Typed Iterators for XML *

Giuseppe Castagna
PPS (CNRS) - Université Paris 7 - Paris, France

Kim Nguyễn
LRI - Université Paris-Sud 11 - Orsay, France

Abstract. XML transformations are very sensitive to types: XML types describe the tags and attributes of XML elements as well as the number, kind, and order of their sub-elements. Therefore, operations on XML documents are performed by iterators that, to be useful, need to be typed by a kind of polymorphism that goes beyond what currently exists. For this reason these iterators are not programmed but, rather, hard-coded in the languages. However, this approach soon reaches its limits, as the hard-coded iterators cannot cover fairly standard usage scenarios.

As a solution to this problem we propose a generic language to define iterators for XML data. This language can either be used as a compilation target (e.g., for XPATH) or it can be grafted on any statically typed host programming language (as long as this has product types) to endow it with XML processing capabilities. We show that our language mostly offers the required degree of polymorphism, study its formal properties, and show its expressiveness and practical impact by providing several usage examples and encodings.

1. Introduction

Research on programming languages to process XML documents is very active. Since the XML specification is essentially typed, an important part of this research area is characterised by the fact of “taking types seriously”. This translates into coding XML transformations by using mostly (but not exclusively, e.g. Xtaic [13]) functional languages in which the use of types is pervasive and goes well beyond the customary partial correctness check: in these languages types are used to select and sieveh data [3, 16, 13], to speed up execution time [8, 20], to optimise code [4] and memory usage [2]. All this is possible because XML transformations are very sensitive to types. XML types (e.g., [7, 30, 25, 17]) describe the tags and attributes of XML elements, as well as the number, kind, and order of their sub-elements. Thus even basic operations, such as changing a tag, renaming an attribute, or adding an element, may imply conspicuous changes from the type of the input documents to the type of the output documents. Such changes may be nested deep inside the structure of documents, which is why good precision of static type checking and/or of type inference is very hard to achieve.

As an example, consider an operation as simple as capitalising a tag, a predefined operator that iterates a given expression on an XML tree is available in many languages (e.g. xtransform in CDuce, map in XDuce, iterate in Xtaic,...), for slightly more complex—but fairly standard—manipulations (e.g. context sensitive document pruning, or the cleaning of XHTML documents to cope with “XHTML-deprecated” elements) this is not the case. Since language designers cannot hard-code in the language as many iterators as needed, the programmer is then left with the sole choice of writing functions specifically typed for a single usage, thus losing the benefits of modularity and code reuse.

The solution to this problem we propose here is to offer the programmer a restricted language, powerful enough to write complex iterators and simple enough to type them precisely. That is, a language of non first-class operators which are not typed (or are just lightly typed) at their definition but, rather, are very precisely typed at the places of their application. This restricted language will then be embedded in an host language and provide it with user-defined iterators.

Several formalisms to define iterators over XML data structures can be found in the literature (see the §6.3 on Related Work). In this work we present a novel solution directly inspired from Hosoya’s regular expression patterns [16] that were later refined by CDuce patterns [10]. Regular expression patterns allow programs to explore and capture sub-parts of an XML tree at an arbitrary depth. Therefore the idea is that if we generalise patterns so that during the exploration they can execute expressions on the explored sub-trees, then we obtain a very simple and compact language to define iterators on XML data structures, iterators that we dub filters.

Although the idea is simple the definition and design of the iterator language is not. In order to fit the usage scenarios we...
outlined above, a language designed to define iterators for different host languages must satisfy precise characteristics.

1. It must be able to call any expression of the host language and therefore its design must be independent from a particular host language.

2. It must be statically typed. This has two consequences on the type system which must be able (i) to associate a domain type to each iterator, that is a set of expressions for which the iterator will not fail (so that, say, an iterator for lists cannot be applied to an XML tree) and (ii) it must be able to deduce a precise type for the output by running the iterator on the type of the input.

3. A consequence of (ii) in the previous point is that the language must define only iterators that always terminate. More precisely, the abstract execution of any iterator on an (input) type — thus the type checking phase — is required to terminate (therefore the application of an iterator to some data may diverge only either because it called a diverging expression of the host language or because it was applied to infinite data).

4. It must be expressive enough to define common sequence and tree operators such as concatenation, reversal, map-functions, various tree-extractions, XPATH expressions, and so on.

5. It must not come at the cost of modularity and code reuse.

Of course there is a clear tension between requirements 3 and 4: expressiveness and termination are contrasting requirements, therefore a trade-off must be found between them. This yields us to one of the main technical problems of this work. In all XML programming languages and proposed standards currently available, XML types essentially are regular trees. We consider as a minimal requirement for a language of iterators for XML to be able to define an expression, say, leaves which extracts all the leaves of a tree. If we accept this minimal requirement, then we must also accept the fact that it is impossible to infer the most precise type for the result of an iterator. To see why, consider the following declarations:

```
type A = <a>[] | [ A T B ]
type B = <b>[]
type T = [] | [ A T B ]
```

which define three types: A and B which type single elements of tag &lt;a&gt; and &lt;b&gt; respectively, that enclose an empty sequence of elements (we use square brackets to denote and delimit sequences, and the content of sequences is described by regular expressions on types); T which types either the empty sequence or (i.e. the vertical bar) sequences of three elements, the first element being of type A, the third of type B and the second a sequence of type T itself. Note that T is regular: its only sub-types are A, B, T, and []. However if we apply to a value of this type the iterator that returns the sequence of the leaves of a tree, then the precise type of the result is \{ [A*B*] | n\geq0 \}, which is not regular. Of course there are regular approximations of this type such as \{ [A*] (B*], or \{ [B*] (A*], or \{ [A*] (B*] | B\}. etc., but there is not a most precise or principal one (it is easy to build an infinite sequence of regular approximations of increasing precision, whose limit is \{ [A*B*] | n\geq0 \}). Our solution is to let the programmer decide which approximation to use, by providing explicit type annotations. We study when such annotations are necessary and make the type-checker use them just in those cases. The study results in the design of a concrete syntax and semantics for a language to define iterators for different host languages. In order to maximise modularity and code reuse (requirement 5), we designed the (concrete) language so that annotations are specified at the application of an iterator rather than at its definition. This allows one to declare filters in a separate library. At the time of application the programmer will add necessary annotations (if needed) which will thus tailor the filter for the specific type of the argument. Modularity is also improved by the addition of parametric filters which are obtained by a technique similar to Wadler’s higher-order macros [29].

