XML Programming

Outline

Outline

XML is just tree-structured data:

```
<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:

```
<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 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

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, ...

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

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_ω (Microsoft)
  - . . .

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_ω (Microsoft)
  - . . .

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_ω (Microsoft)
  - ...

Examples

Level 1: DOM in Javascript

Print the titles of the book in the bibliography

```
<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>
```

Examples

Level 1: DOM in Javascript

Print the titles of the book in the bibliography

```
<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 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 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 suceeds
- 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 suceeds
- 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_ω
- ...

How to add XML types in programming languages?

Types are ignored

- In DOM nothing ensures that the read of a next node suceeds
- 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_ω
- ...

How to add XML types in programming languages?

We need *set-theoretic* type connectives

Outline

Set-theoretic types

We consider the following possibly recursive types:

$$\texttt{T} ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Any} \mid \texttt{(T,T)} \mid \texttt{T} \vee \texttt{T} \mid \texttt{T \& T} \mid \texttt{not(T)} \mid \texttt{T-->T}$$

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 $\texttt{T}_1 \,|\, \texttt{T}_2$ to denote $\texttt{T}_1 \vee \texttt{T}_2$

1. XML types

```
<?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

```
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)
```

2. Precise typing of pattern matching (I)

Consider the following pattern matching expression

$$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$$

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

$$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$$

where patterns are defined as follows:

$$p ::= x \mid (p\,,p) \mid p|p \mid p\&p$$

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

$$\wr p \int = \{v \mid v \text{ is a value that matches } p\}$$

$$\begin{array}{rcl} \wr x \int & = & \texttt{Any} \\ \wr (p_1\,,p_2) \int & = & (\wr p_1 \int\,,\wr p_2 \int) \\ \wr p_1|p_2 \int & = & \wr p_1 \int \vee \wr p_2 \int \\ \wr p_1\&p_2 \int & = & \wr p_1 \int \;\&\; \wr p_2 \int \end{array}$$

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:*

`match` e `with` p_1 `->` e_1 `|` p_2 `->` e_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 $\mathtt{T}_1 \backslash \mathtt{T}_2$ for $\mathtt{T}_1\,\&\,\mathtt{not(T}_2\mathtt{)}$

2. Precise typing of pattern matching (II)

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

$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$

Suppose that $e : \texttt{T}$ and let us write $\texttt{T}_1 \backslash \texttt{T}_2$ for $\texttt{T}_1\,\&\,\texttt{not(T}_2\texttt{)}$

- To infer the type $\texttt{T}_1$ of e_1 we need $\texttt{T}\ \&\ \{\!\!\{ p_1 \}\!\!\}$;

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 $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \backslash \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not}(\mathtt{T}_2)$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T} \,\&\, \wr p_1 \int$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \backslash \wr p_1 \int) \,\&\, \wr p_2 \int$;

2. Precise typing of pattern matching (II)

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

$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$

Suppose that $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \backslash \mathtt{T}_2$ for $\mathtt{T}_1\,\&\,\mathtt{not(T}_2\mathtt{)}$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T}\,\&\,\llbracket p_1 \rrbracket$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \backslash \llbracket p_1 \rrbracket)\,\&\,\llbracket p_2 \rrbracket$;
- The type of the match expression is $\mathtt{T}_1 \vee T_2$.

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 $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \backslash \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not(T}_2\mathtt{)}$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T} \,\&\, \wr p_1 \int$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \backslash \wr p_1 \int) \,\&\, \wr p_2 \int$;
- The type of the match expression is $\mathtt{T}_1 \vee T_2$.
- Pattern matching is exhaustive if $\mathtt{T} \leqslant \wr p_1 \int \vee \wr p_2 \int$;

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 $e : \mathtt{T}$ and let us write $\mathtt{T_1} \backslash \mathtt{T_2}$ for $\mathtt{T_1} \,\&\, \mathtt{not(T_2)}$

