$\mathbb{C}$*Duce: a typeful and efficient language for XML*

Véronique Benzaken, Giuseppe Castagna, Alain Frisch, Marwan Burelle, Cédric Miachon

http://www.cduce.org/

Summary of the talk

- Introduction to XML programming
- XML in ℂ**Duce**: document and types
- Types
- Pattern matching
- Functions
- Type errors
- Query language
- Ongoing work. Around ℂ**Duce**

Programming with XML

- Level 0: textual representation of XML documents
  - AWK, sed, Perl

Programming with XML

- Level 0: textual representation of XML documents
  - AWK, sed, Perl
- Level 1: abstract view provided by a parser
  - SAX, DOM, . . .

Programming with XML

- Level 0: textual representation of XML documents
  - AWK, sed, Perl
- Level 1: abstract view provided by a parser
  - SAX, DOM, . . .
- Level 2: untyped XML-specific languages
  - XSLT, XPath

Programming with XML

- Level 0: textual representation of XML documents
  - AWK, sed, Perl
- Level 1: abstract view provided by a parser
  - SAX, DOM, . . .
- Level 2: untyped XML-specific languages
  - XSLT, XPath
- Level 3: XML types taken seriously (aka: related work)
  - XDuce, Xtatic
  - XQuery
  - . . .

Presentation

ℂ**Duce**:

- XML-oriented
- type-centric
- general-purpose features
- efficient (faster than XSLT, XHaskell, Kawa, Qizx, at least)

Intended uses:

- Small “adapters” between different XML applications
- Larger applications
- Web applications, web services

Status of the implementation

- Public release available for download
  (+ online web prototype to play with).
- Production of an intermediate code and execution with JIT compilation of pattern matching.
- Quite efficient, but more optimizations are possible (and considered, e.g.: generate OCaml code).
- Integration with standards:
  - Unicode, XML, Namespaces: fully supported.
  - DTD: external `dtd2cduce` tool.
  - XML Schema: is implemented at a much deeper level.

XML-oriented + data-centric

- XML literals : in the syntax.
- XML fragments : first-class citizens, not embedded in objects.

```
<program>
  <date day="monday">
    <invited>
       <title>   Conservation of information</title>
       <author>   Thomas Knight, Jr.</author>
    </invited>
    <talk>
       <title>   Scripting the type-inference process</title>
       <author>   Bastiaan Heeren</author>
       <author>   Jurriaan Hage</author>
       <author>   Doaitse Swierstra</author>
    </talk>
  </date>
</program>
```

XML-oriented + data-centric

- XML literals : in the syntax.
- XML fragments : first-class citizens, not embedded in objects.

```
<program>[
  <date day="monday">[
    <invited>[
       <title>[ 'Conservation of information' ]
       <author>[ 'Thomas Knight, Jr.' ]
    ]
    <talk>[
       <title>[ 'Scripting the type-inference process' ]
       <author>[ 'Bastiaan Heeren' ]
       <author>[ 'Jurriaan Hage' ]
       <author>[ 'Doaitse Swierstra' ]
    ]
  ]
]
```

Types

Types are pervasive in $\mathbb{C}$Duce:

- **Static validation**
  - E.g.: does the transformation produce valid XHTML ?
- **Type-driven semantics**
  - Dynamic dispatch
  - Overloaded functions
- **Type-driven compilation**
  - Optimizations made possible by static types
  - Avoids unnecessary and redundant tests at runtime
  - Allows a more declarative style

Typed XML

$$\vdash v : t$$

v ==

```
<program>[
  <date day="monday">[
    <invited>[ <title>[ 'Conservation of information' ]
               <author>[ 'Thomas Knight, Jr.' ] ]
    <talk>[ <title>[ 'Scripting the type-inference process' ]
            <author>[ 'Bastiaan Heeren' ]
            <author>[ 'Jurriaan Hage' ]
            <author>[ 'Doaitse Swierstra' ] ] ] ]
```

t ==

```
<program>[
  <date day="monday">[
    <invited>[ <title>[ 'Conservation of information' ]
               <author>[ 'Thomas Knight, Jr.' ] ]
    <talk>[ <title>[ 'Scripting the type-inference process' ]
            <author>[ 'Bastiaan Heeren' ]
            <author>[ 'Jurriaan Hage' ]
            <author>[ 'Doaitse Swierstra' ] ] ] ]
```

