ℂDuce: An XML-Centric General-Purpose Language

Véronique Benzaken
LRI, (CNRS)
Université Paris-Sud
91405 Orsay, France
Veronique.Benzaken@lri.fr

Giuseppe Castagna
CNRS, Département d'Informatique
École Normale Supérieure
45 rue d'Ulm, Paris, France
Giuseppe.Castagna@ens.fr

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

Abstract. *We present the functional language ℂDuce, discuss some design issues, and show its adequacy for working with XML documents. Distinctive 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 for patterns and error localization, and a natural interpretation of types as sets of values. We also outline some important implementation issues; in particular, a dispatch algorithm that demonstrates how static type information can be used to obtain very efficient compilation schemas.*

Categories and Subject Descriptors: D.3.0 [**Programming Languages**]: General; D.3.2 [**Programming Languages**]: Language Classifications—*Applicative (functional) languages*

General Terms: Languages

Keywords: XML, XML-processing, type systems, CDuce

1. Introduction

ℂDuce is a general purpose typed functional programming language, whose design is targeted to XML applications. The work on ℂDuce started two years ago from an attempt to overtake some limitations of XDuce [11] following three directions:

- *Type system.* XDuce demonstrates the adequacy of some specific features (regular expression types and type-based patterns) to XML applications, but we believe that these features could be integrated in a less specific language. Indeed, as the interface between a XDuce-like language and a mainstream language necessarily looses most of the type information, we aim at minimizing the interactions between ℂDuce and external languages by allowing to define complex applications directly in ℂDuce.

  To this end, we extended XDuce type system by introducing less XML specific type constructions: products, records, general Boolean connectives (union, intersection, difference), and arrow types (first-class functions), continuing and prolongating the semantic approach to define subtyping that was initiated by XDuce. On a practical side, we implemented a type-checker that gives precise localization of error messages and exhibits "samples" to demonstrate type checking failure.

- *Language design.* We added language constructions that we believe to be useful for XML or general purpose processing such as overloaded functions (to allow code sharing and code reuse), iterators on sequences and trees and several extensions of the pattern algebra (in particular to allow extraction of non-consecutive subsequences). We studied precise typing for these constructions.

  We also made several small design decisions that turn out to be very practical. For instance, XML tags are ℂDuce first-class expressions (which allows *computing* on tags), and strings are nothing but sequences of characters (this allows the use of regular expression types and patterns on strings; also, when concatenating two sequences containing characters and XML elements, the two strings at the "boundary" are automatically concatenated).

- *Run-time system.* We tackled the problem of executing ℂDuce programs efficiently. The key issue is the implementation of pattern matching. To this end, we use a new kind of deterministic tree automata which is a combination of top-down and bottom-up automata, and we developed a compilation schema (from patterns to automata) that uses static type information to avoid unnecessary computation at runtime. This allows the programmer to use a much more declarative style in patterns without degrading performances.

This article focuses on language design, shows its adequacy to write applications that handle, transform, and query XML documents, and sketches solutions to implementation issues. To keep the presentation short, we just present some highlights of the language. The homepage for ℂDuce, `http://www.cduce.org/`, includes other references, an online interactive prototype (where the reader can test ℂDuce and check all the examples presented here), a user's manual, many ℂDuce programs (larger and more compelling than the ones presented here), and the ℂDuce distribution (the whole site is generated by a 330-loc ℂDuce program that transforms the XML content description into XHTML: the complete source code is included in the ℂDuce distribution). Theoretical foundations of the ℂDuce's type system can be found in [8].

Related Work

The closest work to ours is, of course, XDuce from which ℂDuce borrows many key features, such as, among others, regular expression types, type-based pattern matching, the semantic inclusion of XML types, the type inference for patterns, the use of recursive patterns. The previous section mentions in which directions the work on ℂDuce extends XDuce.

Since the core definition of ℂDuce was published in [8] both languages evolved in parallel. Independently from us, Haruo Hosoya implemented in XDuce the support for exact typing of non-tail variables, by using a different approach based on automata [10]. Together with Makoto Murata [13] he also defined and extended XDuce types to handle XML attributes in a quite different way from the one we defined for ℂDuce (i.e., extensible records). We believe that the expressive power of the two solutions is comparable, but Hosoya and Murata's solution is able to express compactly dependencies between attributes and elements (while we have to expand those dependencies using union types), and they addressed algorith-

ICFP'03, August 25–29, 2003, Uppsala, Sweden.

mic issues to avoid exponential explosion. However, their algorithm for evaluating a pattern on a sequence uses a first rewriting pass on the automaton, which requires fully materializing this automaton and does not seem compatible with our current efficient compilation algorithm (likewise, Hosoya had to remove pattern optimization from XDuce to incorporate the new attribute support). The ongoing work on XDuce has a constant influence on ours and some of our future plans include adapting to ℂDuce some of Hosoya's insightful ideas on sequence transformations.

ℂDuce is not the only project that started from or was deeply influenced by XDuce. Among the projects inspired by XDuce those closest to ℂDuce are Xtatic [9] and XQuery [3, 7].

The goals for Xtatic and ℂDuce are similar in that they both try to integrate XDuce features in a larger design and less XML-specific language. ℂDuce builds on the functional flavour of XDuce and extends it to a full-fledged functional language, whereas Xtatic goes toward OO principles and combines XDuce types with the class hierarchy of a host language, namely C#. The integration of XDuce and C# in Xtatic is smooth and elegant both on the language side (thanks to the introduction of a special Seq class, and a clever treatment of the concatenation) and on the implementation side (by resorting to an encoding technique reminiscent of Pizza's homogeneous translation [16]). Xtatic and ℂDuce both have to face the problem of combining XDuce's semantic definition of subtyping with a richer type algebra. Xtatic's solution is simpler as it relies on named typing for C# classes, where ℂDuce has to tackle classical issues when dealing with set-theoretic interpretation of arrow types. On a less theoretical side, ℂDuce and Xtatic share many design decisions, including syntactic choices and small decisions mentioned in the Introduction such as first-class XML tags and strings as sequences of characters. ℂDuce and Xtatic designs also have noticeable differences; among them is that in ℂDuce we avoid the stratification of the type algebra between XML types and non-XML types and this permits the use of pattern matching also for non-XML data structures (such as pairs or records). For what concerns implementation issues Levin studied efficient pattern matching implementation for Xtatic [15]. He concentrates on the definition of a generic kind of matching automata and target language suitable as back-ends for pattern matching. Our work is quite orthogonal, as we focus on optimization made possible by static type information. It should be possible to express our compilation algorithm in term of their matching automata. As for expressivity, our algorithm supports in addition ambiguous patterns (disambiguated with first match policy), non-linear capture variables (even when under repetition operators), and XML attributes (implemented as records in ℂDuce).

XQuery is mainly aimed at performing queries on XML documents. Since the XQuery type system took its inspiration from XDuce's one, it is not surprising to find a lot of similarities with ℂDuce. To perform queries XQuery adds to XDuce a for loop (which can be simulated by ℂDuce's transform iteration on sequences) as well as support for XPath, while it removes complex pattern matching with regular expression patterns and, recently, structural typing (replaced by named typing as in XML Schema [17], so as to avoid tree automata to check subtyping or validate documents). Supporting XPath's upward axis yields a copy semantics which is unusual in functional languages: when an element is created in XQuery, its content has to be copied, while in ℂDuce, an XML subtree can be shared by different documents.

The rapidly growing importance of XML has given rise to many other works related to XML processing. Due to lack of space, we decided to focus only on the projects closest to ℂDuce. For an overview of other works and a thorough comparison of these with the XDuce/ℂDuce approach we refer the reader to [11].

2. A sample session

Let us write and comment on a sample ℂDuce program. First, we declare some types[1]:

```
type ParentBook = <parentbook>[Person*]
type Person = FPerson | MPerson
type FPerson = <person gender="F">[ Name Children (Tel | Email)*]
type MPerson = <person gender="M">[ Name Children (Tel | Email)*]
type Name = <name>[ PCDATA ]
type Children = <children>[Person*]
type Tel = <tel kind=?"home"|"work">['0'--'9'+ '-'? '0'--'9'+]
type Echar = 'a'--'z' | 'A'--'Z' | '_' | '0'--'9'
type Email= <email>[ Echar+ ('.' Echar+)* '@' Echar+ ('.' Echar+)+ ]
```

The type ParentBook describes XML documents that store information of persons. A tag <tag attr1=...; attr2=...; ...> followed by a sequence type denotes an XML document type. Sequence types classify ordered lists of heterogeneous elements and they are denoted by square brackets [...] that enclose regular expressions over types (note that a regular expression over types *is not* a type, it just describes the content of a sequence type, therefore if it is not enclosed in square brackets it is meaningless). The definitions above state that a ParentBook element is formed by a possibly empty sequence of persons. A person is either of type FPerson or MPerson according to the value of the gender attribute. An equivalent definition for Person would thus be:

<person gender="F"|"M">[Name Children (Tel | Email)*].