- To infer the type $\mathtt{T_1}$ of e_1 we need $\mathtt{T} \,\&\, \wr p_1 \int$;
- To infer the type $\mathtt{T_2}$ of e_2 we need $(\mathtt{T} \backslash \wr p_1 \int) \,\&\, \wr p_2 \int$;
- The type of the match expression is $\mathtt{T_1} \vee T_2$.
- Pattern matching is exhaustive if $\mathtt{T} \leqslant \wr p_1 \int \vee \wr p_2 \int$;

2. Precise typing of pattern matching (II)

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

$\texttt{match}\ e\ \texttt{with}\ p_1\ \texttt{->}\ e_1\ \texttt{|}\ p_2\ \texttt{->}\ e_2$

Suppose that $e : \mathtt{T}$ and let us write $\mathtt{T}_1 \setminus \mathtt{T}_2$ for $\mathtt{T}_1 \,\&\, \mathtt{not}(\mathtt{T}_2)$

- To infer the type $\mathtt{T}_1$ of e_1 we need $\mathtt{T} \,\&\, \wr p_1 \int$;
- To infer the type $\mathtt{T}_2$ of e_2 we need $(\mathtt{T} \setminus \wr p_1 \int) \,\&\, \wr p_2 \int$;
- The type of the match expression is $\mathtt{T}_1 \vee T_2$.
- Pattern matching is exhaustive if $\mathtt{T} \leqslant \wr p_1 \int \vee \wr p_2 \int$;

Formally:

[MATCH]

$$\frac{\Gamma \vdash e : \mathtt{T} \qquad \Gamma, \mathtt{T} \,\&\, \wr p_1 \int / p_1 \vdash e_1 : \mathtt{T}_1 \qquad \Gamma, \mathtt{T} \setminus \wr p_1 \int / p_2 \vdash e_2 : \mathtt{T}_2}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ p_1 \texttt{->} e_1\ \texttt{|}\ p_1 \texttt{->} e_2 : \mathtt{T}_1 \vee \mathtt{T}_2}(\mathtt{T} \leqslant \wr p_1 \int \vee \wr p_2 \int$$

where $\mathtt{T}/p$ is the type environment for the capture variables in p when the pattern is matched against values in $\mathtt{T}$.

(e.g., $((\mathtt{Int}\,,\mathtt{Int}) \vee (\mathtt{Bool}\,,\mathtt{Char}))/(x\,,y)$ is $x : \mathtt{Int} \vee \mathtt{Bool}, y : \mathtt{Int} \vee \mathtt{Char}$)

3. Overloaded functions

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

```
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):

```
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:

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

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

2+3. Precise typing of OCaml

Exercise:

1. What is the type returned by

   ```
   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

2+3. Precise typing of OCaml

Exercise:

1. What is the type returned by

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

   and what is the problem ?

   `[< `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

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

   and what is the problem ?

   `[< ‘A | ‘B ] * [< ‘A | ‘B ] -> bool` thus `foo( ‘A , ‘A)` fails

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

   `(‘A * ‘B )| ( ‘B * ‘A)  -> bool`

3. Give an intersection type that refines the previous type

2+3. Precise typing of OCaml

Exercise:

1. What is the type returned by

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

   and what is the problem ?

   `[< ‘A | ‘B ] * [< ‘A | ‘B ] -> bool` thus `foo( ‘A , ‘A)` fails

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

   `(‘A * ‘B )| ( ‘B * ‘A)  -> bool`

3. Give an intersection type that refines the previous type

   `((‘A * ‘B ) -> true) & (( ‘B * ‘A)  -> false)`

4. Typing of Mixins

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