More generally, this paper promotes a somewhat radical and unorthodox approach in which the static typing of highly modular/polymorphic code is delayed at the place of its application. This has some clear drawbacks (e.g. a late detection of errors) but it allows a very rich and precise typing, which is a key issue in the manipulation of XML documents. The final result is an iterator language that can be grafted on any statically typed host language (as long as it possesses product types) in order to supply it with, for instance, Hosoya’s regular expression filters (which are XDuce’s iterators), all XDuce’s iterators, precisely-typed operators on heterogeneous sequences, forward XPATH expressions (i.e. only with child and descendant axes) as well as XSLT-like transformations. We implemented filters using XDuce as host language (the implementation uses the same language used for the XDuce compiler, that is OCaml), and the resulting prototype constitutes —in our ken—the first practical implementation of highly polymorphic transformations tested on realistic usage scenarios with non-trivial data-types and applications: comparable approaches (see §6.3) lack usable implementations.

**Outline**. The presentation proceeds as follows. In Section 2 we briefly introduce our language of filters by commenting a couple of practical examples. In Section 3 we start the formal presentation by defining the syntax and operational semantics of a calculus of filters. Section 4 is devoted to the presentation of the type system and of its properties. We address algorithmic issues in Section 5, where we define a typing algorithm for filters which is sound and complete with the type system “up to annotations”: provided that some correct type annotations are given, the algorithm types every typable filter. We also study annotations and precisely point out where they are needed by giving sufficient condition for the success of the algorithm on typable filters. Finally, in Section 6 we formally define the concrete syntax of a language derived from the calculus of Section 3 and demonstrate its use by defining and commenting several examples. Section 6.3 discusses related work. We conclude our presentation in Section 7 where we sketch some directions for future works.

Due to space constraints, proofs are omitted in this presentation but can be found in the second author’s PhD dissertation at [http://www.lri.fr/~kn/iterXML/](http://www.lri.fr/~kn/iterXML/), together with an implementation of the language presented in Section 6.

2. An overview of filters

As we already mentioned, filters can be seen as an extension of pattern matching: as patterns are matched against values to retrieve part of an input value, we address algorithmic issues in Section 5, where we define a typing algorithm for filters which is sound and complete with the type system “up to annotations”: provided that some correct type annotations are given, the algorithm types every typable filter. We also study annotations and precisely point out where they are needed by giving sufficient condition for the success of the algorithm on typable filters. Finally, in Section 6 we formally define the concrete syntax of a language derived from the calculus of Section 3 and demonstrate its use by defining and commenting several examples. Section 6.3 discusses related work. We conclude our presentation in Section 7 where we sketch some directions for future works.

Due to space constraints, proofs are omitted in this presentation but can be found in the second author’s PhD dissertation at [http://www.lri.fr/~kn/iterXML/](http://www.lri.fr/~kn/iterXML/), together with an implementation of the language presented in Section 6.

2. An overview of filters

As we already mentioned, filters can be seen as an extension of pattern matching: as patterns are matched against values to retrieve part of an input value, we address algorithmic issues in Section 5, where we define a typing algorithm for filters which is sound and complete with the type system “up to annotations”: provided that some correct type annotations are given, the algorithm types every typable filter. We also study annotations and precisely point out where they are needed by giving sufficient condition for the success of the algorithm on typable filters. Finally, in Section 6 we formally define the concrete syntax of a language derived from the calculus of Section 3 and demonstrate its use by defining and commenting several examples. Section 6.3 discusses related work. We conclude our presentation in Section 7 where we sketch some directions for future works.

Due to space constraints, proofs are omitted in this presentation but can be found in the second author’s PhD dissertation at [http://www.lri.fr/~kn/iterXML/](http://www.lri.fr/~kn/iterXML/), together with an implementation of the language presented in Section 6.
What we propose is to generalize such a technique by transforming the constructions of the form \( p \rightarrow e \) into first class expressions so that they can be nested, denoted by variables, structured by using the pattern constructors (e.g. the vertical bar, the Kleene star, or the pair constructor), and composed by semicolons.

We show our idea on a representative example: list concatenation. Let us encode lists à la Lisp, that is as nested pairs with a special constant ‘nil’ to denote the empty list. A filter that concatenates two lists can be written as follows:

\[
\text{let filter} ~ \text{concat} = (x, y) \rightarrow (x; \text{let filter} \ aux = \text{‘nil’} \rightarrow y \mid (\text{head}+\text{head}, \text{aux}) \text{ in aux})
\]

To enhance readability we have written keywords and constants in typewriter font, variables that denote filters are underlined and in roman style, while capture variables (and, in Section 6, function names) are written in italics.

The definition of \text{concat} introduces most of the syntax of filters. A filter is: either a pattern filter of the form \( p \rightarrow f \) (e.g. the body of \text{concat}) that when applied to a value matches it against the pattern \( p \) and if this does not fail, then applies the filter \( f \) in the environment returned by the matching of \( p \); or a pair filter \( (f_1, f_2) \) (e.g. the last branch of \text{aux}) that succeeds only if it is applied to a pair of values, it applies each filter \( f_1 \) to the respective value, and returns the pair formed by the results of the two applications; or a union filter \( f_1 | f_2 \) (e.g. the body of \text{aux}) which applies \( f_2 \) on the argument value only if the application of \( f_1 \) failed; or a composition filter \( f_1 \circ f_2 \) (e.g. the right hand filter of the body of \text{concat}) which when applied to a given value, first applies \( f_1 \) to this value and then \( f_2 \) to the result of the previous application; finally, filters can also be either an expression of the host language (in our example the rightmost occurrence of the variables \( x, y, \) and \( \text{head} \)), or a recursive filter declaration \text{let filter} \( x \rightarrow f \).

