil

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

The second branch is executed for values that are in `(List,Int)` **not** in $\wr$`(‘nil,n)`$\int$ **and** in $\wr$`((_,t),n)`$\int$

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ **and** $\wr p \int = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

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

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

The second branch is executed for values that are in `(List,Int)` **not** in $\wr$`(`nil,n)`$\int$ **and** in $\wr$`((_,t),n)`$\int$

`((List,Int)\(`nil,Any))&((Any,Any),Any)`

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ **and** $\llparenthesis p \rrparenthesis = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

```
type List = (Any,List) | ‘nil

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

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

$\texttt{((List,Int)}\setminus\texttt{(‘nil,Any))\&((Any,Any),Any)} = \texttt{((Any,List),Int)}$

Which types should we start from?

$t = \{v \mid v$ value of type $t\}$ **and** $\llparenthesis p \rrparenthesis = \{v \mid v$ matches pattern $p\}$

(this is a type)

```
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 $\llparenthesis$`(`nil,n)`$\rrparenthesis$ **and** in $\llparenthesis$`((_,t),n)`$\rrparenthesis$

`((List,Int)\(`nil,Any))&((Any,Any),Any)` = `((Any,List),Int)`

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ **and** $\llparenthesis p \rrparenthesis = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

```
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?

$t \;=\{v \mid v\;\text{ value of type } t\}$ **and** $\lbag p \rbag = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

```
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

```
Int | Int
```

Which types should we start from?

$$t = \{v \mid v \text{ value of type } t\} \quad \textbf{and} \quad \llparenthesis p \rrparenthesis = \{v \mid v \text{ matches pattern } p\}$$

(this is a type)

```
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

$$\texttt{Int} \mid \texttt{Int} = \texttt{Int}$$

Which types should we start from?

$t = \{v \mid v \text{ value of type } t\}$ **and** $\llparenthesis p \rrparenthesis = \{v \mid v \text{ matches pattern } p\}$

(this is a type)

```
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

$$\texttt{Int} \mid \texttt{Int} = \texttt{Int}$$

The function is well-typed

Unions, intersections, differences

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

  $$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$$

  - To infer the type t_1 of e_1 we need $t\&\llparenthesis p_1 \rrparenthesis$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \llparenthesis p_1 \rrparenthesis)\&\llparenthesis p_2 \rrparenthesis$;
  - 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:**

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

  - To infer the type t_1 of e_1 we need $t \& \lbag p_1 \rbag$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \lbag p_1 \rbag) \& \lbag p_2 \rbag$;
  - 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:**

  $$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$$

  - To infer the type t_1 of e_1 we need $t \& \lbag p_1 \rbag$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \lbag p_1 \rbag) \& \lbag p_2 \rbag$;
  - 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:**

  $\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$

  - To infer the type t_1 of e_1 we need $t \& \lbag p_1 \rbag$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \lbag p_1 \rbag) \& \lbag p_2 \rbag$;
  - 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:**

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

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

- **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:**

  $$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$$

  - To infer the type t_1 of e_1 we need $t \& \lbag p_1 \rbag$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \lbag p_1 \rbag) \& \lbag p_2 \rbag$;
  - 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:**

$$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$$

- To infer the type t_1 of e_1 we need $t \& \wr\, p_1 \int$ (where $e : t$);
- To infer the type t_2 of e_2 we need $(t \setminus \wr\, p_1 \int) \& \wr\, p_2 \int$;
- 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:**

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

  - To infer the type t_1 of e_1 we need $t\&\llparenthesis p_1\rrparenthesis$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t\setminus\llparenthesis p_1\rrparenthesis)\&\llparenthesis p_2\rrparenthesis$;
  - 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.

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:**

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

  - To infer the type t_1 of e_1 we need $t \& \wr p_1 \int$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \wr p_1 \int) \& \wr p_2 \int$;
  - 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.

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:**

  `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.