```
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();
```

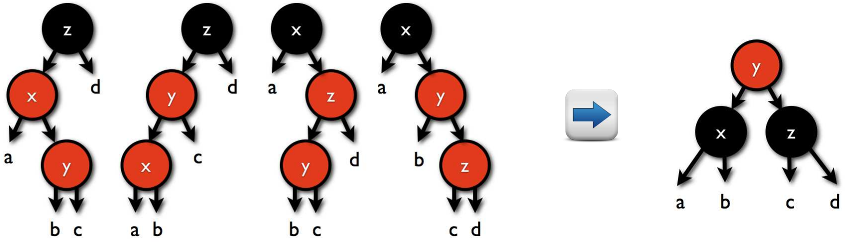
Consider red-black trees. Recall that they must satisfy 4 invariants.

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

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):

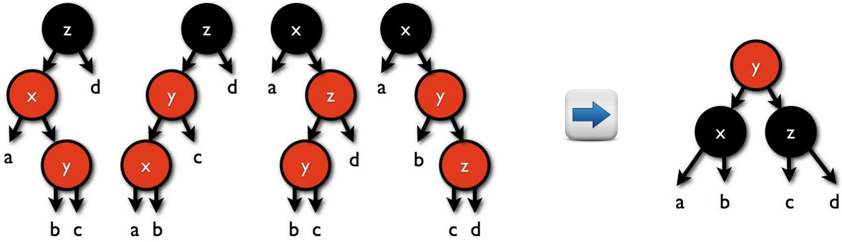


5. General programming paradigms

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 α RBtree =
   | Leaf
   | Red( α , RBtree , RBtree)
   | Blk( α , 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)
```

Cutting edge research

Type checking the previous definitions is not so difficult.
The hard part is to type partial applications:

$$\texttt{map} : (\alpha \to \beta) \to [\,\alpha\,] \to [\,\beta\,]$$

$$\texttt{balance} : (\texttt{Unbal} \to \texttt{Rtree})\ \&\ ((\beta \setminus \texttt{Unbal}) \to (\beta \setminus \texttt{Unbal}))$$

$$\begin{aligned}\texttt{map balance} : &\ (\,[\,\texttt{Unbal}\,] \to [\,\texttt{Rtree}\,]\,)\\ \&&\ (\,[\,\alpha \setminus \texttt{Unbal}\,] \to [\,\alpha \setminus \texttt{Unbal}\,]\,)\\ \&&\ (\,[\,\alpha \mid \texttt{Unbal}\,] \to [\,(\alpha \setminus \texttt{Unbal}) \mid \texttt{Rtree}\,]\,)\end{aligned}$$

Fortunately, programmers (and you) are spared from these gory details.

