XML Programming
Outline

38 XML basics
39 Set-theoretic types
40 Examples in Perl 6
41 Covariance and contravariance
42 XML Programming in CDuce
43 Functions in CDuce
44 Other benefits of types
45 Toolkit
Outline

38 XML basics

39 Set-theoretic types

40 Examples in Perl 6

41 Covariance and contravariance

42 XML Programming in CDuce

43 Functions in CDuce

44 Other benefits of types

45 Toolkit
XML is just tree-structured data:

```xml
<biblio>
  <book status="available">
    <title>Object-Oriented Programming</title>
    <author>Giuseppe Castagna</author>
  </book>
  <book>
    <title>A Theory of Objects</title>
    <author>Martín Abadi</author>
    <author>Luca Cardelli</author>
  </book>
</biblio>
```
XML is just tree-structured data:

```xml
<biblio>
  <book status="available">
    <title>Object-Oriented Programming</title>
    <author>Giuseppe Castagna</author>
  </book>
  <book>
    <title>A Theory of Objects</title>
    <author>Martín Abadi</author>
    <author>Luca Cardelli</author>
  </book>
</biblio>
```

Types describe the set of valid documents

```xml
<?xml version="1.0"?>
<!DOCTYPE biblio [
<!ELEMENT biblio (book*)>
<!ELEMENT book (title, (author|editor)+, price?)>
<!ATTLIST book status (available|borrowed) #IMPLIED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT editor (#PCDATA)>
<!ELEMENT price (#PCDATA)>
]>```
Programming with XML

How to manipulate data that is in XML format in a programming language?
Programming with XML

How to manipulate data that is in XML format in a programming language?

- Level 0: textual representation of XML documents
  - AWK, sed, Perl regexp
How to manipulate data that is in XML format in a programming language?

- **Level 0**: textual representation of XML documents
  - AWK, sed, Perl regexp
- **Level 1**: abstract view provided by a parser
  - SAX, DOM, ...
How to manipulate data that is in XML format in a programming language?

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

How to manipulate data that is in XML format in a programming language?

- Level 0: textual representation of XML documents
  - AWK, sed, Perl regexp
- Level 1: abstract view provided by a parser
  - SAX, DOM, . . .
- Level 2: untyped XML-specific languages
  - XSLT, XPath
- Level 3: XML types taken seriously
  - XDuce, Xtatic
  - XQuery
  - CDuce
  - $C_\omega$ (Microsoft)
  - . . .
How to manipulate data that is in XML format in a programming language?

- Level 0: textual representation of XML documents
  - AWK, sed, Perl regexp
- Level 1: abstract view provided by a parser
  - SAX, DOM, . . .
- Level 2: untyped XML-specific languages
  - XSLT, XPath
- Level 3: XML types taken seriously
  - XDuce, Xtatic
  - XQuery
  - CDuce
  - $C_\omega$ (Microsoft)
  - . . .
How to manipulate data that is in XML format in a programming language?

- **Level 0**: textual representation of XML documents
  - AWK, sed, Perl regexp
- **Level 1**: abstract view provided by a parser
  - SAX, DOM, ...
- **Level 2**: untyped XML-specific languages
  - XSLT, XPath
- **Level 3**: XML types taken seriously
  - XDuce, Xtatic
  - XQuery
  - **CDuce**
  - $C_\omega$ (Microsoft)
  - ...
Examples

**Level 1: DOM in Javascript**

Print the titles of the book in the bibliography

```html
<script>
    xmlDoc=loadXMLDoc("biblio.xml");
    x=xmlDoc.getElementsByTagName("book");
    for (i=0;i<x.length;i++){
        document.write(x[i].childNodes[0].nodeValue);
        document.write("<br>");
    }
</script>
```
**Level 1: DOM in Javascript**

Print the titles of the book in the bibliography

```javascript
<script>
    var xmlDoc = loadXMLDoc("biblio.xml");
    var x = xmlDoc.getElementsByTagName("book");
    for (var i = 0; i < x.length; i++) {
        document.write(x[i].childNodes[0].nodeValue);
        document.write("<br>");
    }
</script>
```

**Level 2: XPath**

The same in XPath:

```
/biblio/book/title
```

Select all titles of books whose price > 35

```
/biblio/book[price>35]/title
```
Level 2: XSLT
XSLT uses XPath to extract information (as a pattern in pattern matching)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:template match="/">
        <html>
            <body>
                <h2>Books Price List</h2>
                <table border="1">
                    <tr bgcolor="#9acd32">
                        <th>Title</th>
                        <th>Price</th>
                    </tr>
                    <xsl:for-each select="biblio/book">
                        <tr>
                            <td><xsl:value-of select="title"/></td>
                            <td><xsl:value-of select="price"/></td>
                        </tr>
                    </xsl:for-each>
                </table>
            </body>
        </html>
    </xsl:template>
</xsl:stylesheet>
```
Types are ignored

- In DOM nothing ensures that the read of a next node succeeds
- In XPath `/biblio/title/book` return an empty set of nodes rather than a type error
- Likewise the use of wrong XPath expressions in XSLT is unnoticed and yields empty XML documents as result (in the previous example the fact that `price` is optional is not handled).
Types are ignored

- In DOM nothing ensures that the read of a next node succeeds
- In XPath /biblio/title/book return an empty set of nodes rather than a type error
- Likewise the use of wrong XPath expressions in XSLT is unnoticed and yields empty XML documents as result (in the previous example the fact that price is optional is not handled).

Level 3: Recent languages take types seriously

- XDuce, Xtatic
- XQuery
- CDuce
- $C_\omega$
- ...

How to add XML types in programming languages?
Types are ignored

- In DOM nothing ensures that the read of a next node succeeds.
- In XPath `/biblio/title/book` return an empty set of nodes rather than a type error.
- Likewise the use of wrong XPath expressions in XSLT is unnoticed and yields empty XML documents as result (in the previous example the fact that `price` is optional is not handled).

**Level 3: Recent languages take types seriously**

- XDuce, Xtatic
- XQuery
- CDuce
- $C_\omega$
- ...

How to add XML types in programming languages?

We need *set-theoretic* type connectives
Outline

38 XML basics

39 Set-theoretic types

40 Examples in Perl 6

41 Covariance and contravariance

42 XML Programming in CDuce

43 Functions in CDuce

44 Other benefits of types

45 Toolkit
Set-theoretic types

We consider the following possibly recursive types:

\[
T ::= \text{Bool} | \text{Int} | \text{Any} | (T, T) | T \lor T | T \land T | \text{not}(T) | T \rightarrow T
\]

Useful for:

1. XML types
2. Precise typing of pattern matching
3. Overloaded functions
4. Mixins
5. General programming paradigms

Let us see each point more in detail

Note: henceforward I will sometimes use \( T_1 \mid T_2 \) to denote \( T_1 \lor T_2 \)
1. XML types

```xml
<?xml version="1.0"?>
<!DOCTYPE biblio [
<!ELEMENT biblio (book*)>
<!ELEMENT book (title, (author|editor)+, price?)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT editor (#PCDATA)>
<!ELEMENT price (#PCDATA)>
]>}
```

Can be encoded with union and recursive types

```perl
type Biblio = ('biblio,X)
type X = (Book,X) ∨ 'nil

type Book = ('book,(Title, Y ∨ Z))
type Y = (Author,Y ∨ (Price,'nil) ∨ 'nil)
type Z = (Editor,Z ∨ (Price,'nil) ∨ 'nil)

type Title = ('title,String)
type Author = ('author,String)
type Editor = ('editor,String)
type Price = ('price,String)
```
Consider the following pattern matching expression

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

where patterns are defined as follows:

\[
p ::= x \mid (p,p) \mid p|p \mid p\&p
\]
2. Precise typing of pattern matching (I)

Consider the following pattern matching expression

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

where patterns are defined as follows:

\[
p ::= x \mid (p, p) \mid p \mid p \mid p \land p
\]

If we interpret types as set of values

\[
t = \{ v \mid v \text{ is a value of type } t \}
\]

then the set of all values that match a pattern is a type

\[
\llbracket p \rrbracket = \{ v \mid v \text{ is a value that matches } p \}
\]

\[
\begin{align*}
\llbracket x \rrbracket &= \text{Any} \\
\llbracket (p_1, p_2) \rrbracket &= (\llbracket p_1 \rrbracket, \llbracket p_2 \rrbracket) \\
\llbracket p_1 \mid p_2 \rrbracket &= \llbracket p_1 \rrbracket \lor \llbracket p_2 \rrbracket \\
\llbracket p_1 \land p_2 \rrbracket &= \llbracket p_1 \rrbracket \land \llbracket p_2 \rrbracket
\end{align*}
\]
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching*:
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching*:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \neg(T_2) \)
Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \neg T_2 \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \{ p_1 \} \);
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 -> e_1 \mid p_2 -> e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \backslash T_2 \) for \( T_1 \& \neg(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \neg(p_1) \);
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \backslash \neg(p_1)) \& \neg(p_2) \);
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

\[
\text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2
\]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \neg(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \{p_1\} \);
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \setminus \{p_1\}) \& \{p_2\} \);
- The type of the match expression is \( T_1 \lor T_2 \).
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching*:

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

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \neg(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \llbracket p_1 \rrbracket \);
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \setminus \llbracket p_1 \rrbracket) \& \llbracket p_2 \rrbracket \);
- The type of the match expression is \( T_1 \lor T_2 \).
- Pattern matching is exhaustive if \( T \leq \llbracket p_1 \rrbracket \lor \llbracket p_2 \rrbracket \);
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to *type pattern matching*:

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

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T \& \neg(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \{ p_1 \} \);
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \setminus \{ p_1 \}) \& \{ p_2 \} \);
- The type of the match expression is \( T_1 \lor T_2 \).
- Pattern matching is exhaustive if \( T \leq \{ p_1 \} \lor \{ p_2 \} \);
2. Precise typing of pattern matching (II)

Boolean type connectives are needed to type pattern matching:

\[ \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \]

Suppose that \( e : T \) and let us write \( T_1 \setminus T_2 \) for \( T_1 \& \neg(T_2) \)

- To infer the type \( T_1 \) of \( e_1 \) we need \( T \& \{p_1\} \);
- To infer the type \( T_2 \) of \( e_2 \) we need \( (T \setminus \{p_1\}) \& \{p_2\} \);
- The type of the match expression is \( T_1 \lor T_2 \).
- Pattern matching is exhaustive if \( T \leq \{p_1\} \lor \{p_2\} \);

Formally:

\[ \begin{align*}
\frac{
\Gamma \vdash e : T \\
\Gamma, T \& \{p_1\} / p_1 \vdash e_1 : T_1 \\
\Gamma, T \setminus \{p_1\} / p_2 \vdash e_2 : T_2
}{\Gamma \vdash \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_1 \rightarrow e_2 : T_1 \lor T_2}
\]  

\( T/p \) is the type environment for the capture variables in \( p \) when the pattern is matched against values in \( T \).

(e.g., \( ((\text{Int}, \text{Int}) \lor (\text{Bool}, \text{Char}))/ (x, y) \) is \( x : \text{Int} \lor \text{Bool}, y : \text{Int} \lor \text{Char} \))
3. Overloaded functions

Intersection types are useful to type overloaded functions (in the Go language):

```go
package main
import "fmt"

func Opposite (x interface{}) interface{} {
    var res interface{}
    switch value := x.(type) {
    case bool:
        res = (!value)   // x has type bool
    case int:
        res = (-value)   // x has type int
    }
    return res
}

func main() { fmt.Println(Opposite(3) , Opposite(true)) }
```

In Go `Opposite` has type `Any-->Any` (every value has type `interface{}`). Better type with intersections `Opposite: (Int-->Int) & (Bool-->Bool)`
3. Overloaded functions

Intersection types are useful to type overloaded functions (in the Go language):

```go
package main
import "fmt"
func Opposite(x interface{}) interface{} {
    var res interface{}
    switch value := x.(type) {
        case bool:
            res = (!value) // x has type bool
        case int:
            res = (-value) // x has type int
    }
    return res
}
func main() { fmt.Println(Opposite(3), Opposite(true)) }
```

In Go `Opposite` has type `Any-->Any` (every value has type `interface{}`). Better type with intersections `Opposite: (Int-->Int) & (Bool-->Bool)`

Intersections can also to give a more refined description of standard functions:

```go
func Successor(x int) { return(x+1) }
```

which could be typed as `Successor: (Odd-->Even) & (Even-->Odd)`. 
Exercise:

1. What is the type returned by

```ocaml
let foo = function
| ('A,'B) -> true
| ('B,'A) -> false
```

and what is the problem?

2. Which type could we give if we had full-fledged union types?

3. Give an intersection type that refines the previous type.
Exercise:

1. What is the type returned by
   
   ```ocaml
   let foo = function
       | ('A,'B) -> true
       | ('B,'A) -> false
   ```

   and what is the problem?

   ```ocaml
   [< 'A | 'B ] * [< 'A | 'B ] -> bool
   ```

   thus `foo('A,'A)` fails.

2. Which type could we give if we had full-fledged union types?

3. Give an intersection type that refines the previous type.
Exercise:

1. What is the type returned by

   ```ocaml
   let foo = function
       | ('A,'B) -> true
       | ('B,'A) -> false
   ```

   and what is the problem?

   

2. Which type could we give if we had full-fledged union types?

   ```ocaml
   (<'A * 'B >| ('B * 'A) -> bool
   ```

3. Give an intersection type that refines the previous type
Exercise:

1. What is the type returned by
   
   ```ocaml
   let foo = function
       | ('A, 'B) -> true
       | ('B, 'A) -> false
   ```
   
   and what is the problem?
   
   ```ocaml
   [< 'A | 'B ] * [< 'A | 'B ] -> bool
   ```
   
   thus `foo('A, 'A)` fails

2. Which type could we give if we had full-fledged union types?

   ```ocaml
   ('A * 'B) | ('B * 'A) -> bool
   ```

3. Give an intersection type that refines the previous type

   ```ocaml
   (('A * 'B) -> true) & (('B * 'A) -> false)
   ```
4. Typing of Mixins

Intersection types are used in Microsoft’s Typescript to type mixins.

```typescript
function extend<T, U>(first: T, second: U): T & U {
    /* <T> exp is a type cast (equivalent: exp as T) */
    let result = <T & U>{};
    for (let id in first) {
        (<any>result)[id] = (<any>first)[id];
    }
    for (let id in second) {
        if (!result.hasOwnProperty(id)) {
            (<any>result)[id] = (<any>second)[id];
        }
    }
    return result;
}

class Person {
    constructor(public name: string) {
    }
}

interface Loggable {
    log(): void;
}

class ConsoleLogger implements Loggable {
    log() { ... }
}

var jim = extend(new Person("Jim"), new ConsoleLogger());
var n = jim.name;
jim.log();
```
5. General programming paradigms

Consider red-black trees. Recall that they must satisfy 4 invariants.

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes
5. General programming paradigms

Consider red-black trees. Recall that they must satisfy 4 invariants.

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes

The key of Okasaki’s insertion is the function `balance` which transforms an unbalanced tree, into a valid red-black tree (as long as a, b, c, and d are valid):
5. General programming paradigms

Consider red-black trees. Recall that they must satisfy 4 invariants.

1. the root of the tree is black
2. the leaves of the tree are black
3. no red node has a red child
4. every path from root to a leaf contains the same number of black nodes

The key of Okasaki’s insertion is the function `balance` which transforms an unbalanced tree, into a valid red-black tree (as long as a, b, c, and d are valid):

In ML we need GADTs to enforce the invariants.
type \( \alpha \) RBtree =
   | Leaf
   | Red( \( \alpha \), RBtree, RBtree)
   | Blk( \( \alpha \), RBtree, RBtree)

let balance =
  function
  | Blk( z , Red( x, a, Red(y,b,c) ) , d )
  | Blk( z , Red( y, Red(x,a,b), c ) , d )
  | Blk( x , a , Red( z, Red(y,b,c), d ) )
  | Blk( x , a , Red( y, b, Red(z,c,d) ) )
    -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
  | x -> x

let insert =
  function ( x , t ) ->
  let ins =
    function
      | Leaf -> Red(x,Leaf,Leaf)
      | c(y,a,b) as z ->
        if x < y then balance c( y, (ins a), b ) else
        if x > y then balance c( y, a, (ins b) ) else z
    in let _(y,a,b) = ins t in Blk(y,a,b)
type Rbtree = Btree | Rtree

type Rtree = Red(α, Btree, Btree)

type Btree = Blk(α, Rbtree, Rbtree) | Leaf


type Wrong = Red(α, (Rtree,Rbtree) | (Rbtree,Rtree))

type Unbal = Blk(α, (Wrong,Rbtree) | (Rbtree,Wrong))

let balance: (Unbal → Rtree) & ((β Unbal) → (β Unbal)) =

 function |
 Blk( z , Red( y, Red(x,a,b), c ) , d )
 | Blk( z , Red( x, a, Red(y,b,c) ) , d )
 | Blk( x , a , Red( z, Red(y,b,c), d ) )
 | Blk( x , a , Red( y, b, Red(z,c,d) ) )
 | -> Red ( y, Blk(x,a,b), Blk(z,c,d) )
 | x -> x

let insert: (α, Btree) → Btree =

 function ( x , t ) ->

 let ins: (Leaf → Rtree) & (Btree → Rbtree\Leaf) & (Rtree → Rtree|Wrong) =

 function |
 Leaf -> Red(x,Leaf,Leaf)
 | c(y,a,b) as z ->
 | if x < y then balance c( y, (ins a), b ) else
 | if x > y then balance c( y, a, (ins b) ) else z
 | in let _ (y,a,b) = ins t in Blk(y,a,b)
Type checking the previous definitions is not so difficult. The hard part is to type partial applications:

\[
\text{map} : (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]
\]

\[
\text{balance} : (\text{Unbal} \rightarrow \text{Rtree}) \& ((\beta \backslash \text{Unbal}) \rightarrow (\beta \backslash \text{Unbal}))
\]

\[
\text{map balance} : ( [\text{Unbal}] \rightarrow [\text{Rtree}] ) \\
\& ( [\alpha \backslash \text{Unbal}] \rightarrow [\alpha \backslash \text{Unbal}] ) \\
\& ( [\alpha \backslash \text{Unbal}] \rightarrow [(\alpha \backslash \text{Unbal}) \backslash \text{Rtree}] )
\]

Fortunately, programmers (and you) are spared from these gory details.
New languages use union and intersections

Facebook’s Flow:

```javascript
// @flow
function toStringPrimitives(val: number | boolean | string) {
    return String(val);
}

type One = { foo: number };

type Two = { bar: boolean };

type Both = One & Two;

var value: Both = {
    foo: 1,
    bar: true
};
```
New languages use union and intersections

**Typed-Racket**

(let ([a-number 37])
  (if (even? a-number)
      'yes
      'no))

- : Symbol [more precisely: (U 'no 'yes)]
  'no

(: f : (case-> (-> True Integer Integer)
  (-> False Boolean Boolean)))

(define (f condition x)
  (if condition
      (if condition
          (add1 x)
          (not x)))))
How to understand/explain set-theoretic type connectives?

