ℂDuce: a white paper

(working document)

Véronique Benzaken
LRI, UMR 8623, C.N.R.S.
Université Paris-Sud, Orsay, France
Veronique.Benzaken@lri.fr

Giuseppe Castagna
C.N.R.S., *Département d'Informatique*
École Normale Supérieure, Paris, France
Giuseppe.Castagna@ens.fr

Alain Frisch
Département d'Informatique
École Normale Supérieure, Paris, France
Alain.Frisch@ens.fr

Version of October 22, 2002
Document regularly updated; the lastest version can be retrieved at www.cduce.org

Abstract

In this paper, we present the functional language ℂDuce*, discuss some design issues, and show its adequacy for working with XML documents. Peculiar features of* ℂDuce *are a powerful pattern matching, first class functions, overloaded functions, a very rich type system (arrows, sequences, pairs, records, intersections, unions, differences), precise type inference and a natural interpretation of types as sets of values. We also discuss how to add constructs for programming XML queries in a declarative (and, thus, optimizable) way and finally sketch a dispatch algorithm to demonstrate how static type information can be used in efficient compilation schemas.*

1 Introduction

In this paper, we present the functional language ℂDuce, discuss some design issues, and show its adequacy for writing applications that handle, transform, and query XML documents. To keep the presentation short, and because part of the design is still in progress, we just present some highlights of the language. Theoretical foundations of the type system can be found in [8]. The homepage for ℂDuce, including an extended version of this paper, other references, and an online interactive prototype, is `http://www.cduce.org`.

ℂDuce is a general purpose typed functional programming language, whose design is guided by keeping XML applications in mind. The work on ℂDuce started from an attempt to overtake some limitations of XDuce [11, 9, 10]:

- *General purpose types.* XDuce is XML-specific: the only datatype it can manipulate is (sequences of) XML documents; this makes it difficult to write complex applications, which are not just simple transformations (filtering, reordering, renaming). XDuce demonstrates how specific features (regular expression types and patterns) may be adequate to XML applications, but we believe that these features could be integrated in a less specific language, improving the interface between XML and the core application. ℂDuce uses a general purpose type algebra, with standard type constructor (product, records, functions), and retains all the power of XDuce regular expression types through explicit use of recursive types and boolean combinations (union, intersection, difference). It is possible with ℂDuce to create complex data structures, model XML document types, and to interface smoothly with other languages.

- *Expressive patterns.* As a by-product, we extended the pattern algebra, allowing to extract by a single pattern non-consecutive subsequences of elements; unlike XDuce[1], we have an *exact* typing even for non-tail variables, and the pattern algorithms are easily derived from simple definitions [8].

- *Higher-order and overloading.* XDuce is a functional language, with a pattern matching reminiscent of ML, but it lacks higher-order functions. ℂDuce type system includes higher-order functions, and also provides late bound overloaded functions. Such functions may dispatch on the dynamic type of their argument, making it possible to use an object-oriented style programming in a functional setting. We are also considering to support in ℂDuce incremental programming through a module system allowing further redefinition/specialization of overloaded functions.

- *XML attributes.* To encode XML attribute sets in a classical language, the natural datatype seems to be record types. To take into account attribute specificities (an attribute may be optional, mandatory, or prohibited), we designed special record type constructors.

- *Type constraints.* An XML document is not only a tree structure; it also has basic information, such as numbers or structured strings, in attributes or element contents. We believe that the type system and the pattern matching should reflect this in some detail, for instance to express (and validate) some constraints on the produced XML documents (for instance, that a date attribute is a string of the form YYYY-MM-DD).

A preliminary and shorter version of this work was presented in the workshop *PLAN-X: Programming Language Technologies for XML*, Pittsburgh, PA, October 2002.

[1] Hosoya has drawn our attention to his independent work (not yet published) on this issue; he has recently implemented in XDuce the support for exact typing of non-tail variables, using an automaton-based approach.

During our work we found out that ℂDuce and more generally XML applications raise some interesting implementation issues. As a matter of fact, we verified that precise type information on the documents the applications work on allows many important optimizations (for instance, searching for an element with a given name in a whole document is of course much easier when one knows where such an element may potentially be).

2 A sample session

Let us write and comment a sample ℂDuce program, on the lines of the typical example of [7]. First, we declare some types:

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

The type `Bib` represents XML documents that store bibliographic information (the corresponding XSchema can be found in [7]). It states that a bibliography is a possibly empty sequence of book elements, each consisting of a sequence formed by one title element, one year element and one or more author elements. Square brackets `[...]` are used to denote sequences, whose type is constrained by a regular expression over types. Regular expressions for character strings are enclosed in `/.../` (they can be omitted when they consist of a single string): for example the type `IntStr` specifies that the content of a `<year>` element is a string representation of an integer. We could have used directly integers by defining `Year` as `<year>[1900..maxint]`

An XML document satisfying the above type is the following

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

If the file is stored in the file, say, `bib.xml`, then it can be loaded with the built-in operator `load_xml`, assigned to a local variable `bib0`, and immediately checked to be of type `Bib` by pattern matching:

```
let bib0 =
  match (load_xml "bib.xml") with
    | (x & Bib) -> x
    | _ -> error "Wrong type !";;
```

when this declaration is entered interactively the system answers:

```
|- bib0 : Bib
```

which indicates that the type checker keeps track that `bib0` is indeed of type `Bib` (`error` raises a fatal exception when the loaded document is not of the correct type). We could have defined `bib0` directly as follows

```
let bib0 =
      <bib>[
         <book>[
            <title>["Persistent Object Systems"]
            <year>["1994"]
            <author>["M. Atkinson"]
            <author>["V. Benzaken"]
            <author>["D. Maier"]
         ]
         <book>[
            <title>["OOP: a unified foundation"]
            <year>["1997"]
            <author>["G. Castagna"]
         ]
      ]
```

Suppose that instead of working with the XML type `Bib` we preferred to use an internal representation for bibliographies based on record types. We can easily define this type and a conversion function as follows:

```
type Intern =
 {title=String; year=Int; authors=[String+]};;

let fun intern (Bib -> [Intern*])
 <bib>l -> map l with
  <book>[<title>[t] <year>[y] a::Author+] ->
   {
    title = t;
    year = int_of_string y;
    authors = map a with <author>[x] -> x
   };;

|- intern : Bib -> [Intern*]

let bib1 = intern bib0;;

|- bib1 : [Intern*]
```

The `let fun` construction defines the function `intern` whose type is `Bib->[Intern*]` as declared in the *interface* of the function which follows its name.

The first pattern extracts the sequence of books and binds it to the variable `l`; the `map l with ...` expression transforms each element of `l` into the corresponding record; the last pattern matching removes the tags `author`. Note that the application of `int_of_string` cannot fail because the `y` value is of type `IntStr`; otherwise, the system would have issued a warning. In the pattern `<book>[...]`, we observe two kinds of capture variables: `t` and `y` capture a single object, whereas `a` captures a sequence of objects (x::E binds to x the whole subsequence matching the regular expression E).

We can now define an overloaded function that extracts the list of authors from either representation; if the argument is an `Intern` object, then we get a sequence of strings; otherwise, we get a sequence of `Author` elements.

```
let fun authors(Intern->[String+]; Book->[Author+])
  | { authors = a }
  | <book>[<title>_ <year>_; a] -> a;;
```

The matching expression in the body of the function is formed by a single branch (with an alternative pattern: that is, `({...}|<book>[...])-> a`). The function interface reflects the overloaded aspect of the function as it declares two distinct arrow types. The underscore symbol _ matches every expression, while the semi-colon followed by a binds the rest of the sequence to a (the semicolon expression ;x is syntactic sugar for x::(_*)): the type system infers that a captures an object of type [Author+] when the argument of the function is of type Book.

```
let fun extract ([(String|Author)+] -> [String+])
   l -> map l with
     | <author>[a] -> a
     | a -> a;;
```

This function takes a list of values that are either strings or author elements and removes the tags; the | in types stands for union. Now, any sequence of type [String+] or [Author+] is *a fortiori* an acceptable argument for this function, so we can define:

```
let fun authors2 ( (Intern|Book) -> [String+] )
 b -> extract (authors b);;
```

(of course, there are more direct ways to implement this function.)

Another possible XML representation for the bibliography would be a flat sequence of elements, as defined by the type:

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

The corresponding pair of conversion functions is:

```
let fun flatten_bib ([Book*] -> Flat)
 l -> transform l with <book>x -> x;;

|- flatten_bib : [Book*] -> Flat

let fun unflatten_bib (Flat -> [Book*])
 [ b::(Title Year Author+); r ] ->
      [<book>b] @ (unflatten_bib r)
|[] -> [];;

|- unflatten_bib : Flat -> [Book*]
```

The @ operator performs sequence concatenation. The transform construction is very much like map; the main difference is that all the returned sequences are concatenated together (that is, it flattens the result that map would return).

