ℂDuce

Véronique Benzaken (LRI U Paris Sud)

Giuseppe Castagna (ENS Paris)

Alain Frisch (ENS Paris)

http://www.cduce.org/

Introduction

- ℂDuce is a general purpose typed and higher-order functional programming language
- Adapted to XML applications: XML features are introduced through syntactic sugar over a core language
- Overtakes some limitations of XDuce
- Put most of the technical difficulties of the type system in the subtyping relation and the type algebra and keep it simple by using a semantic approach.

Overview of the talk

- XDuce and ℂDuce
- ℂDuce language
  - Types
  - Pattern matching
  - Functions (overloaded, first-class)
- Compilation
- Extensions for queries
- Implementation issues

XDuce

(H. Hosoya, B. Pierce, J. Vouillon)

- XDuce: a typed programming language for XML applications
  - value = XML document (tree)
  - type = regular tree language
  - subtyping = inclusion of languages
- A powerful pattern matching operation
  - Recursive patterns to extract information in the middle of a document
  - Precise type inference for capture variables
  - Push/pull duality reflects in patterns/expressions

XDuce $\rightsquigarrow$ CDuce (1/2)

For those who know about XDuce ...

- Recast XDuce features in a general purpose language, with a powerful type algebra
  - Interoperability with non-XML languages
  - Useful outside XML world
- First-class citizen and overloaded functions
  - Expressivity and reusability
  - Late bound overloading reminiscent of OOP style
  - Complex applications written directly in $\mathbb{C}$Duce

XDuce $\rightsquigarrow$ CDuce (2/2)

For those who know about XDuce ...

- Model XML attributes with records, instead of “commutable elements”
- Extend pattern matching
  - Capturing non-consecutive subsequences (exact typing)
  - Default values
- Finer basic types
  - Intervals for integers
  - Regular expressions for strings (with pattern matching)

Types

Types are pervasives in ℂDuce:

- Static validation

Types

Types are pervasives in ℂDuce:

- Static validation
  - E.g.: does the transformation produce valid XHTML ?

Types

Types are pervasives in ℂDuce:

- Static validation
  - E.g.: does the transformation produce valid XHTML ?
- Type-driven semantics

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

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

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

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

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)
  (use `[Char*]` instead of `String` ?)
- types constructors
  product types (t_1, t_2)
  record types $\{\ a_1 = t_1\ ;\ \ldots\ ;\ a_n = t_n\ \}$, $\{\ |\ a_1 = t_1\ ;\ \ldots\ ;\ a_n = t_n\ |\ \}$
  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:

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.

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.

- Natural set-theoretic interpretation of boolean connectives and subtyping relation.

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.

- Natural set-theoretic interpretation of boolean connectives and subtyping relation.
- Circularity because of first-class functions (values whose type is given by the type system, which depends on the subtyping relation).

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.

- Natural set-theoretic interpretation of boolean connectives and subtyping relation.
- Circularity because of first-class functions (values whose type is given by the type system, which depends on the subtyping relation).
- Bootstrap method to break this circularity and continue with the nice semantic definition of subtyping. See:

  Frisch, Castagna, and Benzaken.

  Semantic Subtyping. *LICS'02*

Encoding XML: Sequences

Sequences are encoded using pairs and a terminator `'nil`.
A sequence of values $v_1, \ldots, v_n$ is written

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

and is syntactic sugar for

$$\texttt{(}v_1\texttt{,(...,(}v_n\texttt{, 'nil)...))}.$$

Sequence types are written:

$$[\textit{tyregexp}]$$

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

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

Encoding XML: XML elements

An XML element

<tag $a_1 = v_1$ ... $a_n = v_n$> elem_seq </tag>

is written in ℂDuce as

<tag $a_1 = v_1$ ... $a_n = v_n$>[elem_seq]

which is syntactic sugar for

('tag, ({$a_1 = v_1$; ...; $a_n = v_n$}, [elem_seq]))

(namespaces: use pairs instead of a single atom for tags)

Similarly for types, e.g:

```
type Ul = <ul>[Li+]

type Li = <li>[Flow*]
```

XML 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>
```

ℂDuce syntax

```
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 type

```
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];;
```

Loading XML documents

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 ℂDuce's key features.

```
match e with p1 -> e1 | ...  | pn -> en
fun f (t1 -> s1; ...)  p1 -> e1 | ...  | pn -> en
```

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.

