Programming Languages for XML
Ou
Au delà des standards

Véronique Benzaken

benzaken@lri.fr

LRI (UMR 8623 CNRS) and Université de Paris Sud

Introduction

Information on the web

What you are not able to do with the Web:

- Main Language on the Web is HTML
- HTML used for presenting informations
  - Not suited for data exchange.
  - Not able to perform data manipulation (except displaying).
- Unable to interpret data provided with HTML format.

New applications

- B2B: Companies need to exchange informations (not only for displaying them !)
- Search engines : if one is able to interpret transmitted data one is able to index it efficiently.
- ASP: send data to a server in order to apply them a given treatment.
- Ubiquitous computing: same informations must be displayed differently (HTML, text, WML).
- . . .

New Requirements

Exchanging, publishing and processing data

- Heterogenous network: data must be represented independently from a given machine to another one.
- Various applications: data must be represented independently from a given application.
- Each application has its own (propietary) format: data must be easily transformed from a format to another.

A solution: XML

- Written in ASCII: eases exchange
- Human-readable
- Self-explaining
- Standardisation (W3C)
- Adopted by an increasing number of leading IT companies

XML: an example

```
<bib>
 <book>
   <title> Persistent Object Systems</title>
   <year> 1994</year>
   <author> M. Atkinson</author>
   <author> V. Benzaken</author>
   <author> D. Maier</author>
 </book>
 <book>
   <title> OOP: a unified foundation</title>
   <year> 1997</year>
   <author> G. Castagna</author>
 </book>
</bib>
```

- XML is only a mere format (not a language)
- Constitutes the de facto *“lingua franca”* on the web.

Transformation Languages

Document Processing

Languages are needed to process XML documents.

- Presentation
  XML $\longrightarrow$ XML-FO, XHTML, LaTeX, MathML, ...
- Search
  find all recipes of “Tiramisu”
- Exchange
  prepare a description for search engines
- Integration
  “best recipes on the web”

Document processing

Three different techniques:

- Libraries for tree manipulation in general purpose languages.
- Extension of type systems for existing languages with "XML types".
- Design of XML-specific processing languages.

Library approach

Use of APIs such as SAX (Simple API for XML) or DOM (Document Object Model) in C++ or Java.

- ⊕ No need to train programmers
- ⊕ Tools, support, stability
- ⊖ No use of types: correction difficult to enforce and or guarantee, debugging very hard.
- ⊖ Syntax not adapted, very verbose, code unreadable and very unlikely reusable, very low productivity.

Languages

Extension of type systems of existing languages

Examples: Relaxer and JAXB (based on Java) HaXML (based on Haskell), Xtatic (based on C#).

- ⊕ Use of specific types ensures (partial) correction and eases debbuging.
- ⊕ Tools, support and stability.
- ⊖ Learning curve slower. Need to train programmers in case of not very wide-spread languages.
- ⊖ Syntax not adapted, verbose, code unreadable and unlikely reusable, low productivity.

Languages

XML-specific languages

XML to XML: XSLT, XDuce, YATL, XQuery

General purpose: Xerox's Circus-DTE, ℂDuce, Microsoft's X#.

- ⊕ Use of specific types ensures partial correction and eases debugging.
- ⊕ Syntax very well-adapted, very compact programs, readable, code reuse. High programmer's productivity.
- ⊖ Learning curve very slow. Need to train programmers for new (functional) languages.
- ⊖ XSLT excepted, all those languages are in development phase; only (pre) prototypes or alpha versions are available. Lack of support and stability.

ℂDuce

an XML-Centric General Purpose Language

(www.cduce.org)

Motivation

- XML provides formats for tree-structured data (or documents) and,
- in addition, types (or schemas), e.g.,
  - DTD
  - RELAX NG
  - W3C XML Schema
- However, existing “processing” languages are un-typed (XSLT/XPath).
- How can we process XML documents using types ?

ℂDuce: Introduction

- ℂDuce is a general purpose typed functional programming language.
- The work on ℂDuce started from an attempt to overtake some limitations of XDuce (H. Hosoya, B. Pierce, J. Vouillon).
- Design choice: keeping XML applications in mind.