It is possible to use a *value* in type position to force that specific value; for instance if we declare

```
type Chair_auth = <author>["Pierce"|"Wadler"];;
type Chair = <book>[_* Chair_auth _*];;
```

then Chair is the type of books where either Pierce or Wadler (chairmen of the PLAN-X workshop) appear as one author. Now we can define an extraction function:

```
let fun chair_books (Bib -> [(Chair & Book)*])
 <bib>[(b::Chair | _)*] -> b;;

|- chair_books: Bib -> [(Chair & Book)*]
```

The & denotes the intersection of types, thus the interface of the function declares that the result will be both of type Chair and of type Book (these two being incomparable). Note that the b variable is under a repetition operator (the * Kleene-star in regular expressions); the meaning is that all the matching subsequences are concatenated together (here, each of them has a single element). As explained in Section 3.6 on pattern matching, a variable can occur several times in the same pattern (in the case above because of the repetition operator): when this happens the variable is bound to the recomposition of the bindings of all occurrences by the constructor they appear in (in the case above all the bindings of the occurrences of b are recomposed into a sequence). This example demonstrates how a single pattern can perform a quite complex operation.

The type Chair does not say anything about the year and title elements. But since the argument of the function is known to be of type Bib, it is possible to infer—and ℂDuce does it—that the extracted values are both of type Chair and of type Book (this is indicated by the intersection type operator &).

3 Overview of the ℂDuce language

3.1 The type algebra

ℂDuce type algebra has no specific constructor for sequences and XML documents. The constructions we used in the previous section are encoded, as shown in Section 3.2, in the core type algebra formed by the following types:

- basic scalar types, such as Int, String, Bool, etc., atoms (an atom is a constant of the form ‘*id* where *id* is an arbitrary identifier) and two type constants Empty and Any (the latter is also written _, especially in patterns) that denote respectively the empty (i.e., the smallest) and the universal (i.e., the largest) type;
- classical types constructors: product types (t_1,t_2), record types { a_1=t_1; ... ; a_n=t_n }, and functional types (t_1 -> t_2);
- boolean connectives: intersection t_1&t_2, union $t_1|t_2$ and difference $t_1\backslash t_2$;
- singleton types: for any scalar or constructed (non-functional) value v, v is itself a type (for instance, the value ‘nil denotes also the type of empty sequences, while 18 is the type of the integer 18);
- recursive types: they are defined by recursive toplevel declarations or by the syntax T where T_1=t_1 and ... and T_n=t_n, where T and T_i’ are type identifiers (i.e., identifiers starting by a capitalized letter). For instance, the type of sequences of integers may be written Ilist where Ilist = (Int, Ilist) | ‘nil.

In ℂDuce types have a set-theoretic interpretation: a type is the set of all *values* (i.e. closed irreducible expressions: roughly, expressions that are neither applications, nor field selections, nor matching expressions; sometimes we equivalently use the word “result” instead of “value”) that have that type. For example, the type (t_1,t_2) is the set of all expressions (v_1,v_2) where v_i is a value of type t_i; similarly t_1->t_2 is the set of all functional expressions fun f (s_1;...;s_n)e that have type

t_1`->`t_2 (that is, all the functional expressions that when applied to an expression of type t_1 return a result in t_2). Likewise, when a value is used in a type position (as in `Chair_auth` in the previous section) it denotes the singleton containing that value (whence the name of singleton types).

This interpretation of types is at the basis of the whole intuition of the ℂDuce's type system: the programmer must rely on it to understand all the type constructions and type equivalences of the system. Thus, for example, the difference of two types contains all the values that are contained in the first type but not in the second, the union of two types is formed by all the values of each type, and the intersection of, say, an arrow and a record is *equivalent* (in the sense that it has the same interpretation as) to the empty type.

In particular, subtyping is just set inclusion: a type is a subtype of another if the latter contains all the values that are in the former (for more details see [8]).

There are actually two kinds of record types: the open record type `{` $a_1 = t_1$ `;...;` $a_n = t_n$ `}` that classifies records in which the fields labeled a_i are present with the prescribed type, but other fields may also appear; the closed record type `{|` $a_1 = t_1$ `; ... ;` $a_n = t_n$ `|}`, instead, forbids any label other than the a_i's.[2] It is also possible (both for open and for closed record types) to specify optional fields: the syntax a_i `=?` t_i states that the a_i field may be absent, but when it is present, it must have type t_i. There is a lot of natural subtyping and equivalence relations that hold for record types, like `{|`$a = t$`|}`$\leq$`{`$a = t$`}`, or `{` $a_1 = t_1$ `;` $a_2 = t_2$ `}` $\simeq$ `{` $a_1 = t_1$ `}&{` $a_2 = t_2$ `}`, or `{|`a_1`=`t_1`;`a_2`=?`t_2`|}` $\simeq$ `{|`a_1`=`t_1`|} | {|`a_1`=`t_1`;` a_2`=`t_2`|}`; and once more they can all be deduced by considering the set theoretic interpretation of record types as sets of record values.

For scalar types, we also introduce specific subtypes of `Int` and `String`: i`..`j $\leq$ `Int` is an interval (i and j are integer constants), and `/`*regexp*`/`$\leq$`String` is the set of character strings described by the regular expression *regexp*. Regular expressions are built from string constants, the wild-card "`.`", character classes, and usual regular expression operators `|`,`*`,`?`,`+` and concatenation.

3.2 XML documents

ℂDuce is close in spirit and in syntax to other XML-oriented languages such as XDuce or the algebra introduced in [7]. However, it is a general purpose programming language based on a very small functional core. In particular, XML-related types are encoded in terms of the type constructors we presented in the previous section.

Sequences Sequences are encoded *à la* Lisp using pairs and a terminator `'nil` representing the empty sequence: a sequence of values $v_1, \ldots, v_n$ is written in ℂDuce as `[` $v_1 \ldots v_n$ `]`, but this actually is only syntactic sugar for `(`v_1`,(...,(`v_n`, 'nil)...))`.

In the sample section we saw that regular expressions can be applied to types to define new sequence types. This is just syntactic sugar as the same sequence types could be defined by combining boolean type connectives and recursive types. More precisely it is possible to define sequence types by `[`*tyregexp*`]` where *tyregexp* is a regular expression built from types and usual regular expression operators. For instance, the `Ilist` type in the previous section is equivalent to `[Int*]` while `[Int* String+]` represents sequences built from a possibly empty list of integers concatenated with a non-empty list of strings.

XML elements The value `'nil` is just a special case of atom type `'`*id* where *id* is any identifier. Atoms are also used to encode XML tags. We saw in the sample session that an XML element <*tag*> *elem-seq* </*tag*> can be written in ℂDuce as <*tag*>`[`*elem-seq*`]` where *elem-seq* is a sequence of elements. This latter notation actually is syntactic sugar for `('`*tag*`,({},[`*elem-seq*`]))`. The expression `{}` denotes the empty record value. In its more general form tags can have attributes as for <*tag* $a_1 = v_1$ ... $a_n = v_n$> *elem-seq* </*tag*>. This is written in ℂDuce as <*tag* `{`$a_1 = v_1$`;...;`$a_n = v_n$`}`>`[`*elem-seq*`]` which is syntactic sugar for `('`*tag*`,({`$a_1 = v_1$`;...;`$a_n = v_n$`},[`*elem-seq*`]))`. When appearing in tags curly braces and semicolons can be omitted as in:

```
<a href="click.htm"  target="_top">["Click here"]
```

We applied this convention in all the examples of the sample session as we always omitted the pair of braces `{}` denoting the empty record.

As an illustration, here is a set of declarations for an XML document type representing a (flat) address book where the address tag has the optional attribute `kind`:

```
type AddrBook = <book>Content;;
type Content = [(Name Addr Tel?)*];;
type Name = <name>[String];;
type Addr = <addr {kind=?"home"|"work"}>[String];;
type Tel  = <tel>[String];;
```

The same convention as for record expressions is used for *open* record types occurring in tags of XML types (and patterns, see Section 3.6), thus an equivalent notation for the `Addr` type is:

```
type Addr = <addr kind=?"home"|"work">[String];;
```

On the contrary the `{| |}` parentheses for closed record types cannot be omitted. For example the type

```
type Addr1 = <addr {|kind =? "home"|"work"|}>[...]
```

matches elements with tag `addr` and that have *at most* the `kind` attribute with the specified type (elements of type `Addr`, instead, can have arbitrary attributes with the only restriction that if present the `kind` attribute must have the specified type).

