


OCaml + XDuce

Alain Frisch –  **INRIA**
ROCQUENCOURT



A strongly-typed functional DSL for XML.

- Only **values** = XML fragments.
- Regular expression **types** \simeq DTD.
- Regular expression **patterns**.
- Type system with **implicit subtyping** (language inclusion).
Checks exhaustivity, infers precise type for capture variables.

```
type Addrbook = addrbook[ Entry* ]
type Entry = entry[ Name, Addr, Tel? ]
type Name = name[ String ]
type Addr = addr[ String ]
type Tel = tel[ String ]

let telList (e : Entry*) : (Name, Tel)* = match e with
| entry[ Name, Addr ]*,
  entry[ Name as n, Addr, Tel as t ], Any* as q ->
  n, t, telList q
| entry[ Name, Addr ]* -> []
```

XDuce is fine for XML → XML transformations.

Shows the benefits of **linguistic support** for XML :
more idiomatic and safer code to deal with XML.

Often, one wants **general-purpose** features together with XML-oriented ones.

(Not exhaustive!)

- Linq : .NET + XML literals, queries.
- EcmaScript for XML (E4X) : Javascript + XML literals.
- XJ : Java + XML literals + XML schema.
- Xact : Java + XML templates + XML schema.
- XHaskell : XDuce compiled to Haskell.
- Xtatic : C# + XDuce.

... and ...

- OCamlDuce : Objective Caml + XDuce.

OCaml, XDuce : a natural match ?

- A perfect match : **same programming style**.
Recursive functions, types, patterns, few side effects.
- A difficult relationship : **very different type systems**.
 - OCaml : polymorphism + type inference.
 - XDuce : implicit subtyping + forward type propagation.
- Approach : don't try to mix features, just make them **coexist** smoothly.

OCamlDuce : design principle

Make the XDuce and OCaml type checkers work together (in a controlled way!) and design a type system to account for that.

Existing implementations can be reused and **all of OCaml** exotic features are supported.

Challenge : preserve nice properties of ML, e.g. **clean specification** of type inference (\neq algorithm).

Basic idea : **separation of concerns**.

- Stratification : XML values/types are atomic w.r.t. OCaml.
- Type-checking : instrument the OCaml type-checker to extract the data flow of XML values, which is then processed by the XDuce type-checker.

Limitations :

- No polymorphism, no inference for XML types.
- No implicit subtyping for ML types.

An example

```
let f x = match x with
  a[_]* as y, _ -> y , x
let z1 = f (a[],b[])
let z2 = List.map f [ (a[],a[]); (a[b[]]) ]
```

An example

```
let f x = (* p = a[_]* as y, _ *)
  let y' = match[y;p](x) in y' , x
let z1 = f (a[],b[])
let z2 = List.map f [ (a[],a[]); (a[b[]]) ]
```

Introduce fresh XML type variables

Goal : infer input and output XML types for instances of XML operators / literals.

```
let f x = let y' = match[y;p]l2l1(x) in y' l5l3,l4 x  
let z1 = f (a[],b[])l6  
let z2 = List.map f [ (a[],a[])l7 ; (a[b[]])l8 ]
```

- Introduce “XML type variables” that stand for ground XML types to be inferred.
- Subscript : inputs.
- Superscript : outputs.

Run the ML type-checker

```
let f x = let y' = match[y;p]l2l1(x) in y' ,l5l3,l4 x
let z1 = f (a[],b[])l6
let z2 = List.map f [ (a[],a[])l7 ; (a[b[]])l8 ]
```

- Run the ML type-checker with e.g. , : $l_3 \rightarrow l_4 \rightarrow l_5$.
- It forces unifications :
 - $l_2 = l_3$
 - $l_1 = l_4 = l_6 = l_7 = l_8$
- and produces types for top-level identifiers :

```
val f : l1 → l5
val z1 : l5
val z2 : l5 list
```

Produce constraints = extract data-flow

```
let f x = let y' = match[y;p]l1l2(x) in y' ,l2,l1l5 x
let z1 = f (a[],b[])l1
let z2 = List.map f [ (a[],a[])l1 ; (a[b[]])l1 ]
```

