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 overcome some limitations of XDuce [?] 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 loses 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 [?].

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 [?] 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 [?]. Together with Makoto Murata [?] 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 algorithm-
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 [?] and XQuery [?].

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 OOP 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 [?]). 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 [?]. 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 [?]), 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 [?].

2. A sample session

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

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

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 MPerson or FPerson according to the value of the gender attribute. An equivalent definition for Person would thus be:

<person gender="M"|"F">[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*, so as to avoid tree automata to check subtyping or validate documents. The PCDATA type is of the same kind as in CDuce's types. Support for XML Schema validation has been recently implemented and is in alpha testing (see § ??).

1CDuce distribution includes dtd2duce, 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 § ??).
let parents : ParentBook =
  <person gender="F">
    <name>Clara</name>
    <children>
      <person gender="M">
        <name>Pål André</name>
        <children/>
      </person>
      <person gender="M">
        <name>Bob</name>
        <children/>
      </person>
    </children>
  </person>
  <person gender="M">
    <name>Charlie</name>
    <children/>
  </person>
  <person/>
  <person/>
  <person gender=
    "F">
    <name>Alice</name>
    <children/>
  </person>
  <person/>
  <person gender=
    "F">
    <name>Clara</name>
    <children/>
  </person>
  <person/>
  <person gender=
    "M">
    <name>Clara</name>
    <children/>
  </person>
  <person/>
  <person/>
  <person/>
  <person/>
  <person/>
</parentbook>

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

where @ denotes the concatenation of sequences. Note that in order to recursively call 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 <children[Person Person] _ ] -> [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:

domain (Email->String) <_,x> -> transform x with
  <person> [ n <children[Person Person] _ ] -> [n]@[names c]

declaration (Person -> Phonetel)
  <person> [ n <phones[Phone Person] _ ] ->
    [ name = n ; phones = map x 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.

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 §§?). 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> x -> [map x with <_,x> [ n _ ] -> n].

However this optimization would be useless since it is already done by the implementation (see § ??) 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]

transform x with <person> [ n <children[Person Person] _ ] -> [n]
This is obtained thanks to the pattern \( \text{t::Tel \{\_\}} \) that binds to \( t \) the sequence of all \( \text{Tel} \) elements appearing in the person. By the same rationale the pattern

\[
(\text{w::tel kind=\"work\">\_ | \text{e::email}>\_ )^* \quad \text{partitions the} \quad (\text{Tel} \mid \text{Email}) \quad \text{sequence into three subsequences, binding}
\]

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 \( \mid \) 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 (\text{Tel} \mid \text{Email}) \text{ sequences}) is equivalent: \( (\text{w::tel kind=\"work\">\_ | \text{t::tel}>\_ | \text{e::tel}>\_ )^* \). Both patterns are compiled into \( (\text{w::tel kind=\"work\">\_ | \text{t::tel}>\_ | \text{e::tel}>\_ )^* \), since checking the tag suffices to determine if the element is of type \( \text{Tel} \).

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:

\[
\text{let agendaitem2 (Person -> \{\text{name = String; phones = [Int*]\})}
\]

\[
\text{\{} \text{name = n; phones = map t with \{<\text{tel}>[[\text{\'0\}'--\text{\'9\}]+ \text{\'}\_ \mid \text{\'}t\_ | \text{\'}e\_]]\} -> \text{int_of s } \text{\}} \text{\}\}}
\]

In this case \( s \) extracts the subsequence formed by numerical characters, therefore \( \text{int_of s} \) cannot fail because \( s \) has type \( \{ \text{\'}0\}'--\text{\'9\}+ \} \) (otherwise, the system would have issued a warning).\(^3\)

Consider the type \( \text{PhoneBook = <\text{phonebook} \[\text{PhoneItem*}\]} \) where \( \text{<\_} \) is the type of the integer \( \text{\_} \) the smallest) and the universal (i.e., the largest) type; \( \text{t} \) is any type, \( \text{a} \) and \( \text{b} \) are two integer literals, accordingly. We can specify for instance that \( \text{\{'name = n; phones = map t with \{<\text{tel}>[[\text{\'}0\}'--\text{\'9\}+ \text{\'}\_ | \text{\'}t\_ | \text{\'}e\_]]\} -> \text{int_of s } \text{\}}} \) is the set of all closed functional 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 \( v_1 \) is a value of type \( t_1 \); similarly \( t_1 \rightarrow t_2 \) is the set of all closed functional expressions \( \text{fun1}(s_1, \ldots, s_n) \) that have type \( t_1 \rightarrow t_2 \).