This kind of flat representation (mixing fields for all the entries in the address book) is a little weird; here is the ℂDuce function that transforms it into a sequence of entries (coded as records) where the address attribute is discarded:

[2] There is a small subtlety about singleton record types. For instance, the type `{ x = 3 }`, being open, contains all the records that have field `x = 3` and maybe other fields too. The singleton type corresponding to the value `{ x = 3 }` must be written `{| x = 3 |}`. We have chosen the same notation for record values and *open* record types, because we believe that open record types are much more useful in programming than closed ones.

```
type Entry = { name = String; addr = String;
               tel =? String };;

let fun parse (Content -> [Entry*])
  [<name>[n] <addr>[a] <tel>[t]; r] ->
       ({ name = n; addr = a; tel = t }, parse r)
| [<name>[n] <addr>[a]; r] ->
       ({ name = n; addr = a }, parse r)
| [] -> [];;
```

The function body is made of a pattern matching, that is a sequence of branches p `->` e where p is a *pattern* and e an *expression*. The powerful pattern algebra (discussed in Section 3.6) greatly contributes to ℂDuce expressivity. For instance, a single pattern can filter out of an addressbook all the telephone numbers:

```
let fun notel (Content -> [(Name Addr)*])
  [ (x::(Name Addr) Tel?)* ] -> x;;
```

The several matched subsequences for `x` are concatenated together; the pattern could also be written `[(x::(Name|Addr) Tel?)*]` and the ℂDuce type-checker would still be clever enough to infer that `x` can only bind a sequence of type `[(Name Addr)*]`. The following function splits a list of `Entry` records according to whether the phone number is present or not:

```
type A = Entry & { tel = Any };;
type B = Entry & { tel =? Empty };;

let fun split ([Entry*] -> ([A*],[B*]))
  [((x::A) | (y::B))*] -> (x,y);;
```

Note the definition of `B`. The record type `{tel=?Empty}` means "whenever the field `tel` is present, its value must belong to the type `Empty`"; as there is no value of type `Empty`, this is equivalent to saying "the field `tel` must be absent".

3.3 Overloaded functions

The simplest form for a toplevel function declaration is

$\texttt{let fun}\ f\ \ (t\texttt{->}s)\ \ x\ \texttt{->}\ e$

in which the body of a function is formed by a single branch x`->`e of pattern matching. As we saw in the previous sections, the body of a function may be formed by several branches with complex patterns. The interface $(t\texttt{->}s)$ specifies a constraint on the behavior of the function to be checked by the type system: when applied to an argument of type t, the function returns a result of type s. In general the interface of a function may specify several such constraints, as we did for example in the `authors` function in Section 2.

The general form of a toplevel function declaration is thus:

$\texttt{let fun}\ f(t_1\texttt{->}s_1\texttt{;}\ldots\texttt{;}t_n\texttt{->}s_n)\ \texttt{|}\ p_1\texttt{->}e_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_m\texttt{->}e_m$

(the first vertical bar can be omitted). Such a function accepts arguments of type $(t_1|\ldots|t_n)$, it has all the types t_i`->`s_i, and, thus, it also has their intersection $(t_1\texttt{->}s_1\&\ldots\&t_n\texttt{->}s_n)$.

The use of several arrow types in an interface serves to give to the function a more precise (inasmuch as smaller) type. We can roughly distinguish two different uses of multiple arrow types in an interface:

1. when each arrow type specifies the behavior of a different piece of code forming the body of the function, the compound interface serves to specify the *overloaded* behavior of the function. This is the case for the `author` function of Section 2 or for the function below

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

   where each arrow type in the interface refers to a different branch of the body.
2. when the arrow types specify different behavior for a same code, then the compound interface serves to give a more precise description of the behavior of the function. For example the interface below

   ```
   type IL = [Int*];;                  integer list
   type NE = [Int+];;                  non empty list
   type EE = ([],[]);;                 pair of empty lists

   let fun concat ( ((IL,IL)\EE)->NE; EE->[])
    | ((h,t),l2) -> (h, concat (t,l2))
    | ([],l2) -> l2;;
   ```

   specifies that the function `concat` returns an empty sequence if and only if both the argument sequences are empty (and the type checker verifies it).

Of course, these two uses can be combined allowing the definition of overloaded functions with a precise typing.

A typical use for overloaded functions in the XML setting could be the definition of transformations that operate on documents of different DTDs and, for example, uniformly replace some tag for another (changing in this way the type) like in the following case:

```
type Movie = <movie>[Title <producer>[String]];;
type Film  = <film>[Titre <producteur>[String]];
type Livre = <livre>[Titre Annee Auteur+];;
type Titre = <titre>[String];;
        ⋮

let fun to_french (String -> String;
                   Book -> Livre;
                   Movie -> Film;
                   Title -> Titre; ...)
 <(t)>x ->
   let u  = match t with
   | `book -> `livre | `movie -> `film
   | `title -> `titre | `year -> `annee
   | `author -> `auteur | `producer -> `producteur
   | y -> y
   in
   <(u)>(map x with e -> to_french e);;
```

Similarly overloading can be used to produce documents structured in different ways according to the value of some tag attribute, and this is particularly useful in case of recursive definitions. We will show this by an example that we comment in detail in Section 3.8.

3.4 Higher-order functions

Functions are first class values in ℂDuce. This means that a function can be fed to or returned by a so-called higher-order function. The syntax for a local function is the same as a toplevel function declaration, $\texttt{fun}\,f(t_1\texttt{->}s_1\texttt{;}\ldots\texttt{;}t_n\texttt{->}s_n)\ldots$, with the only difference that f may be omitted if the function is not recursive. A classical example of higher order function is the composition function, which takes two functions and returns as result the function that composes them:

```
let fun composition( (s->t,t->u) -> (s->u) )
  (f,g) -> fun(s->u) x->g(f x)
```

Higher-order holds for all functions, overloaded functions included. For example consider the following function which takes a binary integer function, an `ITree`, and returns an integer:

```
type ITree = Int | (ITree,ITree);;
let fun squeeze((((Int,Int)->Int),ITree) -> Int)
   | (f,(x,y)) -> f (squeeze(f,x),squeeze(f,y))
   | (f,x) -> x;;
|- squeeze : ((Int,Int)->Int),ITree) -> Int
```

it is perfectly correct to pass to `squeeze` the overloaded function `add` defined before:

```
squeeze((3,(4,3)),add);;
|- Int
=> 10
```

we can always pass a function of type `((Int,Int)->Int) & (String,String)->String)` where a function of type `(Int,Int)->Int` is expected, as the latter is a subtype of the former. Once more it would be possible to use a compound interface to specialize the behavior of `squeeze`. If for example `STree = String | (STree,STree)`, then we could have specified for `squeeze` the following interface

```
( ((String,String)->String , STree) -> String ;
  ((Int,Int)->Int , ITree) -> Int )
```

In this case we would have extended the application domain of `squeeze` since `squeeze(add,("a","b"))` would typecheck and return the result `"ab"`.

In the setting of XML applications, a typical use of higher order functions is to parametrize the behavior of another function. For instance, the function `render` below, which is in charge of "rendering" a complex document to HTML, is parametrized by a function of type `Entry2html` that renders specific parts of the document.

```
type Html = ...;;
type Entry2html = [Name Addr Tel?] -> Html;;

let fun render ((Entry2html,AddrBook) -> Html)
...;;
```

We next define the function `print_entry` of type `Entry2html` that is passed to `render` to define a new function `my_render`:

```
let fun print_entry (
  ([Name Addr Tel?] | Entry) -> Html
 )
 | [<name>[n] <addr>[a] <tel>[t]]
 | { name = n; addr = a; tel = t } ->
      ...
 | [<name>[n] <addr>[a]]
 | { name = n; addr = a } ->
      ...;;

let fun my_render (AddrBook -> Html)
 book -> render (print_entry, book);;
```

Note that `([Name Addr Tel?]|Entry) -> Html` is a subtype of `Entry2html`: every function value that when applied to an argument of type `([Name Addr Tel?]|Entry)` returns a result in `Html` is also a function that when applied to an argument of type `[Name Addr Tel?]` returns a result in `Html`. Therefore it is legal to use `print_entry` as the first argument of `render`.

Functions being first-class, it is possible to store them in data structures; for instance, one could dynamically lookup a rendering function in an associative table, according to the value of some field in an input document. This would create a dynamic template system.

3.5 ℂDuce **modules**

In this section we sketch the design guidelines we are following in the definition of the (not yet implemented) ℂDuce's module system.

A ℂDuce module is a sequence of recursive toplevel declarations (types, functions, data, ...); a module may refer to components defined in other modules using a classical dot notation. A module may come with an *interface* that specifies (in a separate file) the type of defined functions and values.

Type declaration All the type declarations in a single module are mutually recursive. A declaration `type T =` t does not *create* a new type; it just gives the name `T` to the type t. This is important because it puts the emphasis on the structure of the values, not the specific type declarations; two unrelated modules may thus communicate and work on the same values, even though they do not share any common declaration.

We are planning to add support for abstract types in two flavors. A *transparent* abstract type gives no information about the structure of its values outside the module where it is defined, but it is still possible to use pattern matching to inspect values of this type. An *opaque* abstract type prevents any such inspection: values are really black boxes outside the module they belong to.