The type connectives union, intersection, and negation are completely defined by the subtyping relation:

- $T_1 \lor T_2$ is the least upper bound of $T_1$ and $T_2$
- $T_1 \land T_2$ is the greatest lower bound of $T_1$ and $T_2$
- $\text{not}(T)$ is the only type whose union and intersection with $T$ yield the Any and Empty types, respectively.

Defining (and deciding) subtyping for type connectives (i.e., $\lor$, $\land$, $\text{not}(\cdot)$) is far more difficult than for type constructors (i.e., $\rightarrow$, $\times$, $\{\ldots\}$, $\ldots$).

Understanding connectives in terms of subtyping is out of reach of simple programmers
The type connectives union, intersection, and negation are completely defined by the subtyping relation:

- $T_1 \lor T_2$ is the least upper bound of $T_1$ and $T_2$
- $T_1 \land T_2$ is the greatest lower bound of $T_1$ and $T_2$
- $\text{not}(T)$ is the only type whose union and intersection with $T$ yield the Any and Empty types, respectively.

Defining (and deciding) subtyping for type connectives (i.e., $\lor$, $\land$, $\text{not}()$) is far more difficult than for type constructors (i.e., $\to$, $\times$, $\{\ldots\}$, $\ldots$).

Understanding connectives in terms of subtyping is out of reach of simple programmers.

Give a set-theoretic semantics to types
Types as sets of values and semantic subtyping

Each type *denotes* a set of values:

- **Bool** is the set that contains just two values \{true, false\}
- **Int** is the set of all the numeric constants: \{0, -1, 1, -2, 2, -3, ...\}.
- **Any** is the set of *all* values.
- \((T_1, T_2)\) is the set of all the pairs \((v_1, v_2)\) where \(v_1\) is a value in \(T_1\) and \(v_2\) a value in \(T_2\), that is \(\{(v_1, v_2) \mid v_1 \in T_1, v_2 \in T_2\}\).
- **\(T_1 \lor T_2\)** is the *union* of the sets \(T_1\) and \(T_2\), that is \(\{v \mid v \in T_1\ or v \in T_2\}\)
- **\(T_1 \land T_2\)** is the *intersection* of the sets \(T_1\) and \(T_2\), i.e. \(\{v \mid v \in T_1\ and v \in T_2\}\).
- **\(\neg(T)\)** is the set of all the values not in \(T\), that is \(\{v \mid v \notin T\}\).

In particular \(\neg(\text{Any})\) is the empty set (written Empty).

- **\(T_1 \rightarrow T_2\)** is the set of all function values that when applied to a value in \(T_1\), if they return a value, then this value is in \(T_2\).
Each type *denotes* a set of values:

- **Bool** is the set that contains just two values \{true, false\}.
- **Int** is the set of all the numeric constants: \{0, -1, 1, -2, 2, -3, ...\}.
- **Any** is the set of *all* values.
- \((T_1, T_2)\) is the set of all the pairs \((v_1, v_2)\) where \(v_1\) is a value in \(T_1\) and \(v_2\) a value in \(T_2\), that is \(\{(v_1, v_2) \mid v_1 \in T_1, v_2 \in T_2\}\).
- **(T \lor T)** is the *union* of the sets \(T_1\) and \(T_2\), that is \(\{v \mid v \in T_1 \text{ or } v \in T_2\}\).
- **(T \land T)** is the *intersection* of the sets \(T_1\) and \(T_2\), i.e. \(\{v \mid v \in T_1 \text{ and } v \in T_2\}\).
- **not(T)** is the set of all the values not in \(T\), that is \(\{v \mid v \not\in T\}\).

In particular, **not(Any)** is the empty set (written Empty).

- **(T \rightarrow T)** is the set of all function values that when applied to a value in \(T_1\), if they return a value, then this value is in \(T_2\).
Outline

38 XML basics
39 Set-theoretic types
40 Examples in Perl 6
41 Covariance and contravariance
42 XML Programming in CDuce
43 Functions in CDuce
44 Other benefits of types
45 Toolkit
A function \textit{value} is a $\lambda$-abstraction. In Perl6 it is any expression of the form:

\begin{verbatim}
sub (parameters) { body }
\end{verbatim}

For instance (functions can be named):

\begin{verbatim}
sub succ(Int $x) { $x + 1 }
\end{verbatim}

the \textit{succ} function is a value in/of type \texttt{Int--\rightarrow Int}. 
A function \textit{value} is a \(\lambda\)-abstraction. In Perl6 it is any expression of the form:

\[
\text{sub (parameters) \{body\}}
\]

For instance (functions can be named):

\[
\text{sub succ(Int $x)\{ $x + 1 \}}
\]

the \texttt{succ} function is a value in/of type \texttt{Int--->Int}.

Subtypes can be defined intensionally:

\[
\text{subset Even of Int where \{ \$_ \% 2 == 0 \}} \\
\text{subset Odd of Int where \{ \$_ \% 2 == 1 \}}
\]

Clearly:

\texttt{both succ:Even--->Odd and succ:Odd--->Even}

therefore:

\texttt{succ : (Even--->Odd) \& (Odd--->Even)}
Subtyping

Notice that every function value in \((\text{Even} \rightarrow \text{Odd}) \& (\text{Odd} \rightarrow \text{Even})\) is also in \(\text{Int} \rightarrow \text{Int}\). Thus:

\[
(\text{Even} \rightarrow \text{Odd}) \& (\text{Odd} \rightarrow \text{Even}) \unteq \text{Int} \rightarrow \text{Int}
\]

The converse does not hold: identity \(\text{sub}(\text{Int} \ x \{ \ x \})\) is a counterexample.
Subtyping

Notice that every function value in \((\text{Even} \rightarrow \text{Odd}) \& (\text{Odd} \rightarrow \text{Even})\) is also in \(\text{Int} \rightarrow \text{Int}\). Thus:

\[(\text{Even} \rightarrow \text{Odd}) \& (\text{Odd} \rightarrow \text{Even}) \ll : \text{Int} \rightarrow \text{Int}\]

The converse does not hold: identity \(\text{sub}(\text{Int} \ \$x\{ \ $x \ \})\) is a counterexample.

The above is just an instance of the following relation

\[(S_1 \rightarrow T_1) \& (S_2 \rightarrow T_2) \ll : (S_1 \lor S_2) \rightarrow (T_1 \lor T_2)\]  \hspace{1cm} (4)

that holds for all types, \(S_1, S_2, T_1,\) and \(T_2,\)
Notice that every function value in \((\text{Even}\rightarrow\text{Odd}) \& (\text{Odd}\rightarrow\text{Even})\) is also in \(\text{Int}\rightarrow\text{Int}\). Thus:

\[
(\text{Even}\rightarrow\text{Odd}) \& (\text{Odd}\rightarrow\text{Even}) <: \text{Int}\rightarrow\text{Int}
\]

The converse does not hold: identity sub(\(\text{Int} \ x\)){ \ x\ } is a counterexample.