This interpretation of types is the basis of \( \text{CDuce} \) 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 \([2]\)).

\[\] 

### Records

There are two different kinds of record types: the open record type, denoted by \( \{a_1 = t_1; \ldots ; a_n = t_n\} \), 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 \( \{a_1 = t_1; \ldots ; a_n = t_n\} \), 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 \( \{a = t\} \subseteq \{a = t\} \), or \( \{a_1 = t_1; a_2 = t_2\} \equiv \{a_1 = t_1 \& \& a_2 = t_2\} \). We can specify for instance that the month attribute of some XML element is an integer between 1 and 12.

### 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 \( a \rightarrow b \) where \( a \) and \( b \) 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.

### 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 \( \text{CDuce} \) as \( \{v_1, v_2, \ldots, v_n, \text{\'nil}\} \), but actually this is syntactic sugar for \( \{v_1, (v_2, \ldots, (v_n, \text{\'nil})\} \).
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 \([\text{Int}]^*\) type used in the function \(\text{agordaintern2}\) is defined as \(T\) where \(T = (\text{Int},T)\) \(\rightarrow\) \(\text{nil}\).

### 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 \([\text{Char}]^*\). 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. Instead, by considering strings as sequences of characters, \([\text{simply 'consider '}<i/>\) this \(</i>\) example\] is equivalent to \([\text{simply consider '}<i/>\) this \(</i>\) 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 §3).

#### 3.3 XML elements

Although in CDuce the type \(t_1 \rightarrow t_2\) 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): \(<x:\text{html}\>\text{hello}</x>\), and although the introduction of some syntax specific to namespaces is surely needed, it is not part of the current definition of CDuce.

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

\[
\langle\text{href="click.htm";} \text{target="_top">}[\text{Click here}]\]

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:

\[
\text{let tag} = \text{a in let link} = \{\text{href = "..."}\} \text{in}\ \text{c = "Click" in} \text{<tag} \text{\langle \text{link}\rangle} \text{c}
\]

or in the following two functions:

\[
\text{let del_target} = \text{\langle Link \rightarrow \text{Link}\rangle} \langle r\rangle \text{x} \rightarrow \langle \ell\ \text{\rightarrow \ell}\rangle \text{x} \text{let add_target} = \text{\langle Link \rightarrow \text{Link}\rangle} \langle r\rangle \text{x} \rightarrow \langle r\rightarrow \text{\rightarrow \text{top}}\rangle \text{x}
\]

4This 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

5In the next version of CDuce we plan to introduce a specific notation such as \(<\text{html}[^*]\), and adapt XML primitives (load_xml,...) to be namespace-compliant.

3.4 Overloaded functions

The simplest form for a toplevel function declaration is

\[
\text{let } f : (t_1 ... t_s) \rightarrow c
\]

in which the body of a function is formed by a single branch \(x \rightarrow c\) 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 \((t_1 ... t_s)\) 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 (§3). The general form of a toplevel function declaration is indeed:

\[
\text{let fun } f : (t_1 \rightarrow s_1 ; \ldots ; t_n \rightarrow s_n) \ \mid \ p_1 \rightarrow e_1 \ \mid \ \ldots \ \mid \ p_m \rightarrow e_m
\]

(\(s\) is the first vertical bar and the \(\text{fun}\) keyword are optional). Such a function accepts arguments of types \(t_1 ; \ldots ; t_n\), it has all the types \(t_i \rightarrow s_i\), and, thus, it also has their intersection \((t_i \rightarrow s_i)\&\ldots\&k\rightarrow s_n\).

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

\[
\text{let add } (\text{\langle Int,Int\rangle} \rightarrow \text{t}) \mid (\text{\langle String,String\rangle} \rightarrow \text{String})
\]

\[
\text{\langle x \& Int, y \& \text{Int}\rangle} \rightarrow x+y \mid \text{\langle x \& \text{String}, y \& \text{String}\rangle} \rightarrow x@y
\]

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 §3.