Incremental programming It is possible to overload a function first defined in another module. For instance, imagine that the functions `add` and `concat` of Section 3.3 were defined in some module `A`, and that instead of having `concat` in a separate function we want to obtain the same behavior by overloading `add`. This can be done in a different module as follows:

```
overload A.add ( ((IL,IL)\EE)->NE; EE->[])
   | (l,l) -> A.concat(l,l);;
```

This affects the global behavior of the function `A.add`, even when called inside the module `A`. Technically, this is referred to as dynamic binding. The new behavior is obtained by adding the new branches before the old ones in the definition of `A.add`. As for typing, we see that it is possible to specify an additional set of constraints in the interface. Of course the old constraints must be satisfied by the new function, too (this automatically enforces the inheritance condition of $\lambda\&$: see [3]). The new constraints are statically visible only in the module where the overloading occurs (and the ones that rely on it), not in A, as this would require to type-check it again. However, sometimes, redoing type-checking might result useful. For instance, if we redefine add so that it always returns positive integers

```
overload A.add ((Int,Int)->(0..maxint))
   | (x&Int,y&Int) -> if x+y>0 then x+y else 1;;
```

and we type check again every function in A that calls add we might obtain more precise typing by using the information that the result of add is always positive. We are investigating ways to allow this powerful kind of incremental programming without sacrificing all the benefit of separate compilation and implementation independence; basically, we are planning to use some kind of *interface* for modules and allow the parts of the implementation that have to be potentially re-typechecked to appear in the interface.

3.6 Pattern matching

Pattern matching is one of ℂDuce's key features. Although it has an ML-like flavor, it is much more powerful, as it allows one to express in a single pattern a complex processing that can dynamically check both the structure and the type of the matched values.

We already saw examples of pattern matching forming the body of a function declaration. As in ML, in ℂDuce there is also a standalone pattern-matching expression match e with p_1->e_1 |...| p_n->e_n. Local binding let p=e_1 in e_2 is just syntactic sugar for match e_1 with p -> e_2.

A pattern may either match or reject a value; when it matches, it binds its *capture variables* to the corresponding parts of the value and the computation can continue with the body of the branch. Otherwise, control is passed to the next branch. Note that this is only a description of the semantics of pattern matching, but the actual implementation uses less naive and more efficient algorithms to simulate it; for instance, we designed an algorithm that uses a single (partial) run on the value to dispatch on the correct branch, and takes profit of static typing information (see Section 6).

Capture variables and deconstructors As in ML, a variable, say, x is a pattern that accepts any value and binds it to x. A pair pattern (p_1,p_2) accepts every value of the form (v_1,v_2) where v_i matches p_i. If a variable x appears both in p_1 and in p_2, then each pattern p_i binds x to some value v'_i; the semantics is here to bind the pair (v'_1,v'_2) to x for the whole pattern (and so recursively). For instance, a pattern matching branch (x,(y,x)) -> (x,y) is equivalent to (x1,(y,x2)) -> ((x1,x2),y). Similarly (x,(x,(y,x))) -> (x,y) is equivalent to (x1,(x2,(y,x3))) -> ((x1,(x2,x3)),y). Such examples are though not very interesting as we really gain in expressivity when the multiple occurrences of a variable are generated by the use of recursive patterns, as we show later on.

Similarly to pair patterns, record patterns are of the form {a_1=p_1;...;a_n=p_n} and {|a_1=p_1;...;a_n=p_n|}: the former matches every record whose fields a_i match the p_i while the latter matches record formed exactly by the a_i fields and whose content matches p_i. We use the same convention as for types and allow to omit the braces for open record parentheses occurring in tags. However, contrary to pair patterns, we do not allow multiple occurrences of a variable in a record pattern (there is no difficulty to structure the results of the different occurrences into a record, but we did not found any interesting use of such a feature).

Type constraint and conjunction Every type can be used as a pattern, whose semantics is to accept only values of this type[3], and create no binding. This is particularly useful because in ℂDuce, a type may reflect precise constraints on the values (structure or content, for instance 0..10). Note that scalar constants can be used, and are just special case of type constraint with singleton types. The wild-card type _ simply is an alternative notation for the type constant Any and as such it matches every type. Similarly <_>t (resp. <_ r>t) is a shorthand for (_,(_,t)) (resp. (_,(r,t))).

To combine a type constraint and a capture variable, one can use the conjunction operator & for pattern, as in (x & Int). The semantics of the conjunction in a pattern is to check both sub-patterns and merge their respective set of bindings. This merging may require a non trivial inference of the type system: for example if we have the pattern (x & <book>_), then in order to deduce the type of x the type system must infer the type of the content of the element <book>.

Alternative and default value There is also an alternative (disjunction) operator p | q with first match policy: it first tries to match the pattern p, and if it fails, it tries with q; the two patterns must have the same set of capture variables. This pattern may be used in conjunction with the pattern (x := c), where c is an arbitrary scalar constant, to provide a default value for a capture variable. For instance, the pattern ((x & Int) | (x := 0)) captures a value and binds it to x when it is an integer, and otherwise ignores it and continue with the binding x := 0.

Recursive patterns As for types, recursive patterns may be defined through the syntax P where P_1=p_1 and ... and P_n=p_n where $P, P_1, \ldots, P_n$ are variables ranging over patterns. Recursive patterns allow one to express complex extraction of information from the matched value. For instance, consider the pattern p where p = (x & Int, _) | (_, p); it extracts the first element of type Int from a sequence (recall that sequences are coded with pairs). The order in the alternative is important, because the pattern p where p = (_,

[3] Conversely, every pattern without capture variables is a type; this intermingle motivated our choice to use the same notation for constructors common to types and patterns. For instance, the term (1,2) in a pattern position can be interpreted (i) as a type constraint where the type is the constructed constant (1,2), (ii) as a type constraint where the type is the product type of of the two scalar constants 1 and 2, or (iii) as a pair pattern formed by two type constraints, 1 and 2. All these interpretations yield the same semantics. Therefore the use of the same constructors for types and patterns reduces the number of possible denotations for a same (from a semantic viewpoint) pattern: had we used a different syntax for product types, say, t_1xt_2, then we would have had two denotations, i.e. (1,2) and 1x2, for the same pattern. The same reason motivates our choice of denoting record types by { a_1=t_1;...;a_n=t_n } rather than by the more common { a_1:t_1;...;a_n:t_n }.

p) | (x & Int, _) would extract the *last* element of type Int.

A pattern may also extract and reconstruct a subsequence, using the convention described before that when a capture variable appears on both side of a pair pattern, the two values bound to this variable are paired together. For instance p where p = (x & Int, p) | (_, p) | (x := ‘nil) extracts all the elements of type Int from a sequence and the pattern p where p = (x & Int, (x & Int, _)) | (_, p) extracts the first pair of consecutive integers.

Regular expression patterns ℂDuce provides syntactic sugar to define patterns working on sequences with regular expressions built from patterns, usual regular expression operators, and *sequence capture variable*. For instance, we have seen the pattern [(x::(Name Addr) Tel?)*]; the variable x captures subsequences of consecutive Name and Addr elements, and concatenates all these subsequences. It is actually compiled into the pattern:

```
p where p = (x & Name, q) | (x & ‘nil)
    and q = (x & Addr, r)
    and r = (Tel, p) | p
```

This example illustrates how *sequence capture variables* are compiled by propagating them down to simple patterns, where they become standard capture variable. The (x & ‘nil) pattern above has a double purpose: it checks that the end of the matched sequence has been reached, and it binds x to ‘nil, to create the end of the new sequence.

Note the difference between [x & Int] and [x :: Int]. Both patterns accept sequences formed of a single integer i, but the first one binds i to x, whereas the second one binds to x the (full) subsequence [i].

Regular expression operators *,+,? are *greedy* in the sense that they try to match the longest possible sequence. Ungreedy version *?, +? and ?? are also provided; the difference in the compilation scheme is just a matter of order in alternative patterns. For instance, [_* (x & Int) _*] is compiled to p where p = (_,p) | (x & Int, _) whereas [_*? (x & Int) _*] is compiled to p where p = (x & Int, _) | (_,p).

It is often useful to bind (or ignore) the tail of the matched sequence; instead of [... r::(_*)] (resp. [... (_*)]), one can use the notation [...; r] (resp. [...; _]).

String patterns The previous paragraph introduced regular expression patterns that match sequences and in which variables may capture subsequences. ℂDuce has also regular expression patterns for strings, with the syntax /*pregexp*/ where *pregexp* is a regular expression built from string constants, wild-cards ., character classes, usual regular expression operators, and *substring capture variables*. For instance, the pattern /y::(....) "-" m::(..) "-" d::(..)/ could be used to extract relevant numeric information from a string representation of a date; if the matched value is known to be of type /['0'-'9']{4} "-"['0'-'9']{2} "-" ['0'-'9']{2}/, then ℂDuce type-checker can infer that the value bound to y has type /['0'-'9']{4}/, and this can be used to check statically that a call to a function int_of_string on y will succeed.

```
transform e with (x & Int) -> [x x];;
(* Select and duplicate any integer *)

transform e with
 | (x & Int) -> [(to_string x)]
 | (x & String) -> [x];;
(* Select only strings and integers
   (transformed to strings) *)
```

Actually, transform can be defined from map and the flatten unary operator, that concatenate a sequence of sequences.