Core pattern algebra

p	$::=$	x	capture
	$\mid$	t	type constraint
	$\mid$	p_1 & p_2	conjunction
	$\mid$	p_1 | p_2	alternative
	$\mid$	(p_1 , p_2)	pair
	$\mid$	{ l = p }	record
	$\mid$	(x := c)	constant

Plus:

- Recursive patterns
- Syntactic sugar for sequences (regular expressions patterns)
- String regular expression patterns

Matching

v/p = match the value v against the pattern p.

The result is either Ω (failure) or a substitution $Var(p) \to$ Values.

Matching

v/p = match the value v against the pattern p.

The result is either Ω (failure) or a substitution $Var(p) \to$ Values.

$$
\begin{array}{lcll}
v/t & = & \{\} & \text{if } v \in t \\
v/t & = & \Omega & \text{if } v \in \neg t \\
v/x & = & \{x \mapsto v\} & \\
v/p_1 \& p_2 & = & v/p_1 \otimes v/p_2 & \\
v/p_1 \,|\, p_2 & = & v/p_1 & \text{if } v/p_1 \neq \Omega \\
v/p_1 \,|\, p_2 & = & v/p_2 & \text{if } v/p_1 = \Omega \\
(v_1, v_2)/(p_1, p_2) & = & v_1/p_1 \otimes v_2/p_2 & \\
v/(p_1, p_2) & = & \Omega & \text{if } v \text{ is not a pair} \\
v/(x := c) & = & \{x \mapsto c\} &
\end{array}
$$

Matching

v/p = match the value v against the pattern p.

The result is either Ω (failure) or a substitution $\mathit{Var}(p) \to$ Values.

$$
\begin{array}{lcll}
v/t & = & \{\} & \text{if } v \in t \\
v/t & = & \Omega & \text{if } v \in \neg t \\
v/x & = & \{x \mapsto v\} & \\
v/p_1 \& p_2 & = & v/p_1 \otimes v/p_2 & \\
v/p_1 \,|\, p_2 & = & v/p_1 & \text{if } v/p_1 \neq \Omega \\
v/p_1 \,|\, p_2 & = & v/p_2 & \text{if } v/p_1 = \Omega \\
(v_1, v_2)/(p_1, p_2) & = & v_1/p_1 \otimes v_2/p_2 & \\
v/(p_1, p_2) & = & \Omega & \text{if } v \text{ is not a pair} \\
v/(x := c) & = & \{x \mapsto c\} &
\end{array}
$$

$\mathbb{C}$Duce Semantics:

$$
\begin{array}{lcll}
\texttt{match}\ v\ \texttt{with}\ p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 & \to & e_1[v/p_1] & \text{if } v/p_1 \neq \Omega \\
\texttt{match}\ v\ \texttt{with}\ p_1 \Rightarrow e_1 \,|\, p_2 \Rightarrow e_2 & \to & e_2[v/p_2] & \text{if } v/p_1 = \Omega
\end{array}
$$

Sequence capture variable

Example:

```
[ ((x ::  Int Int) (y ::  String)?)* ]
```

is syntactic sugar for the recursive pattern:

```
p where p = (x & Int, q) | (x & y & 'nil)
    and q = (x & Int, r)
    and r = (y & String, p) | p
```

- If a variable x appears on both sides of a pair pattern (p_1, p_2), the results from the two patterns are paired together. E.g:

  ```
  match (1,(2,3)) with (x,(_,x)) -> x
  ```

  returns

  ```
  (1,3)
  ```

Patterns

- **Powerful captures**:

```
p where ((x & Int),p) | (_,p) | (x:='nil)
```

When a list L is matched against `p`, then `x` binds the list of all integers occuring in L.