A person element is composed of a sequence formed by a name element, a children element, and zero or more telephone and e-mail elements, in this order. Name elements contain strings. These are encoded as sequences of characters. The PCDATA keyword is equivalent to the regexp Char*, then String, [Char*], [PCDATA], [PCDATA* PCDATA], ..., are all equivalent notations. Children are composed of zero or more Person elements. Telephone elements have an optional (as indicated by =?) string attribute whose value is either "home" or "work" and they are formed by a single string of two non-empty sequences of numeric characters separated by an optional dash character. Had we wanted to state that a phone number is an integer with at least 5 digits (of course this is meaningful only if no phone number starts with 0) we would have used an interval type as in <tel kind=?"home"|"work">[10000--*], where * here denotes $+\infty$. Echar is the type of characters in e-mail addresses. It is used in the regular expression defining Email to precisely constrain the form of the addresses. An XML document satisfying these constraints is shown in the left column of Figure 1.

If the document is stored in the file `parents.xml`, it can be loaded with the built-in operator load_xml, assigned to a local variable parents, and immediately checked to be of the ParentBook type:

```
let parents  =  match (load_xml "parents.xml") with
                  |   (x & ParentBook) -> x
                  |   _ -> raise "Wrong type!"
```

When this declaration is entered interactively the system answers:

```
|- parents : ParentBook
```

which indicates that the type checker remembers that parents is of type ParentBook. To obtain it, the value resulting from the load operation is matched against the pattern x & ParentBook. The &-pattern denotes the simultaneous application of two sub-patterns and it succeeds if both sub-patterns do. In our example the value is matched against the variable x—which always succeeds and binds the value to x—and against the type ParentBook—which succeeds only if the value has the given type. If the &-pattern fails (i.e., if the result of the load is not of type ParentBook), then the value is matched against the pattern _ which always succeeds (_ denotes the type of all values,

[1] ℂDuce distribution includes dtd2cduce, a program that can be used to translate DTDs into ℂDuce's types. Support for XML Schema validation has been recently implemented and is in alpha testing (see § 6).

```
<?xml version="1.0"?>
<parentbook>
  <person gender="F">
    <name>Clara</name>
    <children>
      <person gender="M">
        <name>Pål André</name>
        <children/>
      </person>
    </children>
    <email>clara@lri.fr</email>
    <tel>314-1592654</tel>
  </person>
  <person gender="M">
    <name> Bob </name>
    <children>
      <person gender="F">
        <name>Alice</name>
        <children/>
      </person>
      <person gender="M">
        <name>Anne</name>
        <children>
          <person gender="M">
            <name>Charlie</name>
            <children/>
          </person>
        </children>
      </person>
    </children>
    <tel kind="work">271828</tel>
    <tel kind="home">66260</tel>
  </person>
</parentbook>
```

```
let parents : ParentBook =
<parentbook>[
  <person gender="F">[
    <name>"Clara"
    <children>[
      <person gender="M">[
        <name>['Pål ' 'André']
        <children>[]
      ]
    ]
    <email>['clara@lri.fr']
    <tel>"314-1592654"
  ]
  <person gender="M">[
    <name>"Bob"
    <children>[
      <person gender="F">[
        <name>"Alice"
        <children>[]
      ]
      <person gender="M">[
        <name>"Anne"
        <children>[
          <person gender="M">[
            <name>"Charlie"
            <children>[]
          ]
        ]
      ]
    ]
    <tel kind="work">"271828"
    <tel kind="home">"66260"
  ]
]
```

Figure 1: XML and ℂDuce

so every value matched against it succeeds) and raises an exception (this could be caught by a try...with... construction).

The right column in the figure above represents the binding of the variable parents to the same document written directly as a ℂDuce value. The type annotation (... : ParentBook) is optional, but, in general, it allows an earlier detection of type errors. Sequence values are denoted by the list of elements enclosed in square brackets and separated by blank spaces. For the purpose of the example we used different notations to denote strings since in ℂDuce "xyz", ['xyz'], ['x' 'y' 'z'], ['xy' 'z'], and ['x' 'yz'] define the same string literal (see §3.2). Note also that the "Pål André" string is accepted since ℂDuce supports Unicode characters.

A first example of transformation is names, which extracts the sequences of all names of parents in a ParentBook element:

```
let names (ParentBook -> [Name*])
  <parentbook>x -> (map x with <person>[ n _*] -> n)
```

The name of the transformation is followed by an *interface* that states that names is a function from ParentBook elements to (possibly empty) sequences of Name elements. This is obtained by matching the argument of the function against the pattern <parentbook> x which binds x to the sequence of person elements forming the parentbook. The operator map applies the transformation defined by the subsequent pattern matching to each element of a sequence (in this case x). Here map returns the sequence obtained by replacing each person in x by its Name element. Note that we use the pattern <person>[n _*] to match the person elements: n matches (and captures) the Name element—that is, the first element of the sequence—, _* matches (and discards) the sequence of elements that follow, and <person> matches the tag of the person (although the latter contains an attribute). The interface and the type definitions ensure that the tags will be the expected ones, so we could optimize the code by defining a body that skips the check of the tags:

```
<_> x -> (map x with <_>[ n _*] -> n).
```

However this optimization would be useless since it is already done by the implementation (see § 5) and, of course, it would make the code less readable. If instead of extracting the list of *all* parents we wanted to extract the sublist containing only parents with exactly two children, then we had to replace transform for map:

```
let names2 (ParentBook -> [Name*])
  <parentbook> x ->
    transform x with <person>[ n <children>[Person Person] _*] -> [n]
```

While map must be applicable to all the elements of a sequence, transform filters only those that make its pattern succeed. The right-hand sides return sequences which are concatenated in the final result. In this case transform returns the names only of those persons that match the pattern <person>[n <children>[Person Person] _*]. Here again, the implementation compiles this pattern exactly as <_>[n <_>[_ _] _*], and in particular avoids checking that sub-elements of <children> are of type Person when static-typing enforces this property.

These first examples already show the essence of ℂDuce's patterns: all a pattern can do is to decompose values into subcomponents that are either captured by a variable or checked against a type.

The previous functions return only the names of the outer persons of a ParentBook element. If we want to capture all the name elements in it we have to recursively apply names to the sequence of children:

```
let names (ParentBook -> [Name*])
  <parentbook> x -> transform x with
      <person> [ n <children>c _*] -> [n]@(names <parentbook>c)
```

where @ denotes the concatenation of sequences. Note that in order to recursively call the function on the sequence of children we have to include it in a ParentBook element. A more elegant way to obtain the same behavior is to specify that names can be applied both to ParentBook elements and to Children elements, that is, to the union of the two types denoted by (ParentBook|Children):

```
let names ( ParentBook|Children -> [Name*] )
  <_>x -> transform x with <person>[ n c _*] -> [n]@(names c)
```

Note here the use of the pattern <_> at the beginning of the body which makes it possible for the function to work both on ParentBook and on Children elements.

In all these functions we have used the pattern _* to match, and thus discard, the rest of a sequence. This is nothing but a particular regular expression over types. Type regexps can be used in patterns to match subsequences of a value. For instance the pattern <person>[_ _ Tel+] matches all person elements that specify no Email element and at least one Tel element. It may be useful to bind the sequence captured by a (pattern) regular expression to a variable. But since a regexp is not a type, we cannot write, say, x&Tel+. So we introduce a special notation x::R to bind x to the sequence matched by the type regular expression R. For instance:

```
let domain (Email->String) <_>[ _*? d::(Echar+ '.' Echar+) ] -> d
```

returns the last two parts of the domain of an e-mail (the *? is an ungreedy version of *, see §3.6). If these ::-captures are used *inside* the scope of the regular expression operators * or +, or if the same variable appears several times in a regular expression, then the variable is bound to the concatenation of all the corresponding matches. This is one of the distinctive and powerful characteristics of ℂDuce, since it allows to define patterns that in a single match capture subsequences of non-consecutive elements. For instance:

```
type PhoneItem = {name = String; phones = [String*]}
```

```
let agendaitem (Person -> PhoneItem)
  <person>[<name>n _ (t::Tel | _)*] ->
    { name = n ; phones = map t with <tel> s ->s }
```

transforms a person element into a record value with two fields containing the element's name and the list of all the phone numbers.