The behaviour of the filter above is equivalent to the following recursive function (given in pseudo-ML, where we use the same typesetting conventions):

\[
\text{let } \text{concat} \ (x, y) = \\text{let rec aux z} = \text{match } z \text{ with} \text{'nil'} \rightarrow y \mid (\text{head}+\text{tail}, \text{aux}) \text{ in aux } x
\]

The definition of the filter \text{concat} can be understood by referring to its ML equivalent. First, the arguments (the pair of the two lists one wants to concatenate) are bound to two variables \( (x \text{ and } y) \) via the pattern \( (x, y) \). Then, a second recursive filter \text{aux} is defined and applied (via the composition operator \( ; \)) to the result of the expression \( \text{‘nil’} \rightarrow y \), that is to the first list. Note the similarity between \( x; \text{let filter} \ aux = \ldots \) and \text{let rec aux } \( z = \ldots \text{in aux } x \). The recursive filter \text{aux} is the union (“∪”) of two filters, playing the same role as two branches of a pattern matching. If the argument is the constant \text{‘nil’}, (this case is handled by the pattern filter \( \text{‘nil’} \rightarrow y \)), then the second list \( y \) is returned. If the argument is a pair, (handled by the pair filter \( (\cdot, \cdot) \)), then the first component is left unchanged (by the use of an identity filter: \( \text{head}+\text{head} \)) while the second component, which is the tail of the list, is recursively iterated over by the \text{aux} filter. The result of each component is then recomposed as a pair.

Let us now consider the type analysis by first trying to type the ML function:

• In a type system with parametric polymorphism à la ML, this function has type: \( \forall \alpha. \alpha \text{ list } \times \alpha \text{ list } \rightarrow \alpha \text{ list } \). The constraint is that both arguments must have the same type and the result will be of that type. This precludes the use of such a typing discipline with heterogeneous lists: even in the presence of sub-typing, the type of the elements of the result would be crushed to the least upper bound of all element types, losing in this way any precision. This does not fit XML document processing where heterogeneous sequences (of elements) are pervasive.

  • In a type system with regular expression types, the most general type of the function would be: \( \text{Any*} \times \text{Any*} \rightarrow \text{Any*} \) (\( \text{Any} \)) being the super type of all types). Again, precision is lost because while any lists is accepted as argument (thanks to sub-typing its type is “up-casted” to \( \text{Any*} \)), the output type is uninformative about the type of the elements of the result.

For filters we use a rather different type discipline. When we define a filter, we do not try to characterise the type of all its possible results. Instead, we just check that there exists some input type for which the application of the filter cannot yield a type error or a failure. For instance consider the filter \text{concat}. It is easy to see that the subfilter \text{aux} will not fail as long as it is applied to a list. Therefore the composition in \text{concat} will work if \( z \) is bound to a list. From this we deduce that \text{concat} will not fail as long as its first argument is a list. Thus there exists an input type which ensures a safe application of \text{concat}.

Precision of type inference is achieved by using for \text{concat} the same typing policy as the one used for the hard-coded concatenation operator \( \& \) of XDuce or C.Duce. That is, instead of typing the operator, one types each single application of the operator. In terms of the filter \text{concat} this corresponds to type the application \text{concat}(l_1, l_2) for some specific expressions \( l_1 \) and \( l_2 \). This allows us to achieve very precise typing. For example, if \( l_1 \) has type \( \text{String Bool*} \) and \( l_2 \) has type \( \text{Bool Int?} \), then the type system infers for the result the type \( \text{String Bool* Int?} \), in this case, is the most precise one. If \text{concat} is applied elsewhere to an input of different type, then the output type is again computed from the specific input type and a precise type is given to the whole application.

We complete this overview by an example of XML transformation. To that end we add to the filters presented so far the filter \( f_1 \circ f_2 \circ f_1 \) that accepts XML elements as input, applies the sub-filter \( f_1 \) on the element tag, \( f_2 \) on the element attributes and \( f_3 \) on the sequence of children. We will often omit \( f_2 \) to ease the reading of the examples, in which case the attributes are copied to the result.

Consider Figure 1. We want to convert the file recipe.xml, whose content is sketched at the top of the figure, into an XHTML document to publish on a website. Below it, we see first the type declaration for such a document, then the definition of two mutually recursive filters. The first one, map_elem, is the union of two filters. If its argument is an XML element (character for example), then it is just copied to the output. The map_list filter is just an iterator over sequences which calls map_elem on each element of the input sequence. The last part is the special construct apply \( f \) to \( e \) which applies the filter \( f \) on the result of an expression \( e \). Here, the expression is the document, returned by the built-in load_xml function. This document is fed to a filter which changes the root tag \( <\text{recipe}> \) to \( <\text{html}> \), extracts the content of the root tag (by the pattern \( x \rightarrow \ldots \)), and rebuilds a new element \( <\text{body}> \) whose content is the application of map_list to the variable \( x \).

It is worth noticing that precise typing is achieved without resorting to any explicit type annotation. This precision is obtained in the typing of the apply to construction: in our example the input has type Recipe, thus the type system abstractly executes the filter on it and deduces for the result the type:

\( \text{Recipe} \rightarrow \text{Recipe} \)
produced by the following grammar:

### Definition 1

of filters and of their operational semantics.

and patterns as defined in [5] and [10], followed by the definition

Had we applied the filter to more a complex expression in which

with two additional requirements:

b

We use

X

Section 5). The contractivity condition rules out meaningless terms

\[\text{Types of the data:}\]

\[
\text{type Itemize = } \langle \text{itemize} \rangle \ [\text{Item+} ]
\]

\[
\text{type Enumerate = } \langle \text{enumerate} \rangle \ [\text{Item+} ]
\]

\[
\text{type Item = } \langle \text{item} \rangle \ [\text{Char*} ]
\]

The types of the data:

\[
\text{type Item} = \langle \text{item}\rangle [\text{Char*} ]
\]

\[
\text{type Enumerate} = \langle \text{enumerate}\rangle[\text{Item+}]
\]

\[
\text{type Recipe} = \langle \text{recipe} \rangle \ [\text{Itemize Enumerate}]
\]

### Definition 2

A pattern is a (possibly infinite) term produced by the following grammar

\[
p ::= x \mid t \mid (p_1 \land p_2) \mid p_1 \lor p_2 \mid p_1 \xor p_2.
\]

\[\text{that is regular, conjunctive (as in Definition 1), and in which every}\]

\[\text{subtree of the form } p_1 \land p_2 \text{ or } p_1 \lor p_2 \text{ satisfies }
\]

\[\text{Sat}(p_1) \cap \text{Sat}(p_2) = \emptyset\]

\[\text{and } \text{Sat}(p_1) = \text{Sat}(p_2), \text{ respectively (where } \text{Sat}(p)\text{ is the set of}\]

\[\text{capture variables occurring in } p).\]