Patterns

- **Powerful captures**:
  `p where ((x & Int),p) | (_,p) | (x:='nil)`
  When a list L is matched against `p`, then `x` binds the list of all integers occuring in L.
- **Precise typing**:
  $(t/\mathtt{p})$ = type environment for the variables in `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 Int]`	`[(0..10)+ Int]`
`[(Int String)+]`	`[Int+]`

Typing rule for Pattern matching

- Let B denote the sequence of branches p_1 `->` e_1 `|` ... `|` p_n `->` e_n.
- The rule below derives the typing judgments $\Gamma \vdash t/B \Rightarrow s$, whose intended meaning is: matching a value of type t against the sequence of branches B always succeeds and every possible result is of type s.

$$\frac{\begin{array}{l} t \leq \llbracket p_1 \rrbracket \mid \ldots \mid \llbracket p_n \rrbracket \\ t_i = ((t \setminus \llbracket p_1 \rrbracket) \setminus \ldots \setminus \llbracket p_{i-1} \rrbracket) \,\&\, \llbracket p_i \rrbracket \\ \begin{cases} \Gamma, (t_i/p_i) \vdash e_i : s_i & \text{if } t_i \not\simeq \texttt{Empty} \\ s_i = \texttt{Empty} & \text{if } t_i \simeq \texttt{Empty} \end{cases} \\ s = s_1 \mid \ldots \mid s_n \end{array}}{\Gamma \vdash t/B \Rightarrow s}$$

Pattern type inference

- The real work is done by $\llbracket p \rrbracket$ and (t/p).
- They are defined as the least solution of some set of equations.

$$
\begin{array}{lcllcl}
\llbracket x \rrbracket & = & \mathsf{Any} & \llbracket p_1 \mid p_2 \rrbracket & = & \llbracket p_1 \rrbracket \mid \llbracket p_2 \rrbracket \\
\llbracket t \rrbracket & = & t & \llbracket p_1 \& p_2 \rrbracket & = & \llbracket p_1 \rrbracket \,\&\, \llbracket p_2 \rrbracket \\
\llbracket (x := c) \rrbracket & = & \mathsf{Any} & \llbracket (p_1, p_2) \rrbracket & = & (\llbracket p_1 \rrbracket, \llbracket p_2 \rrbracket)
\end{array}
$$

$$
\begin{array}{lcl}
(t/x)(x) & = & t \\
(t/(p_1 \mid p_2))(x) & = & ((t \& \llbracket p_1 \rrbracket)/p_1)(x) \mid ((t \setminus \llbracket p_1 \rrbracket)/p_2)(x) \\
 & \vdots &
\end{array}
$$

List comprehension

map l with p_1 -> e_1 | ... | p_n -> e_n
transform l with p_1 -> e_1 | ... | p_n -> e_n

- `map` applies some transformation to each element of a sequence.
  (implicit default branch: `x -> x`)
- Very precise typing. E.g:

```
l:[(Int String)*] |- map l with Int -> ‘A | String -> ‘B : [(‘A ‘B)*]
```

- `transform` is a variant where each branch of the pattern is supposed to return a (possibly empty) sequence, and all the returned sequences, for each element in the source sequence, are concatenated together.
  (implicit default branch: `x -> []`)

Functions

- The general form for a function is:
  fun f (t_1 -> s_1; ...)
  p_1 -> e_1
  | ...
  | p_n -> e_n
- Functions are first-class values.
- f is optional (used for recursive functions)
- The t_i -> s_i are constraints to be checked by the type checker.
- All top-level function declarations (`let fun ...`) are mutually recursive.

Simple functions

```
fun add5 ( 0..100 -> 0..100 ) x -> x + 5;;
```

is rejected with:

```
Type error.
This expression has type 5..105
Expected type: 0..100
Residual type: 101..105
Constraint not satisfied: 0..100 -> 0..100
```

Simple functions

```
fun wrap (0..100 -> 0..10)
  | x & 0..10 -> x
  | x -> wrap (x - 10);;

type Expr =
    (‘add | ‘mul | ‘sub | ‘div) * Expr * Expr
  | Int;;

let fun eval ( Expr -> Int )
  | (‘add,x,y) -> eval x + eval y
  | (‘mul,x,y)  -> eval x * eval y
  | (‘sub,x,y) -> eval x - eval y
  | (‘div,x,y)  -> eval x / eval y
  | n -> n;;   (* here n has type Int *)
```

Simple functions

```
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

Simple functions

```
type Flat_bib = [(Title Year Author+)*];;

let fun unflatten ( Flat_bib -> [Book*])
 | [ b::(Title Year Author+); r ] ->
        (<book>b, unflatten r)
 | [] -> [];;
```

- b is a sequence capture variable

Simple functions