Typed XML

$$\boxed{\vdash v : t}$$

v ==

```
<program>[
  <date day="monday">[
    <invited>[ <title>[ 'Conservation of information' ]
               <author>[ 'Thomas Knight, Jr.' ] ]
    <talk>[ <title>[ 'Scripting the type-inference process' ]
            <author>[ 'Bastiaan Heeren' ]
            <author>[ 'Jurriaan Hage' ]
            <author>[ 'Doaitse Swierstra' ] ] ] ]
```

t ==

```
<program>[
  <date day=String>[
    <invited>[ <title>[ PCDATA ]
               <author>[ PCDATA ] ]
    <talk>[ <title>[ PCDATA ]
            <author>[ PCDATA ]
            <author>[ PCDATA ]
            <author>[ PCDATA ] ] ] ]
```

Typed XML

$$\boxed{\vdash v : t}$$

```
v  ==
  <program>[
    <date day="monday">[
      <invited>[ <title>[ 'Conservation of information' ]
                 <author>[ 'Thomas Knight, Jr.' ] ]
      <talk>[ <title>[ 'Scripting the type-inference process' ]
              <author>[ 'Bastiaan Heeren' ]
              <author>[ 'Jurriaan Hage' ]
              <author>[ 'Doaitse Swierstra' ] ] ] ]

t  ==
  <program>[
    <date day=String>[
      <invited>[ Title Author ]
      <talk>[ Title Author Author Author  ] ] ]

type Author = <author>[ PCDATA ]
type Title  = <title>[ PCDATA ]
```

Typed XML

$$\boxed{\vdash v : t}$$

v ==

```
<program>[
  <date day="monday">[
    <invited>[ <title>[ 'Conservation of information' ]
               <author>[ 'Thomas Knight, Jr.' ] ]
    <talk>[ <title>[ 'Scripting the type-inference process' ]
            <author>[ 'Bastiaan Heeren' ]
            <author>[ 'Jurriaan Hage' ]
            <author>[ 'Doaitse Swierstra' ] ] ] ]
```

t ==

```
<program>[
  <date day=String>[
    <invited>[ Title Author+ ]
    <talk>[ Title Author+ ] ] ]

type Author = <author>[ PCDATA ]
type Title  = <title>[ PCDATA ]
```

Typed XML

$$\vdash v : t$$

```
v  ==
  <program>[
    <date day="monday">[
      <invited>[ <title>[ 'Conservation of information' ]
                 <author>[ 'Thomas Knight, Jr.' ] ]
      <talk>[ <title>[ 'Scripting the type-inference process' ]
              <author>[ 'Bastiaan Heeren' ]
              <author>[ 'Jurriaan Hage' ]
              <author>[ 'Doaitse Swierstra' ] ] ] ]

t  ==    Program

type Program = <program>[ Day* ]
type Day = <date day=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]
type Author = <author>[ PCDATA ]
type Title  = <title>[ PCDATA ]
```

Types

- Types describe values.
- A natural notion of subtyping:

$$t \leq s \iff [\![t]\!] \subseteq [\![s]\!]$$

where

$$[\![t]\!] = \{v \mid \vdash v : t\}$$

- Problem: circular definition between subtyping and typing
  - Bootstrap method to remain set-theoretic (cf. LICS '02).

Pattern Matching: ML-like flavor

- ML-like flavor:

```
match e with <date day=d>_ -> d

type E = <add>[Int Int] | <sub>[Int Int]
fun eval (E -> Int)
 | <add>[ x y ] -> x + y
 | <sub>[ x y ] -> x - y
```

Pattern Matching: ML-like flavor

- ML-like flavor:

```
match e with <date day=d>_ -> d

type E = <add>[Int Int] | <sub>[Int Int]
fun eval (E -> Int)
 | <add>[ x y ] -> x + y
 | <sub>[ x y ] -> x - y
```

- Beyond ML: patterns are “types with capture variables”

```
match e with
 | x & Int -> ...
 | x & Char -> ...

let doc =
  match (load_xml "doc.xml") with
    | x & DocType -> x
    | _ -> raise "Invalid input !";;
```

Pattern Matching: beyond ML

- Regular expression and capture:

```
fun (Invited|Talk -> [Author+])
   <_>[ Title x::Author* ] -> x
```

Pattern Matching: beyond ML

- Regular expression and capture:

```
fun (Invited|Talk -> [Author+])
   <_>[ Title x::Author* ] -> x
```

```
fun ([(Invited|Talk|Event)*] -> ([Invited*], [Talk*]))
   [ (i::Invited | t::Talk | _)* ] -> (i,t)
```