Our transform construction is very similar to the for construction in [7]; a loop for x in e_1 do e_2 would be simply translated to transform e_1 with x -> e_2.

3.7 Extra support for sequences

Although there is no special support for sequences in the core type and pattern algebras (regular expression types and patterns are just syntactic sugar), ℂDuce provides some language constructions to support them.

Map A common operation on sequences is to apply some transformation to each element. In ML, this kind of operation may be defined as an higher-order function map, taking as arguments a list and a function that operates on the elements of the list; the type of map is $\forall\alpha,\beta.(\alpha \to \beta) \times \alpha$ list $\to \beta$ list. In ℂDuce, one can define for instance:

```
let fun int_str
((Int -> String, [Int*]) -> [String*])
| (f,(x,l)) -> (f x, int_str (f,l))
| _ -> ‘nil;;
```

However, there are two drawbacks: (i) we can only define a monomorphic instantiation of the ML counterpart, hence code duplication; and (ii) the application of such a function is more verbose, because the abstraction (local function) passed as first argument has to provide an explicit interface in ℂDuce (for instance, fun (Int -> Int) x -> x + 1 instead of the ML fun x -> x + 1).

Actually (ii) is related to the following issue: the complex type algebra of ℂDuce makes type inference probably infeasible in practice (inferred types may be completely unreadable); moreover, as the semantics is driven by types, there is not necessarily a notion of *best* type for an expression. However, we are planning to consider limited form of inference, such as local inference, to handle simple cases as above.

To address (i), one can try to add some form of parametric polymorphism to ℂDuce; we already started to consider the problem and adapted ℂDuce type algebra to handle type variables. But it turns out that ML-like polymorphism is not always sufficient for XML-oriented processing and ℂDuce like type algebra. The function fun x -> x + 1 has of course type Int->Int, but also all the types i..j -> $i+1$..$j+1$ for any integers i and j. As a different example, suppose we have a sequence of type [$(t_1\ t_2)$*] and want to obtain a

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

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


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

Figure 1: A ℂDuce program

sequence of type [(t'_1 t'_2)*] (say, t_i and t'_i correspond to XML elements), by applying two distinct transformations for elements of type t_1 and those of type t_2. This is beyond the power of ML polymorphism; we could instantiate in the type of map α with t_1 | t_2 and β with t'_1 | t'_2, and this would give [(t'_1 | t'_2)*] for the type of the result, which forgets the order of elements.

As a pragmatic solution, we adopted the following construction in ℂDuce: map e with p_1 -> e_1 | ... | p_n -> e_n. Here, the expression e must evaluate to a sequence, and each of its elements will go through the pattern matching and get transformed if matched by some branch (otherwise, it is left unchanged, as if there were an implicit x -> x branch at the end of the pattern matching). Here is an example where this kind of polymorphism is required: the following function uses two different tags to represent home addresses and work addresses (default is “work”):

```
type AddrBook1 = <book>[(Name Addr1 Tel?)*];;
type Addr1 = <home>[String] | <work>[String];;

let fun patch_addr (AddrBook -> AddrBook1)
 <book>x -> let y = map x with
              | <addr kind="home">z -> <home>z
              | <addr>z -> <work>z
            in <book>y;;
```

Transform The map construction does not affect the length of the sequence, and each element is mapped to a single element in the result. ℂDuce also provides a variant of map, written transform, where each branch of the pattern is supposed to return a (possibly empty) sequence, and all the returned sequences, for each element in the source sequence, are concatenated together. The implicit default branch is now _ -> [] (so, unhandled elements are discarded). Here are some examples:

3.8 A detailed example

In Figure 1 we have written a small program that illustrates the expressivity and the peculiar characteristics of ℂDuce. Let us go throught it in detail.

We have XML-documents of type Person that are used to store information of persons. To that end they use an attribute gender, an element name, and an element children that is a sequence of Person elements. This is expressed by the first five type declarations of Figure 1. A completely equivalent definition for Person would have been:

```
type Person = <person gender=("F"|"M")>[Name Children]
```

Imagine now that we want to transform our representation so to get rid of the attribute by specifying either the tag <man> or the tag <woman>. We also want to distinguish the children of persons into two different sequences, one of sons, composed by men, and the other of daughters, composed by women. Of course we want to recursively apply this transformation to the children of a person. In practice, we want to define a function sort of type Person ->(Man | Woman) where Man and Woman are the types defined in the last four type declarations of Figure 1. The definition of the transformation ends the figure. It is declared to be an overloaded function that when applied to a MPerson returns an XML document of type Man and that applied to a FPerson it returns a value of type Woman. The body is composed by a single pattern matching whose pattern binds four variables: gen that is bound to the gender of the argument of the function, n that is bound to its the name, mc that is bound to the sequence of all children that are of type MPerson, and fc that is bound to the sequence of all children that are of type FPerson. Here we see the first use of a peculiar feature of ℂDuce namely the ability of patterns to capture subsequences of non consecutive elements of a sequence. On the next line we define tag to be `man or `woman according to the value of gen, and then we recursively apply sort to the elements of mc and fc. And here it is the crucial use of a second feature of ℂDuce, that is overloading: since mc is of type [MPerson*] then by the overloaded type of sort we can deduce that s is of type [Man*]; similarly we deduce for d the type [Woman*]. From this the type checker deduces that the expressions <sons>s and <daughters>d are of type Sons and Daughters, and therefore it returns for the sort function the type (MPerson -> Man) & (FPerson -> Woman). Note that the use of overloading here is critical: although sort has type Person ->(Man | Woman) (as sort is of type MPerson->Man & FPerson->Woman which is a subtype of the former) had we declared `sort` of that type the function would not have type checked: in the recursive calls the type we would have been able to deduce for `s` and for `d` is `[ (Man | Woman)* ]`, which is not enough to type check the result.

Another point that is worth to be stressed is that if for example we had defined the `Children` type as follows

```
type Children = <children>[Person+]
```

with the intention of asserting that all the persons in the base have at least a children, then the type checker emits a Warning stating that the types `Person`, `FPerson`, `MPerson`, `Children` are empty. As a matter of facts there does not exist any *value* with these type, for the simple reason that there is no value to start from (we are in a purely function language).[4]

4 Types

The type system is at core of ℂDuce. The whole language was conceived and designed on it. From a practical point of view the most interesting and useful characteristic of the type system is the semantic interpretation we described before, according to which a type is nothing but a set of values denoted by some syntactic expression[5]. This simple intuition is all is needed to grasp the semantics of the ℂDuce's type system and, in particular, of:

Subtyping: subtyping is simply defined as inclusion of sets of values: a type t is a subtype of s if and only if every value which has type t has also type s; when this does not hold, the type system can always exhibit a value of type t and not of type s.

Boolean connectives: boolean connectives in the type algebra are simply interpreted as their set-theoretic counterpart on sets of values: intersection `&`, union `|`, and difference `\` are the usual set theoretic operations.

Type equivalences: two types are equivalent if and only if all the values in the former are values in the latter and vice-versa. So for example `[ Int (String Int)* ]` $\simeq$ `[ (Int String)* Int ]`.

Understanding types is fundamental to ℂDuce programming as they are pervasive. In particular, pattern matching is based on types: all a pattern can do is to capture a value, deconstruct it, or check its type. So pattern matching can be basically seen as dynamic dispatch on types, combined with information extraction. This gives to ℂDuce a type-driven semantics reminiscent of object-oriented languages as overloaded functions can mimic dynamic dispatch on method invocations. Note however that a class based approach (mapping each XML element type to a class) would be infeasible, as the standard dispatch mechanism in OO-languages is much less powerful than pattern matching (which can look for and extract information deep inside the value). By keeping "methods" outside objects, we also get the equivalent of multi-methods (dispatch on the type of all the arguments, not just on the type of a distinguished "self").

Besides this dynamic function, types play also a major role in the static counterpart of the language. Type correctness of all ℂDuce transformations can be statically ensured. This is an important point: although many type systems have been proposed for XML documents (DTD, XML-Schema, ...), most XML applications are still written in languages (e.g. XSLT) that, unlike XDuce or ℂDuce, cannot ensure that a program will only produce XML documents of the expected type. Furthermore, in ℂDuce pattern matching has *exact* type inference, in the sense that the typing algorithm assigns to each capture variable exactly the set of all values it may capture. This yields a very precise static type system, that provides a better description of the dynamic behavior of programs.