New languages use union and intersections

Facebook's Flow:

```
// @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
       (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:
  - $T_1 \vee T_2$ is the least upper bound of T_1 and T_2
  - $T_1 \,\&\, T_2$ is the greatest lower bound of T_1 and T_2
  - `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., $\vee$, $\&$, `not()`) is far more difficult than for *type constructors* (i.e., `-->`, $\times$, $\{...\}, \ldots$).
- Understanding connectives in terms of subtyping is out of reach of simple programmers

How to understand/explain set-theoretic type connectives?

- The type connectives union, intersection, and negation are completely defined by the subtyping relation:
  - $\mathtt{T}_1 \vee \mathtt{T}_2$ is the least upper bound of $\mathtt{T}_1$ and $\mathtt{T}_2$
  - $\mathtt{T}_1 \,\&\, \mathtt{T}_2$ is the greatest lower bound of $\mathtt{T}_1$ and $\mathtt{T}_2$
  - $\mathtt{not}(T)$ is the only type whose union and intersection with $\mathtt{T}$ yield the `Any` and `Empty` types, respectively.
- Defining (and deciding) subtyping for *type connectives* (i.e., $\vee$, $\&$, $\mathtt{not}()$) is far more difficult than for *type constructors* (i.e., `-->`, $\times$, $\{\ldots\}$, $\ldots$).
- 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

$\mathtt{T} ::= \mathtt{Bool} \mid \mathtt{Int} \mid \mathtt{Any} \mid \mathtt{(T\,,T)} \mid \mathtt{T} \vee \mathtt{T} \mid \mathtt{T} \,\&\, \mathtt{T} \mid \mathtt{not(T)} \mid \mathtt{T\text{-->}T}$

Each type *denotes* a set of values:

`Bool` is the set that contains just two values $\{\mathtt{true}\,,\mathtt{false}\}$

`Int` is the set of all the numeric constants: $\{\mathtt{0}\,,\ \mathtt{-1}\,,\ \mathtt{1}\,,\ \mathtt{-2}\,,\ \mathtt{2}\,,\ \mathtt{-3}\,,\ \mathtt{...}\}$.

`Any` is the set of *all* values.

$\mathtt{(T_1\,,T_2)}$ is the set of all the pairs (v_1, v_2) where v_1 is a value in $\mathtt{T_1}$ and v_2 a value in $\mathtt{T_2}$, that is $\{(v_1, v_2) \mid v_1 \in \mathtt{T_1}\,,\ v_2 \in \mathtt{T_2}\}$.

$\mathtt{T_1} \vee \mathtt{T_2}$ is the *union* of the sets $\mathtt{T_1}$ and $\mathtt{T_2}$, that is $\{v \mid v \in \mathtt{T_1} \text{ or } v \in \mathtt{T_2}\}$

$\mathtt{T_1} \,\&\, \mathtt{T_2}$ is the *intersection* of the sets $\mathtt{T_1}$ and $\mathtt{T_2}$, i.e. $\{v \mid v \in \mathtt{T_1} \text{ and } v \in \mathtt{T_2}\}$.

$\mathtt{not(T)}$ is the set of all the values not in $\mathtt{T}$, that is $\{v \mid v \notin \mathtt{T}\}$.
In particular `not(Any)` is the empty set (written `Empty`).

$\mathtt{T_1\text{-->}T_2}$ is the set of all function values that when applied to a value in $\mathtt{T_1}$, if they return a value, then this value is in $\mathtt{T_2}$.

Types as sets of values and semantic subtyping

$\texttt{T} ::= \texttt{Bool} \mid \texttt{Int} \mid \texttt{Any} \mid \texttt{(T , T)} \mid \texttt{T} \vee \texttt{T} \mid \texttt{T} \,\&\, \texttt{T} \mid \texttt{not(T)} \mid \texttt{T-->T}$

Each type *denotes* a set of values:

`Bool` is the set that contains just two values $\{\texttt{true},\texttt{false}\}$

`Int` is the set of all the numeric constants: $\{\texttt{0, -1, 1, -2, 2, -3, ...}\}$.

`Any` is the set of *all* values.

$\texttt{(T}_1\texttt{ , T}_2\texttt{)}$ is the set of all the pairs (v_1, v_2) where v_1 is a value in $\texttt{T}_1$ and v_2 a value in $\texttt{T}_2$, that is $\{(v_1, v_2) \mid v_1 \in \texttt{T}_1 , v_2 \in \texttt{T}_2\}$.

$\texttt{T}_1 \vee \texttt{T}_2$ is the *union* of the sets $\texttt{T}_1$ and $\texttt{T}_2$, that is $\{v \mid v \in \texttt{T}_1 \text{ or } v \in \texttt{T}_2\}$

$\texttt{T}_1 \,\&\, \texttt{T}_2$ is the *intersection* of the sets $\texttt{T}_1$ and $\texttt{T}_2$, i.e. $\{v \mid v \in \texttt{T}_1 \text{ and } v \in \texttt{T}_2\}$.

`not(T)` is the set of all the values not in $\texttt{T}$, that is $\{v \mid v \notin \texttt{T}\}$.
In particular `not(Any)` is the empty set (written `Empty`).

$\texttt{T}_1\texttt{-->T}_2$ is the set of all function values that when applied to a value in $\texttt{T}_1$, if they return a value, then this value is in $\texttt{T}_2$.

Semantic subtyping

Subtyping is set-containment

Outline

Set-theoretic types in Perl 6

A function *value* is a λ-abstraction. In Perl6 it is any expression of the form:

sub (*parameters*){*body*}

For instance (functions can be named):

```
sub succ(Int $x){ $x + 1 }
```

the `succ` function is a value in/of type `Int-->Int`.

Set-theoretic types in Perl 6

A function *value* is a λ-abstraction. In Perl6 it is any expression of the form:

`sub` (*parameters*){*body*}

For instance (functions can be named):

```
sub succ(Int $x){ $x + 1 }
```

the `succ` function is a value in/of type `Int-->Int`.

Subtypes can be defined intensionally:

```
subset Even of Int where { $_ % 2 == 0 }
subset Odd  of Int where { $_ % 2 == 1 }
```

Clearly:

both `succ:Even-->Odd` and `succ:Odd-->Even`

therefore:

`succ :  (Even-->Odd) & (Odd-->Even)`

Subtyping

Notice that every function value in `(Even-->Odd) & (Odd-->Even)` is also in `Int-->Int`. Thus:

```
(Even-->Odd) & (Odd-->Even) <: Int-->Int
```

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

Subtyping

Notice that every function value in `(Even-->Odd)` & `(Odd-->Even)` is also in `Int-->Int`. Thus:

$$\texttt{(Even-->Odd)}\ \&\ \texttt{(Odd-->Even)}\ \texttt{<:}\ \texttt{Int-->Int}$$

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

The above is just an instance of the following relation

$$(\mathtt{S}_1\texttt{-->}\mathtt{T}_1)\ \&\ (\mathtt{S}_2\texttt{-->}\mathtt{T}_2)\ \texttt{<:}\ (\mathtt{S}_1 \vee \mathtt{S}_2)\texttt{-->}(\mathtt{T}_1 \vee \mathtt{T}_2) \tag{4}$$

that holds for all types, $\mathtt{S}_1$, $\mathtt{S}_2$, $\mathtt{T}_1$, and $\mathtt{T}_2$,

Subtyping

Notice that every function value in `(Even-->Odd) & (Odd-->Even)` is also in `Int-->Int`. Thus:

$$\texttt{(Even-->Odd) \& (Odd-->Even) <: Int-->Int}$$

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

The above is just an instance of the following relation

$$(\mathtt{S}_1\texttt{-->}\mathtt{T}_1)\ \&\ (\mathtt{S}_2\texttt{-->}\mathtt{T}_2)\ \texttt{<:}\ (\mathtt{S}_1 \vee \mathtt{S}_2)\texttt{-->}(\mathtt{T}_1 \vee \mathtt{T}_2) \qquad (4)$$

that holds for all types, $\mathtt{S}_1$, $\mathtt{S}_2$, $\mathtt{T}_1$, and $\mathtt{T}_2$,

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

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

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

$$\texttt{sum} : (\texttt{(Int,Int)-->Int}) \;\&\; (\texttt{(Bool,Bool)-->Bool}), \qquad (5)$$

Overloaded functions

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

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

$$\texttt{sum} : (\texttt{(Int , Int)-->Int}) \ \& \ (\texttt{(Bool , Bool)-->Bool}), \qquad (5)$$

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

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

Overloaded functions

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

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

$$\texttt{sum} : ((\texttt{Int , Int})\texttt{-->Int}) \ \& \ ((\texttt{Bool , Bool})\texttt{-->Bool}), \tag{5}$$

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

```
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:

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

Both definitions of `sumC` have type

$$(\texttt{Int-->}(\texttt{Int-->Int})) \ \& \ (\texttt{Bool-->}(\texttt{Bool-->Bool})). \tag{6}$$

though partial application is possible only with the first definition of `sumC`

Dynamic dispatch

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

Dynamic dispatch

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

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

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

```
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

```
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: *v* is the type that contains just value *v*)

$$
\begin{array}{rl}
 & (\texttt{(Even , Odd)-->1}) \\
\& & (\texttt{(Odd , Even)-->1}) \\
\& & (\texttt{(Int,Int)-->0} \vee \texttt{1})
\end{array}
$$

Exercise

Find a more precise type and justify how the type checker can deduce it.

Formation rules for multi-subroutines: Ambigous Selection

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 ambigous: the computation is stuck on arguments of type `(Even,Odd)`.

Formation rules for multi-subroutines: Ambigous Selection

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 ambigous: 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: Ambigous Selection

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 ambigous: 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`.