Pattern Matching: beyond ML

- Regular expression and capture:

```
fun (Invited|Talk -> [Author+])
   <_>[ Title x::Author* ] -> x
```

```
fun ([(Invited|Talk|Event)*] -> ([Invited*], [Talk*]))
   [ (i::Invited | t::Talk | _)* ] -> (i,t)
```

```
fun parse_email (String -> (String,String))
   | [ local::_* '@' domain::_* ] -> (local,domain)
   | _ -> raise "Invalid email address"
```

Compilation of pattern matching

- **Problem:** implementation of pattern matching
- **Result:** A new kind of push-down tree automata.
  - $\rightsquigarrow$ Non-backtracking implementation
  - $\rightsquigarrow$ Uses static type information
  - $\rightsquigarrow$ Allows a more declarative style.

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

fun ( A|B -> Int)    A -> 0 | B -> 1
   ≃
fun ( A|B -> Int) <a>_ -> 0 | _ -> 1
```

Functions

- Overloaded, first-class, subtyping, name sharing, code sharing...

```
type Program = <program>[ Day* ]
type Day = <date day=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]

let patch_program
  (p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program] =
  xtransform p with (Invited | Talk) & x -> [ (f x) ]
```

Functions

- Overloaded, first-class, subtyping, name sharing, code sharing...

```
type Program = <program>[ Day* ]
type Day = <date day=String>[ Invited? Talk+ ]
type Invited = <invited>[ Title Author+ ]
type Talk = <talk>[ Title Author+ ]

let patch_program
  (p :[Program], f :(Invited -> Invited) & (Talk -> Talk)):[Program] =
  xtransform p with (Invited | Talk) & x -> [ (f x) ]

let first_author ([Program] -> [Program];
                  Invited -> Invited;
                  Talk -> Talk)
| [ Program ] & p -> patch_program (p,first_author)
| <invited>[ t  a  _* ] -> <invited>[ t a ]
| <talk>[ t  a  _* ] -> <talk>[ t a ]

(* we can replace the last two branches with:
  <(k)>[ t a _* ] -> <(k)>[ t a ]
*)
```

Precise type errors

```
type Title = <title>String
type Author = <author>String
type Talk = <talk>[ Title Author+ ]


let x : Talk = <talk>[ <author>[ 'G. Castagna' ] <title>[ 'CDuce' ] ]
```

$\rightsquigarrow$

```
let x : Talk = <talk>[ <author>[ 'G. Castagna' ] <title>[ 'CDuce' ] ]
This expression should have type:
'title
but its inferred type is:
'author
which is not a subtype, as shown by the sample:
'author
```

Precise type errors

```
type Title = <title>String
type Author = <author>String
type Talk = <talk>[ Title Author+ ]

fun mk_talk(s : String) : Talk = <talk>[ <title>s ]
```

$\rightsquigarrow$

```
fun mk_talk(s : String) : Talk = <talk>[ <title>s ]
This expression should have type:
[ Author+ ]
but its inferred type is:
[  ]
which is not a subtype, as shown by the sample:
[  ]
```

Precise type errors

```
type Title = <title>String
type Author = <author>String
type Talk = <talk>[ Title Author+ ]
type Invited = <invited>[ Title Author+ ]
type Day = <date>[ Invited? Talk+ ]

fun (Day -> [Talk+]) <date>[ _ x::_*] -> x
```

$\rightsquigarrow$

```
fun (Day -> [Talk+]) <date>[ _ x::_*] -> x
This expression should have type:
[ Talk+ ]
but its inferred type is:
[ Talk* ]
which is not a subtype, as shown by the sample:
[  ]
```

Subtle type errors

- Removing `<li>` elements from XHTML ?

```
fun (x : [Xhtml]) : [Xhtml] = xtransform x with <li>s -> [ ]
```

is ill-typed (since empty `<ul>` = invalid XHTML)

- Empty types detected

```
type Person = <person>[ Name Children ]
type Children = <children>[Person+]
type Name = <name>[PCDATA]

Warning at chars 57-76:
type Children =  <children>[Person+]
This definition yields an empty type for Children
Warning at chars 14-39:
type Person =  <person>[ Name Children ]
This definition yields an empty type for Person
```

Other features

- General-purpose: records, tuples, integers, exceptions, references, . . .
- String + regular expressions (types/patterns)
- Boolean connectives (types/patterns)
- Other iterators

CDuce Query Language: ℂQL

XQuery:

```
<books-with-prices>
  { for
        $b in $biblio//book,
        $a in $amazon//entry
    where $b/title = $a/title and $b/@year > 1990
    return
        <book-with-prices>
            { $b/title }
            <price-amazon>{ $a/price/text() }
            </price-amazon>
            <price-bn>{ $b/price/text() }
            </price-bn>
        </book-with-prices>  }
</books-with-prices>
```

CDuce Query Language: $\mathbb{C}$QL

XQuery = SQL for XML? More likely = OQL for XML

```
<books-with-prices>
  { for
        $b in $biblio//book,
        $a in $amazon//entry
    where $b/title = $a/title and $b/@year > 1990
    return
        <book-with-prices>
            { $b/title }
            <price-amazon>{ $a/price/text() }
            </price-amazon>
            <price-bn>{ $b/price/text() }
            </price-bn>
        </book-with-prices>  }
</books-with-prices>
```

CDuce Query Language: ℂQL

CDuce:

```
<books-with-prices>
   select <book-with-price>[t1
                            <price-amazon>p2
                            <price-bn>p1 ]
     from b  in [biblio]/<book>_ ,
          y  in [b]/@year,
          t1 in [b]/<title>_,
          e  in [amazon]/<entry>_,
          t2 in [e]/<title>_,
          p2 in [e]/<price>_/_,
          p1 in [b]/<price>_/_
    where t1=t2 and y>>1990
```

CDuce Query Language: ℂQL

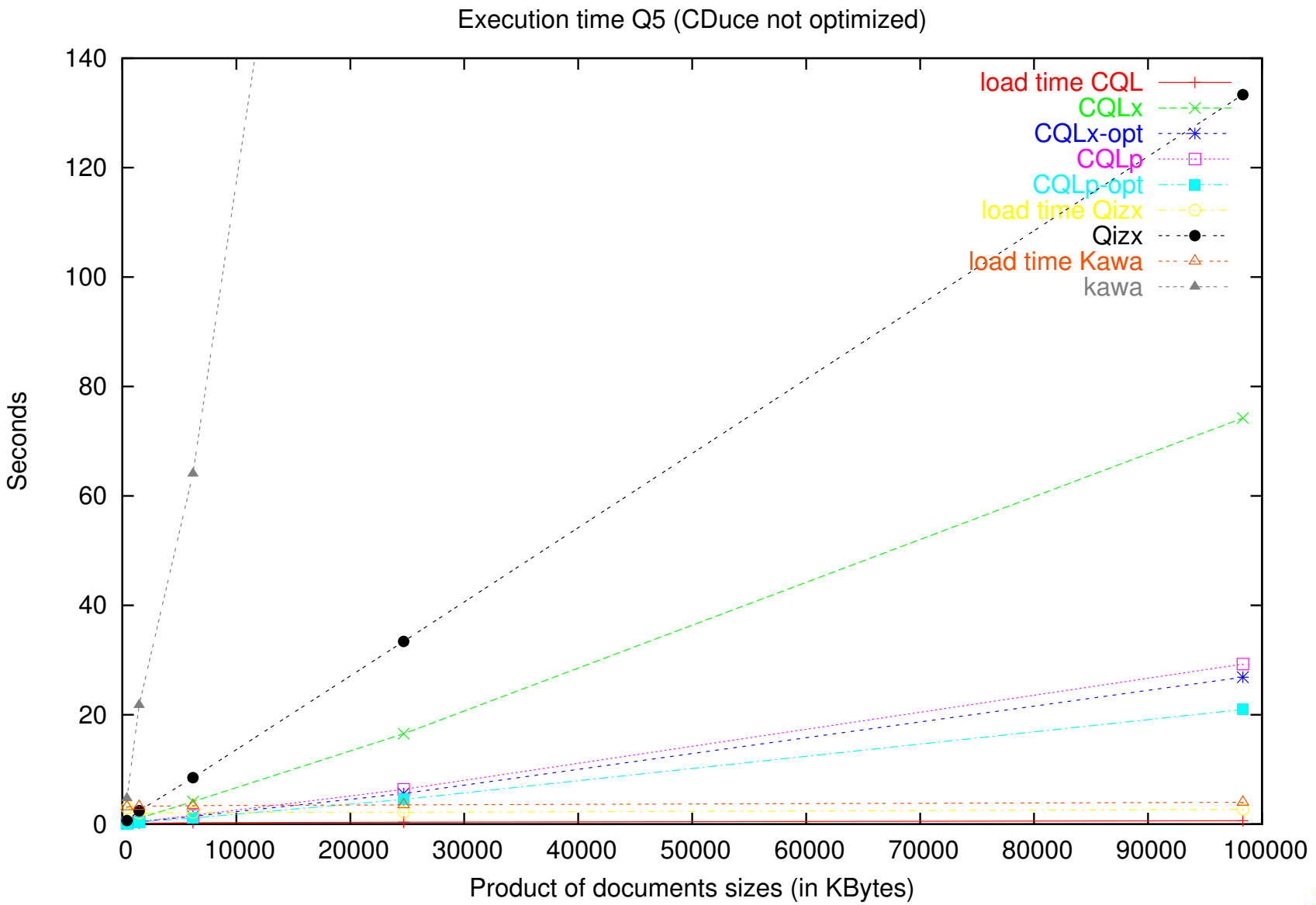
```
<books-with-prices>
   select <book-with-price>[t1
                            <price-amazon>p2
                            <price-bn>p1 ]
    from <bib>[b::Book*] in [biblio],
         <book year=y&(1991--*)>[t1&Title _* <price>p1] in b,
         <reviews>[e::Entry*] in [amazon],
         <entry>[t2&Title <price>p2 ;_] in e
   where t1=t2;;
```

CDuce Query Language: ℂQL