Each XML operator produces a constraint (input \rightarrow output flow) :

$$l_2 \geq \text{match}[y;p](l_1)$$

$$l_5 \geq l_2, l_1$$

$$l_1 \geq (a[],b[])$$

$$l_1 \geq (a[],a[])$$

$$l_1 \geq (a[b[]])$$

Solve constraints

The constraints are acyclic :

$$l_2 \geq \text{match}[y;p](l_1)$$

$$l_5 \geq l_2, l_1$$

$$l_1 \geq (a[], b[])$$

$$l_1 \geq (a[], a[])$$

$$l_1 \geq (a[b[]])$$

We can compute the smallest solution (union of lower-bounds) :

$$l_1 = a[], b[] \mid a[], a[] \mid a[b[]]$$

$$l_2 = \text{match}[y;p](l_1) = a[] \mid a[], a[] \mid a[b[]]$$

$$l_5 = l_2, l_1$$

Basic type-checking algorithm

- Introduce fresh XML type variables.
- Run the OCaml type checker to unify these variables and extract the data-flow.
- Run the XDuce type checker on the acyclic data-flow.

Note : this is done for a **whole compilation unit**.

Implicit subtyping ?

With the basic algorithm, subsumption is only allowed at the output of XML operators.

```
val x : a[]  
val y : b[]
```

```
if ... then x else y (* failure! *)  
if ... then  $\text{id}_{l_1}^{l_2}(x)$  else  $\text{id}_{l_3}^{l_4}(y)$  (* ok! *)
```

- id = identity on XML values.
- Unifications : $l_2 = l_4$, $l_1 = a[]$, $l_3 = b[]$.
- Constraints : $l_2 \geq l_1$, $l_2 \geq l_3$.
- Solution :
 $l_2 = a[] \mid b[]$

Strengthening = preprocessing pass to add applications of the `id` operator around sub-expressions of kind XML.

Q : How to detect these sub-expressions ?

A : With a **preliminary run of the OCaml type-checker**, where all the XML types collapse.

The data-flow can be cyclic.

Example :

```
let f x = let y = match[y;p](x) in y , x
let z1 = f (a[],b[])
let z2 = List.map f [ ... ; ... ]
let z3 = f z1
```

After strengthening :

```
let f x = let y = match[y;p]l2l1(x) in yl5l3,l4 x  
let z1 = f (a[],b[])l6  
let z2 = List.map f [ ...l7 ; ...l8 ]  
let z3 = f idl12l11(z1)
```

- Unifications :

$$l_2 = l_3, l_1 = l_4 = l_6 = l_7 = l_8, l_{11} = l_5, l_{12} = l_1.$$

- Cyclic constraints :

$l_1 \geq l_5$ (the output of f can flow back to its input)

$l_5 \geq l_2, l_1$

$l_2 \geq \text{match}[y;p](l_1)$

$l_1 \geq a[], b[] \mid a[], a[] \mid a[b[]]$

Cycle in constraints = cycle in the data flow of XML values modulo the approximation of the ML type system (a single flow point for each sub-expression). This can be **predicted by the programmer**. Typical examples :

- Recursive functions.
- Fold-like functions with an XML accumulator.
- Self-composition of XML functions.

To **break cycles**, the programmer can provide :

- Explicit type annotations.
- Data-type declarations.
- Module signatures.

Annotations are propagated by the ML type system (more robust and predictable than ad hoc propagation rules).

Conclusion

- All of OCaml + most of CDuce.
- A minimalistic yet realistic approach.
- In the paper : **abstract extension** of ML with foreign types, foreign operators, subtyping.
- The **programmer can understand** where to put annotations.
- Main limitation : **no polymorphism on XML** types (but : universal type + implicit subtyping + dynamic type checking).
- A **practical implementation** (drop-in replacement for OCaml tools, binary compatibility).
- (Not in the paper :) an automatic type-directed translation between ML and XML values (e.g. $\tau \text{ list} \leftrightarrow \tau^*$)

Thank you !



- OCamlDuce homepage :
<http://www.cduce.org/ocaml>