Formal Foundation: "Semantic Subtyping" in [LICS'02]

Design and Implementation: "ℂDuce an XML-Centric General-Purpose Language"in [ICFP'03]

$\mathbb{C}$Duce overview

- Type algebra
  Core (low level) representation of XML documents, Transformation typing
- Support for XML documents: sequences and elements
  XML friendly syntax
- Pattern matching
  Complex extraction of information with exact typing
- Overloaded Functions
  Code reusability, OOP style

ℂDuce overview

- Higher order functions
- Queries
  Highly declarative programming interface
- Benchmarks
- Current status

Types

Types are pervasives in ℂDuce:

- Static validation
  - E.g.: does the transformation produce valid XHTML ?
- Type-driven semantics
  - Pattern matching can dispatch on types, overloaded functions
- Type-driven compilation and optimizations
  - Makes use of static type information to avoid unnecessary and redundant tests at runtime
  - Allows a more declarative style without degrading performance
  - Extremely useful with tag-coupled XML types (e.g.: DTDs)

Core type algebra

- basic types
  `Int, String, Atom`
  (an atom is a constant of the form `‘`*id* where *id* is an arbitrary identifier)
- types constructors
  product types `(`t_1`,`t_2`)`
  record types $\{$ $a_1 = t_1$`;` ... `;` $a_n = t_n$ $\}$
  functional types t_1 `->` t_2
- boolean connectives
  empty and universal types `Empty` and `Any`
  intersection t_1 `&` t_2
  union t_1 `|` t_2
  and difference $t_1 \backslash t_2$

Core type algebra

- finer basic types
  integer interval `i..j` (e.g.: `0..9`)
  string regexp `/regexp/` (e.g.: `/['a'-'z']*/`)
- singleton types
  for any scalar or constructed value v, v is itself a type (for instance `` `nil `` is the type of empty sequences, and `18` is the type of the integer `18`)
- recursive types
  e.g.: integer lists:

```
Ilist where Ilist = (Int, Ilist) | `nil
```

Set-theoretic interpretation of types

- To handle complexity of the type algebra, we need a simple interpretation of types:

A type is a set of values.

  - type `Int` is the set $\{\ldots, -1, 0, 1, 2, 3, \ldots\}$;
  - type (t_1, t_2) is the set of all expressions `(`v_1`,`v_2`)` where v_i is a value of type t_i;
  - type t_1`->`t_2 is the set of all expressions `fun f (`s_1`;...;`s_n`)e` that applied to a value in t_1 return a result (if any) in t_2.
- Natural set-theoretic interpretation of boolean connectives and subtyping relation.

Formal foundations in [LICS'02]

XML: Sequences and Elements

Sequences are encoded *à la* Lisp by pairs and a terminator `‘nil`.

A sequence of values $v_1, \ldots, v_n$ is written

$$\textbf{[}\ v_1 \textbf{...} v_n\ \textbf{]}$$

which is syntactic sugar for

$$\textbf{(}v_1\textbf{,(}\ldots\textbf{,(}v_n\textbf{, ‘nil)}\ldots\textbf{)).}$$

XML: Sequences and Elements

Define sequence types by

[*tyregexp***]**

where *tyregexp* is a **regular expression** built from types.

E.g.: **[Int*]** , **[Int* String+ Int?]**

An XML element

$<$*tag* $a_1 = v_1 \quad \ldots \quad a_n = v_n>$ *elem_seq* $</$*tag*$>$

is written in $\mathbb{C}$Duce as

$<$*tag* $a_1 = v_1 \quad \ldots \quad a_n = v_n>$ [*elem_seq*]

XML-Friendly Syntax

```
<bib>
  <book>
    <title>Persistent Object Systems</title>
    <year>1994</year>
    <author>M. Atkinson</author>
    <author>V. Benzaken</author>
    <author>D. Maier</author>
  </book>
  <book>
    <title>OOP: a unified foundation</title>
    <year>1997</year>
    <author>G. Castagna</author>
  </book>
</bib>
```

XML

```
let bib0 = <bib>[
             <book>[
                 <title>["Persistent Object Systems
                 <year>["1994"]
                 <author>["M. Atkinson"]
                 <author>["V. Benzaken"]
                 <author>["D. Maier"] ]

              <book>[
                 <title>["OOP: a unified foundation
                 <year>["1997"]
                 <author>["G. Castagna"]]
                 ]
```