It is a *formation rule*. It belongs to language design not to the type system:

$$\texttt{((Even,Int) --> } 0\vee 1 \texttt{) \& ((Int,Odd) --> } 0\vee 1 \texttt{)}$$

the type above is perfectly ok (and a correct type for `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, $\Rightarrow$
- 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, $\Rightarrow$
- the code selected for a multi-subroutine changes, $\Rightarrow$
- 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, $\Rightarrow$
- the code selected for a multi-subroutine changes, $\Rightarrow$
- the type of application changes

Types may *only* decrease along the computation

Consider again:

```
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

$$(\texttt{(Even,Odd)-->1}) \;\&\; (\texttt{(Odd,Even)-->1}) \;\&\; (\texttt{(Int,Int)-->0} \vee \texttt{1})$$

Formation rules for multi-subroutines: Specialization

Because of dynamic dispatch during the execution:

- the type of the argument changes, $\Rightarrow$
- the code selected for a multi-subroutine changes, $\Rightarrow$
- the type of application changes

Types may *only* decrease along the computation

Consider again:

```
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

$$\big((\texttt{Even},\texttt{Odd})\texttt{-->}1\big)\ \&\ \big((\texttt{Odd},\texttt{Even})\texttt{-->}1\big)\ \&\ \big((\texttt{Int},\texttt{Int})\texttt{-->}0\vee 1\big)$$

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

- static time: third code selected; static type is $0\vee 1$
- run time: first code selected; dynamic type is 1 (notice 1 `<:` $0\vee 1$)

Formation rules for multi-subroutines: Specialization

“Types may *only* decrease along the computation”

Formation rules for multi-subroutines: Specialization

“Types may *only* decrease along the computation”

Why does it matter?

```
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.

Formation rules for multi-subroutines: Specialization

"Types may *only* decrease along the computation"

Why does it matter?

```
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:

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

Specialization sound: If $S_1 <: S_2$ then $T_1 <: T_2$.

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

$$(\mathtt{S}_1\mathtt{-->}\mathtt{T}_1)\ \&\ (\mathtt{S}_2\mathtt{-->}\mathtt{T}_2)$$

even if $\mathtt{S}_1\mathtt{<:}\mathtt{S}_2$ and $\mathtt{T}_1$ and $\mathtt{T}_2$ are not related. However consider all the possible cases of applications of a function of this type:

1. If the argument is in $\mathtt{S}_1\ \&\ \mathtt{S}_2$, then the application has type $\mathtt{T}_1\ \&\ \mathtt{T}_2$.
2. If the argument is in $\mathtt{S}_1\backslash\mathtt{S}_2$ and case 1 does not apply, then the application has type $\mathtt{T}_1$.
3. If the argument is in $\mathtt{S}_2\backslash\mathtt{S}_1$ and case 1 does not apply, then the application has type $\mathtt{T}_2$.
4. If the argument is in $\mathtt{S}_1 \vee \mathtt{S}_2$ and no previous case applies, then the application has type $\mathtt{T}_1 \vee \mathtt{T}_2$.

Formation rules for multi-subroutines: Specialization

This case

1. If the argument is in $\texttt{S}_1 \;\&\; \texttt{S}_2$, then the application has type $\texttt{T}_1 \;\&\; \texttt{T}_2$.

may confuse the programmer when $\texttt{S}_2\texttt{<:}\texttt{S}_1$, since in this case $\texttt{S}_2 = \texttt{S}_2 \;\&\; \texttt{S}_1$:

When a function of type $(\texttt{S}_1\texttt{-->}\texttt{T}_1) \;\&\; (\texttt{S}_2\texttt{-->}\texttt{T}_2)$ with $\texttt{S}_2\texttt{<:}\texttt{S}_1$, is applied to an argument of type $\texttt{S}_2$, then the application returns results in $\texttt{T}_1 \;\&\; \texttt{T}_2$.

Design choice: to avoid confusion force (wlog) the programmer to specify that the return type for a $\texttt{S}_2$ input is (some subtype of) $\texttt{T}_1 \;\&\; \texttt{T}_2$.