Finally, types play an important role also in the compiler back-end, as the type-driven computation raises interesting issues about the execution model of ℂDuce and opens the door to type-aware compilation schemas and type-driven optimizations that we hint at in Section 6.

4.1 Highlights of the type system

Since the whole intuition of the ℂDuce type system relies on interpreting types as set of values, it is important to explain how values are typed. This is straightforward in most cases apart from function values. So we explain below the typing rule for functions and, in order to ease the presentation we split it in two rules, a subrule for typing function bodies (that is list of pattern matching branches) whose derivation is then used in the typing rule for functions.

A typing judgment has the form $\Gamma \vdash e : t$ where Γ is a typing environment (a map from variables to types), e is a ℂDuce expression, and t is a type; the intended meaning is that if the free variables of e are assigned values that respect Γ, then every possible result of e will be a value in t.

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

[4] While there does not exist any *value* in this type, there do exist *expressions* with this types. Of course they are all diverging. For example consider the mutually recursive functions:

```
let mkP( _->Person)
   x -> <person gender='F'>["Infinite Loop" mkC(x)]
and mkC( _->Children)
   x -> <children>[ mkP(x) ];;
```

Then `mkC(1)` is an expression of the type `Children` above.

[5] Of course, every type system induces a set-theoretic interpretation of types as sets of values. The point is that ℂDuce type system is *built* on such an interpretation. As a result, for instance, the subtyping relation of ℂDuce is both sound and complete w.r.t. set-inclusion, whereas in other type systems just soundness holds (if a type t is a subtype of s then all the values in t are also in s, but the converse does not hold).

succeeds and every possible result is of type s.

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

Let us look at this rule in detail. The matched value is supposed to be of type t. The first line checks that the pattern matching is exhaustive; for each pattern p_i, $\langle p_i \rangle$ is a type that represents exactly *all* the values that are matched by p_i. The exhaustivity condition is just saying that every value that belongs to type t must be accepted by some pattern.

Now we have to type-check each branch. At runtime, when the branch p_i->e_i is considered, one already knows that the value has been rejected by all the previous patterns $p_1, \ldots, p_{i-1}$; if the branch succeeds, one also knows that the value is of type $\langle p_i \rangle$. So, when type-checking the expression of the branch, one knows that the value will be of type t_i, that is, of type t and of type $\langle p_i \rangle$ but not of any of the types $\langle p_1 \rangle, \ldots, \langle p_{i-1} \rangle$. Now we type-check the body e_i of the branch; to do so, one must collect some type information about the variables bound by p_i. This is the purpose of (t_i/p_i): it is a typing environment that associates to each variable x in p_i a type that collects all the values that can be bound to x by matching some value of type t_i against p_i.

It is evident that all the "magic" of type inference resides in the operators $\langle p \rangle$ and (t/p). These operators were introduced in [8]. Their definition reflects their intuitive semantics and is also used to derive the algorithms that compute them. In the next section examples are given to illustrate some non-trivial computations performed by these algorithms.

The result of the pattern matching will be the result of one of the branches that can potentially be used. This is expressed by taking for the type of the pattern matching the union of the result type of each branch i such that t_i is not empty; indeed, if t_i is empty, the branch cannot be selected, and we take $s_i =$ Empty as its contribution.

Functions What unused branches (i.e., those with $t_i \simeq$ Empty) are useful for in a pattern matching? The answer is in the typing rule for abstractions:

$$\frac{\begin{array}{c} t = t_1\texttt{->}s_1 \;\texttt{\&}\; \ldots \;\texttt{\&}\; t_n\texttt{->}s_n \\ (\forall i)\;\; \Gamma, f{:}t \vdash t_i/B \Rightarrow u_i \leq s_i \end{array}}{\Gamma \vdash \texttt{fun}\;\; f(t_1\texttt{->}s_1\texttt{;}\ldots\texttt{;}t_n\texttt{->}s_n)B : t}$$

The type system simply checks all the constraints given in the interface (as the function can call itself recursively, we remember when typing the body that f is a function of the type given by the interface). So the body is type-checked several times, and for some type t_i, it may be the case that some branch in B is not used. Let us illustrate this by a simple example:

```
fun (Int -> Int; String -> String)
  | Int -> 42
  | (x & String) -> x
```

When type-checking the body for the constraint `String -> String`, the first branch is not used, and even though its return type is not empty (it is `42`, which is the type assigned to the constant `42`), it must not be taken into account to prove the constraint.

This is not a minor point: the fact of not considering the return type of unused branches is the main difference between dynamic overloading and a type-case (or equivalently the `dynamic` types of [1]). The latter always returns the union of the result types of all the branches and, as such, it is not able to discriminate on different input types.

4.2 Pattern type inference: examples

We saw that $\langle p \rangle$ and (t/p) are the core of the type system. They are defined as the least solution of some set of equations (there may be several solutions when considering recursive patterns). These definitions are quite straightforward as they reflect the intuitive semantics of the operators. For example, $\langle p \rangle$ is defined by the following set of equations

$$\begin{array}{lcllcl} \langle x \rangle & = & \texttt{Any} & \langle p_1 \,|\, p_2 \rangle & = & \langle p_1 \rangle \mid \langle p_2 \rangle \\ \langle t \rangle & = & t & \langle p_1 \& p_2 \rangle & = & \langle p_1 \rangle \,\&\, \langle p_2 \rangle \\ \langle (x\texttt{:=}c) \rangle & = & \texttt{Any} & \langle (p_1\texttt{,}p_2) \rangle & = & (\langle p_1 \rangle\texttt{,}\langle p_2 \rangle) \end{array}$$

which simply states that a pattern formed by a variable matches (the type formed by) all values, that a pattern type matches all the values it contains, that an alternative pattern matches the union of the types matched by each pattern and so on. The same intuition guides the definition of (t/p). So for example:

$$\begin{array}{lcl} (t/x)(x) & = & t \\ (t/(p_1 \,|\, p_2))(x) & = & ((t\& \langle p_1 \rangle)/p_1)(x) \mid ((t\backslash \langle p_1 \rangle)/p_2)(x) \\ & \vdots & \end{array}$$

states that when we match the pattern x against values ranging over the type t then the values captured by x will be exactly those in t, similarly when we match values ranging over t against an alternative pattern, then the values captured by a variable x will be those captured by x when the first pattern is matched against those values of t that are accepted by p_1, union those captured by x when the second pattern is matched against the values in t that are accepted by p_2 but not by p_1.

The most important result is that the equations above can be used to define two algorithms that compute $\langle p \rangle$ and (t/p). Rather than entering in the details of the algorithms[6], we prefer to give a couple of examples that show the subtlety of the computation they are required to perform.

Filter Consider the pattern `p where ((x & Int),p) | (_,p) | (x:=‘nil)` that extracts from a sequence all the integers occurring in it [7]. In the table below we show the types of all values that are captured by the variable x of the pattern

[6] See [8] for more details about these algorithms as well as the algorithms of typing and subtyping. Here we just want to signal that with our semantic approach the proofs of completeness and termination for the algorithms seems considerably easier than those of XDuce's ones.

[7] More precisely, if p is matched against a sequence L, then x is bound to the subsequence of L containing all the integers in L

p when this latter is matched against (values ranging over) different types:

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

Pairs Consider the following types already introduced in Section 3.3

```
type IL = [Int*];;                    integer list
type NE = [Int+];;                    not empty list
type T = (IL,IL)\([],[]);;            pair of lists not both empty
```

and the patterns $p =$ ((x,y),z) and $q =$ ([],z). Then:

$$((\mathtt{T}\&\llparenthesis p \rrparenthesis)/p) \;=\; [\mathtt{x} \mapsto \mathtt{Int}\ ,\ \mathtt{y} \mapsto \mathtt{IL}\ ,\ \mathtt{z} \mapsto \mathtt{IL}]$$
$$((\mathtt{T}\backslash\llparenthesis p \rrparenthesis)/q) \;=\; [\mathtt{z} \mapsto \mathtt{NE}]$$

The typing of patterns, pattern matching, and functions is essentially all is needed to understand how the type algorithm works, as the remaining rules are straightforward. The only exception to that are the typing of the constructions map and transform which need to compute the transformations of regular expressions (over types) and for which the same techniques as those of [7] can be used.

5 Queries

In the world of XML, the boundary between programming languages, transformation languages, and query languages/algebras is not easy to draw and as pointed out in [12] there is no definitive standard for query languages for XML. Indeed, a query can be seen as a transformation that filters the XML documents to extract the relevant information and presents it with a given structure, and a transformation is just a special kind of application. A declarative language such as XSLT [4] is clearly not on the "programming" side, but systems such as XQuery [2] or the one in [7] are very close in spirit to XDuce, and they can be seen as real programming languages for XML.

ℂDuce was designed as a programming language, recasting some XML specific features from XDuce in the more general setting of higher-order functional languages. But it turns out that a small set of extra constructions can also endow it with query-like facilities that are standard in the database world: projection, selection, and join.[8]

We already mentioned that the transform construction allows us to encode the for iteration from [7] in ℂDuce.