The above is just an instance of the following relation

\[
(S_1\rightarrow T_1) \& (S_2\rightarrow T_2) <: (S_1 \vee S_2)\rightarrow(T_1 \vee T_2) \tag{4}
\]

that holds for all types, \(S_1, S_2, T_1\), and \(T_2\),

The relation (4) shows why defining subtyping for type connectives is far more difficult than just with constructors: connectives mix types of different forms.
Overloaded functions are defined by giving multiple definitions of the same function prefixed by the `multi` modifier:

\[
\text{multi sub } \text{sum}(\text{Int } \$x, \text{Int } \$y) \{ \; \$x + \$y \; \}
\text{multi sub } \text{sum}(\text{Bool } \$x, \text{Bool } \$y) \{ \; \$x \&\& \$y \; \}
\]

\[
\text{sum}:((\text{Int},\text{Int})\rightarrow\text{Int}) \; \& \; ((\text{Bool},\text{Bool})\rightarrow\text{Bool}), \quad (5)
\]
Overloaded functions

Overloaded functions are defined by giving multiple definitions of the same function prefixed by the `multi` modifier:

\[
\begin{align*}
\text{multi sub } \text{sum}(\text{Int } $x, \text{Int } $y) & \{ \hspace{1ex} $x + $y \hspace{1ex} \\
\text{multi sub } \text{sum}(\text{Bool } $x, \text{Bool } $y) & \{ \hspace{1ex} $x \&\& $y \hspace{1ex} \\
\end{align*}
\]

\[
\text{sum} : (\text{Int,Int} \rightarrow \text{Int}) \& (\text{Bool,Bool} \rightarrow \text{Bool}), \tag{5}
\]

Just one parameter is enough for selection. The *curried* form is equivalent.

\[
\begin{align*}
\text{multi sub } \text{sumC}(\text{Int } $x) & \{ \hspace{1ex} \text{sub } (\text{Int } $y)\{$ $x + $y \hspace{1ex} \} \hspace{1ex} \\
\text{multi sub } \text{sumC}(\text{Bool } $x) & \{ \hspace{1ex} \text{sub } (\text{Bool } $y)\{$ $x \&\& $y \hspace{1ex} \} \hspace{1ex} \\
\end{align*}
\]
Overloaded functions

Overloaded functions are defined by giving multiple definitions of the same function prefixed by the `multi` modifier:

```perl
multi sub sum(Int $x, Int $y) { $x + $y }
multi sub sum(Bool $x, Bool $y) { $x && $y }
```

\[ \text{sum} : ((\text{Int},\text{Int})\rightarrow\text{Int}) \quad \& \quad ((\text{Bool},\text{Bool})\rightarrow\text{Bool}), \quad (5) \]

Just one parameter is enough for selection. The *curried* form is equivalent.

```perl
multi sub sumC(Int $x){ sub (Int $y){$x + $y } }
multi sub sumC(Bool $x){ sub (Bool $y){$x && $y} }
```

In Perl we can use `;;` to separate parameters used for code selection from those passed to the selected code:

```perl
multi sub sumC(Int $x ;; Int $y) { $x + $y }
multi sub sumC(Bool $x ;; Bool $y) { $x && $y }
```

Both definitions of `sumC` have type

\[ ((\text{Int})\rightarrow((\text{Int})\rightarrow\text{Int})) \quad \& \quad ((\text{Bool})\rightarrow((\text{Bool})\rightarrow\text{Bool})). \quad (6) \]

though partial application is possible only with the first definition of `sumC`
The code to execute for a multisubroutine is chosen at run-time according to the type of the argument. The multi-subroutine with the best approximating input type is executed.
Dynamic dispatch

The code to execute for a multisubroutine is chosen at run-time according to the type of the argument. The multi-subroutine with the *best* approximating input type is executed.

- All examples given so far can be resolved at static time.
- Dynamic dispatch is sensible only when types change during computation.
Dynamic dispatch

The code to execute for a multisubroutine is chosen at run-time according to the type of the argument.

The multi-subroutine with the best approximating input type is executed.

- All examples given so far can be resolved at static time.
- Dynamic dispatch is sensible only when types change during computation.

In a statically-typed language with subtyping, the type of an expression may decrease during the computation.
Dynamic dispatch

The code to execute for a multisubroutine is chosen at run-time according to the type of the argument. The multi-subroutine with the best approximating input type is executed.

- All examples given so far can be resolved at static time.
- Dynamic dispatch is sensible only when types change during computation.

In a statically-typed language with subtyping, the type of an expression may decrease during the computation.

Example:

```
( sub(Int $x) { $x % 4 } )(3+2)
```

Int at compile time; Even after the reduction.
Dynamic dispatch

Example

```perl
multi sub mod2sum(Even $x , Odd $y) { 1 }
multi sub mod2sum(Odd $x , Even $y) { 1 }
multi sub mod2sum(Int $x , Int $y) { 0 }
```
Dynamic dispatch

Example

```perl
multi sub mod2sum(Even $x, Odd $y) { 1 }
multi sub mod2sum(Odd $x, Even $y) { 1 }
multi sub mod2sum(Int $x, Int $y) { 0 }
```

Its type (with singleton types: \( \nu \) is the type that contains just value \( \nu \))

\[
((\text{Even}, \text{Odd}) \rightarrow 1) \\
\& ((\text{Odd}, \text{Even}) \rightarrow 1) \\
\& ((\text{Int}, \text{Int}) \rightarrow 0 \lor 1)
\]

Exercise

Find a more precise type and justify how the type checker can deduce it.
Alternative definition for `mod2sum`:

```
multi sub mod2sum(Even $x, Int $y) { $y % 2 }
multi sub mod2sum(Int $x, Odd $y) { ($x+1) % 2 }
```

Mathematically correct but selection is ambiguous: the computation is stuck on arguments of type `(Even, Odd)`. 
Alternative definition for mod2sum:

\[
\text{multi sub } \text{mod2sum}(\text{Even } x, \text{Int } y)\{ y \% 2 \} \\
\text{multi sub } \text{mod2sum}(\text{Int } x, \text{Odd } y)\{ (x+1) \% 2 \}
\]

Mathematically correct but selection is ambiguous: the computation is stuck on arguments of type (Even, Odd).

Formation rule 1: Ambiguity

A multi-subroutine is *free from ambiguity* if whenever it has definitions for input $S$ and $T$, and $S \& T$ is not empty, then it has a definition for input $S \& T$. 
Formation rules for multi-subroutines: Ambiguous Selection

Alternative definition for \texttt{mod2sum}:

\begin{verbatim}
    multi sub mod2sum(Even $x$, Int $y$) { $y \% 2$ }
    multi sub mod2sum(Int $x$, Odd $y$) { ($x+1) \% 2$ }
\end{verbatim}

Mathematically correct but selection is ambiguous: the computation is stuck on arguments of type \((\text{Even, Odd})\).

Formation rule 1: Ambiguity

A multi-subroutine is \textit{free from ambiguity} if whenever it has definitions for input \(S\) and \(T\), and \(S \& T\) is not empty, then it has a definition for input \(S \& T\).

It is a \textit{formation rule}. It belongs to language design not to the type system:

\[
(\text{Even, Int}) \rightarrow 0 \lor 1 \quad \& \quad (\text{Int, Odd}) \rightarrow 0 \lor 1
\]

the type above is perfectly ok (and a correct type for \texttt{mod2sum}).
Formation rules for multi-subroutines: Specialization

Because of dynamic dispatch during the execution:
- the type of the argument changes,
Formation rules for multi-subroutines: Specialization

Because of dynamic dispatch during the execution:
- the type of the argument changes, \( \implies \)
- the code selected for a multi-subroutine changes,
Formation rules for multi-subroutines: Specialization

Because of dynamic dispatch during the execution:

- the type of the argument changes, ⇒
- the code selected for a multi-subroutine changes, ⇒
- the type of application changes

Types may *only* decrease along the computation
Formation rules for multi-subroutines: Specialization

Because of dynamic dispatch during the execution:

- the type of the argument changes, ⇒
- the code selected for a multi-subroutine changes, ⇒
- the type of application changes

Types may \textit{only} decrease along the computation

Consider again:

\begin{verbatim}
multi sub mod2sum(Even $x$, Odd $y$) { 1 }
multi sub mod2sum(Odd $x$, Even $y$) { 1 }
multi sub mod2sum(Int $x$, Int $y$) { 0 }
\end{verbatim}

which has type

\[
((\text{Even}, \text{Odd}) \rightarrow 1) \& ((\text{Odd}, \text{Even}) \rightarrow 1) \& ((\text{Int}, \text{Int}) \rightarrow 0 \lor 1)
\]
Formation rules for multi-subroutines: Specialization

Because of dynamic dispatch during the execution:

- the type of the argument changes, ⇒
- the code selected for a multi-subroutine changes, ⇒
- the type of application changes

Types may **only** decrease along the computation

Consider again:

```plaintext
multi sub mod2sum(Even $x , Odd $y) { 1 }
multi sub mod2sum(Odd $x , Even $y) { 1 }
multi sub mod2sum(Int $x , Int $y) { 0 }
```

which has type

```
((Even,Odd)--->1) & ((Odd,Even)--->1) & ((Int,Int)--->0 ∨ 1)
```

For the application `mod2sum(3+3,3+2)`:  

- **static time**: third code selected; static type is 0 ∨ 1  
- **run time**: first code selected; dynamic type is 1 (notice 1 <: 0 ∨ 1)
“Types may only decrease along the computation”
Formation rules for multi-subroutines: Specialization

“Types may only decrease along the computation”

Why does it matter?

```perl
multi sub foo(Int $x) { $x+42 }
multi sub foo(Odd $x) { true }
```

Consider $10+(\text{foo}(3+2))$: statically well-typed but yields a runtime type error.
Formation rules for multi-subroutines: Specialization

“Types may only decrease along the computation”

Why does it matter?

```perl
multi sub foo(Int $x) { $x+42 }
multi sub foo(Odd $x) { true }
```

Consider `10+(foo(3+2))`: statically well-typed but yields a runtime type error.

How to ensure it for dynamic dispatch?