This is obtained thanks to the pattern (t::Tel | _)* that binds to t the sequence of all Tel elements appearing in the person. By the same rationale the pattern

(w::<tel kind="work">_ | t::<tel kind=?"home">_ | e::<email>_)*

partitions the (Tel | Email)* sequence into three subsequences, binding the list of work phone numbers to w, the list of other numbers to t, and the list of e-mails to e. Alternative patterns | follow a first match policy (the second pattern is matched only if the first fails). Thus we can write a shorter pattern that (applied to (Tel | Email)* sequences) is equivalent: (w::<tel kind="work">_ | t::Tel | e::_)*. Both patterns are compiled into (w::<tel kind="work">_ | t::<tel>_ | e::_)*, since checking the tag suffices to determine if the element is of type Tel.

Storing phone numbers in integers rather than in strings requires minimal modifications. It suffices to use a pattern regular expression to strip off the possible occurrence of a dash:

```
let agendaitem2 (Person -> {name=String; phones=[Int*]})
  <person>[ <name>n _ (t::Tel|_)* ] ->
     { name = n; phones = map t with <tel>[(s::'0'--'9'|_)*] -> int_of s }
```

In this case s extracts the subsequence formed only by numerical characters, therefore int_of s cannot fail because s has type ['0'- -'9'+] (otherwise, the system would have issued a warning)[2].

Consider the type PhoneBook = <phonebook>[PhoneItem*]. If we add a new pattern matching branch in the definition of the function names, we make it work both with ParentBook and PhoneBook elements. This yields the following *overloaded* function:

```
let names3 (ParentBook -> [Name*] ; PhoneBook->[String*])
      | <parentbook> x -> map x with <person>[ n _* ] -> n
      | <phonebook> x -> map x with { name=n } -> n
```

The overloaded nature of names3 is expressed by its interface, which states that when the function is applied to a ParentBook element it returns a list of names, while if applied to a PhoneBook element it returns a list of strings. We can factorize the two branches in a unique alternative pattern:

```
let names4 (ParentBook -> [Name*] ; PhoneBook->[String*])
    <_> x -> map x with ( <person>[ n _* ] | { name=n } ) -> n
```

The interface ensures that the two representations will never mix.

3. Presentation of the ℂDuce language

3.1 The type algebra

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

- three native scalar types, Int, Char, and Atom (atoms are symbolic constants 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;
- types constructors: record types { $a_1 = t_1$; . . . ; $a_n = t_n$ }, product types (t_1 , t_2), functional types (t_1 -> t_2), and XML types < t_1 t_2 > t_3 (where t_1,t_2,t_3 specify respectively the possible tags, attribute sets, and element contents);
- Boolean connectives: intersection t_1 & t_2, union t_1 | t_2 and difference $t_1 \setminus t_2$;
- singleton types: for any scalar or constructed (non-functional) value v, v is itself a type (for instance, ‘nil denotes 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's are type identifiers (that is, identifiers starting by a capital letter).

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 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 closed functional expressions fun f (s_1;. . . ;s_n)e that have type t_1–>t_2.

This interpretation of types is the basis of ℂDuce type system: the programmer must rely on it to understand all the type constructions and type equivalences of the system. 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* to (in the sense that it has the same interpretation as) 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]).

Records.
There are two different kinds of record types: the open record type, denoted by { $a_1 = t_1$; . . . ; $a_n = t_n$ }, that classifies records in which the fields labeled a_i are present with the prescribed type, but other fields may also appear, and the closed record type, denoted by {| $a_1 = t_1$; . . . ; $a_n = t_n$ |}, which forbids any label other than the a_i's [3]. 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 for instance {| $a = t$ |}≤{ $a = t$ }, or { $a_1 = t_1$; $a_2 = t_2$ } ≃ { $a_1 = t_1$ }&{ $a_2 = t_2$ }, or {| a_1=t_1 ; a_2=?t_2 |} ≃ {| a_1=t_1 |} | {| a_1=t_1 ; a_2=t_2 |}, where $\simeq = \leq \cap \geq$; once more, they all can be deduced from the set theoretic interpretation of record types as sets of record values.

Scalars.
We took special care of the definition and implementation of scalar types: integers have arbitrary precision (we believe that in XML applications, it is more important to have exact results than to optimize intensive numerical computations), and Char represents the whole Unicode character set. Moreover, we have subtypes of Int and Char, namely intervals of the form i- -j where i,j are two integer literals or two character literals, accordingly. We can specify for instance that the month attribute of some XML element is an integer between 1 and 12.

3.2 Encoded types

Sequences.
As in Lisp, sequences are encoded by pairs and an atom ‘nil representing the empty sequence: a sequence of values $v_1, v_2, \ldots, v_n$ is written in ℂDuce as [v_1 v_2 . . . v_n], but actually this is syntactic sugar for (v_1 , (v_2 (. . . , (v_n , ‘nil) . . .))).

[2] Actually the type system deduces for s the following type ['0'- -'9'+ '0'- -'9'+] (subtype of the former) since there always are at least two digits.

[3] There is a 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.

In the sample section we saw that regular expressions on types can be used to define new sequence types. Once more, this is just syntactic sugar since the sequence types are, in reality, defined by combining Boolean type connectives and recursive types. For instance, the [Int*] type used in the function agendaitem2 is defined as T where T = (Int,T) | 'nil.

Strings.
Although strings are nothing but a special case of sequences, the intensive use of this datatype in XML documents makes them worth special care. The main design choice we made for strings is to not have String as a native basic type; it is encoded as [Char*]. This seemed to us necessary for dealing with real world XML documents where strings often alternate with XML (e.g. XHTML) elements (simply consider <i> this </i> example). Having String as a basic type would be problematic, because no automatic concatenation would be performed.[4] Instead, by considering strings as sequences of characters, ['simply ' 'consider' <i>[' this '] 'example'] is equivalent to ['simply consider' <i>[' this '] 'example'] (both equivalent to the sequence with all characters separated). Of course, for sequences formed only by character literals, we allow the more classical double quote notation (values of XML attributes often fall into this case). We paid special attention to the implementation of strings for which we use a compact representation as long as possible and convert it to a sequence of characters only when necessary, and always transparently to the user (see §5).

3.3 XML elements

Although in ℂDuce the type <t_1 t_2> t_3 is primitive and the t_i's can range over all types, in practice, when working with XML documents, t_1 is usually an atom, t_2 a record type, and t_3 a sequence type. Thus XML types are conceptually encoded in terms of atoms, records and pairs and their introduction as a new type constructor is only justified by the need to avoid possible interferences.

That said, it is still important to notice that the type system does not restrict the possible values in tag position to being atoms. For instance, it is possible to use pairs to simulate namespaces; the namespace could be denoted by another atom (namespace normalization), or by a string (namespace URI): <(`xhtml,`li)>[ ] or <("http://...",`li)>[ ], and although the introduction of some syntax specific to namespaces is surely needed[5], in its current definition ℂDuce can already handle XML namespaces.

An XML element, <*tag* a_1=v_1 ... a_n=v_n> *elem-seq* </*tag*>, is written in ℂDuce as <*tag* {a_1 = v_1 ; ... ; a_n = v_n}> [*elem-seq*]. When appearing in tags, the back-quote of atoms and the curly brackets of records may be omitted:

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

We applied this convention to all the examples of the sample session (the same notations apply to types as shown by the same examples). An XML element construction has three "holes": the tag, the attribute record, and the content although they can actually be filled with any arbitrary expression, as in:

```
let tag = `a in let link = { href = "..." } in let c = "Click" in <(tag) (link)>c
```

or as in the two following functions

```
let del_target  (Link -> Link) <t (r)>x  ->  <t (r\target)>x
let add_target (Link -> Link) <t (r)>x  ->  <t (r+{target= "_top"})>x
```

that respectively remove and add some "target" attribute to an HTML element ($r \setminus \ell$ removes the ℓ field from r, if any, while r+r' denotes the destructive addition of the fields of r' to those of r).

It is possible to restrict the authorized attribute names by using a closed record type instead of the default open type; for instance, an element of type

```
type Tel2 = <tel {| kind=?"home"|"work" |}>['0'--'9'+ '-'? '0'--'9'+]
```

may only have a kind attribute and no other. Similarly we can also specify that a specific attribute must be absent:

```
type TelNokind = <tel kind=?Empty>['0'--'9'+ '-'? '0'--'9'+]
```

which can be read: "whenever the attribute kind is present, its value must be of type Empty"; as there is no such value, this means that the attribute kind cannot be present.