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 §3, using different tags \(<\text{man}>\) and \(<\text{woman}>\) 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 \(<\text{man}>\)), 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 \(\text{Person} \rightarrow (\text{Man} \mid \text{Woman})\) where \text{Man} and \text{Woman} are the types:
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=1>[]<_children>[](mp: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 cf with x -> split x in

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 XDuce 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 nocInc increase, but only at the expense of loosing static type safety and type based optimization: see Section ?? for preliminary benchmarks.

3.5 Higher-order functions

In XDuce 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 (t1->s1;...;tn->sn), ..., 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 t1 -> s1 or t1&n -> s1&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 XDuce scripts. Instead of producing XHTML pages, the scripts would generate a variant of XHTML with XDuce 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 XDuce variables. The web application server would transform this pseudo-XML document with embedded XDuce functions to a real XML document before sending it to the client, and storing in its internal tables the XDuce 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 [2]. 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 XDuce expression), and relays 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 XDuce’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 [2,?] and our semantic approach to subtyping [2] has succeeded in mixing classical arrow types and XML types.

3.6 Pattern matching

Pattern matching is one of XDuce’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 XDuce there is a standalone pattern-matching expression match e with p1::p2::...::pn ->c, Local binding let p = e in e2 is just syntactic sugar for match c1 with p -> e2.

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 ??).

Capture variables and destructors.

As in ML, a variable, say x, is a pattern that accepts and binds every value to x. A pair pattern (p1, p2) accepts every value of the form (v1, v2) where v1 matches p1. If a variable x appears both in p1 and in p2, then each pattern p1 binds x to some value v; the semantics is here to bind the pair (v1, v2) to x for the whole pattern. For instance, a pattern matching branch (x,y) -> (x,y) is equivalent to (x1,x2) -> (x1,x2) & y. Similarly (x1,x2,y) -> (x,y) is equivalent to (x1,x2,y) -> (x1,x2,y) & 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 (a1=pi1;...;an=pni); the former matches every record with at least the fields ai, whose content matches pi, 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 \( (x \& \text{Int}) \). 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 \( p_1|p_2 \) with first match policy: it first tries to match the pattern \( p_1 \), and if it fails, it tries with \( p_2 \); 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 \( P = (x \& \text{Int}, _, _) \) \((P)\); 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 \( P = (_, P) \) \((P)\) 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 \( P = (x \& \text{Int}, P) \) \((P)\) \((x := 'nil')\) extracts all the elements of type Int from a sequence \( x \) is bound to the sequence containing them) and the pattern \( P \) where \( P = (x \& \text{Int}, (x \& \text{Int}, _)) \) \((P)\) 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 \textit{sequence capture variables} of the form \( x:R \) (where \( R \) is a pattern regular expression).

Regular expression operators \(*\), +, ? are \textit{greedy} in the sense that they try to match as many times as possible. Ungreedy versions \(\star\), +?, and ? are also provided; the difference in the compilation scheme is just a matter of order in alternative patterns. For instance, \( [\star^* (x \& \text{Int}, _) ] \) is compiled to \( P \) where \( P = (\_, P) \) \((x \& \text{Int}, _)\) while \( [\star^* (x \& \text{Int}, _) ] \) is compiled to \( P \) where \( P = (x \& \text{Int}, _) \) \((P)\).

Let us detail the compilation of an example with a sequence capture variable: \( [\star^* d:(\text{Echar} \cdot \cdot\cdot) \star^* \cdot] \). The first step is to propagate the grammar down to simple patterns: \( [\star^* d:(\text{Echar}) \cdot\cdot\cdot \star^* \cdot] \) where \( d:(\text{Echar}) \cdot\cdot\cdot \star^* \cdot \) is then compiled to the recursive pattern: \( P \) where \( P = (d \& \text{Echar}, Q) \) \((P)\) and \( Q = (d \& \cdot, d \& \text{Echar}, R) \) \((P)\) and \( R = (d \& \cdot \& \text{Echar}, R) \) \((P)\). The \((d \& \cdot)\) 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 \( x \& \text{Int} \) and \( [x \& \text{Int}] \). 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 \([i]\) [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":

\[
\text{let preferred_contact(Person->String)} \\
<_{x>[\star^* d:(\text{Echar})\cdot\cdot\cdot \star^* \cdot]\rightarrow x | \_\rightarrow \text{"no contact"}}
\]