Formation rule 2: Specialization

A multi-subroutine is *specialization sound* if whenever it has definitions for input `S` and `T`, and `S <: T`, then the definition for input `S` returns a type smaller than the one returned by the definition for `T`.

Example:

```perl
multi sub foo(S₁ $x) returns T₁ { ... }
multi sub foo(S₂ $x) returns T₂ { ... }
```

Specialization sound: If `S₁ <: S₂` then `T₁ <: T₂`. 
Formation rules for multi-subroutines: Specialization

Once more, a *formation rule*: concerns language design, not the type system. The type system is perfectly happy with the type

\[(S_1 \to T_1) \& (S_2 \to T_2)\]

even if \(S_1 <: S_2\) and \(T_1\) and \(T_2\) are not related. However consider all the possible cases of applications of a function of this type:

1. If the argument is in \(S_1 \& S_2\), then the application has type \(T_1 \& T_2\).
2. If the argument is in \(S_1 \backslash S_2\) and case 1 does not apply, then the application has type \(T_1\).
3. If the argument is in \(S_2 \backslash S_1\) and case 1 does not apply, then the application has type \(T_2\).
4. If the argument is in \(S_1 \lor S_2\) and no previous case applies, then the application has type \(T_1 \lor T_2\).
This case

1. If the argument is in \( S_1 \ & S_2 \), then the application has type \( T_1 \ & T_2 \).

may confuse the programmer when \( S_2 <: S_1 \), since in this case \( S_2 = S_2 \ & S_1 \):

When a function of type \( (S_1 \rightarrow T_1) \ & (S_2 \rightarrow T_2) \) with \( S_2 <: S_1 \), is applied to an argument of type \( S_2 \), then the application returns results in \( T_1 \ & T_2 \).

**Design choice:** to avoid confusion force (wlog) the programmer to specify that the return type for a \( S_2 \) input is (some subtype of) \( T_1 \ & T_2 \).

This can be obtained by accepting only specialization sound definitions and greatly simplifies the presentation of the type discipline of the language.
Outline

38 XML basics
39 Set-theoretic types
40 Examples in Perl 6
41 Covariance and contravariance
42 XML Programming in CDuce
43 Functions in CDuce
44 Other benefits of types
45 Toolkit
Homework assignment:

1. **Mandatory:** Study the covariance and contravariance problem described in the first 3 sections of the following paper (click on the title).


2. **Optional:** if you want to know what is under the hood, you can read Section 4 of the same paper, which describes a state-of-the-art implementation of a type system with set-theoretic types.
Outline

38  XML basics
39  Set-theoretic types
40  Examples in Perl 6
41  Covariance and contravariance
42  XML Programming in CDuce
43  Functions in CDuce
44  Other benefits of types
45  Toolkit
CDuce is built on types

The main motivation for studying set-theoretic types is to define strongly typed programming languages for XML.

CDuce is a programming language for XML whose design is completely based on set-theoretic types.

In CDuce set-theoretic types are pervasive:

1. XML types are encoded in set-theoretic types
2. Patterns are types with capture variables
3. Set-theoretic types are used for informative error messages
4. Types are used for efficient JIT compilation
<bib>
  <book year="1997">
    <title> Object-Oriented Programming </title>
    <author>
      <last> Castagna </last>
      <first> Giuseppe </first>
    </author>
    <price> 56 </price>
    Bikhäuser
  </book>
  <book year="2000">
    <title> Regexp Types for XML </title>
    <editor>
      <last> Hosoya </last>
      <first> Haruo </first>
    </editor>
    UoT
  </book>
</bib>
XML syntax

```xml
<bib>
  <book year="1997">
    <title>Object-Oriented Programming’
    <author>
      <last>Castagna’
      <first>Giuseppe’
    </author>
    <price>56’
    'Bikhäuser’
  </book>
  <book year="2000">
    <title>Regexp Types for XML’
    <editor>
      <last>Hosoya’
      <first>Haruo’
    </editor>
    'UoT’
  </book>
</bib>
```
XML syntax

type Bib = <bib>[  
    <book year="1997">[  
    <title>['Object-Oriented Programming']  
    <author>[  
      <last>['Castagna']  
      <first>['Giuseppe']  
    ]  
    <price>['56']  
    'Bikhäuser'  
  ]  
  <book year="2000">[  
    <title>['Regexp Types for XML']  
    <editor>  
      <last>['Hosoya']  
      <first>['Haruo']  
    ]  
    'UoT'  
  ]  
]
]
XML syntax

type Bib = <bib>
    <book year=String>
        <title>
        <author>
            <last>[PCDATA]
            <first>[PCDATA]
        
        <price>[PCDATA]
        PCDATA
        
    </author>
    <title>[PCDATA]
    <editor>
        <last>[PCDATA]
        <first>[PCDATA]
        
        PCDATA
        
    ]

]
XML syntax

type Bib = <bib>[Book Book]
type Book = <book year=String>[Title
   (Author  | Editor )
   Price?
   PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
XML syntax

type Bib = <bib>[Book*]
type Book = <book year=String>[Title
  (Author+ | Editor+)
  Price?
  PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
XML syntax

type Bib = <bib>[Book*]
type Book = <book year=String>[Title
   (Author+ | Editor+)
   Price?
   PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
XML syntax

type Bib = <bib>[Book*]

Kleene star

type Book = <book year=String>[Title

(Author+ | Editor+)

Price?

PCDATA]

type Author = <author>[Last First]

type Editor = <editor>[Last First]

type Title = <title>[PCDATA]

type Last = <last>[PCDATA]

type First = <first>[PCDATA]

type Price = <price>[PCDATA]
XML syntax

Type Bib = <bib>[Book*]

Type Book = <book year=String>[Title
  (Author+ | Editor+)
  Price?
  PCDATA]

Type Author = <author>[Last First]

Type Editor = <editor>[Last First]

Type Title = <title>[PCDATA]

Type Last = <last>[PCDATA]

Type First = <first>[PCDATA]

Type Price = <price>[PCDATA]
XML syntax

type Bib = <bib>[Book*]
type Book = <book year=String>[Title
   (Author+ | Editor+)
   Price?
   PCDATA]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]

nested elements
XML syntax

```xml
<book year=String>[Title
  (Author+ | Editor+)
  Price?
  PCDATA]

<author>[Last First]
<editor>[Last First]
<title>[PCDATA]
<last>[PCDATA]
<first>[PCDATA]
<price>[PCDATA]
```

This and: singletons, intersections, differences, Empty, and Any.
XML syntax

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

optional elems

type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]
XML syntax

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

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 Price = <price>[PCDATA]
XML syntax

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

type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title = <title>[PCDATA]
type Last = <last>[PCDATA]
type First = <first>[PCDATA]
type Price = <price>[PCDATA]

This and: singletons, intersections, differences, Empty, and Any.
XML syntax

```xml
<book year=String>
  Title
  (Author+ | Editor+)
  Price?
  PCDATA
</book>
```

This includes: singletons, intersections, differences, Empty, and Any.

We saw that all this can be encoded with recursive and set-theoretic types.
Types & patterns: the functional languages perspective

- **Types** are sets of **values**
- Values are decomposed by **patterns**
- Patterns are roughly values with **capture variables**
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

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

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

with patterns one can write

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

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

with patterns one can write

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

which is syntactic sugar for

```plaintext
match e with (x,y) -> (y,x)
```
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

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

with patterns one can write

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

which is syntactic sugar for

```plaintext
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:

```haskell
type List = (Any, List) | 'nil
```
Example: tail-recursive version of length for lists:

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

fun length (x:(List,Int)): Int =
  match x with
  | ('nil , n) -> n
  | (_,t), n) -> length(t,n+1)
```
Example: tail-recursive version of \texttt{length} for lists:

\begin{verbatim}
  type List = (Any,List) | 'nil

  fun length (x:(List,Int)): Int =
    match x with
    | ('nil , n) -> n
    | (_,t), n) -> length(t,n+1)
\end{verbatim}
Example: tail-recursive version of \texttt{length} for lists:

\begin{verbatim}
type List = (Any,List) | 'nil

fun length (x:(List,Int)): Int =
  match x with
  | ('nil , n) -> n
  | ((_ __ __ _,t), n) -> length(t,n+1)
\end{verbatim}

So patterns are values with
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
  | ((_ __ __ _ ,ttt),nnn) -> length(t,n+1)
```

So patterns are values 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,
Example: tail-recursive version of \texttt{length} for lists:

\begin{verbatim}
  type List = (Any,List) | 'nil

  fun length (x:(List,Int)): Int =
      match x with
      | ('nil', n) -> n
      | ((_,t), n) -> length(t,n+1)
\end{verbatim}

So patterns are values with \texttt{capture variables}, \texttt{wildcards}, \texttt{constants}. 
Example: tail-recursive version of \texttt{length} for lists:

\begin{verbatim}
  type List = (Any,List) | 'nil

  fun length (x:(List,Int)): Int =
    match x with
    | ('nil , n) -> n
    | ((_ __ __ _,t), n) -> length(t,n+1)
\end{verbatim}

So patterns are values with \texttt{capture variables}, \texttt{wildcards}, \texttt{constants}.

\textbf{But if we:}
Example: tail-recursive version of length for lists:

```ml
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 \(s \times t\))
Example: tail-recursive version of length for lists:

```ocaml
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:

- use for types the same constructors as for values
  (e.g. \((s, t)\) instead of \(s \times t\))
Example: tail-recursive version of \texttt{length} for lists:

\begin{verbatim}
  type List = (Any,List) | ‘nil

  fun length (x:(List,Int)): Int =
    match x with
    | (‘nil , n) -> n
    | ((__,t), n) -> length(t,n+1)
\end{verbatim}

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);
Example: tail-recursive version of `length` for lists:

```haskell
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);
Example: tail-recursive version of \texttt{length} for lists:

\begin{verbatim}
 type List = (Any,List) | 'nil

 fun length (x:(List,Int)): Int =
    match x with
    | ('nil , n) -> n
    | ((_ __ __ _,t), n) -> length(t,n+1)
\end{verbatim}

So patterns are values with \texttt{capture variables, wildcards, constants}.