3.4 Overloaded functions

The simplest form for a toplevel function declaration is

let f (t->s) x -> e

in which the body of a function is formed by a single branch x->e of pattern matching. As we have seen in the previous sections, the body of a function may be formed by several branches with complex patterns. The interface (t->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 the names3 example (§2). The general form of a toplevel function declaration is indeed:

let fun f(t_1->s_1;...;t_n->s_n) | p_1->e_1 | ... | p_m->e_m

(the first vertical bar and the fun keyword are optional). Such a function accepts arguments of type (t_1|...|t_n); it has all the types t_i->s_i, and, thus, it also has their intersection (t_1->s_1&...&t_n->s_n).

The use of several arrow types in an interface serves to give the function a more precise 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 function below

   ```
   let 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 the same code, then the compound interface serves to give a more precise description of the behavior of the function. An example is the function names4 from §2.

There is no clear separation between these two situations since, in general, an overloaded function has body branches that specify behaviors of different arrow types of the interface but share some common portions of the code.

Let us examine a more complex example. We want to transform the representation of persons introduced in §2, using different tags <man> and <woman> instead of the gender attribute and, conversely, using an attribute instead of an element for the name. We also want to distinguish the children of a person into two different sequences, one of sons, composed of men (i.e. elements tagged by <man>), and the other of daughters, composed of women. Of course we also want to apply this transformation recursively to the children of a person. In practice, we want to define a function split of type Person –>(Man | Woman) where Man and Woman are the types:

[4] This is a classical problem: XML parsers do not usually guarantee that text nodes represent maximal textual portions of the documents; they are free to split adjacent characters into several text nodes

[5] In the next version of ℂDuce we plan to introduce a specific notation such as <xhtml:li>[ ], and adapt XML primitives (load_xml,...) to be namespace-compliant.

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

Here is a possible way to implement such a transformation:

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

The function split is declared to be an overloaded function that, when applied to a MPerson, returns an element of type Man and that, when applied to a FPerson, returns an element of type Woman. The body is composed of a single pattern matching whose pattern binds four variables: g is bound to the gender of the argument of the function, n is bound to its name, mc is bound to the sequence of all children that are of type MPerson, and fc is bound to the sequence of all children that are of type FPerson. On the next line we define tag to be `man or `woman according to the value of g. Then we apply split recursively to the elements of mc and fc. Here is the use of overloading: since mc is of type [MPerson*], then by the overloaded type of split 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 split function the type (MPerson -> Man) & (FPerson -> Woman). Note that the use of overloading here is critical: although split *also* has type Person ->(Man | Woman) (since split is of type MPerson->Man & FPerson->Woman, which is a subtype), had we declared split of that type, the function would not have type-checked: in the recursive calls we would have been able to deduce for s and for d the type [(Man | Woman)*], which is not enough to type-check the result. If, for example, we wanted to define the same transformation in XDuce we would need first to apply a filter (that is our transform) to the children so as to separate male from females (while in ℂDuce we obtain it simply by a pattern) and then resort to two auxiliary functions that have nearly the same definition and differ only on their type, one being of type MPerson –> Man, the other of type FPerson –> Woman. The same transformation can be elegantly defined in XSLT with a moderate nloc increase, but only at the expense of loosing static type safety and type based optimization: see Section 5 for preliminary benchmarks.

3.5 Higher-order functions

In ℂDuce all functions, including the overloaded ones, are first class expressions. 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, that is fun $f\,(t_1$->s_1;...;t_n->s_n)..., the only difference being that f can be omitted if the function is not recursive. Note that, using subtyping, such a function can be used wherever a function of type, say t_1 -> s_1 or t_1&t_n -> s_1&s_n, is expected.

Higher-order functions improve code reusability by sharing common code and providing specialized parts as functional arguments. In the setting of XML applications, a typical use would be to parameterize a generic printing function (that displays documents of some DTD to XHTML) by providing specialized functions to print various subparts of the documents (for instance, one of these functions could be in charge of displaying dates in a format chosen by the user).

Another use of first-class functions we have in mind (and we are going to implement) is a web application server that runs ℂDuce scripts. Instead of producing XHTML pages, the scripts would generate a variant of XHTML with ℂDuce code replacing <input> elements. For instance, instead of <input type=submit ...>, the scripts would directly generate a function (seen as a first class value) that has to be triggered when the user clicks on the button; this function can use identifiers bound to form-fields as normal ℂDuce variables. The web application server would transform this pseudo-XML document with embedded ℂDuce functions to a real XML document before sending it to the client, and storing in its internal tables the ℂDuce functions to be called. At the next HTTP request, it would call the function corresponding to the button the user pressed. The advantage of such an approach is twofold: thanks to static typing we know that the value handled by the transformation has the correct type and, more importantly, the web administrator no longer has to maintain CGI programs consistent with the HTML pages since this is already done by the server. This example demonstrates the use of first-class functions embedded in XML documents, in particular it addresses the same problems as (and for some aspects it extends) the JWIG approach [5]. The latter uses XML templates, that is, valid documents embedded with "gaps" that are to be filled by other templates (whereas we propose to embed higher order function—or any other ℂDuce expression), and relies on global data flow analysis to statically enforce both the validity of the generated XHTML and the correspondence between generated forms and received fields (whereas we plan to use the current ℂDuce's type system to check it).

Since the advantages of higher-order programming are well-known to the functional programming language community (and space is limited) we will not elaborate further. The lack of first-class functions in XML languages has been identified in several papers [12, 7] and our semantic approach to subtyping [8] has succeeded in mixing classical arrow types and XML types.

3.6 Pattern matching

Pattern matching is one of ℂDuce's key features. Although it resembles ML's, 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 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 either matches or rejects a value; when it matches, it binds its *capture variables* to the corresponding parts of the value and the computation continues with the body of the branch. Otherwise, control is passed to the next branch. This is a simple description of the behavior 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) traversal on the value to dispatch on the correct branch, and benefits from static typing information to avoid redundant checks (see §5).

Capture variables and deconstructors.

As in ML, a variable, say x, is a pattern that accepts and binds every value 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. 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). More interesting examples showing the expressiveness of this construction in the presence of recursive patterns will be presented later in this paper.

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 with at least the fields a_i whose content matches p_i while the latter matches records formed exactly by the a_i fields and whose content matches p_i. We use the same convention that we used for types and allow omitting curly brackets for open records occurring in tags. However, contrary to pair patterns, we do not allow multiple occurrences of a variable in a record pattern.

Type constraint and conjunction.
Every type can be used as a pattern. The semantics of such a pattern is to accept only values of that type and creates no binding.[6] This is particularly useful because in ℂDuce a type may reflect precise constraints on the values (structure and content). Note that scalar constants can also be used as patterns, as they are a special case of type constraints with singleton type. The wild-card type "_" is simply an alternative notation for the type constant Any and as such it matches every value.

To combine a type constraint and a capture variable, one can use the conjunction operator & for patterns, as in (x & Int). The semantics of the conjunction in a pattern is to check both sub-patterns and merge their respective sets of bindings. Since the two patterns of a conjunction must have disjoint sets of capture variables, no conflict can arise during the merging.

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. Alternative patterns are often used in conjunction with the pattern (x := c), where c is an arbitrary scalar or constructed constant, which provides a default value for a capture variable.

Recursive patterns.
Recursive patterns use the same syntax as recursive types: P where $P_1 = p_1$ and ... and $P_n = p_n$ with $P, P_1, \ldots, P_n$ being variables ranging over pattern identifiers (i.e. identifiers starting by a capital letter). 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 from a sequence the first element of type Int (recall that sequences are encoded with pairs). The order is important, because the pattern P where P = (_, P) | (x & Int, _) extracts 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 sides 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 (x is bound to the sequence containing them) 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 for defining patterns working on sequences with regular expressions built from patterns, usual regular expression operators, and *sequence capture variables* of the form x::R (where R is a pattern regular expression).

[6] Conversely, every pattern without capture variables is a type, which 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 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 the same (from a semantic viewpoint) pattern: had we used a different syntax for product types, say, $t_1 \mathrm{x}\, t_2$, 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 `}`.

Regular expression operators *, +, ? are *greedy* in the sense that they try to match as many times as possible. Ungreedy versions *?, +? 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, _) while [_*? (x & Int) _*] is compiled to P where P = (x & Int, _) | (_,P).

Let us detail the compilation of an example with a sequence capture variable: [_*? d::(Echar+ '.' Echar+)]. The first step is to propagate the variable down to simple patterns: [_*? (d::Echar)+ (d::'.') (d::Echar)+], which is then compiled to the recursive pattern:

```
P where P = (d & Echar, Q) | (_,P)
   and Q = (d & Echar, Q) | (d & '.', (d & Echar, R))
   and R = (d & Echar, R) | (d & ‘nil)
```

The (d & ‘nil) pattern above has a double purpose: it checks that the end of the matched sequence has been reached, and it binds d 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 sequence $[i]$.

A mix of greedy and ungreedy operators with the first match policy of alternate patterns allows the definition of powerful extractions. For instance, one can define a function that for a given person returns the first work phone number if any, otherwise the last e-mail, if any, otherwise any telephone number, or the string "no contact":