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:**

  $$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$$

  - To infer the type t_1 of e_1 we need $t \& \llparenthesis p_1 \rrparenthesis$ (where $e : t$);
  - To infer the type t_2 of e_2 we need $(t \setminus \llparenthesis p_1 \rrparenthesis) \& \llparenthesis p_2 \rrparenthesis$;
  - 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.

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

```
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]
```

Kleene star

This and: singletons, intersections, differences, `Empty`, and A

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]
```

This and: singletons, intersections, differences, Empty, and A

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]                          PCDATA
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Publisher = String
type Price = <price>[PCDATA]
```

This and: singletons, intersections, differences, `Empty`, and A

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 A

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]
```

This and: singletons, intersections, differences, `Empty`, and A

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

```
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

Patterns = Types + Capture variables

TYPES

PATTERNS

Patterns

Patterns = Types + Capture variables

TYPES

PATTERNS

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

Patterns

Patterns = Types + Capture variables

TYPES

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

PATTERNS

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

Patterns

Patterns = Types + Capture variables

TYPES

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

PATTERNS

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

The pattern binds `x` to the *sequence* of all books in the bibliography

Patterns

Patterns = Types + Capture variables

TYPES

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

PATTERNS

```
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

TYPES

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

PATTERNS

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

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

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

PATTERNS

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

Patterns

Patterns = Types + Capture variables

TYPES

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

PATTERNS

```
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

```
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

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

$\vdots$

x_n in e_n

where c

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

Select-from-where

Instead of just variables use patterns

select e from
$\quad \mathtt{p}_1$ in $\mathtt{e}_1$
$\quad\quad \vdots$
$\quad \mathtt{p}_n$ in $\mathtt{e}_n$
where c

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

Select-from-where

Instead of just variables use patterns

select e from
p_1 in e_1
$\vdots$
p_n in e_n
where c

```
<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?]
```

Select-from-where

Instead of just variables use patterns

```
select e from
     $p_1$ in $e_1$
        ⋮
     $p_n$ in $e_n$
where c
```

```
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

```
select e from
    $p_1$ in $e_1$
        ⋮
    $p_n$ in $e_n$
where c
```

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

Selects from `bibs` the titles of all books of 1990 and of price 69.99

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

Select-from-where

Instead of just variables use patterns

select e from

p_1 in e_1

$\vdots$

p_n in e_n

where c

```
fun getTitles(bibs : Biblio) : [(<book>[Title])*]
  select <book>t from
    <bib>[b::Book*] in bibs,
    <book year="1990">[ t::Title _+ <price>"69.99" ] in b
```

Selects from `bibs` the titles of all books of 1990 and of price 69.99 and has type `Biblio -> [(<book>[Title])*]`

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

XPath encoding

For instance in $\mathbb{CQL}$ (…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.

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.

XPath encoding

For instance in $\mathbb{C}$QL (…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.

XPath encoding

For instance in $\mathbb{C}$QL (…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.

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 . . .

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

$$t \;::=\; \mathtt{Int} \mid v \mid (t,t) \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathtt{Any}$$

Patterns

$$p \;::=\; t \mid x \mid (p,p) \mid p \vee p \mid p \wedge p$$

Example:

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

encoded as

$$\begin{aligned} Book &= (\text{`}book, (Title, X \vee Y)) \\ X &= (Author, X \vee (Price, \text{`}nil) \vee \text{`}nil) \\ Y &= (Editor, Y \vee (Price, \text{`}nil) \vee \text{`}nil) \end{aligned}$$

. . . it is all syntactic sugar!

Types

$$t \;::=\; \texttt{Int} \;\mid\; v \;\mid\; (t,t) \;\mid\; t \vee t \;\mid\; t \wedge t \;\mid\; \neg t \;\mid\; \texttt{Any}$$

Patterns

$$p \;::=\; t \;\mid\; x \;\mid\; (p,p) \;\mid\; p \vee p \;\mid\; p \wedge p$$

Example:

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

encoded as

$$\begin{aligned} Book &= (\text{‘}book, (Title, X \vee Y)) \\ X &= (Author, X \vee (Price, \text{‘}nil) \vee \text{‘}nil) \\ Y &= (Editor, Y \vee (Price, \text{‘}nil) \vee \text{‘}nil) \end{aligned}$$

. . . it is all syntactic sugar!