The semantics of both types and patterns is expressed in terms of

\[\text{values. In the framework of XML processing languages, values are }\]

\[\text{XML documents and, following Hosoya et al. [18], an XML}\]

\[\text{type is (interpreted as) the set of XML documents that have}\]

\[\text{that type. In this paper we do not fix a particular set of values (since}\]

\[\text{it depends on the host language filters are used in) but we rather}\]

\[\text{suppose its existence and implicitly assume that it contains all}\]

\[\text{XML documents. Then we consider a type as the set of values}\]

\[\text{that have that type, the union, intersection, and negation types as}\]

\[\text{the corresponding set-theoretic operations, and the subtyping relation,}\]

\[\text{noted } \preceq, \text{ as set-containment. Since the use of subsumption}\]

\[\text{makes two equivalent types (that is, two types denoting the same set of}\]

\[\text{values) operationally indistinguishable, then we will always work}\]

\[\text{up to type equivalence and consider, e.g. } t\|t, \text{ Any} \& t \text{ and } t\]

\[\text{as the same type.}\]

The semantics of patterns is defined in terms of matching. Informally,

\[\text{the matching of a value } v \text{ against a pattern } p, \text{ that we note}\]

\[v/p,\text{ is either a failure (noted } ?) \text{ or a substitution from the}\]

\[\text{capture variables of } p \text{ to values. The substitution is then used as an}\]

\[\text{environment in which some expression is evaluated. If the pattern is a}\]

\[\text{type, then the matching fails if and only if the pattern is matched against}\]

\[\text{a value that has not that type. If it is a variable, then the matching}\]

\[\text{fails if and only if the pattern is matched against a value that}\]

\[\text{has not that type. If it is a variable, then the matching}\]

\[\text{always succeeds and returns the substitution that assigns the}\]

\[\text{matched value to the variable. The pair pattern } (p_1, p_2)\text{ succeeds if and only}\]

\[\text{if it is matched against a pair of values and each sub-pattern}\]

\[\text{succeeds on the corresponding projection of the value (the union of the}\]

\[\text{two substitutions is then returned). An intersection pattern } p_1 \& p_2\]

\[\text{succeeds if and only if both patterns succeed (the union of the two}\]

\[\text{substitutions is then returned). The union pattern } p_1 \lor p_2\text{ first tries to}\]

\[\text{match the pattern } p_1 \text{ and if it fails it tries the pattern } p_2.\]

For instance the pattern \((\text{Int} \& x,y)\) succeeds only if the matched value

\[\text{is a pair of values } (v_1, v_2) \text{ in which } v_1 \text{ is an integer } — \text{in}\]

\[\text{which case it returns the substitution } \{ x:=v_1, y:=v_2\} — \text{and fails}\]

\[\text{otherwise.}\]
This informal semantics of matching (see [10] for the formal
definition) explains the reasons of the restrictions on capture variables
in Definition 2: in intersections both patterns must be matched
so that they have to assign distinct variables, while in union patterns
just one pattern will be matched, hence the same set of variables
must be assigned, whichever alternative is selected.

Types are sets of values, but of course not every set of values is
a type. However there are some useful sets of values that happen
to be types. These are the sets formed by all and only those values
that make some pattern succeed:

**Theorem 1 (Accepted type [10]).** For all \( p \in \mathbb{P} \), the set of all values \( v \) such that \( v/p \neq \Omega \) is a type. We call this set the accepted
type of \( p \) and note it by \([p]\).

The fact that the exact set of values for which a matching succeeds
is a type is not obvious and is of the utmost importance for a precise
typing of pattern matching. In particular, given a pattern \( p \) and a
type \( t \) contained in \([p]\), it allows us to compute the exact
type of the capture variables of \( p \) when it is matched against a value in \( t \):

**Theorem 2 (Type environment [10]).** There exists an algorithm
that for all \( p \in \mathbb{P} \), and \( t \leq [p] \), returns a type environment
\( t/p \to \text{Types such that } (t/p)(x) = \{(v/p)(x) \mid v : t\}. \)

### 3.2 Filter calculus

**Definition 3 (Filters).** A filter \( f \) is a (possibly infinite) regular tree
coinductively generated by the following production rules (where \( e \) ranges over expressions of the host language)

\[
\begin{align*}
f &::= e \quad \text{expression} \\
    &| p \to f \quad \text{pattern}, p \in \mathbb{P} \\
    &| f_1 ; f_2 \quad \text{composition} \\
    &| (f_1 , f_2) \quad \text{product} \\
    &| f | f \quad \text{union}
\end{align*}
\]

and which satisfies the following conditions:

1. (contractivity) for every infinite branch of \( f \), there the number
   of occurrences of the pair constructor \((\_ , \_ )\) is infinite.
2. (composition) for every subterm \( f' \) of \( f \), if \( f' \) is of the form
   \( f_1 ; f_2 \), then \( f' \) is not a subterm of \( f_2 \).

Here, \( e \) is an expression of the host language. The condition on
contractivity is the usual one which rules out meaningless terms.
The condition on composition is however rather new and involved.
In a nutshell it states that recursion cannot traverse composition
semicorns \( \_ , \_ \). For example, \( f = (f_1 , f_2) g \) with \( g = (g_1 , g_2) x \to x \)

### 3.3 Operational semantics

We define a big step operational semantics for filters and show the
termination of the evaluation of filters \( f \) on a finite value \( v \).

The dynamic semantics is given by the inference rules for the
expression \( \gamma \vdash_e \) (Free variables) We define the set of
\( \mathbb{F} \) and show the

**Definition 5 (Free variables).** We define the set of free variables
of a filter \( f \), \( \mathbb{F}(f) \) as:

\[
\mathbb{F}(f) = \mathbb{V}(f) \cup \mathbb{F}(f_1) \cup \mathbb{F}(f_2)
\]

We suppose that \( \mathbb{F}(f) \) is defined (provided by the host language).