```
type Flat_bib = [(Title Year Author+)*];;
type TitleYear = [(Title Year)*];;

let fun remove_authors1 (Flat_bib -> TitleYear)
 [ (Author | x::Any)* ]  -> x;;

let fun remove_authors2 (Flat_bib -> TitleYear)
 [ (x::(Title Year) | _)* ]  -> x;;

let fun remove_authors3 (Flat_bib -> TitleYear)
 [ ((x::Title) | (x::Year) | _)* ] -> x;;
```

- All the matched subsequences for a sequence capture variable are concatenated together.

Overloading

```
let fun add( (Int,Int) -> Int; (String,String) -> String )
 | (x & Int, y) -> x + y
 | (x & String, y ) -> x ^ y;;

type Person = <person gender=("M"|"F")>[...];;

let fun title (Book -> String; Person -> `mister | `miss)
 | <book>[<author>[a]; _] -> a
 | <person gender="M">_ -> `mister
 | <person gender="F">_ -> `miss;;
```

Overloading

```
type Movie = <movie>[Title <producer>[String]];;
type Film  = <film>[Titre <producteur>[String]];
type Livre = <livre>[Titre Annee Auteur+];;
type Titre = <titre>[String];;
...
let fun to_french (String -> String; Book -> Livre; Movie -> Film;
                   Title -> Titre; ...)
 <(t)>x ->
   let t'  = match t with
   | 'book -> 'livre | 'movie -> 'film
   | 'title -> 'titre | 'year -> 'annee
   | 'author -> 'auteur | 'producer -> 'producteur
   | y -> y
   in
   <(t')>(map x with e -> to_french e);;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;


let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)* ] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;


let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)* ] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)* ] ] ->
    let tag = match gen with "M" -> `man | "F" -> `woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)* ] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)* ] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;


let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)* ] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;


let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)* ] ] ->
    let tag = match gen with "M" -> 'man | "F" -> 'woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;


let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in        mc:[MPerson*] ⇒ s:[Man*]
    let d = map fc with x -> sort x in        fc:[FPerson*] ⇒ d:[Woman*]
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;

let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Overloading: serious example

```
type Person = FPerson | MPerson;;
type FPerson = <person gender ="F">[ Name Children ];;
type MPerson = <person gender ="M">[ Name Children ];;
type Children = <children>[Person*];;
type Name = <name>[String];;

type Man = <man>[ Name Sons Daughters ];;
type Woman = <woman>[ Name Sons Daughters ];;
type Sons = <sons>[ Man* ];;
type Daughters = <daughters>[ Woman* ];;
```

```
let fun sort (MPerson -> Man ; FPerson -> Woman)
  <person gender=gen>[ n <children>[ (mc::MPerson | fc::FPerson)*] ] ->
    let tag = match gen with "M" -> ‘man | "F" -> ‘woman in
    let s = map mc with x -> sort x in
    let d = map fc with x -> sort x in
    <(tag)>[ n <sons>s <daughters>d ];;
```

Although `sort:Person -> Man | Woman`, the declaration
`fun sort (Person -> Man | Woman)`
wouldn’t type-check (fails for the recursive calls).

Overloading

- Static overloading: same name for a similar action in different types.
- Dynamic dispatch: reminiscent of OO programming (without encapsulation ;))
  - Separation of overloading in function interface and in implementation (pattern matching) allows code sharing between different "classes".
  - Combine advantage of pattern-matching (that can look deep inside the value to be dispatched) 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.
- (To be investigated) Incremental programming to follow the evolution of schemas.

Higher-order

```
type Bool = ‘true | ‘false;;
let fun present_bib ( (Bib, (Book -> Bool)) -> Html )
  (bib, highlight) ->
     transform bib with
        b & <book>[<title>[t]; _] ->
          match (highlight b) with
            | ‘true -> [ <em>[t] <br>[] ]
            | ‘false -> [ t <br>[] ];;
...
present_bib
  (bib0, fun (Book -> Bool)
     | <book>[<title>[/.* "Object" .*/]; _] -> ‘true
     | _ -> ‘false);;
  );;
...
present_bib
  (bib0, fun (Book -> Bool)
     <book>[_* <year>[y] _*] -> (int y >= 2000));;
```

First-class functions

Other examples in the field of XML processing:

- Parametrize the behavior of complex transformations with basic transformations over subparts of the document.
- Dynamic template system (compute or extract from a database a “presentation function”).
- Compile a user-provided function at runtime and check its type.

```
let present =
 match eval (cgi_arg "present") with
  | ‘compile_error -> error "invalid CDuce code"
  | ‘runtime_error -> error "error when evaluating code"
  | (‘value, f & (Book -> Html)) -> f
  | _ -> error "invalid type";;

|- present : Book -> Html
```

Compiling pattern matching

- Given n patterns $p_1, \ldots, p_n$, produce code to select the branch of a pattern matching p_1 -> e_1 | ... | p_n -> e_n.
- The type system gives a static type t for the matched expression and type safety ensures that the matched value has type t. Use this information.

Compiling pattern matching

- Given n patterns $p_1, \ldots, p_n$, produce code to select the branch of a pattern matching p_1->e_1 | ... | p_n->e_n.
- The type system gives a static type t for the matched expression and type safety ensures that the matched value has type t. Use this information.
- Example:

```
fun (<c>[A+|B+] -> Int)
    <c>[A+] -> 0
  | <c>[B+] -> 1;;
```

Compiling pattern matching

- Given n patterns $p_1, \ldots, p_n$, produce code to select the branch of a pattern matching p_1->e_1 | ... | p_n->e_n.
- The type system gives a static type t for the matched expression and type safety ensures that the matched value has type t. Use this information.
- Example:

```
fun (<c>[A+|B+] -> Int)
      <c>[A+] -> 0
    | <c>[B+] -> 1;;
```

- Naive compilation schema: the argument may be run through completely several times.

Compiling pattern matching

- Given n patterns $p_1, \ldots, p_n$, produce code to select the branch of a pattern matching p_1->e_1 | ... | p_n->e_n.
- The type system gives a static type t for the matched expression and type safety ensures that the matched value has type t. Use this information.
- Example:

```
fun (<c>[A+|B+] -> Int)
    <c>[A+] -> 0
  | <c>[B+] -> 1;;
```

- Naive compilation schema: the argument may be run through completely several times.

How fast if `A=<a>[...]` **and** `B=<b>[...]` **!!**

```
fun (<c>[A+|B+] -> Int) <_>[<a>_ _*] -> 0
| _ -> 1
```

Compiling pattern matching

- In general, a naive approach to compiling pattern matching may yield multiple runs and backtracking through the matched value.
- Without capture variable, this is the problem of recognizing regular (binary) tree languages.
- Idea: deterministic bottom-up tree automata can eliminate backtracking.
- Determinization creates huge and intractable automata (uniform computation, disregarding the current position in the tree)

Compiling pattern matching

- Idea: propagate a “state” from the root and from the left (hybrid top-down and bottom-up automata).
- When matching a value (v_1, v_2), perform some computation of v_1 and, according to the result, perform another computation on v_2.
- By using static type information, it is possible to avoid checking whole parts of the matched value.
- E.g.: to decide whether $v = (v_1, v_2)$ has type (t_1, t_2) when it is known to have type $(t_1, t_2) \,|\, (s_1, s_2)$ (with $t_1 \& s_1 \simeq \texttt{Empty}$), one just has to look at v_1.
- Vague description of the algorithm: put patterns in some kind of disjunctive normal form.

Streaming

- Streaming processing can be achieved by giving ℂDuce a lazy semantics.
- To make this feasible, pattern matching must not inspect parts of the value which are not needed.

Extensions for queries

- For XML, the boundary between programming languages, transformation languages and query languages/algebras is not easy to draw.
- 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.
- We can describe encodings for these constructions, but we want to give the compiler some freedom and a large latitude in query optimization.
- Core ℂDuce contribution to this potential query language is: static typing + efficient compilation schema.