Types

$$t ::= \mathtt{Int} \mid v \mid (t, t) \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathtt{Any}$$

Patterns

$$p ::= t \mid x \mid (p, p) \mid p \vee p \mid p \wedge p$$

Example:

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

encoded as

$$\begin{aligned} Book &= (\text{`}book, (Title, X \vee Y)) \\ X &= (Author, X \vee (Price, \text{`}nil) \vee \text{`}nil) \\ Y &= (Editor, Y \vee (Price, \text{`}nil) \vee \text{`}nil) \end{aligned}$$

. . . it is all syntactic sugar!

Types

$$t \ ::= \ \mathtt{Int} \ | \ v \ | \ (t,t) \ | \ t \vee t \ | \ t \wedge t \ | \ \neg t \ | \ \mathtt{Any}$$

Patterns

$$p \ ::= \ t \ | \ x \ | \ (p,p) \ | \ p \vee p \ | \ p \wedge p$$

Example:

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

encoded as

$$\begin{aligned} Book &= (\text{`}book, (Title, X \vee Y)) \\ X &= (Author, X \vee (Price, \text{`}nil) \vee \text{`}nil) \\ Y &= (Editor, Y \vee (Price, \text{`}nil) \vee \text{`}nil) \end{aligned}$$

Some reasons to consider regular expression types and patterns

Some good reasons to consider regexp patterns/types

- **Theoretical reason: very compact**
-
    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 ($\neq$ simple)**
-
    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 ($\neq$ 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 ($\neq$ 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:

  [illegible]

  [illegible]

- Types can be composed:

  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:

  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:

  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:

```
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 elements.

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:

```
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:

```
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:

```
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.

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

```
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

```
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

```
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

```
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?]
```

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>_|<edtor>_)+ _* ] 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.

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>_|<edtor>_)+ _* ] 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.

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>_|<edtor>_)+ _* ] 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.

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>_|<edtor>_)+ _* ] 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>_|<edtor>_)+ _* ] 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>_|<edtor>_)+ _* ] 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>_|<edtor>_)+ _* ] 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>_|<edtor>_)+ _* ] 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.

```
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*]

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.

```
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.

```
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
```

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

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

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 `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

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