**Figure 2. Dynamic semantics of filters**

\[
\begin{align*}
\Gamma \vdash e(v) \sim r &\quad \text{if } \mathbb{V}(e) \subseteq \mathbb{F}(v) \\
\Gamma \vdash f_1 (v_1) \sim r_1 &\quad \text{if } r_1 \neq \Omega \\
\Gamma \vdash (f_1 , f_2)(v_1 , v_2) \sim r_2 &\quad \text{if } r_2 \neq \Omega \\
\Gamma \vdash (f_1 ; f_2)(v_1 , v_2) \sim r_1 &\quad \text{if } r_1 \neq \Omega
\end{align*}
\]
4. Static semantics

4.1 Type system

We present here a type system for filters. We start by extending the notion of accepted type to filters:

**Definition 6 (Accepted type).** For every filter $f \in F$ we define the type $\{f\}$ as follows:

$$
\begin{align*}
\{f\} & = \text{Any} \\
\{f_1 \cup f_2\} & = \{f_1\} \cup \{f_2\} \\
\{f_1 f_2\} & = \{f_1\} \times \{f_2\} \\
\{f_1\} & = \{f_1\}
\end{align*}
$$

Input inclusion in the accepted type of a filter is a necessary condition for filter application to succeed: $\forall v \notin \{f\}, f(v) \sim \Omega$. Unfortunately it is not also sufficient since, for instance, the accepted type of $\text{Any} \rightarrow 3 \cdot 5$ is $\text{Any}$, but every application of this filter fails, since it tries to match 3 against a pair pattern. The problem lies in the composition operator, $f_1 f_2$. Indeed, a necessary condition is that the output type of $f_1$ is a subtype of the input type of $f_2$. To ensure type safety, we need to infer the output type of the filter $f_1$. To that end we define the inference rules of Figure 3 in which we use the notation $\bigvee_{i=1}^n t_i$ as a shorthand for the finite union $t_1 \cdots t_n$.

![Figure 3. Deduction system associated with $\mathcal{F}$](image)

The system proves judgements of the form $\Gamma \vdash f(t) = s$ meaning that in a type environment $\Gamma$ a filter $f$ applied to an expression of type $t$ returns (if any) a value of type $s$. We call $\mathcal{F}$ the associated deduction system and only consider (possibly infinite) regular derivations of this system. Regularity both for filters and for deductions prevents $\Gamma$ from growing indefinitely (regularity of filters guarantees that the number of distinct variables on an infinite branch is finite and regularity of deductions ensures that these variables can be assigned only to finitely many types). However, the system is not algorithmic (that is, there is not a unique derivation for every provable judgement) since neither it is syntax-directed (because of the subsumption rule, the form of the term does not univocally determine which rule to apply) nor does it satisfy the subformula property (the composition rule is a logical cut, thus the pivotal type in the premise do not occur in the conclusion).

Most of the rules require that the input type is compatible with the accepted type of the considered filter. To type an (host language) expression the rule $(t\text{-expr})$ calls the type system of the host language with the current environment. Typing of the union pattern $(t\text{-union})$ is straightforward, since it types each branch for the values that it can be applied to, but only the results of branches that have a chance to be selected (i.e. those for which $t_i$ is not empty) are considered for the result (see filter $\text{mymatch}$ in Section 6 for an example that justifies this discipline and [3] for a detailed discussion). To type the filter pattern $p \rightarrow f$ the system types $f$ under an environment enriched with $t/p$; the latter—intensionally defined by Theorem 2—is the type environment that assigns to each capture variable in $p$ the most precise type that can be deduced for it when the pattern is matched against a value of type $t$ (refer to [10] for formal definition). The subsumption rule $(t\text{-subs})$, allows the system to approximate an intermediary type $s'$ with a broader type $s$. This rule is one of the reasons this type system is not algorithmic.

The difficult rules are those for composition and products. $(t\text{-comp})$ resembles a logical cut since it introduces an intermediary type $t_1$ (which is the other reason why the type system is non-algorithmic). The standard example is the leaves filter and type $\text{T}$ informally discussed in Section 1. There is an infinite number of regular derivations for $\Gamma \vdash \text{leaves}(T) = s$, each one giving a different $s$ with no lower bound (the limit being the context free language: $\{[a^*b^n] \mid n \geq 0\}$). As for the rule for products, it is not so straightforward as one could naively expect: to achieve a precise typing in the presence of union types we must resort to a very subtle and “surgically precise” technique.

The difficulty arises because the only constraint we have on the input type $t$ of a product filter is that $t$ is a subtype of $(\text{Any, Any})$. However this does not imply that $t$ is a product of just two types: $t$ in general is an arbitrary finite union of products. This yields to two original aspects of our approach that, as we argue in Section 6, allow us to achieve very precise typing. First, as we decompose a type in a finite union, we apply the same filter to each type of the decomposition ($f_1$ and $f_2$ in the $(t\text{-prod})$ rule are typed many times, against different input types) and recompose the result in the output type. This already allows us to obtain a fine grained typing of the transformation. The second aspect is the decomposition itself. Indeed, while every subtype of $(\text{Any, Any})$ can be decomposed in a union of products, the decomposition is not unique. However, there exists a decomposition (that we dub maximal product decomposition) given by the operator $\mathcal{G}$ (pronounced $\mid pi: \ intrinsic\ types\ make\ the\ system\ more\ precise\ and\ respect\ typing$). We devote the next section to define it.