As in [7], the projection operator—denoted by /— can be defined from this construction: if e is a ℂDuce expression representing a sequence of elements and t is a type, e/t is syntactic sugar for[9]:

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

This new syntax can be used to obtain a notation close to XPath [5]. For example consider the type AddrBook of Section 3.2 modified so that the <addr> elements contain subelements such as <street>, <town> and so on. If addrbook is of type AddrBook, then the expression [addrbook]/<addr kind="home">_/<town>_ extracts from addrbook the sequence of all town elements that occur in a "home" address. To enhance readability we use the syntactic convention that in such paths the wild-cards _ that follow tags can be omitted and write [addrbook]/<addr kind="home">/<town>. This corresponds to the XPath expression /addr[@kind="home"]/town. Our type system allows us to push further the simulation of XPath, for example to consider the position of the elements. So we may imagine to write [addrbook]/<addr kind="home">{2}/<town> to select exactly the town of the second "home" address element of addrbook which would correspond to the XPath expressions /addr[2][@kind="home"]/town and coded into the following (unreadable but efficient) expression:

```
transform addrbook with
  <_>[(_*?) <addr kind="home">_ (_*?)
       <addr kind="home">[ (x::<town> | _)* ]; _
     ] -> x
```

But ℂDuce path expressions are not just there to mimic XPath. Since our syntax allows us to specify the type of element contents, we can use this option to express complex conditions in paths; for instance, using the types defined in Section 2, the path

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

extracts the titles of all the books with exactly two authors.

A ℂDuce compiler could take profit of equivalences similar to those mentioned in [7] to optimize complex queries; a typical example of optimization rule is: transform (transform e with p1 -> e1) with p2 -> e2 $\rightsquigarrow$ transform e with p1 -> transform e1 with p2 -> e2 . Considering query-like features in a programming language leads to introducing a query optimizer in the code generator.

To implement joins, we introduce a cartesian product operator in ℂDuce; if $s_1, s_2, \ldots, s_n$ are sequences, then prod(s_1,s_2,...,s_n) evaluates to a sequence containing all the (v_1,v_2,...,v_n) where v_i appears in s_i. We let the order of the resulting sequence unspecified so to allow optimizations. For instance, if s_1 = [1 2] and s_2 = ["A" "B"], then the expression prod(s_1,s_2) evaluates to some permutation of [(1,"A") (1,"B") (2,"A") (2,"B")]. A typical join creates the cartesian product of sequence of XML documents,

[8] The fact that ℂDuce can implement such constructions is not surprising: any Turing-complete language can obviously do it. The point is that, instead of defining a fixed implementation of these constructions, one can use the semantic foundations of ℂDuce to obtain different implementations and natural (insofar as semantic) transformations that open the door to query optimization.

[9] We can take advantage of the fact that in ℂDuce a single pattern can perform the complex operation of extracting all the elements of a given type, to define the following more compact encoding:

```
transform e with <_>[ (x::t | _)* ] -> x
```

and filters it (with `transform` or a pattern). The compiler is free to apply algebraic optimizations or use available indexes to implement the join efficiently.

The typing rule for `prod` is the following:

$$\frac{\Gamma \vdash e_1 : [t_1 *] \quad \dots \quad \Gamma \vdash e_n : [t_n *]}{\Gamma \vdash \mathtt{prod}(e_1, \dots, e_n) \;:\; [(t_1, \dots, t_n) *]}$$

Note that we restrict the e_i's to be homogeneous sequences (i.e, sequences whose elements are all of the same type) which yields the product to be an homogeneous sequence, as well. A `select` construction can then be defined easily. The meaning of `select` e `from` x_1 `in` $e_1, \dots, x_n$ `in` e_n `where` e' is defined to be the same as: `transform prod(`$e_1, \dots, e_n$`)` `with (`$x_1, \dots, x_n$`) -> if` e' `then` e `else []`. The compiler can implement this more efficiently; for instance, if e' does not involve all the x_i, the query optimizer can, as usual, push selections (and/or projections) on some components before creating the full cartesian product. The following example, inspired from [7], illustrates the join between two documents `bib0` and `rev0` of types `Bib` and `Review` respectively.

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

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


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

yielding the following result.

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

Sometimes we need to access to the content of an element when this is a sequence of just one element. This is done by the expression e`.<tag>` as in the following example where we add the review as an attribute of the book tag:

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

The type system authorizes to access the content of an element only if the element occurs exactly once and it contains a sequence of length 1, as stated by the following typing rule (where $\neg t$ is by definition `Any`$\setminus t$)

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

(r is an optional attribute specification) that can be easily deduced from the encoding of e`.<t` r`>`:

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

This construct corresponds to the XSLT element value-of where, say, `(/<a>/<b>/<c>).<c>` would be written as `<xsl:value-of select="/a/b/c">`.

On the lines of what precedes we could easily show how to use ℂDuce to encode the various examples presented in the current W3C proposal for XML Query and, in general, obtain a more precise typing. However the point is not there, as we do not want to fix a precise implementation for queries. On the contrary, in this section we wanted to single out constructions that left the compiler with maximal freedom in the implementation and, therefore, a large latitude in query optimization. In other terms, we sought for constructions that allowed us to express queries as most "declarative" as possible, especially because the set-theoretic semantic foundations of ℂDuce constitutes an adequate support for defining optimizations. In the frame of the ℂDuce type system, the `prod(...)` construction above looks like a promising choice.

6 Implementation

We described the static type checking of ℂDuce programs in [8]. In particular, we defined a high-level specification of a subtyping algorithm by a notion of coinductive simulation that characterizes empty types (t is a subtype of s if the difference type $t \setminus s$ is empty). Many optimizations and non-trivial implementation techniques can be applied to pass from this specification to a practical algorithm: among these are those mentioned in [11] (for the knowledgeable reader, note that it is also possible to cache negative results of the subtyping algorithm in a destructive structure, as they cannot be invalidated even under different assumptions) and the definition of maximal sharing and unique representation for recursive types which improve the impact of caching mechanisms.

In ℂDuce type checking is not just a preliminary verification. We believe that static typing is a key information for designing an efficient execution model for ℂDuce and XML languages in general. To motivate this idea let us consider the following example. Suppose that `A` and `B` are two types and consider the function:

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

A naive compilation schema would yield the following behavior for the function. First check whether the first pattern matches the argument. To do this: (i) check that it has the form `(‘a,(_,`l`))` and (ii) run through l to verify that it is a non-empty sequence of elements of type `A` (checking that an element is of type `A` may be complex, if `A` represents for instance a complex DTD). If this fails, try the second branch and do all these tests again with `B`. The argument may be run through completely several times.

There are many useless tests; first, one knows statically that the argument is necessarily a pair with first component `‘a`:

there is no need to check this. Then, one knows that the second is a pair `(_,`l`)` where l is a non-empty sequence whose elements are either all of type `A` or all of type `B`. To know in which situation we are, one just has to look at the first element and perform some tests to discriminate between `A` and `B` (for instance, if `A = <x>[...]` and `B = <y>[...]`, it is enough to look at the head tag). Using these optimizations, only a small part of the argument is looked at (and just once).

In general, a naive approach to compiling pattern matching may yield multiple runs and backtracking through the matched value. Forgetting for a moment functions and records, values can be seen as binary trees, and types simply represent regular tree languages. It is a well-known fact that such tree languages may be recognized by deterministic bottom-up tree automata; this indicates that backtracking can be eliminated. It is possible to adapt the theory of tree automata to handle the full range of ℂDuce patterns (with capture variables) and values. However, determinization may create huge and intractable automata (number of states and transition function); this is due to the fact that such automata perform a uniform computation, disregarding the current position in the tree. When matching a pair (v_1, v_2), one can choose to perform different computations on v_1 and on v_2 (each with a smaller automaton), but classical bottom-up tree automata do not have this flexibility. Also, we want to take into account static type information about the matched value to help pattern matching. This can be combined with the previous remark: when matching (v_1, v_2), one can start for example with v_1; according to the result of this computation, we get more information about the (dynamic) type of the matched value and this information can simplify the work on v_2: for instance, if we statically know that (v_1, v_2) has type (t_1, t_2) `|` (s_1, s_2) with non-intersecting types t_1 and s_1, and if the computation on v_1 tells us that v_1 has type t_1, then we know that v_2 has necessarily the type t_2, and no further check is required. By using static type information, it is thus possible not only to avoid backtracking, but also to avoid checking whole parts of the matched value. This is particularly useful when working with tag-coupled document types (as DTD types) where the tag of an XML element already provides a lot of information about its content.