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

Outline

Covariance and contravariance

Homework assignment:

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

   G. Castagna. Covariance and Contravariance: a fresh look at an old issue. *Draft manuscript, 2014.*

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

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

XML syntax

```
<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

```
<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="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
             ]
             <book year=String>[
               <title>[PCDATA]
               <editor>
                 <last>[PCDATA]
                 <first>[PCDATA]
               ]
               PCDATA
             ]
           ]
```

String = [PCDATA] = [Char*]

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>[                    attribute types
                      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                        nested elements
                      (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+)                        unions
                      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?                          optional elems
                      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]                          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

```
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`**.**

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

```
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

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

with patterns one can write

```
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

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

with patterns one can write

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

which is syntactic sugar for

```
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

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

with patterns one can write

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

which is syntactic sugar for

```
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:

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

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)
```

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)
```

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

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,

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 `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, constants.

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, constants.

But if we:

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, 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:

```
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:

```
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 `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, 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:

```
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);**
3. **consider the wildcard “_” as synonym of** `Any`

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, 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);**
3. **consider the wildcard "_" as synonym of** `Any`

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, constants.~~

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:

```
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 `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, constants.~~

Key idea behind regular patterns

Patterns are types with capture variables

Define types: patterns come for free.

Patterns in CDuce

Patterns = Types + Capture variables

Patterns in CDuce

Patterns = Types + Capture variables

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

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

```
<bib>[x::Book*]
```

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

```
<bib>[x::Book*]
```

The pattern binds `x` to the *sequence* of all books in the bibliography

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

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

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

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

Returns the content of `bibs`.

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

```
    <bib>[( x::<book year="2005">_ | y::_ )*]
```

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

```
<bib>[( x::<book year="2005">_ | y::_ )*]
```

Binds `x` to the sequence of all this year's books, and `y` to all the other books.

Patterns = Types + Capture variables

TYPES

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

PATTERNS

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

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

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

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

Patterns = Types + Capture variables

TYPES

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

PATTERNS

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

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

```
<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 = Types + Capture variables

TYPES

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

PATTERNS

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

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

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

Returns all the captured books

Patterns in CDuce

Patterns = Types + Capture variables

TYPES

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

PATTERNS

```
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

TYPES

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

PATTERNS

```
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+]`

Outline

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

```
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

```
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

```
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

```
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...

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 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 ]
```

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

```
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 Polymorphic CDuce

```
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 ]
```

Outline

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
```

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
```

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>[ ] ]
```

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>[ ] ]
```

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.

```
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
```

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
```

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
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.

```
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.

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
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 = <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

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.

```
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

Outline

Toolkit

Every programming language needs tools / libraries / DLS extensions.

Available for CDuce:

- OCaml full integration
- Web-services API
- Navigational patterns (à la XPath) [experimental]

CDuce↔OCaml Integration

A CDuce application that requires OCaml code

CDuce↔OCaml Integration

A CDuce application that requires OCaml code

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

A CDuce application that requires OCaml code

- Reuse existing librairies
  - 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 librairies
  - 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↔OCaml Integration

A CDuce application that requires OCaml code

- Reuse existing librairies
  - 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 librairies
  - 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↔OCaml Integration

A CDuce application that requires OCaml code

- Reuse existing librairies
  - 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

Need to seamlessly call OCaml code in CDuce and viceversa

Main Challenges

Main Challenges

1. **Seamless integration:**

Main Challenges

1. **Seamless integration:**
   No explicit conversion function in programs:

Main Challenges

1. **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

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 → CDuce

How to integrate the two type systems?

The translation can go just one way: OCaml → 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: 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;
  $\Rightarrow$ *CDuce typing would be lost.*

- **CDuce has unions, intersections, differences, heterogeneous lists; OCaml does not**
  $\Rightarrow$ *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;
  $\Rightarrow$ *CDuce typing would be lost.*
- ⊕ **CDuce has unions, intersections, differences, heterogeneous lists; OCaml does not**
  $\Rightarrow$ *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 → 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).**

- ⇒ *Polymorphic OCaml libraries/functions must be first instantied to be used in CDuce*