```
<books-with-prices>
   select <book-with-price>[t1
                            <price-amazon>p2
                            <price-bn>p1 ]
     from <bib>[b::Book*] in [biblio],
          <book year=y&(1991--*)>[t1&Title _* <price>p1] in b,
          <reviews>[e::Entry*] in [amazon],
          <entry>[t2&Title <price>p2 ;_] in e
    where t1=t2;;
```

```
<books-with-prices>
   select <book-with-price>[t2
                            <price-amazon>p2
                            <price-bn>p1 ]
     from <book year=y&(1991--*)>[t1&Title _* <price>p1] within [biblio],
          <entry>[t2&Title <price>p2 ;_] within [amazon]
    where t1=t2;;
```

ℂQL compared



Security: an example

```
<exam_base>
  <person gender="M">
    <name>Durand</name>
    <birth>
      <year>1970</year>
      <month>Aug</month>
      <day>10</day>
    </birth>
    <grade>110</grade>
  </person>
  <person gender="M">
    <name>Dupond</name>
    <birth>
      <year>1953</year>
      <month>Apr</month>
      <day>22</day>
    </birth>
  </person>
  <person gender="F">
    <name>Dubois</name>
    <birth>
      <year>1965</year>
      <month>Sep</month>
      <day>2</day>
    </birth>
    <grade>120</grade>
  </person>
</exam_base>
```

Security: an example

```
<exam_base>
  <person gender="M">
    <name>Durand</name>
    <birth>
      <year>1970</year>
      <month>Aug</month>
      <day>10</day>
    </birth>
    <grade>110</grade>
  </person>
  <person gender="M">
    <name>Dupond</name>
    <birth>
      <year>1953</year>
      <month>Apr</month>
      <day>22</day>
    </birth>
  </person>
  <person gender="F">
    <name>Dubois</name>
    <birth>
      <year>1965</year>
      <month>Sep</month>
      <day>2</day>
    </birth>
    <grade>120</grade>
  </person>
</exam_base>
```

- Only academic users can have information both on names and grades or on names and birthdays simultaneously.
- The administrative users can check whether a person passed the examination (that is, they can check for the presence of a `<grade>` element) but cannot access the result.
- Every user can ask for statistical results on grades upon criteria limited to year of birth and gender (so that they cannot select sufficiently restrictive sets to infer personal data).

Language level security

- Set security classifications for data and users (queries)
- Define non-interference: data that interferes with the result of a query
- Detect information flow: dependency analysis

Language level security

- Set security classifications for data and users (queries)
- Define non-interference: data that interferes with the result of a query
- Detect information flow: dependency analysis

Examples

Q1. A query which returns the average grade of all persons born after 1960 is non-interferent with nominative data.

Q2. A query which returns the average grade of all persons whose name contains the string “bois” is **interferent**.

Language level security

- Set security classifications for data and users (queries)
- Define non-interference: data that interferes with the result of a query
- Detect information flow: dependency analysis

Examples

Q1. A query which returns the average grade of all persons born after 1960 is non-interferent with nominative data.