(Recall that \( <_{\text{tel}} \) 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 \( \text{split} \) defined in Section??

\[
\text{let f([[MPerson f:Person] -> [Man Woman]])} \\
\text{s -> map s with x -> split x}
\]

The type system is able to infer that the result of the map has type \([\text{Man Woman}]\). If the argument had type \([\text{MPerson f:Person}]\), the type inferred for the result would be \([\text{Man Woman}]\). 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 \( _\rightarrow [] \) added on at the end so that unmatched elements are discarded. Our transform is very similar to the for of \([2]\): their generic loop for \( x \) in \( e_1 \) return \( e_2 \) can be simply translated to transform \( e_1 \) with \( x \rightarrow e_2 \).
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:

\[
\text{let bold}[X\text{html}][X\text{html}] = \text{xtransform} x \text{ with } \langle a(y)\rangle \mapsto \text{[<a (y)>]<b>t]} \text{ (note the use of the variable y to preserve the attributes, e.g. href, rel, target, . . . of the link). Note that without xtransform we would be obliged to iterate on the whole DTD of XHTML. In short, xtransform combines the flexibility of XSLT template programming with the precise static typing and efficient compilation of C\text{Duce}'s transform.}
\]

Queries.

C\text{Duce} was designed by recasting some XML specific features from X\text{Duce} 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.\textsuperscript{7}

As we mentioned above, our transform generalizes the for iteration from \textsuperscript{7}. As in \textsuperscript{7}, the projection operator—denoted by /—can be defined from this construction: if \( e \) is a C\text{Duce} sequence expression and \( t \) is a type, then \( e/t \) is syntactic sugar for: \textsuperscript{8}

\[
\text{transform} \ e \ \text{with} \ e \_\_ \rightarrow \text{transform} \ c \ \text{with} \ (x \ & \ t) \rightarrow \ [x]
\]

This new syntax can be used to obtain a notation close to XPath \textsuperscript{9}:

\[
\text{[parents]} \langle \text{parentbook} \_ \_ \_ \_ \ r \_ \_ \_ \_ \ \langle \text{person} \_ \_ \_ \_ \ \langle \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_
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 \( p_i \), \([p_i]t\) 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 \( p_1 \rightarrow e_1 \) 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 \([p_i]t\). 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 \([p_i]t\) but not of any of the types \([p_1]t, \ldots, [p_{i-1}]t\). Now we type-check the body \( e_i \) of the branch; to do so, we must collect some type information about the variables bound by \( p_i \). This is the purpose of \((t/p_i)\): 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 \([p]t\) and \((t/p)\). These operators were introduced in [?]. 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 \( t_i \sim \text{Empty} \)) in a pattern matching? The answer is in the typing rule for functions:

\[
(t = t_1 \rightarrow s_1 \ldots \& t_n \rightarrow s_n) \\
Γ, f :: t \Rightarrow u_i \leq s_i \\
Γ \vdash f : t_1 \rightarrow \ldots \rightarrow t_n \Rightarrow B \Rightarrow t
\]

The typing 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:

\[
\text{fun (Int -> Int; String -> String)} \\
\mid \text{Int -> 42} \\
\mid \text{(x & String) -> x}
\]

When type-checking the body for the constraint \( \text{String -> String} \), 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 dynamicic types of [?]). 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 \([p]t\) 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, \([p]t\) is defined by the following set of equations

\[
([p_1]t) = \text{Any} \\
([p_1 \& p_2]t) = ([p_1]t) \& ([p_2]t) \\
([t/p_1]t) = t \\
([p_1 \& p_2]t) = ([p_1]t) \& ([p_2]t) \\
([\text{int} = \text{int}] t) = \text{Any}
\]

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:

\[
(t/x) = t \\
(t/(p_1/p_2)) = (((t & [p_1])/(p_2))(t((t & [p_1])/(p_2))))(x)
\]

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 [?] 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 \([p]t\) 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 \( P = ((x & \text{Int}) , P) | (\_ , P) \) (\( x::\text{nil}! \)) 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.