In practice

1. Define a mapping $\mathbb{T}$ from OCaml types to CDuce types.

In practice

1. Define a mapping $\mathbb{T}$ from OCaml types to CDuce types.

t (*OCaml*)	$\mathbb{T}(t)$ (*CDuce*)
`int`	`min_int--max_int`
`string`	`Latin1`
$t_1 * t_2$	$(\mathbb{T}(t_1), \mathbb{T}(t_2))$
$t_1 \to t_2$	$\mathbb{T}(t_1) \to \mathbb{T}(t_2)$
t `list`	$[\mathbb{T}(t)*]$
t `array`	$[\mathbb{T}(t)*]$
t `option`	$[\mathbb{T}(t)?]$
t `ref`	`ref` $\mathbb{T}(t)$
A_1 `of` $t_1 \mid \ldots \mid A_n$ `of` t_n	$(\text{`}A_1, \mathbb{T}(t_1)) \mid \ldots \mid (\text{`}A_n, \mathbb{T}(t_n))$
$\{l_1 = t_1; \ldots; l_n = t_n\}$	$\{l_1 = \mathbb{T}(t_1); \ldots; l_n = \mathbb{T}(t_n)\}$

In practice

1. Define a mapping $\mathbb{T}$ from OCaml types to CDuce types.

t (*OCaml*)	$\mathbb{T}(t)$ (*CDuce*)
`int`	`min_int--max_int`
`string`	`Latin1`
$t_1 * t_2$	$(\mathbb{T}(t_1), \mathbb{T}(t_2))$
$t_1 \to t_2$	$\mathbb{T}(t_1) \to \mathbb{T}(t_2)$
t `list`	$[\mathbb{T}(t)*]$
t `array`	$[\mathbb{T}(t)*]$
t `option`	$[\mathbb{T}(t)?]$
t `ref`	`ref` $\mathbb{T}(t)$
A_1 `of` $t_1 \mid \ldots \mid A_n$ `of` t_n	$(\text{‘}A_1, \mathbb{T}(t_1)) \mid \ldots \mid (\text{‘}A_n, \mathbb{T}(t_n))$
$\{l_1 = t_1; \ldots; l_n = t_n\}$	$\{l_1 = \mathbb{T}(t_1); \ldots; l_n = \mathbb{T}(t_n)\}$

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

In practice

1. Define a mapping $\mathbb{T}$ from OCaml types to CDuce types.

t (*OCaml*)	$\mathbb{T}(t)$ (*CDuce*)
`int`	`min_int--max_int`
`string`	`Latin1`
$t_1 * t_2$	$(\mathbb{T}(t_1), \mathbb{T}(t_2))$
$t_1 \to t_2$	$\mathbb{T}(t_1) \to \mathbb{T}(t_2)$
t `list`	$[\mathbb{T}(t)*]$
t `array`	$[\mathbb{T}(t)*]$
t `option`	$[\mathbb{T}(t)?]$
t `ref`	`ref` $\mathbb{T}(t)$
A_1 `of` $t_1 \mid \ldots \mid A_n$ `of` t_n	$(\text{‘}A_1, \mathbb{T}(t_1)) \mid \ldots \mid (\text{‘}A_n, \mathbb{T}(t_n))$
$\{l_1 = t_1; \ldots; l_n = t_n\}$	$\{l_1 = \mathbb{T}(t_1); \ldots; l_n = \mathbb{T}(t_n)\}$

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

`ocaml2cduce`: $t \to \mathbb{T}(t)$

`cduce2ocaml`: $\mathbb{T}(t) \to t$

Calling OCaml from CDuce

Easy

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

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

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

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

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
- applies `ocaml2cduce` to the result 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
- applies `ocaml2cduce` to the result of the call

Example: use ocaml-mysql library in CDuce

```
let db = Mysql.connect Mysql.defaults;;

match Mysql.list_dbs db `None [] with
| (`Some,l) -> print [ 'Databases: ' !(string_of l) '\ n' ]
| `None -> [];;
```

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.

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 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 $\mathbb{T}(t)$

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 $\mathbb{T}(t)$
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 `val` *f*:*t* in the `.mli` file, then the CDuce type of *f* is a *subtype* of $\mathbb{T}(t)$
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: *)
val fact: Big_int.big_int -> Big_int.big_int

(* File cdnum.cd: *)
let aux ((Int,Int) -> Int)
| (x, 0 | 1) -> x
| (x, n) -> aux (x * n, n - 1)

let fact (x : Int) : Int = aux(1,x)
```