```
let preferred_contact(Person->String)
   <_>[ _ _ ( _*? <tel kind="work">x) | (_* <email>x) | <tel>x ] -> x
  | _ -> "no contact"
```

(note that <tel>x does not need to be preceded by any wildcard pattern as it is the only possible remaining case).

3.7 Extra support for sequences and queries

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, transform, and xtransform.
ℂDuce features a construction map e with p_1->e_1 | ... | p_n->e_n. The expression e must evaluate to a sequence, and each of its elements will go through the pattern matching and get transformed by the first matching branch. Static typing ensures the existence of such a branch. The typing of map is very precise, even when working with heterogeneous sequences, since it keeps track of the order of the elements in the input sequence, as shown by this example that uses the function split defined in Section 3.4

```
let f ([ MPerson* FPerson* ] -> [ Man* Woman* ])
   s -> map s with x -> split x
```

The type system is able to infer that the result of the map has type [Man* Woman*]. If the argument had type [MPerson? FPerson+], the type inferred for the result would be [Man? Woman+]. This precision in typing is out of reach of the user-definable polymorphic map function in ML; even with parametric polymorphism incorporated to ℂDuce (we have already started studying it), this built-in map would probably not be user-definable (because its very precise typing closely characterizes its complex behavior).

The map construction does not affect the length of the sequence, since each element is mapped to a single element. It is often useful not only to map the elements of a sequence but also to filter them, for which ℂDuce provides a variant of map, written transform, where each branch of the pattern returns a (possibly empty) sequence, and all the returned sequences, for each element in the source sequence, are concatenated together. There is an implicit default branch _ -> [] added on at the end so that unmatched elements are discarded. Our transform is very similar to the for of [7]: their generic loop for x in e_1 return e_2 can be simply translated to transform e_1 with x->e_2.

The last special sequence operator is xtransform, which works on (sequences of) XML trees. It matches the patterns against the root element of each XML tree and, if it fails, it recursively applies itself to the sequence of sons of the root. Thanks to xtransform a function that puts in boldface all the links of an XHTML document can be simply defined as:

```
let bold(x:[Xhtml]):[Xhtml]=xtransform x with <a (y)>t -> [ <a (y)>[<b>t] ]
```

(note the use of the variable y to preserve the attributes, e.g. href, rel, target,..., of the link). Note that without xtransform we would be obliged to iterate on the whole DTD of XHTML. In short, xtransform combines the flexibility of XSLT template programming with the precise static typing and efficient compilation of ℂDuce's transform.

Queries.
ℂDuce was designed by recasting some XML specific features from XDuce in a more general setting of higher-order and overloaded 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.[7]

As we mentioned above, our transform generalizes the for iteration from [7]. As in [7], the projection operator—denoted by /—can be defined from this construction: if e is a ℂDuce sequence expression and t is a type, then e/t is syntactic sugar for:[8]

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

This new syntax can be used to obtain a notation close to XPath [6]:

```
[parents]/<parentbook>_/
        <person>_/<children>_/<person>_/<tel kind="home">_
```

returns the sequence of all "home" phone numbers of children in our parents base (thanks to static typing we could have used _/_ in the path instead of <parentbook>_/<person>_ as no ambiguity is possible).

The function names2 in §2 implements exactly the same query. But while the use of transform somewhat freezes the implementation, the more declarative nature of path expressions spots out the places in the code where query optimization (using for instance equivalences similar to those mentioned in [7]) should be applied.

More generally, we are currently adding to the "algorithmic" constructions of ℂDuce a set of more "declarative" query-like constructions amenable to optimization techniques. In particular, we are currently adding classical select-from-where expressions but where the from clause can take advantage of the powerful ℂDuce's pattern algebra. Since order in from clauses is left unspecified, the system will be free to apply algebraic and/or cost-model based optimizations and/or use available indexes to implement joins efficiently.

4. Types

The type system is at the core of ℂDuce. The whole language was conceived and designed around it. From a practical point of view, the most interesting and useful characteristic of the type system is the semantic interpretation we described before, in which a type is nothing but a set of values denoted by some syntactic expression[9]. This simple intuition is all is needed to grasp the semantics of the ℂDuce's type system and, in particular, of:

- *Subtyping*: subtyping is defined as the 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 sample of type t but not of type s.
- *Boolean connectives:* Boolean connectives in the type algebra are interpreted simply 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. For example: [Int (String Int)*] $\simeq$ [(Int String)* Int].

It is important to understand types since they are pervasive in ℂDuce. In particular, pattern matching can be basically seen as dynamic dispatch on types, combined with information extraction, which gives ℂDuce a type-driven semantics reminiscent of object-oriented languages, since 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 since 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 the "methods" separate from the 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, RELAX NG, ...), 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 XDuce/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 in the compiler back-end. 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 §5.

4.1 Highlights of the type system

Since ℂDuce type system relies on interpreting types as sets of values, it is important to explain how values are typed. Apart from function values, this is straightforward, so we will focus on the typing rule for functions. In order to simplify the presentation, we split it into two rules, a subrule for typing function bodies (these are lists of pattern matching branches) whose derivation is then used in the typing rule for functions.

Pattern matching (function bodies).
Let B denote the sequence of branches p_1->e_1 | ... | p_n->e_n. The rule below derives the typing judgment $\Gamma \vdash t/B \Rightarrow s$, meaning "matching a value of type t against the sequence of branches B always succeeds and every possible result is of type s".

$$(t_i = t \backslash \wr p_1 \int \backslash \ldots \backslash \wr p_{i-1} \int \& \wr p_i \int)$$
$$\frac{t \leq \wr p_1 \int | \cdots | \wr p_n \int \qquad \Gamma, (t_i/p_i) \vdash e_i : s_i}{\Gamma \vdash t/B \Rightarrow \bigcup_{\{i \mid t_i \not\simeq \mathsf{Empty}\}} s_i}$$

[7]The fact that ℂDuce can implement such constructions is not surprising: any Turing-complete language can 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 pave the way to query optimization.

[8]We can take advantage of the fact that in ℂDuce a single pattern can extract all the elements of a given type, to define the following more compact encoding: `transform e with <_>[ (x::t | _ )* ] -> x`

[9]Of course, every type system induces a set-theoretic interpretation of types as sets of values. The point is that ℂDuce's type system is *built* on such an interpretation. As a result, the subtyping relation of ℂDuce is both sound and complete w.r.t. set-inclusion, whereas in other type systems only soundness holds (that is, 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).

Let us look at this rule in detail. The matched value is of type t. The left premise checks that the pattern matching is exhaustive; for each pattern p_i, $\llparenthesis p_i \rrparenthesis$ is a type that represents exactly *all* the values that are matched by p_i. The exhaustivity condition states that every value that belongs to 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 $\llparenthesis p_i \rrparenthesis$. So, when type-checking the expression of the i-th branch, one knows that the value is of type t_i, that is to say, of type t and of type $\llparenthesis p_i \rrparenthesis$ but not of any of the types $\llparenthesis p_1 \rrparenthesis, \ldots, \llparenthesis p_{i-1} \rrparenthesis$. 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 $\llparenthesis p \rrparenthesis$ 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 complex 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 the union of the result type of each branch i such that t_i is not empty (the notation $\bigcup_{i=1..n} s_i$ stands for $s_1|\ldots|s_n$); indeed, if t_i is empty, the branch cannot be selected, and the corresponding s_i is not included in the union.

Functions.

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

$$\frac{(t = t_1\text{->}s_1\&\ldots\&t_n\text{->}s_n) \qquad \Gamma, f{:}t \vdash t_i/B \Rightarrow u_i \leq s_i}{\Gamma \vdash \mathsf{fun}\ f(t_1\text{->}s_1;\ldots;t_n\text{->}s_n)B : t}$$

The type system simply checks all the constraints given in the interface (because the function can call itself recursively, when typing the body we record in the type environment 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 with 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 check the constraint.

This is not a minor point: not considering the return type of unused branches is the main difference between dynamic overloading and 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 different input types.

From the point of view of the programmer, it is quite easy to determine the type of a function value: it is simply the intersection of all the types specified in its interface.

4.2 Pattern type inference: examples

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

$$\begin{array}{lcl@{\qquad}lcl}
\llparenthesis x \rrparenthesis & = & \mathsf{Any} & \llparenthesis p_1|p_2 \rrparenthesis & = & \llparenthesis p_1 \rrparenthesis \,|\, \llparenthesis p_2 \rrparenthesis \\
\llparenthesis t \rrparenthesis & = & t & \llparenthesis p_1\&p_2 \rrparenthesis & = & \llparenthesis p_1 \rrparenthesis \,\&\, \llparenthesis p_2 \rrparenthesis \\
\llparenthesis (x{:=}c) \rrparenthesis & = & \mathsf{Any} & \llparenthesis (p_1,p_2) \rrparenthesis & = & (\llparenthesis p_1 \rrparenthesis,\llparenthesis p_2 \rrparenthesis)
\end{array}$$

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. Recursive patterns are handled by considering their infinite unfolding which, thanks to regularity, generate by the equations only finite systems. Other data constructors (records, XML elements) are treated like pairs. The same intuition guides the definition of (t/p). For example:

$$\begin{array}{lcl}
(t/x)(x) & = & t \\
(t/(p_1|p_2))(x) & = & ((t\&\llparenthesis p_1 \rrparenthesis)/p_1)(x)|((t\backslash\llparenthesis p_1 \rrparenthesis)/p_2)(x) \\
& \vdots &
\end{array}$$

states that when we match the pattern x against values ranging over the type t, the values captured by x will be exactly those in t. Similarly when we match values ranging over t against an alternative pattern, 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, and 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 (see the appendix of [8] for the rest of the definitions).

The most important result for these definitions is that the equations above can be used to define two algorithms that compute $\llparenthesis p \rrparenthesis$ and (t/p). Rather than going into the details of the algorithms, we prefer to give some examples that show the subtlety of the computation they are required to perform.

Consider again the pattern P where P = ((x & Int) , P) | (_ , P) | (x:='nil) that extracts all the integers occurring in a sequence[10]. In the table below we show the types of all values that are captured by the variable x of the pattern P when this latter is matched against (values ranging over) different types.

t	$(t/\mathsf{P})(\mathsf{x})$
[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+]

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, transform, and xtransform which need to compute the transformations of regular expressions (over types) and for which the same techniques as those of [7] are used.

5. Implementation

In this section we briefly highlight some important implementation issues and solutions specific to our approach. We have developed a prototype in Objective Caml; it compiles when needed pattern matchings to an internal automaton-like representation ("just-in-time"). Despite the interpretative overhead, it exhibits satisfactory performances. Typing ℂDuce programs is theoretically complex (the subtyping relation itself is already exponential in the size of involved types), and it is indeed possible to find short programs that kill the type-checker (as it is the case for ML, for instance). In designing ℂDuce we put the emphasis on the expressiveness of the language and the efficiency of the produced code, accepting the theoretical complexity of type-checking. XDuce has proved that type systems

[10] 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 in the order in which they appear.

for XML programs with regular expression types are workable, and our prototype gives us confidence that ℂDuce's new features will be too. In the future, we plan to develop a compiler for ℂDuce, and perform serious benchmarking and performance tests. Preliminary benchmarks on our prototype (see below) are encouraging.

Localization of error messages.

We formalized the static type checking of ℂDuce programs in [8], using the classical presentation of a type system with bottom-up inference rules. It is well-known that for a type-checking failure, direct implementation of such rules cannot provide well-localized error messages. For instance, consider fun f $(t$->$s)$ p_1->e_1 | p_2->e_2, where e_1 turns out not to be of type s. The typing rule that triggers the error is the one for abstractions. Since it sees the body branches as black-boxes, it cannot localize the error on e_1 but only on the whole function expression. A different example is given by the application of a function of type, say, Xhtml -> Xhtml to the document <html>[<head>[ ] <bdy>[ ]]. The typo in the last tag makes the typing of the application fail, and a bottom-up algorithm is not precise enough to track the misspelled <bdy>.

The solution is to type-check programs with a mixed top-down and bottom-up algorithm. In particular, we have adopted a technique that types expressions by propagating a *constraint* through the abstract syntax tree, from the root to the leaves. This constraint gives an upper-bound on the resulting type for the sub-expressions which catches errors earlier and localizes them precisely. This solution is highly effective when coupled with the ℂDuce typing as the latter is very precise and, quite interestingly (although not unexpectedly), typing precision induces a similar precision in error localization. In the first example, our prototype localizes an error somewhere in e_1 (depending on where the error is), and in the second it localizes it in the tag <bdy> (highlighted in red on the screen) and emits an error message stating that it should be <body>. This precision of error localization combined with the ℂDuce capacity to exhibit a value of the expected type not matching the definition produces very informative error messages. We invite the reader to test this issue on the online prototype or on the ℂDuce distribution.

Formally, the type system with propagation of constraint is defined by a typing relation $\Gamma \vdash e|t : s$, where Γ is the typing environment, e the expression to type-check, t the constraint and s the resulting type, which must be a subtype of t. For instance, the typing rule for pairs is:

$$\frac{\Gamma \vdash e_1|\pi_1(t) : t_1 \quad \Gamma \vdash e_2|\pi_2^{t_1}(t) : t_2}{\Gamma \vdash (e_1, e_2)|t : (t_1, t_2)}$$

where π_1 represents the set-theoretic first projection ($\pi_1(t)$ is the smallest solution τ to $t \leq (\tau, \mathsf{Any})$) and $\pi_2^{t_1}(t)$ represents the best refinement of the constraint knowing a more precise type for e_1 (it is defined as the largest solution τ to $(t_1, \tau) \leq t$). For instance, if $t =$ (<a>_,<a>_) | (<b>_,<b>_), and $e =$ (<a>[ ],<b>[ ]), the error will be localized on <b>[ ] which is type-checked with the constraint <a>_.

Implementation of typing algorithms.

In [8], 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 subtle implementation techniques can be applied in order to move from this specification to a practical algorithm, including those mentioned in [14] (for the informed reader, note that their algorithm can be easily improved by caching negative results in a destructive structure, since they cannot be invalidated even under different assumptions). The algorithm in [14] uses a top-down approach to avoid exponential explosion (due to the presence of union types); however, it requires backtracking and is thus suboptimal with respect to its theoretical complexity (and backtracking prevents the use a persistent data structure). Our implementation uses instead an efficient local solver for monotonic Boolean constraints that we developed for the occasion. It is always as efficient as the backtracking top-down approach, both theoretically and in practice (we implemented both). We conjecture that it guarantees optimal theoretical complexity as well.

The set-theoretic denotational foundation of ℂDuce is a primary source for studying implementation issues since quite often simple set-theoretic observations allow important optimizations. For instance, the type $(t, s)\backslash(t_1, s_1)\backslash\ldots\backslash(t_n, s_n)$ often appears in the execution of the typing algorithm. A naive development, consisting of the repeated use of the rule $(t, s)\backslash(t', s') = (t \setminus t', s)$ | $(t, s \setminus s')$ would systematically yield 2^n terms. To avoid this exponential explosion, we noticed that the result can be written as $(t \setminus t_1 \setminus \ldots \setminus t_n, s)$ | $(t$ & $t_1, s \setminus s_1)$ | ... | $(t$ & $t_n, s \setminus s_n)$ *provided* that the t_i's are pairwise non-intersecting (the formula can be further simplified by erasing empty terms). It is always possible to enforce the condition by "splitting" some of the t_i types. For the general case, we fall back to 2^n, but in practice we are closer to the linear case. Checking or enforcing the condition requires many applications of the subtyping algorithm, but our results have shown that is it worthwhile. This kind of "split and distribute" technique can be found in several places of our implementation and is reminiscent of analogous techniques mentioned in [13] for Boolean algorithms.

Pattern-matching compilation, type-driven optimizations.

ℂDuce type checking is not just a preliminary verification. We believe that static typing is key for designing an efficient execution model for ℂDuce (and XML languages in general). To grasp the idea, consider two types A and B and the function:

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

A naive compilation schema would yield the following behavior. It will first check whether the first pattern matches the argument. To do this, it will: (i) check that it is an XML element of the form <a>c and (ii) run through c to verify that it is a non-empty sequence of elements of type A (checking that an element is of type A may be very expensive if A represents for instance a complex DTD). If this fails, it will 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, it is known statically that the argument is necessarily an XML element with tag 'a, so there is no need to check this. Also, the content c must be a non-empty sequence whose elements are either all of type A or all of type B. To determine the situation, simply look at the first element and perform some tests to discriminate between A and B (for instance, if A = <x>[...] and B = <y>[...], looking at the head tag is sufficient, so the match could be compiled as <_>[<x>_ _*] -> 0 | _ -> 1). 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 can 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), different computations can be performed on v_1 and v_2 (each with a smaller automaton), but classical bottom-up tree automata do not have this flexibility. Also, to help pattern matching, we want to take static type information about the matched value into account. 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, which 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.

We have designed and implemented an efficient compilation schema that incorporates all these optimizations. Lack of space and complexity of the algorithms keep us from describing them here, and they will be presented in a future publication (but see the online extended version of this paper for an outline). Let us just state one interesting property: the compilation schema is semantic with respect to types, in the sense that the produced code does not depend on the syntax of the types that appear in patterns, but only on their interpretation. Therefore, there is no need to simplify types—for instance by applying any of the many type equivalences—before producing code, since such simplifications are all "internalized" in the compilation schema itself.

Representation of run-time values.

In ℂDuce, patterns check at runtime whether a value has a specific type. Hence one must be able to distinguish efficiently the representation of say, a pair value, an integer value, and a record value. In our prototype values are implemented by an OCaml sum type:

```
type value = Pair of value*value | Char of int | Integer of ... | ... .
```

This gives us the possibility to introduce *derived forms* of runtime values in order to optimize specific cases that cannot be easily detected at compile-time. When a derived form is inspected, it can be dynamically coerced to its canonical representation. For instance, in ℂDuce, character strings are conceptually sequences of Unicode characters: the value ['AB' <x>[ ]] is represented as Pair(Char 65, Pair(Char 66, Xml(...))). In order to avoid the allocation of many sequence cells (pairs) for long strings, we have added to the sum type for values a special form for strings (actually, we have several forms corresponding to different internal encodings), namely,

```
type value = ... | String of int * string * value
```

where the integer points to the beginning of the string in the buffer of characters at the second argument, and the third argument corresponds to the rest of the sequence (the 'nil atom in case of a sequence with only characters). The value above can thus be represented more compactly as String(0,"AB", Xml(...)) ; when this value is inspected by a pattern, it is seen as Pair(Char 65, String(1,"AB", Xml(...))) (the OCaml string buffer "AB" is not copied). The idea is to work with the compact representation as long as the string is not inspected.

Another example of special form is a lazy version of the concatenation operator @ for sequences; indeed, it is not unusual to build a long sequence by extending it repeatedly to the right. The canonical representation of sequences requires copying the first argument of @, and this may yield a quadratic behavior where a linear one is expected. The solution is to delay the computation of @ until the result is actually inspected; the binary tree with @ nodes can then be efficiently linearized. Of course, these derived forms are completely transparent to the programmer, and they do not affect the semantics of the language.

Benchmarks.

We performed preliminary benchmarks to evaluate ℂDuce's performance and validate our type-driven approach for compiling pattern matching.[11] In this section, we present a comparison between ℂDuce and an XSLT processor (the `xsltproc` program from Gnome libxslt library). The tables below display execution time in seconds (user time as reported by the Unix `time` command) for several sizes of input XML documents and several implementations of the same transformation. The execution times we give for ℂDuce programs include the times of XML parsing, type-checking, and validation of the input documents, while the times for XSLT programs include only XML parsing time (since XSLT is untyped).

The transformation `addrbook` is a simple filtering of flat XML documents. In order to test the effectiveness of ℂDuce type-driven optimization we performed some auto-benchmarking by testing different ℂDuce programs implementing the same transformation. In particular we considered two different versions of the transformation `addrbook`, one that uses explicit recursion to implement the traversal of the document and a second that uses instead the transform construction; furthermore for each version we considered a variant (denoted by "opt") which was optimized by hand by replacing dynamic type checking by minimal tests on tags.

`addrbook`	0.1 Mb	0.5 Mb	1.2 Mb	6 Mb	12 Mb
ℂDuce1	0.11	0.33	0.65	3.36	7.15
ℂDuce1 opt	0.11	0.33	0.64	3.39	7.14
ℂDuce2	0.11	0.32	0.61	2.95	5.85
ℂDuce2 opt	0.11	0.32	0.61	2.96	5.83
XSLT	0.08	0.38	0.76	3.74	7.38

The second transformation we considered is a simplified version of the `split` function in §3.4. The first ℂDuce version uses the "standard" pattern <person gender=g>[<name>n <children>[(mc::MPerson | fc::FPerson)*]]. The second one uses the hand-optimized pattern <_ gender=g>[<_>n <_>[(mc::<_ gender="M">_ | fc::_)*]]. The third ℂDuce version duplicates the main function to avoid overloading and useless computations on tags. The two XSLT versions use slightly different styles (two templates, or a single template with computations on the tag).

`split`	60Kb	0.3 Mb	0.6 Mb	2.5 Mb	5.2 Mb
ℂDuce 1	0.10	0.30	0.52	1.92	3.95
ℂDuce 2	0.11	0.30	0.50	1.92	3.92
ℂDuce 3	0.10	0.29	0.49	1.85	3.81
XSLT 1	0.15	0.79	1.42	5.95	12.85
XSLT 2	0.18	0.93	1.68	6.90	14.33

Although preliminary, these benchmarks already allow us to draw some conclusions. First of all, ℂDuce exhibits good performances: ℂDuce programs are usually faster than equivalent XSLT transformations (using a quite efficient XSLT processor written in C) and on the files of the tests they show execution times which are linear in the size of data. Compiling ℂDuce programs will remove interpretative overhead and type-checking from runtime, and so we expect some further improvements. Also, in a real usage scenario, several transformations will be composed in a single ℂDuce program to avoid parsing/validating/printing of intermediate XML documents.

Secondly, the negligible difference of execution times between normal and hand-optimized versions of ℂDuce programs demonstrates the effectiveness of the type-driven compilation approach and the uselessness of hand-coded optimization. This means that ℂDuce's runtime avoids the burden of coding optimizing patterns, and allows the programmer to use a more declarative and robust style of pro-

[11] The test machine was an Athlon 750 with 128 Mbytes of RAM. The code of all benchmarks is available at `http://www.cduce.org/bench.html`.

gramming.

Finally, we want to stress that none of the examples we used in the benchmarks above exploits the full power of ℂDuce's type-based optimization: in all these examples the only possible gain brought by ℂDuce's optimization came from avoiding useless tests of tag names; however the examples are so compact that ℂDuce's optimization could not be used to ignore whole subtrees of the input documents. Since the latter is the case in which ℂDuce runtime system is expected to succeed best, we expect that in real-case use the advantage of using ℂDuce will be even more evident also in terms of pure performance.

We do not include in this benchmark section a comparison with XDuce because XDuce has not been optimized for runtime. As for type-checking—an aspect we cannot compare with XSLT—complex transformations seem to be type-checked significantly faster in ℂDuce than in XDuce (for instance, a simplified version of `html2latex` from the XDuce distribution takes 0.44s for ℂDuce, versus 9.40s for XDuce 0.4.0, 1.33s for XDuce 0.2.4, and 18.30s for XDuce 0.2.4 with pattern optimization turned on).

We are currently performing more extensive benchmarking and comparing performances with respect to XSLT and XQuery. We plan to report complete results on ℂDuce site's benchmark pages.

6. Conclusions and ongoing work

ℂDuce is an extension of XDuce with a richer set of basic types (Char, String, Int, intervals) and of constructed types (open and closed records, intersections, differences, singletons), while adding to the language overloaded and higher order functions, powerful sequence extracting patterns, records, and tags as first-class expressions. It is important to notice that this is obtained smoothly by using a very small core of semantically defined key features [8] . ℂDuce also relies on a theoretic construction quite different from that of XDuce and this construction had a deep impact on the implementation of the language itself and on the typing and subtyping algorithms. A separate article to present them is in preparation.

We have also begun formally studying security issues of ℂDuce with preliminary results available on the ℂDuce web site.

As for language, we have just finished implementing and we are presently testing XML Schema integration and validation. This is obtained by importing schemas into ℂDuce types and by validating ℂDuce expressions against them. This greatly simplifies the typed import of XML documents which can then be treated by using the ℂDuce data model (in XQuery the choice was made to work directly on XML Schema data model).

We are also implementing the query language we hinted at in Section 3.7, developing simple logical optimizations and benchmarking it against the Bell-labs' XQuery implementation [2] using the XQuery Use Cases (http://www.w3.org/TR/xmlquery-use-cases/) as testbed. We expect to present results and merge the CVS branch into the main ℂDuce branch before the end of the year.

We are also currently studying a module system that supports incremental programming via cross-module specialization. The basic idea can be understood by considering the toplevel definition of the function add given in Section 3.4. A new let fun declaration for add would hide the older one. We are experimenting with a specializing declaration

```
let method add ((Char,String)->String) (x,y) ->[x] @ y
```

where the new definition specializes the definition of add in the sense that it is as if we had defined from the beginning add as follows