CDuce

Loading XML documents

```
type IntStr = /['0'-'9']+/;;
type Bib    = <bib>[Book*];;
type Book   = <book>[Title Year Author+];;
type Year   = <year>[IntStr];;
type Title  = <title>[String];;
type Author= <author>[String];;
```

An XML document can be loaded with `load_xml` and checked to be of the correct type by pattern matching:

```
let bib0 =

   match (load_xml "bib.xml") with
    | (x & Bib) -> x
    | _ -> error "Wrong type !";;

|- bib0 : Bib
```

Pattern Matching

One of $\mathbb{C}$Duce's key features.

match e with p_1 -> e_1 | ... | p_n -> e_n

fun f (t_1 -> s_1; ...) p_1 -> e_1 | ... | p_n -> e_n

A pattern may either match or reject a value. When it matches:

- Binds its *capture variables* to the corresponding parts of the value and the computation can continue with the body of the branch.

Otherwise: Control is passed to the next branch.

- ML-like flavor, but much more powerful
- Express in a single pattern a computation that dynamically checks both the **structure and the type** of the matched values, and extracts deep information.

Pattern Algebra

$$
\begin{array}{llll}
p & ::= & x & \textit{capture}, x \in \mathbb{V} \\
 & | & t & \textit{type constraint}, t \in T \\
 & | & p_1 \wedge p_2 & \textit{conjunction} \\
 & | & p_1 | p_2 & \textit{alternative} \\
 & | & (p_1,p_2) & \textit{pair} \\
 & | & (x := c) & \textit{constant}, c \in \mathbb{C} \textit{ with } [\![t_c]\!] = \{c\}
\end{array}
$$

Formal foundations in [LICS'02]

Recursive patterns

Multiple occurrences of the same variable are useful in recursive patterns:

- p where p = (x & Int, _) | (_, p)
  extracts the *first* element of type Int from a sequence.
- p where p = (_, p) | (x & Int, _)
  extracts the *last* element of type `Int`.

Order is important

- **Powerful captures**:
  p where ((x & Int),p) | (_,p) | (x:=‘nil)
  when L is matched against `p`, then `x` binds the list of all integers occuring in L.

Syntactic sugar: [(x::Int | _)*]

More on Patterns

- **Precise typing**: $(t/\mathtt{p})$ = type environment for the variables in $\mathtt{p}$ when matching a value in t

t	$(t/\mathtt{p})(\mathtt{x})$
[Int String Int]	[Int Int]
[Int|String]	[Int?]
[Int* String Int]	[Int+]
[Int+ String Int]	[Int+ Int]
[(0..10)+ String]	[(0..10)+]
[(Int String)+]	[Int+]

XML-friendly Patterns

```
type IntStr = /['0'-'9']+/;;
type Bib    = <bib>[Book*];;
type Book   = <book>[Title Year Author+];;
type Year   = <year>[IntStr];;
type Title  = <title>[String];;
type Author= <author>[String];;

let fun book_title ( Book -> String )
 <book>[ <title>[t]; _ ] -> t;;

let fun book_author ( Book -> [String+])
 <book>l -> transform l with <author>[a] -> [a];;
```

- `l`, `t` and `a` are capture variables

Pattern Compilation

- Key issue to execute ℂDuce programs efficiently
- New kind of deterministic tree automata: Non Uniform Tree Automata (combination of top-down and bottom-up automata).
- Compilation schema (from patterns to automata) which uses static type info to avoid unecessary run-time checks.

$\Downarrow$

Allows for a declarative programming style

Extra support for sequences

map e with p_1 -> e_1 | ... | p_n -> e_n
transform e with p_1 -> e_1 | ... | p_n -> e_n
xtransform e with p_1 -> e_1 | ... | p_n -> e_n

- map applies some transformation to each element of a sequence.(implicit default branch: `x -> x`)
- transform each branch of the pattern is supposed to return a sequence, and all the returned sequences are concatenated together. (implicit default branch: `x -> []`)
- xtransform works on (sequences of) XML-trees. Match the patterns on each root of each tree and if it fails recursively apply to the sequence of sons.

xtransform

- Thanks to xtransform a function that puts in boldface all the links of an Xhtml document can be defined

```
let bold(x:[Xhtml]):[Xhtml]=
                  xtransform x with <a,(x)>t -> [<a,(x)>[<b>t]]
```

- Without xtransform we would be obliged to iterate on the whole DTD of XHTML.
- xtransform combines the flexibility as XSLT template programming, with the precise static typing and efficient compilation of ℂDuce's transform.