```
average(select x from
   <person>[ _ <birth>[<year>[1960-*] ;_] <grade>x ] in mybase)
```

Q2. A query which returns the average grade of all persons whose name contains the string “bois” is **interferent**.

```
average(select x from
   <person>[ <name>[ _*?  'bois' _*?  ]  _ <grade>x ] in mybase)
```

Flows in Pattern matching

Dependency analysis in pattern matching:

- Presence of direct information flows: binding of variables

```
match e with <person>[ <name>[x] _ <grade>[y] ] -> ..x..  |  ...
```

- Presence of indirect information flows: type constraints, structure deconstruction.

```
match e with <person>[<name>[_*?  'bois' _*?]  _ <grade>x] -> e1  |   ...
```

- First match policy: information can be acquired by the failure of the previous branch has failed.

```
match e with <person>[<name>[_*?  'bois' _*?]  _ <grade>x] -> e1 | _ ->e2
```

Flows in Pattern matching

Dependency analysis in pattern matching:

- Presence of direct information flows: binding of variables

```
match e with <person>[ <name>[x] _ <grade>[y] ] -> ..x..  |  ...
```

- Presence of indirect information flows: type constraints, structure deconstruction.

```
match e with <person>[<name>[_*?  'bois' _*?]  _ <grade>x] -> e1  |   ...
```

- First match policy: information can be acquired by the failure of the previous branch has failed.

```
match e with <person>[<name>[_*?  'bois' _*?]  _ <grade>x] -> e1 | _ ->e2
```

We need a fine grained analysis of the information used by patterns

Back to the example

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

Add to the syntax the expression

$$\ell_t : e$$

Back to the example

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

Add to the syntax the expression

$$\ell_t : e$$

- For a normal user, our analysis must accept only queries whose result does not depend on values labeled by `name` and by `result` at the same time.

Back to the example

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

Add to the syntax the expression

$$\ell_t : e$$

- For a normal user, our analysis must accept only queries whose result does not depend on values labeled by `name` and by `result` at the same time.
- Use labels to perform dependency analysis.

What we have done (dynamic)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

What we have done (dynamic)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

Dynamic analysis: defined label propagation so that if the result depends on a labeled expression, its label occurs in the result.

What we have done (dynamic)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

The first query returns:

<average>[$stat_{Int}$:$result_{Int}$:115]

What we have done (dynamic)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

The second query returns:

<average>[$name_{Int}$:$result_{Int}$:120]

What we have done (dynamic)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

The second query returns:

<average>[$name_{Int}$:$result_{Int}$:120]

Interference is detected

What we are doing (static)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

What we are doing (static)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

Static analysis: Infer label propagation so that if the result of the query depends on a labeled expression, its label occurs in the type of the query.

What we are doing (static)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

The first query has type:

<average>[$stat_{Int}$:$result_{Int}$:Int]

What we are doing (static)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

The second query has type:

<average>[$name_{Int}$:$result_{Int}$:Int]

What we are doing (static)

```
<exam_base>[
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Durand]
    <birth>[
      <year>[stat_Int:1970]
      <month>[private_String:Aug]
      <day>[private_Int:10]
    ] passed_Grade:
    <grade>[result_Int:110]
  ]
  <person gender=stat"M"|"F":"M">[
    <name>[name_String:Dupond]
    <birth>[
      <year>[stat_Int:1953]
      <month>[private_String:Apr]
      <day>[private_Int:22]
    ]
  ]
  <person gender=stat"M"|"F":"F">[
    <name>[name_String:Dubois]
    <birth>[
      <year>[stat_Int:1965]
      <month>[private_String:Sep]
      <day>[private_Int:2]
    ] passed_Grade:
    <grade>[result_Int:120]
  ]
]
```

The second query has type:

<average>[$name_{Int}$:$result_{Int}$:Int]

Interference is detected

Ongoing work on language design

Currently investigated:

- XSLT-like primitives, in depth patterns, combinators for polymorphic iterators (Nguyen)
- Interface with external languages (Demouth)
- Parametric polymorphism (Castagna, Hosoya, Frisch)
- Binding with persitent stores (Benzaken, Manolescu)
- Concurrency (Castagna, De Nicola, Varacca)

Soon (??) inverstigated:

- Module system, incremental programming
- Webservices
- Streaming
- Assertions.

Thank you!

http://www.cduce.org/

For hardcore coders:

```
cvs -d":pserver:anonymous@cvs.cduce.org:/cvsroot" co cduce
```