Queries: Projection

- As an example, we introduce projection.
- E.g.: projection can be defined from the `transform` construction.
- If e is a $\mathbb{C}$Duce 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]
```

- Examples:

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

[bib]/<book>[Title Year Author Author]/<title>
```

Equivalences

- ℂDuce compiler could take profit of equivalences:

```
transform (transform e with p1 -> e1) with p2 ->
                                              e2
```

$\leadsto$

```
transform e with p1 -> transform e1 with p2 -> e2
```

Queries: joins

- To implement joins, we introduce a cartesian product operator.
- if $s_1, \ldots s_n$ are sequences, `prod(`s_1`,`s_2`,...,`s_n`)` evaluates to a sequence containing all the `(`v_1`,...,`v_n`)` where v_i appears in s_i.
- We let the order of this sequence unspecified to allow optimizations.
- **Example:** let s_1 `= [ 1 2 ]` and s_2 `= [ "A" "B" ]`, then the expression `prod(`s_1`,`s_2`)` evaluates to some permutation of `[ (1,"A") (1,"B") (2,"A") (2,"B") ]`.

Queries

The typing rule for `prod` is the following:

$$\frac{\Gamma \vdash e_1 : [t_1*] \quad \Gamma \vdash e_2 : [t_2*] \quad \ldots \quad \Gamma \vdash e_n : [t_n*]}{\Gamma \vdash \texttt{prod(}e_1\texttt{, ..., }e_n\texttt{) : [(}t_1\texttt{,}t_2\texttt{,...,}t_n\texttt{)*]}}$$

- We restrict the e_i's to be homogeneous sequences (i.e, sequences whose elements are all of the same type)
- which yields the product to be an homogeneous sequence, as well.

Select from where

- A `select` construction can then be easily defined since

  `select` e `from` x_1 `in` e_1`,...,`x_n `in` e_n `where` e'

- can be defined to be the same as:
  `transform prod(`e_1`,...,`e_n`) with`
  `(`x_1`,...,`x_n`) -> if` e' `then` e `else []`

Select from where

join between two documents `bib0` and `rev0` of types `Bib` and `Review` respectively.

```
type Review =<reviews>[BibRev*];;
type BibRev = <book>[<title>[String]
                     <review>[String]];;

let rev0 = <reviews>[
    <book>[
       <title>["Persistent Object Systems"]
       <review>["Good topic"]]
    <book>[
       <title>["Les illusions perdues"]
       <review>["A promising writer"]]];;
```

Select from where

```
select <bookr>([b]/<title> @ [b]/<author> @ [r]/<review>)
from b in [bib0]/<book> , r in [rev0]/<book>
where [b]/<title> = [r]/<title>
```

yielding the following result:

```
[
<bookr>[
   <title>["Persistent Object Systems"]
   <author>["M. Atkinson"]
   <author>["V. Benzaken"]
   <author>["D. Maier"]
   <review>["Good topic"]]
<bookr>[
   <title>...]
]
```

Select from where

Need to access to the content of an element when it is a sequence of just one element.

e.<tag>

```
select
   <book review=r.<review>>
     ([b]/<title>@
      [b]/<book>@
      [<price>["unknown"]])
from b in [bib0]/<book> , r in [rev0]/<book>
where b.<year>="2000" and (b.<title> = r.<title>)
```

Typing rule

- The type system authorizes to access the content of an element only if the element occurs exactly once and it contains a sequence of length 1, as stated by:

$$\frac{\Gamma \vdash e : [(\neg\texttt{<t}\ r\texttt{>})^*\ \texttt{<t}\ r\texttt{>}[s]\ (\neg\texttt{<t}\ r\texttt{>})^*]}{\Gamma \vdash e\texttt{.<t}\ r\texttt{>} : s}$$

- (r is an optional attribute specification)

Typing rule

- easily deduced from the encoding of e`.<t` r`>`:

  `match` e `with [(¬<t` r`>)* <t` r`>[x] (¬<t` r`>)*] -> x.`

- This construct corresponds to the XSLT element value-of where,

  `(/<a>/<b>/<c>).<c>`

- would be written as

  <xsl:value-of select="/a/b/c">.

Implementation

- Current prototype at `http://www.cduce.org`
- A lot of non-trivial implementation issues. E.g.:
  - Subtyping algorithm: subtle caching mechanisms, short-cuts, ...
  - Maximal sharing and unique representation of recursive types.

Open issues

- Polymorphism and inference (Alain and Giuseppe)
- Language oriented security (Giuseppe, Marwan, Véronique)
- Formal study of the query sub-part and potential optimisations (Véronique and whoever interested)
- Dependent types (Alain, Giuseppe)
- Module system and incremental programming (Alain, Giuseppe)
- Interfacing with other languages
- Interfacing with native XML databases (indexes, in-place modifications)