4.2 Typing of Cartesian products

Typing Cartesian products can be tricky since not every decomposition of a product in a finite union of Cartesian products behaves equally with respect to the subtyping relation. We study the case of the maximal product decomposition, which is stable with respect to the subtyping relation and provide better typing properties to the filter language. Let us illustrate this with interval types. Consider the following filters (the interval notation $i..j$ being syntactic sugar for the finite union of integers $\{i+i+1+\cdots+j\}$):

$$
f_1 = 0..4 \rightarrow \lambda[s..5.8 \rightarrow B] \\
f_2 = 0..3 \rightarrow c[0..7 \rightarrow D] \\
f = (f_1, f_2)
$$

and the types $t$ and $s$:

$$
t = (0..4, 0..3)(5..8, 0..7) \\
s = (4..6, 1..2)
$$

It is clear that $s \leq t$: by drawing all intervals on a plane, as in Figure 4, it is easy to check that the rectangle $s$ is contained in the “$t$”-shaped $t$. However, $s$ overlaps the two rectangles which form $t$. If we decompose naively (i.e., syntactically) both types and compute the result type of $f$ by separately applying the filter on each com-
ponent of the obtained decomposition, then we have:
\[ \emptyset \vdash f(t) = (A,|C|)[B,D] \]
but also:
\[ \emptyset \vdash f(s) = (A|B|,|C|D) \]
the latter being a supertype of the former. Indeed, in \( f_1 \), a value

\begin{align*}
\text{Lemma 1 (Stability of filtering).} & \quad \text{For every filter } f, \text{ types } s \text{ and } t, \text{ and type environment } \Gamma, \text{ if } s \leq t \text{ and } \Gamma \vdash f(t) = t', \text{ then } \\
& \quad \Gamma \vdash f(s) = s' \text{ and } s' \leq t'.
\end{align*}

As a concluding remark we want to stress that stability is a key property to ensure modularity. If a programmer chooses to refine a type in some existing code, then stability ensures that the result of the computation of any filter on an input of the refined type will be a value in a subtype of the previous output type: the behaviour on the old type is preserved without modifying any piece of code.

We now state the main property of this type system:

\textbf{Theorem 4 (Subject reduction).} Let \( f, t, s \text{ and } t \) be such that \( \Gamma \vdash f(t) = s \). For every \( \gamma : \Gamma \text{ and } \nu : t \), we have that \( \gamma \vdash_e f(\nu) \sim \nu' \text{ implies } \nu' : s \).

It should be noted that even though stability with respect to the type-system is an important property, it is not required to prove subject reduction. Hence, for any decomposition of a product in a finite union of Cartesian products, the language is type-safe.

\section{Typing algorithm}

\subsection{Presentation}

In the previous section we presented a type system for the filter algebra which enjoys the desired properties of type safety and precision. However, in its present state, the system does not translate directly into a typing algorithm. In fact, for some input types and particular filters, there exists an infinite number of valid regular derivations in the set \( \mathcal{F} \). To have an effective language, we need to turn this set of rules into an algorithm. We obtain it by adding type annotations to recursive filters. We claim that in many useful cases such annotations are not needed (mainly all map-like filters), while for other cases (e.g., tree leaves extraction), these annotations make it possible to type a filter for which there is no regular output type. We will then show that this algorithm is sound and complete with respect to the type-system.

Since the algorithm needs to work on finite representations of (possibly infinite) regular types, we use the classic "µ" notation to exploit the recursive binder. µ-types are inductively generated by the following grammar:

\[
\tau ::= \mu\alpha.\tau \mid \alpha \mid b \mid \tau_1 \cdot \tau_2 \mid \tau_1 \& \tau_2 \mid \tau \Rightarrow \emptyset \mid \text{Any}
\]

We use Greek letters \( \tau, \sigma \) to range over \( \mu \)-types and to distinguish them from regular tree types; recursion variables are ranged over by \( \alpha, \beta, \ldots \). Contractivity translates into requiring that every occurrence of a type variable is separated from its binder by at least one product constructor. Regular trees and explicit binders are two equivalent representations for types (cf. \cite{12, 6}). It is well-known that every recursive term ("µ-term") represents a recursive tree and, conversely, we can choose a canonical µ-term that represents a regular tree.

\textbf{Definition 8 (Infinite expansion).} Given a recursive term \( \tau \) we note \([\tau]_\infty\) its infinite expansion.

\textbf{Definition 9 (Recursive folding).} Given a regular tree \( t \) we note \([t]_\mu\) the equivalent recursive term with the least number of variables.

We extend \([\cdot]_\mu\) and \([\cdot]_\infty\) to typing environments in a straightforward way by applying the aforementioned functions to each type in the image of the environment. These two functions ensure that \( \tau/p \), the maximal product decomposition, and the subtyping relation are well-defined for \( \mu \)-types, as well. We extend the definition of filters with annotations:

\textbf{Definition 10 (Annotated filters).}
\[
f ::= e \mid p \Rightarrow f \mid f; f \mid (f, f) \mid f/f \quad \text{unchanged annotation}
\]
### Structural rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| (a-expr) | $\text{type}_\text{af}(\Gamma, \sigma) = \sigma$  
$\Gamma, \Delta \vdash_{\text{af}} e(\tau) = \sigma$  
\[ \frac{ i = 1, \text{rank}(\tau) \quad j = 1, 2 }{ \Gamma, \Delta \vdash_{\text{af}} f_i(f_j(\tau)) = \sigma_j } \]  
| (a-prod) | $\prod$ \text{ and more precisely, fails due to a lack of annotations (or to incorrect } \text{annot} \text{ions required are minimal, that is if the programmer does not have to “guess too much”}. We now formalise the algorithm in order to not cluter it with tedious backtracking rules and environments. The order of application of \textit{the rules} is the following: one must apply a memoization rule \textit{(if possible)} before a structural rule, and a memoization rule must be followed by a structural rule \textit{(if the rule is not terminal)}. This order \textit{of evaluation} is important, since it ensures the termination of the algorithm. The only rule \textit{which requires some attention}, since it does not derive directly from the non-\text{algorithmic} type system \textit{is the (a-annot) rule which cope with all the cases that made the type system behave non-\text{algorithmically}. Recall that there are two such cases. The first is the subsumption rule (t-sub), where the system “guesses” the output super-type. This is reflected in the (a-annot) rule by checking that the output type of the filter is a subtype of the annotation $\sigma$. Secondly, and most importantly, in rule (t-comp) the intermediary type of the composition is also guessed. As we discuss hereafter, we require that in that case, the left-hand side filter of a composition is annotated. To be complete, it should be noted that, given a derivation in the non \text{algorithmic} type system, every instance of the rule (t-sub) can be erased, except for those occurring at the left-hand side of a composition. This situation is akin to the one of the simply typed lambda calculus with subtyping, where every use of the subsumption rule can be erased, but those used to type the application of a function.  