Overloading

- Static overloading: same name for a similar action in different types.
- Dynamic dispatch: reminiscent of OO programming.
  - Separation of overloading in function interface and in implementation (pattern matching) allows code sharing between different "classes".
  - Combine advantage of pattern-matching and multi-methods (dispatch according to the run-time type of several arguments)
- With higher-order: pass a single overloaded function argument to a function instead of several functions.

Extensions for queries

- ℂDuce was designed as a **programming language**.
- A small set of extra constructions (or syntactic sugar) can endow it with **query-like facilities: projection, selection, join.**
- Core ℂDuce contribution to this query language is: static typing + efficient compilation schema.

Highly Declarative Programming Interface

Projection

- projection can be defined from the transform construction.
- If e is a ℂDuce expression representing a sequence of elements and t is a type,

$$e/t$$

is syntactic sugar for:

```
transform e with <_>c ->
            transform c with (x & t) -> [x]
```

Note that **c** is bound to the content of each element in the sequence

e

Projection

Consider

```
type AddrBook = <book>content;;
type content = [(Name Addr Tel?)*];;
type Name = <name>[String];;
type Addr = <addr kind =? "home"|"work">[Street Town];;
type Street  = <street>[String];;
type Town  = <town>[String];;
type Tel  = <tel>[String];;
```

If addr_book is of type AddrBook, then

```
[addr_book]/<addr kind="home">_/<town>_
```

denotes the sequence of all town elements that occur in a “home” address in `addr_book`.

This corresponds to the XPath expression

/addr[@kind="home"]/town.

Select from where

- A `select` construction can then be defined:

  select e from p_1 in e_1,...,p_n in e_n where e'

- can be defined to be the same as:

```
transform e1 with p1 -> ...
       transform en with pn ->
          if e' then -> e else []
```

- Important

  Order is unspecified to exploit usual query optimization techniques.

Select from where

ℂDuce Style

```
select [<resultats1>[<letitre>[t] <lacrit>[r] ]]
from
        <bibliography>[ <heading>_ p::Paper*]  in [bib]
        <paper>[a::Author+ <title>t _*]        in p
        <author>"Honore de Balzac"             in a
        <reviews>[b::BibRev*]                  in [rev0]
        <book>[<title>t1 <review>r]            in b
where t1 = t ;;
```

Xquery Style

```
select <resultat2>[<letitre>([t]/_) <lacrit>([r]/_) ]
from
        p   in [bib]/<paper>_ ,
        t   in [p]/<title>_ ,
        a   in [p]/<author>_ ,
        b   in [rev0]/<book>_ ,
        t1  in [b]/<title>_ ,
        r   in [b]/<review>_
where t1=t and a=<author>"Honore de Balzac" ;;
```

Benchmarks

`xsltproc` parser for XSLT.

split	60Kb	0.3 Mb	0.6 Mb	2.5 Mb	5.2 Mb
ℂDuce 1	0.10	0.30	0.52	1.92	3.95
ℂDuce 2	0.11	0.30	0.50	1.92	3.92
ℂDuce 3	0.10	0.29	0.49	1.85	3.81
XSLT 1	0.15	0.79	1.42	5.95	12.85
XSLT 2	0.18	0.93	1.68	6.90	14.33

- The first ℂDuce version uses the pattern <person gender=g>[<name>n <children>[(mc::MPerson | fc::FPerson)*]].
- The second one uses the hand-optimized pattern <_ gender=g>[<_>n <_>[(mc::<_ gender="M">_ | fc::_)*]].
- The third ℂDuce version duplicates the main function to avoid overloading and useless computations on tags.
- The two XSLT versions use slightly different styles (two templates, or a single template with computation on tag).

Current Status and Perspectives

- Current status
  - DTD, Schema validation, Namespace, Unicode, Web Services, Interactive Sessions.
  - Distribution under MIT Licence, for Linux/Unix, Mac OS10, Windows XP (.exe).
- Perspectives
  - Polymorphism and inference
  - Modules
  - Language oriented security
  - Persitent Engine,

Current prototype (MIT Licence) at `www.cduce.org`

</Be ℂDuce'd>