The typing of patterns, pattern matching, and functions is essentially all needed is 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 [?] 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 CDuces 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 CDuces 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 [2], 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 ((\rightarrow s) p1 -> e1 | p2 -> e2), where e1 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 e1 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><a>head</a></head></html>]. 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 e1 (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 $\Gamma \vdash e : t$, where $\Gamma$ 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:

$$\Gamma \vdash e_1[\pi_1(t) : t_1] \quad \Gamma \vdash e_2[\pi_2(t) : t_2]$$

$$\Gamma \vdash (e_1, e_2) : (t_1, t_2)$$

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

Implementation of typing algorithms.

In [2], 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 [2] (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 [2] 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-theoretical 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 $((t, s) \setminus (t_1, s_1)) \ldots \setminus (t_n, s_n)$ often appears in the execution of the typing algorithm. A naive development, consisting of the repeated use of the rule $((t, s) \setminus (t', s') = ((t', s') \setminus (t, s \setminus s'))$ would systematically yield $2^n$ terms. To avoid this exponential explosion, we noticed that the result can be written as $((t : t_1 \ldots \setminus t_n, s) [ (t & t_1, s \setminus s_1) \ldots (t & t_n, s \setminus s_n) ]$ 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 it is 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 [2] for Boolean algorithms.

Pattern-matching compilation, type-driven optimizations.

CDuce typing 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:

$$\text{fun } (\rightarrow A & B) \rightarrow \text{int}$$

$$\text{where } \rightarrow A \rightarrow 0 \land \rightarrow B \rightarrow 1$$

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$>$ 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 $A = \rightarrow A [...]$ and $B = \rightarrow B[...]$, looking at the head tag is sufficient, so the match could be compiled as $\rightarrow A \rightarrow _\text{int} > 0 \land \rightarrow B \rightarrow 1$. 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 au-
tomato 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 \([t_1, t_2]\) \(\rightarrow [s_1, s_2]\) 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 scheme 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 Duce, 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 Duce, character strings are conceptually sequences of Unicode characters: the value \('[AB]<>[]'\) is represented as \(\text{Pair}(\text{Char} \ 65, \text{Pair}(\text{Char} \ 66, \text{Xml}(...)\))\). 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, 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 \(\text{String}(0,"AB", \text{Xml}(...)\)\); when this value is inspected by a pattern, it is seen as \(\text{Pair}(\text{Char} \ 65, \text{String}(1,"AB", \text{Xml}(...)\))\) (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 Duce’s performance and validate our type-driven approach for compiling pattern matching. In this section, we present a comparison between Duce and an XSLT processor (the xsltproc program from Gnome libxml 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 Duce 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 addbook is a simple filtering of flat XML documents. In order to test the effectiveness of Duce type-driven optimization we performed some auto-benchmarking by testing different Duce programs implementing the same transformation. In particular we considered two different versions of the transformation addbook, 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.

This allows Duce to use a “standard” pattern \(<\text{person gender=g}\>\ [<\text{name}>n <\text{children}>c]\). The second version one uses the hand-optimized pattern \(<\text{person gender=g}\>\ [<\text{name}>n <\text{children}>c]\). The third Duce 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).

<table>
<thead>
<tr>
<th>split</th>
<th>60Kb</th>
<th>0.3 Mb</th>
<th>0.6 Mb</th>
<th>2.5 Mb</th>
<th>5.2 Mb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duce1</td>
<td>0.10</td>
<td>0.30</td>
<td>0.52</td>
<td>1.92</td>
<td>3.95</td>
</tr>
<tr>
<td>Duce2</td>
<td>0.11</td>
<td>0.30</td>
<td>0.50</td>
<td>1.92</td>
<td>3.92</td>
</tr>
<tr>
<td>Duce3</td>
<td>0.10</td>
<td>0.29</td>
<td>0.49</td>
<td>1.85</td>
<td>3.81</td>
</tr>
<tr>
<td>XSLT1</td>
<td>0.15</td>
<td>0.79</td>
<td>1.42</td>
<td>5.95</td>
<td>12.85</td>
</tr>
<tr>
<td>XSLT2</td>
<td>0.18</td>
<td>0.93</td>
<td>1.68</td>
<td>6.90</td>
<td>14.33</td>
</tr>
</tbody>
</table>

Although preliminary, these benchmarks already allow us to draw some conclusions. First of all, Duce exhibits good performances: Duce 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 Duce programs will remove interpretive 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 Duce program to avoid parsing/validating/printing of intermediate XML documents.