### Memoization rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| (a-base-rec) | $(f, \tau, \sigma) \in \Delta$  
$\Gamma, \Delta \vdash_{\text{af}} f(\tau) = \sigma$  
\[ \frac{ (f, \mu \alpha, \tau, \beta) \notin \Delta \quad \alpha \text{ and } \beta \text{ fresh var.} }{ \Gamma, \Delta \cup \{(f, \mu \alpha, \tau, \beta)\} \vdash_{\text{af}} f(\tau[\alpha \leftarrow \mu \alpha \tau]) = \sigma } \]  
| (a-unfold-rec) | $\text{choose}(E)$ and $(f, \tau, \sigma) \notin \Delta$  
$\Gamma, \Delta \vdash_{\text{af}} f(\tau) = \sigma$  
\[ \frac{ \text{choose}(E) \quad \sigma = \text{choose}(E) }{ \Gamma, \Delta \vdash_{\text{af}} f(\tau) = \sigma } \]  
| (a-unfold-non-rec) | $\text{choose}(E)$ and $(f, \tau, \sigma) \notin \Delta$  
$\Gamma, \Delta \vdash_{\text{af}} f(\tau) = \sigma$  
\[ \frac{ \text{choose}(E) \quad \sigma = \text{choose}(E) }{ \Gamma, \Delta \vdash_{\text{af}} f(\tau) = \sigma } \]  
| (a-annot) | $\text{choose}(E)$ and $(f, \tau, \sigma) \notin \Delta$  
$\Gamma, \Delta \vdash_{\text{af}} f(\tau) = \sigma$  
\[ \frac{ \text{choose}(E) \quad \sigma = \text{choose}(E) }{ \Gamma, \Delta \vdash_{\text{af}} f(\tau) = \sigma } \]  

---

### Figure 5. Deduction system associated with $T_{af}$

An annotation is a set $E$ of $(\mu)$-types in which the algorithm will pick an output type for the annotated filter. The algorithm is described in Figure 5 as a set of deduction rules for the judgement $\Gamma, \Delta \vdash_{\text{af}} f(\tau) = \sigma$, where $\Gamma$ denotes a type environment for pattern variables and $\Delta$ is a memoization environment (which ensures the termination of the algorithm), that is, a set of triples $(\sigma, \tau, \alpha)$ where $\tau$ is a filter and $\tau$ and $\sigma$ types (intuitively, they respectively are an input and an output type). We assume that the choose() function in rule (a-annot) always chooses the right type in the annotation set if it exists. In practice, this is implemented by backtracking, the algorithm trying all the annotations one after the other until a valid one is found (or a type error is raised). We chose to hide this aspect of the algorithm in order not to clutter it with tedious backtracking rules and environments. The order of application of \textit{the rules} is the following: one must apply a memoization rule \textit{(if possible)} before a structural rule, and a memoization rule must be followed by a structural rule \textit{(if the rule is not terminal)}. This order \textit{of evaluation} is important, since it ensures the termination of the algorithm. The only rule \textit{which requires some attention}, since it does not derive directly from the non-\text{algorithmic} type system \textit{is the (a-annot) rule which cope with all the cases that made the type system behave non-\text{algorithmically}. Recall that there are two such cases. The first is the subsumption rule (t-sub), where the system “guesses” the output super-type. This is reflected in the (a-annot) rule by checking that the output type of the filter is a subtype of the annotation $\sigma$. Secondly, and most importantly, in rule (t-comp) the intermediary type of the composition is also guessed. As we discuss hereafter, we require that in that case, the left-hand side filter of a composition is annotated. To be complete, it should be noted that, given a derivation in the non \text{algorithmic} type system, every instance of the rule (t-sub) can be erased, except for those occurring at the left-hand side of a composition. This situation is akin to the one of the simply typed lambda calculus with subtyping, where every use of the subsumption rule can be erased, but those used to type the application of a function.

### 5.2 Properties

Termination and soundness of the algorithm are both straightforward to state (and prove):

**Theorem 5** (Termination of the typing algorithm). \textit{For all filters $f$ and types $\tau$, the typing algorithm for $f(\tau)$ terminates.}

**Theorem 6** (Soundness of the typing algorithm). \textit{For all $\Gamma, \Delta, f, \tau$, and $\sigma$, if $\Gamma, \Delta \vdash_{\text{af}} f(\tau) = \sigma$, then $\Gamma \vdash f([\tau]_\infty) = [\sigma]_\infty$.}

Showing completeness is more challenging, though. Indeed, we have seen that some filters do not have a unique output type for a given input type and that, in such cases, the filter must be annotated for the algorithm to succeed. The notion of completeness we choose is the following. Let us consider $\Gamma, f, t$. If there exists a regular type $s$ (and hence a regular derivation) such that $\Gamma \vdash f(t) = s$, and if we annotate $f$ with types coming from the derivation of this judgement, then the algorithm finds a type such that: $[\Gamma]_\mu, \emptyset \vdash_{\text{af}} f([t]_\mu) = [s]_\mu$ (the algorithm works on explicit recursive types instead of regular trees for the system, hence the $[.]_\mu$). Informally, we state that if we “guide” the algorithm in the good direction, it will find the expected type. Of course such an algorithm is useful in practice only if the annotations required are minimal, that is if the programmer does not have to “guess too much”. We now formalise all these notions and state the completeness theorem. We proceed in three steps. First, we highlight the cases where the algorithm fails, and more precisely, fails due to a lack of annotations (or to incorrect annotations). Then, we give a sufficient condition on annotations such that a filter annotated in this way can either be typed or be detected as ill-typed. Finally, we state the completeness theorem, ensuring that if a filter is well typed in the type system, with respect to a certain input type $t$, then the algorithm succeeds in typing the filter, provided that the latter is sufficiently annotated. Let us start by pinpointing the cases where the algorithm fails:

**Theorem 7** (Failure cases). The algorithm fails if and only if at least one of the following three conditions holds:

1. One of the side conditions for the current rule is not true (e.g. the input type of a product filter is not a product).
2. One of the meta four operations $\tau/p$, or $\exists^+(\tau)$, or testing for equality, or subtyping is applied to a type $\tau$ such that $\text{FV}(\tau) \neq \emptyset$.
3. The choice operator cannot find a suitable type amongst the given annotations for a certain filter $f_x$.  