But if we:

\begin{enumerate}
\item use for types the same constructors as for values \hfill (e.g. \((s,t)\) \texttt{instead of} \(s \times t\))
\item use values to denote singleton types \hfill (e.g. \texttt{‘nil in the list type});
\item consider the wildcard \texttt{“\_\_\_\_\_” as synonym of \texttt{Any}}
\end{enumerate}
Example: tail-recursive version of `length` for lists:

```haskell
type List = (\text{\texttt{Any}},\text{\texttt{List}}) \mid \text{\texttt{\textquoteright nil}}

fun length (x:(\text{\texttt{List}},\text{\texttt{Int}})):\text{\texttt{Int}} =
  match x with
  | (\text{\texttt{\textquoteright 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. \(\text{\texttt{\textquoteright nil}}\) in the list type);

3. consider the wildcard “\_” as synonym of \texttt{Any}
Example: tail-recursive version of length for lists:

```plaintext
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.
Example: tail-recursive version of \texttt{length} for lists:

\begin{verbatim}
type List = (Any,List) | 'nil

fun length (x:(List,Int)): Int = 
  match x with
  | ('nil , n) -> n
  | ((_ __ __ _,t), n) -> length(t,n+1)
\end{verbatim}

So patterns are values with capture variables, wildcards, constants.

---

\textbf{Key idea behind regular patterns}

Patterns are types with capture variables

Define types: patterns come for free.
Example: tail-recursive version of `length` for lists:

```ocaml
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.
Patterns in CDuec

Patterns = Types + Capture variables
Patterns in CDuce

Patterns = Types + Capture variables

type Bib = <bib>[Book*]
Patterns in CDuce

Patterns = Types + Capture variables

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

<x::Book*>
```

G. Castagna (CNRS)
Patterns = Types + Capture variables

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

The pattern binds \( x \) to the \textit{sequence} of all books in the bibliography.
Patterns in CDuce

Patterns = Types + Capture variables

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

match bibs with
  <bib>[x::Book*] -> x
```
Patterns in CDuce

Patterns = Types + Capture variables

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

match bibs with
  <bib>[x::Book*] -> x

Returns the content of bibs.
```
Patterns = Types + Capture variables

```cduce

import Bib:

type Bib = <bib>[Book*]

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

G. Castagna (CNRS)
Patterns in CDuce

Patterns = Types + Capture variables

\[
\text{type Bib = <bib>[Book*]}
\]

\[
<\text{bib}>[\text{( x:<book year="2005">_ _ | y:_ })^*]
\]

Binds \( x \) to the sequence of all this year’s books, and \( y \) to all the other books.
Patterns in CDuce

Patterns = Types + Capture variables

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

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

**Types**

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

**Patterns**

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

Returns the concatenation (i.e., “@”) of the two captured sequences.
Patterns in CDuce

Patterns = Types + Capture variables

---

```haskell
-- TYPES

type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String

-- PATTERNS

<bib>[(x::<book year="1990">[_* Publisher\"ACM"] | _)*]```

---

G. Castagna (CNRS)
Patterns = Types + Capture variables

```haskell
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String

<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 in CDuce

Patterns = Types + Capture variables

```plaintext
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String

match bibs with
  <bib>[(x::<book year="1990">[_* Publisher"ACM"] | _)*)] -> x
```
Patterns = Types + Capture variables

type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String

match bibs with
  <bib>[(x::<book year="1990">[ _* Publisher\"ACM"] | _*)*] -> x

Returns all the captured books
Patterns in CDuce

Patterns = Types + Capture variables

```
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String
```

```
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 in CDuce

Patterns = Types + Capture variables

```haskell
type Bib = <bib>[Book*]
type Book = <book year=String>[Title Author+ Publisher]
type Publisher = String

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+]`
Functions in CDuce
Functions: basic usage

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

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

Extract subsequences (union polymorphism)

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

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

Extract subsequences (union polymorphism)

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

Extract subsequences of non-consecutive elements:

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

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

Extract subsequences (union polymorphism)

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

Extract subsequences of non-consecutive elements:

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

Perl-like string processing (String = [Char*])

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

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

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

Functions can be **higher-order** and **overloaded**

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

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

Functions can be **higher-order** and **overloaded**

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

``` OCaml 

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

Functions can be higher-order and overloaded

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

**Functions**: advanced usage

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

Functions can be **higher-order** and **overloaded**

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

Higher-order, overloading, subtyping provide name/code sharing...

Functions: advanced usage

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

Functions can be higher-order and overloaded

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

Higher-order, overloading, subtyping provide name/code sharing...

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 ]
```
Functions: advanced usage

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

Functions can be **higher-order** and **overloaded**

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

Higher-order, overloading, subtyping provide name/code sharing...

```
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 ]
```
Functions: advanced usage

Type definitions:

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

Functions can be **higher-order** and **overloaded**

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

Higher-order, overloading, subtyping provide name/code sharing...

```plaintext
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 ]
```

Even more compact: replace the last two branches with:

```plaintext
<(k)[ t a _* ] -> <(k)[ t a ]
```
Functions: advanced usage

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

Functions can be higher-order and overloaded

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

Higher-order, overloading, subtyping provide name/code sharing...

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 ]

Even more compact: replace the last two branches with:

<(k)[ t a _* ] -> <(k)[ t a ]
Red-black trees in CDuce

type RBtree = Btree | Rtree;;
type Btree = <black elem=Int>[ RBtree RBtree ] | [] ;;
type Rtree = <red elem=Int>[ Btree Btree ];;

type Wrongtree = Wrongleft | Wrongright;;
type Wrongleft = <red elem=Int>[ Rtree Btree ];;
type Wrongright = <red elem=Int>[ Btree Rtree ];;
type Unbalanced = <black elem=Int>([Wrongtree RBtree] | [RBtree Wrongtree])

let balance ( Unbalanced -> Rtree ; Rtree -> Rtree ; Btree \[] -> Btree \[] ;
  \[] -> \[] ; Wrongleft -> Wrongleft ; Wrongright -> Wrongright )
| <black (z)>[ <red (y)>[ <red (x)>[ a b ] c ] d ]
| <black (z)>[ <red (x)>[ a <red (y)>[ b c ] ] d ]
| <black (x)>[ a <red (z)>[ <red (y)>[ b c ] d ] ]
| <black (x)>[ a <red (y)>[ b <red (z)>[ c d ] ] ] ->
  <red (y)>[ <black (x)>[ a b ] <black (z)>[ c d ] ]
| x -> x

let insert (x : Int) (t : Btree) : Btree =
let ins_aux ( \[] -> Rtree ; Btree \[] -> RBtree \[] ; Rtree -> Rtree|Wrongtree)
| \[] -> <red elem=x>[ \[] \[] ]
| (<<(color) elem=y>[ a b ]) & z ->
  if x << y then balance <<(color) elem=y>[ (ins_aux a) b ]
  else if x >> y then balance <(color) elem=y>[ a (ins_aux b) ]
  else z
in match ins_aux t with
| _ (y)>[ a b ] -> <black (y)>[ a b ]
Red-black trees in CDuce

```haskell
type RBtree = Btree | Rtree;;
type Btree = <black elem=Int>[ RBtree RBtree ] | [] ;;
type Rtree = <red elem=Int>[ Btree Btree ];;

let balance ( Unbalanced -> Rtree ; Rtree -> Rtree ; Btree\[\] -> Btree\[\] ;
  [\] -> [\] ; Wrongleft -> Wrongleft ; Wrongright -> Wrongright)
  | <black (z)>[<red (y)>[<red (x)>[a b] c] d]
  | <black (z)>[<red (x)>[a <red (y)>[b c] ] d]
  | <black (x)>[a <red (z)>[<red (y)>[b c] d]]
  | <black (x)>[a <red (y)>[b <red (z)>[c d] ] ] ->
    <red (y)>[<black (x)>[a b] <black (z)>[c d] ]
  | x -> x

let insert (x : Int) (t : Btree) : Btree =
  let ins_aux ( [] -> Rtree ; Btree\[\] -> RBtree\[\] ; Rtree -> Rtree|Wrongtree)
    | [] -> <red elem=x>[ [] [] ]
    | (<<(color) elem=y>[ a b ]) & z ->
      if x << y then balance (<<(color) elem=y>[ (ins_aux a) b ]
    else if x >> y then balance (<<(color) elem=y>[ a (ins_aux b) ]
    else z
  in match ins_aux t with
    | <_(y)>[a b] -> <black (y)>[a b ]
```

G. Castagna (CNRS)
Red-black trees in Polymorphic CDuce