```
let fun add ((Char,String)->String; (Int,Int)->Int; (String,String)->String)
    | (x & Char, y & String) -> [x !y]
    | (x & Int, y & Int) -> x+y
    | (x & String, y & String) -> x@y
```

Such definitions can be smoothly encoded in the core of ℂDuce by a combination of dynamic scoping and a technique similar to the one used for Java's parasitic methods [4]; an incremental programming style can then be obtained by allowing cross module specialization of the form let method SomeModule.add ... .

Besides studying the module system and query language extensions we have hinted at in this paper, future plans include the study of polymorphic, lazy, and reference types, the exploration of interactions with other languages and type systems (typically for using existing libraries) and the development of tools for better interfacing ℂDuce with XML tools and standards.

Acknowledgments. We want to warmly thank Dario Colazzo, Haruo Hosoya, Stijn Vansummeren, Jérôme Vouillon, and Philip Wadler for the useful and constant feedback on this work, Pietro Di Lena for his help with XSLT, Stefano Zacchiroli for his work on XML Schema, Abigail Pope for proof-reading this paper, and the ICFP referees whose remarks greatly contributed to improve the presentation of this article. Very special thanks go to Benli Pierce for "sheparding" this work and for the many useful suggestions, discussions, inputs on this and other topics.

References

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

[2] Bell-labs. *Galax*. `http://db.bell-labs.com/galax/`.

[3] S. Boag, D. Chamberlin, M. Fernandez, D. Florescu, J. Robie, J. Siméon, and M. Stefanescu. *XQuery 1.0: An XML Query Language*. W3C Working Draft, `http://www.w3.org/TR/xquery/`, 2003.

[4] J. Boyland and G. Castagna. Parasitic methods: Implementation of multi-methods for Java. In *OOPSLA '97*, 32(10), pages 66–76, 1997.

[5] A. Christensen, A. Møller, and M. Schwartzbach. Extending Java for high-level web service construction. *ACM TOPLAS*, 2003. To appear.

[6] J. Clark and S. DeRose. *XML Path Language (XPath)*. W3C Recommendation, `http://www.w3.org/TR/xpath/`, 1999.

[7] Mary Fernández, Jérôme Siméon, and Philip Wadler. An algebra for XML query. In *FST&TCS*, LNCS n. 1974, pages 11–45, 2000.

[8] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *17th IEEE Symp. on Logic in Computer Science*, pages 137–146, 2002.

[9] V. Gapayev and B.C. Pierce. Regular object types. In *Proceedings of the 10th workshop FOOL*, 2003.

[10] H. Hosoya. Regular expressions pattern matching: a simpler design. Unpublished manuscript, February 2003.

[11] H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM TOIT*, 2003. To appear. `http://xduce.sourceforge.net/`.

[12] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, 2001.

[13] Haruo Hosoya and Makoto Murata. Validation and boolean operations for attribute-element constraints. In *PLAN-X*, 2002.

[14] H. Hosoya, J. Vouillon, and B. Pierce. Regular expression types for XML. In *Proc. of ICFP '00*, *SIGPLAN Notices* 35(9), 2000.

[15] Michael Y. Levin. Matching automata for regular patterns. Tech. rep., 2003. `http://www.cis.upenn.edu/~bcpierce/xtatic/`.

[16] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proc. of 24th ACM POPL*, 1997.

[17] J. Siméon and P. Wadler. The essence of XML. In *Proc. of 30th ACM POPL*, 2003.

A.1 Syntax of ℂDuce

Names

X *Names complying with the XML recommendation*$^{(*)}$ *but starting by an upper case literal*

Variables

x *Names complying with the XML recommendation*$^{(*)}$ *but starting by a colon, an underscore, or a lower case literal*

Constants

$c ::= n$	n a signed integer
$\mid$ $'a'$	a a unicode literal

Atoms

$a ::= \text{‘}X \mid \text{‘}x$

Base types

$B ::=$ String $\mid$ Int $\mid$ Char $\mid$ Bool

Singleton types

$S ::= a \mid c$

Types

$T ::= B$	base types
$\mid$ S	singleton
$\mid$ $T \mid T$	union
$\mid$ $T \& T$	intersection
$\mid$ $T \setminus T$	difference
$\mid$ $T \to T$	functions
$\mid$ $\langle T\, T \rangle\, T$	XML tree
$\mid$ $\langle T\, \ell(T) \rangle\, T$	XML friendly
$\mid$ X	type variable
$\mid$ T where $X = T$ and . . . and $X = T$	recursion
$\mid$ $\{\ \ell(T)\ \}$	open record
$\mid$ $\{\mid \ell(T) \mid\}$	closed record
$\mid$ $[R]$	sequences
$\mid$ l--l	interval
$\mid$ Empty	empty type
$\mid$ Any	all values
$\mid$ _	all values

Attribute list

$\ell(\alpha) ::= \epsilon$	empty attribute list
$\mid$ $x = \alpha$	mandatory attribute
$\mid$ $x\ =?\ \alpha$	optional attribute
$\mid$ $\ell\ ;\ \ell$	list of α attributes

Interval limit

$l ::= c \mid *$

Type regular expressions

$R ::= T \mid R \mid R \mid R\, R \mid R\, po$

Pattern regular expressions

$r ::= p \mid (x :: r) \mid r \mid r \mid r\, r \mid r\, po$

Sequence content

$s ::= \epsilon \mid e \mid\ !e \mid s\, s$

$^{(*)}$http://www.w3.org/TR/REC-xml#NT-Name

Patterns

$p ::= x$	capture
$\mid$ T	type constraint
$\mid$ X	pattern variable
$\mid$ $p \& p$	conjunction
$\mid$ $p \mid p$	alternative
$\mid$ (p, p)	pair
$\mid$ $\langle\, p\ p \,\rangle\, p$	XML pattern
$\mid$ $\langle\, p\ \ \ell(p) \,\rangle\, p$	XML friendly
$\mid$ $\{\ \ell(p)\ \}$	open record
$\mid$ $\{\mid \ell(p) \mid\}$	closed record
$\mid$ $[r]$	sequences
$\mid$ $(x{:=}c)$	default
$\mid$ p where $X = p$ and . . . and $X = p$	recursion

Functions

$f ::=$ fun $x(p : T, \ldots, p : T){:}T = e$

$\mid$ fun $x(T \to T; \ldots; T \to T)\ p \to e \mid \ldots \mid p \to e$

Expressions

$e ::= c$	constant
$\mid$ a	atom
$\mid$ x	variable
$\mid$ f	function
$\mid$ $e_1 e_2$	application
$\mid$ (e_1, e_2)	pair
$\mid$ $\langle\, e\ \ e \,\rangle\, e$	XML element
$\mid$ $\langle\, e\ \ \ell(e) \,\rangle\, e$	XML friendly
$\mid$ $o\ e$	prefix operator
$\mid$ $e\ op\ e$	infix operator
$\mid$ $\{\ \ell(e)\ \}$	record
$\mid$ $e.x$	field select
$\mid$ $e \setminus x$	field remove
$\mid$ $[s]$	sequences
$\mid$ e/T	projection
$\mid$ $(e : T)$	coercion
$\mid$ let $p = e$ in e	let
$\mid$ let $p : T = e$ in e	coerced let
$\mid$ if e then e else e	if_then_else
$\mid$ raise e	raise exception
$\mid$ try e with $p \to e \mid \ldots \mid p \to e$	trap exception
$\mid$ match e with $p \to e \mid \ldots \mid p \to e$	match
$\mid$ map e with $p \to e \mid \ldots \mid p \to e$	map
$\mid$ transform e with $p \to e \mid \ldots \mid p \to e$	filter
$\mid$ xtransform e with $p \to e \mid \ldots \mid p \to e$	XML-tree transf

Prefix operators

$o ::=$ load_xml $\mid$ load_html $\mid$ load_file $\mid$ load_file_utf8

$\mid$ print $\mid$ print_xml $\mid$ print_xml_utf8

$\mid$ dump_to_file $\mid$ dump_to_file_utf8

$\mid$ int_of $\mid$ string_of $\mid$ atom_of $\mid$ flatten

Infix operators

$op ::=$ @ $\mid$ + $\mid$ * $\mid$ − $\mid$ div $\mid$ mod

$\mid$ = $\mid$ <= $\mid$ << $\mid$ >> $\mid$ >=

Postfix operators

$po ::=\ ? \mid + \mid * \mid\ ?? \mid +? \mid *?$

Conventions: we write [PCDATA] for [Char*], write ['xy' 'z'] or ['x' 'yz'] or ['xyz'] or "xyz" for ['x' 'y' 'z'], and write $<x>$ and $<X>$ for $<\text{‘}x>$ and $<\text{‘}X>$. Also, let $f(\ldots)\ldots$ and let fun $f(\ldots)\ldots$ are both allowed instead of let f = fun $f(\ldots)\ldots$. The types String and Bool are defined as [Char*] and ‘true | ‘false, respectively.