Let us give another example, where we use atoms to simulate ML datatype constructors:

```
type Ex =
  (`number,Int)
| (`plus,(Ex,Ex))
| (`mul,(Ex,Ex));;

let fun eval(Ex->Int)
  (`number,(n & Int)) -> n
| (`plus,(x & Ex,y & Ex)) -> (eval x) + (eval y)
| (`mul,(x & Ex,y & Ex)) -> (eval x) * (eval y);;
```

The type constraints `& Ex` and `& Int` are useless here, but the programmer may want to specify them. A naive implementation would check that the "constructors arguments" are of the correct type (and this requires to look at the whole subexpressions), even though this is guaranteed by the static type of the function argument. An efficient implementation would simply look at the tag, dispatch according to its value, and then assume correct types for the arguments.

A third example is given by the selection expression e`.<t `r`>` introduced in the previous section that would be rather implemented by `match` e `with [(¬<t `r`>_)* <t `r`>[x] ;_] -> x` as the static type-checking already ensures that `<t>` does not occur in the rest of the sequence.

Even more, combining static information and query optimization techniques such as rewriting, the expression

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

mentioned in the previous Section could be translated into:

```
match bib with <_>[ (<_>[(x::_) _ _ _] | _)* ] -> x
```

where tests are reduced to the essential.

An efficient dispatch algorithm Let us sketch an efficient dispatch compilation schema; we will just present some ideas, as the full formalization is outside the scope of this paper. Given n *disjoint* types $t_1,\ldots,t_n$, and a value v belonging to their union to the problem is to decide which t_i the value belongs to. For instance, if the program has to test at runtime whether a value belongs to a type t, and the type-checker proves that the value is necessarily of type s, then the compiler can just produce code that decides between the types $t_1 = s\&t$ and $t_2 = s\backslash t$. If either t_1 or t_2 is empty, then there is no code to produce at all. More generally, a possible implementation of pattern matching could first determine which branch to choose, by making a choice among the types t_i defined as in the typing rule of the pattern matching ($t_i = t\backslash \langle\!| p_1 |\!\rangle \ldots \backslash \langle\!| p_{i-1} |\!\rangle \& \langle\!| p_i |\!\rangle$).

The algorithm we outline has the important property to be semantic, in the sense that its result does not change if the t_i are replaced by equivalent types (two types are equivalent if they denote the same set of values). As a consequence, for instance, there is no need to apply algebraic rewriting rules to syntactically simplify the types t_i before applying the algorithm.

The compilation schema consists in descending deep in the value, starting from the root, and accumulate enough information to stop the process as soon as possible. For instance, if the value is a pair (v_1, v_2), the idea is to generate a new set of disjoint types $s_1,\ldots,s_m$ and use v_1 to make a choice among them; according to the result i (meaning "v_1 is in s_i"), a new set of types $u^i_1,\ldots,u^i_{l_i}$ is examined and v_2 is used to single out a new result j that will be enough to determine the t_i the pair (v_1, v_2) belongs to.

All the types s_i, u^i_j are computed at compile time from $t_1,\ldots,t_n$.[10] The choice of the s_i's must be done so that (i) any possible value v_1 will belong to one s_i, and (ii) the information extracted from v_1 during the selection of the s_i, is enough to avoid backtracking to v_1 (that is, the knowledge of v_2 and of the i such that v_1 is in s_i must be enough to decide the t_j the value (v_1, v_2) belongs to).

For example, consider $t_1 =$ `((0..100),Int)` and $t_2 =$ `((50..150),String)`. A possible choice is $s_1 =$ `(0..49)`, $s_2 =$ `(50..100)`, and $s_3 =$ `(101..150)`. Given $v = (v_1, v_2)$,

[10] Actually, it is possible to take $l_i = n$ and arrange so that the dispatch on v_2 is a tail-recursive call (this means: the result j of dispatching among the u^i_j is also the result for dispatching among $t_1,\ldots,t_n$).

if v_1 is in s_1 (resp. s_3), then v is in t_1 (resp. t_2); if v_1 is in s_2, then we look at v_2 and check whether it is in $u_1^2 = \texttt{Int}$ or in $u_2^2 = \texttt{String}$.

The choice of the s_i's is not unique and must be done heuristically. Indeed, the capture of more information from v_1 than the strictly necessary may allow to cut down the exploration of v_2. For instance, suppose that $t_1 = (a_1, a_2)$, $t_2 = (b_1, b_2)$, that a_1 is disjoint from b_1 and a_2 is disjoint from b_2. In order to decide in which t_i a value (v_1, v_2) is, one can either look at v_1 (making the choice between a_1 and b_1) and ignore v_2, or look at v_2 (making the choice between a_2 and b_2) and ignore v_1: there is not a better choice in general. For the implementation of ℂDuce we chose to extract as much information from v_1 as possible (that is, any subdivision of a s_i would give the same set of possible values for v_2); our choice is based on the fact that in ℂDuce sequences are coded as right associating pairs and we do not want to run through a whole sequence when the first elements may be enough to conclude. Similarly for the encoding of XML elements we first descend on the left of a pair so that we examine in turn the tag, the attributes, and only at last the children. We do not detail here formally how to choose the s_i's, nevertheless it is important to signal that this can be done "semantically" so that the choice will be invariant to the replacement of the t_i's by equivalent types.

7 Other and future issues

We summarize here some open questions in the design of ℂDuce and sketch further research directions.

Polymorphism and inference Section 3.7 discusses lack of polymorphism and type inference; it suggests that ℂDuce could benefit from some kind of powerful polymorphism mechanism, such as higher-order types; of course, introducing such features would make everything more complicated. For instance, we would probably be obliged to get rid of simple semantic definitions for the typing of pattern matching, and use more or less ad hoc approximations.

The current approach is to add specific constructions such as `map` or `transform` to handle typical cases where polymorphism is needed; a possible direction is to allow the programmer to define new constructions (syntax, typing rules, compilation schemes) in a meta language (ML for instance), and use some kind of plug-in mechanism to extend the ℂDuce core compiler.

Concrete interaction with XML The natural behavior of an XML transformation written in ℂDuce would be to parse the input XML document, validate it with a given ℂDuce type, run the transformation, and output a final XML document. However, we can consider improvements to this simple scenario:

- combining programs: frameworks such as Transmorpher [6] are proposed to combine several XML transformations together to form a complex application. Between two transformations, it may be useless to output and then re-parse immediately documents; instead, XML transformations engines (and ℂDuce could be considered as one of them) should be able to communicate directly by exchanging internal representation of documents. Moreover, if the output of a transformation is proven to have the expected type (and ℂDuce type system can enforce such a constraint), it is not necessary to validate the input;
- textual representation of XML is important for exchanging documents between several organizations, but it may be inadequate for some applications. For instance, in the setting of XML databases, we do not want to parse the full database for each query. A binary representation of XML would match current practice of non-XML databases; the representation should be optimized to take into account the structure of documents, and types will of course be important in this setting. For instance, if the *content model* of an element is completely fixed by the type, compact and efficient storage as in relational databases could be used.
- a specific ℂDuce program, such as a query, may not need to look at the whole input document (database); it may be interesting to design communication protocols between XML programs (the ℂDuce interpreter or programs) and (binary) XML storage managers that allow to extract only needed part of documents.

Missing XML features XML recommendation specifies a way to refer from an element to another element with ID, IDREF and IDREFS attributes. Currently, there is no specific support in ℂDuce for these. We are planning to add (mutable) reference types to ℂDuce to reflect the pointer structure defined by ID and IDREF inside a document. This feature would be close to ML references, thus allowing the definition of complex and spurious data structures (such as graphs).

We are also planning to add support for namespaces (using pairs instead of simple atoms to represent element tags).

References

[1] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Scott Boag, Don Chamberlin, Mary Fernandez, Daniela Florescu, Jonathan Robie, Jérôme Siméon, and Mugur Stefanescu. *XQuery 1.0: An XML Query Language*. W3C Recommendation, `http://www.w3.org/TR/xslt`, November 1999.

[3] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[4] James Clark. *XSL Transformations (XSLT)*. W3C Recommendation, `http://www.w3.org/TR/xslt`, November 1999.

[5] James Clark and Steve DeRose. *XML Path Language (XPath)*. W3C Recommendation, `http://www.w3.org/TR/xpath`, November 1999.

[6] Jérôme Euzenat and Laurent Tardif. XML transformation flow processing. In *2nd conference on Extreme markup languages*, pages 61–72, 2001. Available at `http://transmorpher.inrialpes.fr/wpaper/`.

[7] Mary Fernández, Jérôme Simeon, and Philip Wadler. An algebra for XML query. In *Foundations of Software Technology and Theoretical Computer Science*, number 1974 in Lecture Notes in Computer Science, pages 11–45, 2000.

[8] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[9] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, 2000.

[10] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.

[11] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, volume 35(9) of *SIGPLAN Notices*, 2000.

[12] V. Vianu. A web odissey: from Codd to XML. In *Proc. of International Conference on Principles of Database Systems (PODS '01)*, pages 1–15. ACM Press, 2001.