```ocaml
type RBtree = Btree | Rtree;;
type Btree = <black elem=Int>[ RBtree RBtree ] | [];;;
type Rtree = <red elem=Int>[ Btree Btree ];;;

type Wrongtree = <red elem=Int>[ Rtree Btree ]
     | <red elem=Int>[ Btree Rtree ];;;
type Unbalanced = <black elem=Int>([Wrongtree RBtree] | [RBtree Wrongtree]);

let balance ( Unbalanced -> Rtree ; α\Unbalanced -> α\Unbalanced )
 | <black (z)>[ <red (y)>[ <red (x)>[ a b ] c ] d ]
 | <black (z)>[ <red (x)>[ a <red (y)>[ b c ] ] d ]
 | <black (x)>[ a <red (z)>[ <red (y)>[ b c ] d ] ]
 | <black (x)>[ a <red (y)>[ b <red (z)>[ c d ] ] ] ->
     <red (y)>[ <black (x)>[ a b ] <black (z)>[ c d ] ]
 | x -> x

let insert (x : Int) (t : Btree) : Btree =
let ins_aux ( [] -> Rtree ; Btree\[] -> RBtree\[]; Rtree -> Rtree|Wrongtree)
 | [] -> <red elem=x>[ [] [] ]
 | ((color) elem=y>[ a b ]) & z ->
     if x << y then balance (color) elem=y>[ (ins_aux a) b ]
     else if x >> y then balance (color) elem=y>[ a (ins_aux b) ]
     else z
in match ins_aux t with
 | _ (y)>[ a b ] -> <black (y)>[ a b ]
```

G. Castagna (CNRS)
Outline

38 XML basics
39 Set-theoretic types
40 Examples in Perl 6
41 Covariance and contravariance
42 XML Programming in CDuce
43 Functions in CDuce
44 Other benefits of types
45 Toolkit
Informative error messages
Informative error messages

List of books of a given year, stripped of the Editors and Price
Informative error messages

List of books of a given year, stripped of the Editors and Price

fun onlyAuthors (year:Int, books:[Book*]):[Book*] =

Informative error messages

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)
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>[
] ]
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
Informative error messages

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>[  ] ]
Informative error messages

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>[ ] ]
List of books of a given year, stripped of the Editors and Price

```haskell
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>[ ] ]
```
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>[ ] ]
Efficient execution
**Efficient execution**

**Idea:** if types tell you that something cannot happen, don’t test it.
**Efficient execution**

**Idea:** if types tell you that something cannot happen, don’t test it.

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

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
Idea: if types tell you that something cannot happen, don’t test it.

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

fun check(x : A|B) = match x with A -> 1 | B -> 0
```
Efficient execution

Idea: if types tell you that something cannot happen, don’t test it.

```plaintext
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
```
Efficient execution

Idea: if types tell you that something cannot happen, don’t test it.

\[
\text{type A} = \langle a\rangle[A^*] \\
\text{type B} = \langle b\rangle[B^*]
\]

fun check(x : A\|B) = match x with A \to 1 | B \to 0

fun check(x : A\|B) = match x with \langle a\rangle_\_ \to 1 | \_ \_ \to 0

- No backtracking.
**Efficient execution**

**Idea:** if types tell you that something cannot happen, don’t test it.

```ocaml
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.
Efficient execution

Idea: if types tell you that something cannot happen, don’t test it.

type A = \langle a \rangle [A*]
type B = \langle b \rangle [B*]

fun check(x : A|B) = match x with A -> 1 | B -> 0
fun check(x : A|B) = match x with \langle a \rangle _ _ -> 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
Efficient execution

**Idea:** if types tell you that something cannot happen, don’t test it.

```ml
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

**Specific kind of push-down tree automata**
Every programming language needs tools / libraries / DLS extensions.

Available for CDuce:
- OCaml full integration
- Web-services API
- Navigational patterns (à la XPath) [experimental]
A CDuce application that requires OCaml code
A CDuce application that requires OCaml code

- Reuse existing libraries
  - Abstract data structures: hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries: databases, networks, ...

CDuce ↔ OCaml Integration
A CDuce application that requires OCaml code

- Reuse existing libraries
  - Abstract data structures: hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries: databases, networks, ...

- Implement complex algorithms
CDuce ↔ OCaml Integration

A CDuce application that requires OCaml code

- Reuse existing libraries
  - Abstract data structures: hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries: databases, networks, ...

- Implement complex algorithms

An OCaml application that requires CDuce code
A **CDuce** application that requires **OCaml** code

- Reuse existing libraries
  - Abstract data structures: hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries: databases, networks, ...

- Implement complex algorithms

An **OCaml** application that requires **CDuce** code

- **CDuce** used as an XML input/output/transformation layer
CDuce↔OCaml Integration

A CDuce application that requires OCaml code

- Reuse existing libraries
  - Abstract data structures: hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries: databases, networks, ...

- Implement complex algorithms

An OCaml application that requires CDuce code

- CDuce used as an XML input/output/transformation layer
  - Configuration files
  - XML serialization of datas
  - XHTML code production
CDuce\textleft\rightarrow\textright OCaml Integration

A CDuce application that requires OCaml code

- Reuse existing libraries
  - Abstract data structures: hash tables, sets, ...
  - Numerical computations, system calls
  - Bindings to C libraries: databases, networks, ...
- Implement complex algorithms

An OCaml application that requires CDuce code

- CDuce used as an XML input/output/transformation layer
  - Configuration files
  - XML serialization of data
  - XHTML code production

Need to seamlessly call OCaml code in CDuce and vice versa
Main Challenges
Main Challenges

1. Seamless integration:
Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:
Seamless integration:
No explicit conversion function in programs:
the compiler performs the conversions
Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:
   the compiler performs the conversions

2. **Type safety:**
Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs: the compiler performs the conversions.

2. **Type safety:**
   No explicit type cast in programs.
Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs: 
   the compiler performs the conversions

2. **Type safety:**
   No explicit type cast in programs: 
   the standard type-checkers ensure type safety
Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs: the compiler performs the conversions

2. **Type safety:**
   No explicit type cast in programs: the standard type-checkers ensure type safety

**What we need:**

A mapping between OCaml and CDuce *types and values*
How to integrate the two type systems?

The translation can go just one way: OCaml $\rightarrow$ CDuce
How to integrate the two type systems?

The translation can go just one way: OCaml $\rightarrow$ CDuce

CDuce uses (semantic) subtyping; OCaml does not
How to integrate the two type systems?

The translation can go just one way: OCaml → CDuce

+ CDuce uses (semantic) subtyping; OCaml does not
  
  If we translate CDuce types into OCaml ones:
  - soundness requires the translation to be monotone;
  - no subtyping in Ocaml implies a constant translation;
How to integrate the two type systems?

The translation can go just one way: OCaml $\rightarrow$ CDuce

- CDuce uses (semantic) subtyping; OCaml does not
- If we translate CDuce types into OCaml ones:
  - soundness requires the translation to be monotone;
  - no subtyping in Ocaml implies a constant translation;
  $\Rightarrow$ CDuce typing would be lost.
How to integrate the two type systems?

The translation can go just one way: OCaml $\rightarrow$ CDuce

- CDuce uses (semantic) subtyping; OCaml does not
  - If we translate CDuce types into OCaml ones:
    - soundness requires the translation to be monotone;
    - no subtyping in Ocaml implies a constant translation;
    $\Rightarrow$ CDuce typing would be lost.

- CDuce has unions, intersections, differences, heterogeneous lists; OCaml does not
How to integrate the two type systems?

The translation can go just one way: \( \text{OCaml} \rightarrow \text{CDuce} \)

- **CDuce uses (semantic) subtyping; OCaml does not**
  - If we translate CDuce types into OCaml ones:
    - soundness requires the translation to be monotone;
    - no subtyping in Ocaml implies a constant translation;
    \( \Rightarrow \text{CDuce typing would be lost.} \)

- **CDuce has unions, intersections, differences, heterogeneous lists; OCaml does not**
  \( \Rightarrow \text{OCaml types are not enough to translate CDuce types.} \)
How to integrate the two type systems?

The translation can go just one way: OCaml → CDuce

- CDuce uses (semantic) subtyping; OCaml does not
  If we translate CDuce types into OCaml ones:
  - soundness requires the translation to be monotone;
  - no subtyping in Ocaml implies a constant translation;
  ⇒ CDuce typing would be lost.

- CDuce has unions, intersections, differences, heterogeneous lists; OCaml does not
  ⇒ OCaml types are not enough to translate CDuce types.

- OCaml supports type polymorphism; CDuce does not yet (it does in the development version).
How to integrate the two type systems?