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

---

1. The test machine was an Athlon 750 with 128 Mbytes of RAM. The code of all benchmarks is available at http://www.duce.org/bench.html.
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 [7]. 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 Schemata data model).

We are also implementing the query language we hinted at in Section 4.2, developing simple logical optimizations and benchmarking it against the Bell-labs’ XQuery implementation [7] 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 4.2. A new let fun declaration for add would hide the older one. We are experimenting with a specializing declaration

```haskell
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

```haskell
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 [7]; 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 inter-actions 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

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
- \( c ::= n \) a signed integer
- \( a ::= 'a' \) a unicode literal

Atoms
- \( a ::= 'X | x \)

Base types
- \( B ::= \) String | Int | Char | Bool

Singleton types
- \( S ::= a | c \)

Types
- \( T ::= B \)
- \( S \)
- \( T | T \)
- \( T \& T \)
- \( T \setminus T \)
- \( T \to T \)
- \( (T T) T \)
- \( T \ell (T) \)
- \( X \)
- \( T \) where \( X = T \) and ... and \( X = T \)
- \( \ell (T) \)
- \( \ell (T) T \)
- \( T \ell (T) \)
- \( [T] \)
- \( l \cdot l \)
- \( Empty \)
- \( Any \)
- \( - \)

Attribute list
- \( \ell (a) ::= \)
- \( x = a \)
- \( x =? a \)
- \( \ell ; \ell \)

Interval limit
- \( l ::= c | \ast \)

Type regular expressions
- \( R ::= T | R \bar{R} | R \bar{R} | R \bar{R} p\)

Pattern regular expressions
- \( r ::= p | (x \geq r) | r \bar{r} | r \bar{r} \bar{r} | r \bar{r} \bar{r} \bar{r} \)

Sequence content
- \( s ::= \}
- \( e | e ! e | s s \)

Patterns
- \( p ::= x \)
- \( T \)
- \( X \)
- \( p \& p \)
- \( p | p \)
- \( (p p) p \)
- \( (\ell (p) p) \)
- \( (\ell (p)) \)
- \( [r] \)
- \( (x = c) \)
- \( p \) where \( X = p \) and ... and \( X = p \)

Functions
- \( f ::= \) fun \( x : \) p | T | T | ... ; T | T \)
- \( \) p \to e \)

Expressions
- \( e ::= e \)
- \( a \)
- \( x \)
- \( f \)
- \( e_1 e_2 \)
- \( (e, e) e \)
- \( (e, l(e)) e \)
- \( e \alpha p e \)
- \( \ell (e) \)
- \( \ell (e) \)
- \( e x \)
- \( e \setminus x \)
- \( [s] \)
- \( e T \)
- \( (e : T) \)
- \( T \)
- \( let p = e \) in e
- \( let p : T = e \) in e
- \( if e then e else e \)
- \( if _ { \text { then } _ { \text { else } } } e \)
- \( raise e \)
- \( try e \) with \( p \to e \) ... \( p \to e \)
- \( try e \) with \( p \to e \) ... \( p \to e \)
- \( match e \) with \( p \to e \) ... \( p \to e \)
- \( map e \) with \( p \to e \) ... \( p \to e \)
- \( transform e \) with \( p \to e \) ... \( p \to e \)
- \( x transform e \) with \( p \to e \) ... \( p \to e \)

Prefix operators
- \( o ::= \) load_xml | load_html | load_file | load_file_utf8
- \( \) print | print_xml | print_xml_utf8
- \( dump_to_file | dump_to_xml_utf8
- \( \) int_of | string_of | atom_of | flatten

Infix operators
- \( ap ::= \) + | * | \- | div | mod
- \( = | \langle \rangle | \langle \rangle | \langle \rangle | \langle \rangle |

Postfix operators
- \( po ::= ? | + | * | \? | \? | \? | \? |

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

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