```
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

```
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).

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.

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 -> 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).

`select` e `from` p `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 -> e" (rather than over patterns).

$$\texttt{select}\ e\ \texttt{from}\ p\ \texttt{in}\ e' = \texttt{filter[(}p\texttt{->}e\texttt{|\_->[])*](}e'\texttt{)}$$

- 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 “$p \texttt{->} e$” (rather than over patterns).

$$\texttt{select}\ e\ \texttt{from}\ p\ \texttt{in}\ e' = \texttt{filter[(}p\texttt{->}e\texttt{|\_->[])*](}e'\texttt{)}$$

$$\texttt{map}\ e\ \texttt{with}\ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n$$

- 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).

$$
\begin{aligned}
\texttt{select}\ e\ \texttt{from}\ p\ \texttt{in}\ e' &= \texttt{filter[(}p\texttt{->}e\texttt{|\_->[])*](}e'\texttt{)}\\
\texttt{map}\ e\ \texttt{with}\ \ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &= \texttt{filter[(}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{)*](}e\texttt{)}
\end{aligned}
$$

- 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).

$$\begin{aligned}
\texttt{select}\ e\ \texttt{from}\ p\ \texttt{in}\ e' &= \texttt{filter[(}p\texttt{->}e\texttt{|\_->[])*](}e'\texttt{)}\\
\texttt{map}\ e\ \texttt{with}\ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &= \texttt{filter[(}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{)*](}e\texttt{)}\\
\texttt{match}\ e\ \texttt{with}\ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &
\end{aligned}$$

- 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).

$$\begin{array}{rcl}
\texttt{select}\ e\ \texttt{from}\ p\ \texttt{in}\ e' &=& \texttt{filter[(}p\texttt{->}e\texttt{|\_->[])*](}e'\texttt{)} \\
\texttt{map}\ e\ \texttt{with}\ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &=& \texttt{filter[(}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{)*](}e\texttt{)} \\
\texttt{match}\ e\ \texttt{with}\ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &=& \texttt{filter[}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{]([}e\texttt{])}
\end{array}$$

- 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).

$$\begin{aligned}
\texttt{select}\ e\ \texttt{from}\ p\ \texttt{in}\ e' &= \texttt{filter[(}p\texttt{->}e\texttt{|\_->[])*](}e'\texttt{)}\\
\texttt{map}\ e\ \texttt{with}\ \ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &= \texttt{filter[(}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{)*](}e\texttt{)}\\
\texttt{match}\ e\ \texttt{with}\ \ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &= \texttt{filter[}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{]([}e\texttt{])}
\end{aligned}$$

- 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).

$$\begin{aligned}
\texttt{select}\ e\ \texttt{from}\ p\ \texttt{in}\ e' &= \texttt{filter[(}p\texttt{->}e\texttt{|\_->[])*](}e'\texttt{)}\\
\texttt{map}\ e\ \texttt{with}\ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &= \texttt{filter[(}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{)*](}e\texttt{)}\\
\texttt{match}\ e\ \texttt{with}\ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &= \texttt{filter[}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{]([}e\texttt{])}
\end{aligned}$$

- 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).

$$\begin{aligned}
\texttt{select}\ e\ \texttt{from}\ p\ \texttt{in}\ e' &= \texttt{filter[(}p\texttt{->}e\texttt{|\_->[])*](}e'\texttt{)}\\
\texttt{map}\ e\ \texttt{with}\ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &= \texttt{filter[(}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{)*](}e\texttt{)}\\
\texttt{match}\ e\ \texttt{with}\ p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n &= \texttt{filter[}p_1\texttt{->}e_1\texttt{|}\ldots\texttt{|}p_n\texttt{->}e_n\texttt{]([}e\texttt{])}
\end{aligned}$$

- 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.

compiled as

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.

compiled as

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**

compiled as

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**

```
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**

```
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**

```
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**

```
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
```

Data matched by wildcards "_" not in the scope of a capture variable are not necessary to the evaluation.

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
```

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:
  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:
    for
    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.

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

```
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 this**

```
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 runtime: in main memory it outperforms efficiency-oriented XQuery processors such as Qizx and Qexo [XMark and XQuery Use Cases [illegible]]
  - 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 [illegible].
  - 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 [illegible].
    - 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, ...)

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,. . .)

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,. . .)

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

. . . 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, . . .)
- A good place to start from is PLAN-X, *ACM SIGPLAN Workshop on Programming Languages Technologies for XML*.
- To try all this install ℂDuce

  `http://www.cduce.org`

  (on Linux, `yum/apt-get install cduce`)

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, . . .)
- A good place to start from is PLAN-X, *ACM SIGPLAN Workshop on Programming Languages Technologies for XML.*
- To try all this install ℂDuce

  `http://www.cduce.org`

  (on Linux, `yum/apt-get install cduce`)

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, . . .)
- A good place to start from is PLAN-X, *ACM SIGPLAN Workshop on Programming Languages Technologies for XML.*
- To try all this install ℂDuce

  `http://www.cduce.org`

  (on Linux, `yum/apt-get install cduce`)

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, . . .)
- A good place to start from is PLAN-X, *ACM SIGPLAN Workshop on Programming Languages Technologies for XML.*
- To try all this install ℂDuce

  `http://www.cduce.org`

  (on Linux, `yum/apt-get install cduce`)

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, . . .)
- A good place to start from is PLAN-X, *ACM SIGPLAN Workshop on Programming Languages Technologies for XML.*
- To try all this install ℂDuce

  `http://www.cduce.org`

  (on Linux, `yum/apt-get install cduce`)