The translation can go just one way: OCaml $\rightarrow$ CDuce

- **CDuce** uses (semantic) subtyping; **OCaml** does not
  
  If we translate **CDuce** types into **OCaml** ones:
  - soundness requires the translation to be monotone;
  - no subtyping in OCaml implies a constant translation;
  \[ \Rightarrow \text{CDuce typing would be lost.} \]

- **CDuce** has unions, intersections, differences, heterogeneous lists;
  **OCaml** does not
  
  \[ \Rightarrow \text{OCaml types are not enough to translate CDuce types.} \]

- **OCaml** supports type polymorphism; **CDuce** does not yet (it does in the development version).
  
  \[ \Rightarrow \text{Polymorphic OCaml libraries/functions must be first instantiated to be used in CDuce} \]
In practice

Define a mapping $T$ from OCaml types to CDuce types.
In practice

Define a mapping \( T \) from **OCaml** types to **CDuce** types.

<table>
<thead>
<tr>
<th>( t )</th>
<th><strong>(OCaml)</strong></th>
<th>( T(t) )</th>
<th><strong>(CDuce)</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td></td>
<td>min_int--max_int</td>
<td></td>
</tr>
<tr>
<td>string</td>
<td></td>
<td>Latin1</td>
<td></td>
</tr>
<tr>
<td>( t_1 * t_2 )</td>
<td></td>
<td>( (T(t_1), T(t_2)) )</td>
<td></td>
</tr>
<tr>
<td>( t_1 \rightarrow t_2 )</td>
<td></td>
<td>( T(t_1) \rightarrow T(t_2) )</td>
<td></td>
</tr>
<tr>
<td>( t ) list</td>
<td></td>
<td>( [T(t)]^* )</td>
<td></td>
</tr>
<tr>
<td>( t ) array</td>
<td></td>
<td>( [T(t)]^* )</td>
<td></td>
</tr>
<tr>
<td>( t ) option</td>
<td></td>
<td>( [T(t)]? )</td>
<td></td>
</tr>
<tr>
<td>( t ) ref</td>
<td></td>
<td>( \text{ref } T(t) )</td>
<td></td>
</tr>
<tr>
<td>( A_1 ) of ( t_1 )</td>
<td>...</td>
<td>( A_n ) of ( t_n )</td>
<td>( {A_1, T(t_1)}</td>
</tr>
<tr>
<td>( {l_1 = t_1; \ldots; l_n = t_n} )</td>
<td></td>
<td>( {l_1 = T(t_1); \ldots; l_n = T(t_n)} )</td>
<td></td>
</tr>
</tbody>
</table>
In practice

1. Define a mapping $T$ from **OCaml** types to **CDuce** types.

<table>
<thead>
<tr>
<th>$t$ (OCaml)</th>
<th>$T(t)$ (CDuce)</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>min_int--max_int</td>
</tr>
<tr>
<td>string</td>
<td>Latin1</td>
</tr>
<tr>
<td>$t_1 \ast t_2$</td>
<td>$\langle T(t_1), T(t_2) \rangle$</td>
</tr>
<tr>
<td>$t_1 \rightarrow t_2$</td>
<td>$T(t_1) \rightarrow T(t_2)$</td>
</tr>
<tr>
<td>$t$ list</td>
<td>$[T(t)\ast]$</td>
</tr>
<tr>
<td>$t$ array</td>
<td>$[T(t)\ast]$</td>
</tr>
<tr>
<td>$t$ option</td>
<td>$[T(t)\ ?]$</td>
</tr>
<tr>
<td>$t$ ref</td>
<td>ref $T(t)$</td>
</tr>
<tr>
<td>$A_1$ of $t_1</td>
<td>\ldots</td>
</tr>
<tr>
<td>${l_1 = t_1; \ldots; l_n = t_n}$</td>
<td>${l_1 = T(t_1); \ldots; l_n = T(t_n)}$</td>
</tr>
</tbody>
</table>

2. Define a retraction pair between **OCaml** and **CDuce** values.
In practice

1. Define a mapping $T$ from OCaml types to CDuce types.

<table>
<thead>
<tr>
<th>OCaml Type</th>
<th>CDuce Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>min_int--max_int</td>
</tr>
<tr>
<td>string</td>
<td>Latin1</td>
</tr>
<tr>
<td>$t_1 * t_2$</td>
<td>$(T(t_1), T(t_2))$</td>
</tr>
<tr>
<td>$t_1 \rightarrow t_2$</td>
<td>$T(t_1) \rightarrow T(t_2)$</td>
</tr>
<tr>
<td>t list</td>
<td>$[T(t)]^*$</td>
</tr>
<tr>
<td>t array</td>
<td>$[T(t)]^*$</td>
</tr>
<tr>
<td>t option</td>
<td>$[T(t)]?$</td>
</tr>
<tr>
<td>t ref</td>
<td>$\text{ref } T(t)$</td>
</tr>
<tr>
<td>$A_1 \text{ of } t_1 \mid \ldots \mid A_n \text{ of } t_n$</td>
<td>$(\langle A_1, T(t_1) \rangle \mid \ldots \mid \langle A_n, T(t_n) \rangle)$</td>
</tr>
<tr>
<td>${l_1 = t_1; \ldots; l_n = t_n}$</td>
<td>${l_1 = T(t_1); \ldots; l_n = T(t_n)}$</td>
</tr>
</tbody>
</table>

2. Define a retraction pair between OCaml and CDuce values.

\[
\text{ocaml2cduce: } t \rightarrow T(t) \\
\text{cduce2ocaml: } T(t) \rightarrow t
\]
## Calling OCaml from CDuce

**Easy**

Use $M.f$ to call the function $f$ exported by the OCaml module $M$. 
Easy

Use $\textit{M.f}$ to call the function $\textit{f}$ exported by the OCaml module $\textit{M}$.

The CDuce compiler checks type soundness and then
Easy

Use \texttt{M.f} to call the function \texttt{f} exported by the OCaml module \texttt{M}

The CDuce compiler checks type soundness and then
- applies \texttt{cduce2ocaml} to the arguments of the call
Calling OCaml from CDuce

**Easy**

Use `M.f` to call the function `f` exported by the OCaml module `M`.

The CDuce compiler checks type soundness and then
- applies `cduce2ocaml` to the arguments of the call
- calls the OCaml function
Using `M.f` to call the function `f` exported by the OCaml module `M`.

The CDuce compiler checks type soundness and then:
- applies `cduce2ocaml` to the arguments of the call
- calls the OCaml function
- applies `ocaml2cduce` to the result of the call
Calling OCaml from CDuce

Easy

Use \texttt{M.f} to call the function \texttt{f} exported by the OCaml module \texttt{M}

The CDuce compiler checks type soundness and then
- applies \texttt{cduce2ocaml} to the arguments of the call
- calls the OCaml function
- applies \texttt{ocaml2cduce} to the result of the call

Example: use ocaml-mysql library in CDuce

\begin{verbatim}
let db = Mysql.connect Mysql.defaults;;

match Mysql.list_dbs db 'None [] with
| ('Some,l) -> print [ 'Databases: ' !(string_of l) '\n' ]
| 'None -> [];;
\end{verbatim}
Needs little work

Compile a CDuce module as an OCaml binary module by providing a OCaml (.mli) interface. Use it as a standard Ocaml module.
Calling CDuce from OCaml

Needs little work

Compile a CDuce module as an OCaml binary module by providing a OCaml (.mli) interface. Use it as a standard Ocaml module.

The CDuce compiler:
Calling Cdue from OCaml

Needs little work

Compile a Cdue module as an OCaml binary module by providing a OCaml (.mli) interface. Use it as a standard Ocaml module.

The Cdue compiler:
1. Checks that if \texttt{val f:t} in the \texttt{.mli} file, then the Cdue type of \(f\) is a subtype of \(T(t)\)
Calling **CDuce** from **OCaml**

### Needs little work

Compile a CDuce module as an OCaml binary module by providing a OCaml (.mli) interface. Use it as a standard Ocaml module.

The CDuce compiler:

1. Checks that if `val f:t` in the .mli file, then the CDuce type of `f` is a *subtype of* `T(t)`.

2. Produces the OCaml glue code to export CDuce values as OCaml ones and bind OCaml values in the CDuce module.
Calling CDuce from OCaml

Needs little work

Compile a CDuce module as an OCaml binary module by providing a OCaml (.mli) interface. Use it as a standard Ocaml module.

The CDuce compiler:

1. Checks that if \texttt{val f:t} in the \texttt{.mli} file, then the CDuce type of \texttt{f} is a subtype of \texttt{T(t)}

2. Produces the OCaml glue code to export CDuce values as OCaml ones and bind OCaml values in the CDuce module.

Example: use CDuce to compute a factorial:

(* File cdnum.mli: *)
\begin{verbatim}
val fact: Big_int.big_int \rightarrow Big_int.big_int
\end{verbatim}

(* File cdnum.cd: *)
\begin{verbatim}
let aux ((Int,Int) \rightarrow Int)
| (x, 0 | 1) \rightarrow x
| (x, n) \rightarrow aux (x * n, n - 1)

let fact (x : Int) : Int = aux(1,x)
\end{verbatim}