8

2008/6/16
Case (i.) means that the term is ill-typed and the algorithm fails with a type error. In case (ii.), the algorithm is deconstructing a type which contains free recursion variables, that is, a type which it is currently computing. It therefore fails due to a lack of information and more annotations are required. In case (iii.) the annotation provided are either insufficient or wrong. We want to avoid cases (ii.) and (iii.) while keeping the annotations as minimal as possible.

The intuition, that we formalise hereafter, is that the only case where such a problem occurs is when, in a composition filter, the first filter is “recursive” (hence necessitating the introduction of a type variable to express its output type) and the second filter deconstructs its input (which in that case might be a type with open variables). This is the only case where annotations are needed and we will see in Section 6 that, in practice, these annotations are not cumbersome. We can now formalise the intuition mentioned above:

**Definition 11** (Deconstructing subterms). A filter \( f \) **deconstructs** its input if and only if \( f \) is not an expression filter. A recursive filter \( f \) is a filter such that the associated regular tree is not finite. Let \( f \) be a filter. We define the set of all deconstructing sub-terms of \( f \), noted \( \delta_f \), as the set of all sub-terms \( g \) of \( f \) such that \( g \equiv f_1; f_2 \) where \( f_1 \) is recursive and \( f_2 \) deconstructs its input.

We can now present the algorithm from failing in case (ii.) by requiring that in all deconstructing sub-terms of a filter the leftmost one is annotated:

**Lemma 2** (Mandatory annotations). Let \( \tau \) be an input type and \( f \) a filter such that for all \( f_1; f_2 \in \delta_f \), \( f_1 \equiv f_1' \), for some \( E \). For all type \( \tau' \) occurring in the derivation of \( \Gamma \Delta \vdash \sigma \), if \( FV(\tau') \neq \emptyset \), then \( \tau' = \sigma \) is never deconstructed.

Now that we know the only places where it may be necessary to annotate a filter, it remains define how to annotate these places, that is to find the correct annotations and thus avoid the last case (iii.) of failure. Once done, it remains nothing but to state the completeness theorem. To have the right annotations for a well-typed filter it suffices to pick their types in the corresponding regular derivation of the type system. This is formally defined by the following t-labelling procedure

**Definition 12** (t-labelling). Let \( f \) be a filter and \( t \) a type such that a regular derivation for \( \Gamma \vdash f(t) = s \) exists, for some type \( s \). Let \( \delta_f = \{ f_1; f_2; \ldots; f_i; f_j; \ldots; f_n \} \), we call \( E_t \) the set of all the output types for \( f_i; f_j \) in this derivation. A t-labelling of \( f \), noted \( [T]_t \), is obtained from the filter \( f \) by replacing all its \( f_i; f_j \) sub-terms in \( \delta_f \) by the corresponding \( f_1; f_2 \) corresponding \( f_1; f_2 \).

It is important to note that thanks to the regularity of the derivation of \( \Gamma \vdash f(t) = s \), all the sets \( E_t \), mentioned in the definition are finite. We can now use Definition 12 to state the completeness of the algorithm with respect to t-labellings.

**Theorem 8** (Completeness). The algorithm given by the set of rules \( \mathcal{F}_{ct} \) is complete with respect to the type system \( \mathcal{F} \), that is: if \( \Gamma \vdash f(t) = s \), then \( \Gamma \vdash [T]_t \equiv [s]_t \).

### 6. Concrete language

#### 6.1 Syntax

We have implemented our language into the CDue compiler. While efficient compilation of our language or possible syntax enhancements are still matter of study, the type-checking algorithm proves to be usable in practice\(^5\). In this section we give various examples of filters, to show how they can be used to implement and

---

\(^5\)It should be noted that this early prototype makes use of the product decomposition operator already available in CDue, which is not stable with respect to subtyping but still provides type safety.
Comparison with Hosoya’s filters: The example in Figure 1 is a typical example of tree mapping, the kind of transformation that can be programmed by Hosoya’s regular expression filters [14]. In order to illustrate how different our and Hosoya’s tying disciplines are, even for the cases that can be handled also by Hosoya’s filters, let us simplify the example and consider the following filter:

```haskell
let filter replace a = [a] -> []
    | ((<a>[m] -> <d>[]) x -> x) , replace a )
```

This filter simply takes a sequence of XML elements and replaces every `<a>` tag by a `<d>` tag, leaving the other tags unchanged via the identity filter `x -> x`. In Hosoya’s framework, every single expression filter (here the rightmost `x`) is typed only once. Its type must thus reflect all the possible values this expression may evaluate to. If this filter is applied to a value of type `[<a>[1] <b>[1] <c>[1] <d>[1] ]`, then `x` can be bound to values of type `<b>[1]`, `<c>[1]`, or `<d>[1]`, which in Hosoya’s system yields the output type `<d>[1] (b[c]d[]) (c[b]d[]) (b[c]d[])`. In our system instead thanks to rule(s) (`*prod`) the same expression may be typed several times under different hypothesis (recall that sequences are nested pairs) which in this case yields that the output type is the expected `<d>[1] <b>[1] <c>[1] <d>[1]`. Hosoya justifies his typing policy by stressing that in some cases there is no lower bound to the output type\(^7\) and that using the union of all types leads to a clean specification of the algorithm and to a simple notion of completeness. This is true, but the loss of typing precision that this choice implies seems to us a too high price to pay: filters such as `x -> x` are in practice used almost everywhere, since they define a default behaviour in transformations and we cannot afford to lose precision by using them. The solution we retain is to deduce some particular type which is always more precise than taking the union of all possible types for a given expression and which we claim to be in practice precise enough for common transformations.

Flattening: To illustrate the use of annotations we define a filter for unbounded flattening of XML elements, that is, a filter that accepts a sequence of arbitrary nested XML elements and returns the sequence of all these elements:

```haskell
let filter flatten =
    | [] -> []
    | (x, y) -> ((<d> id flatten , flatten ) ,
                ((<c> y , z) -> ((x,y,z);concat))
    | (id , flatten )
```

If the argument is an empty sequence, then the filter returns the empty sequence. If the argument is a sequence with an XML element as head, then it captures the head (in `x`), recursively flattens the children and the tail of the list, captures both results respectively in `y` and `z`, and concatenates everything into a list. Finally if the first element is not an XML element, then the filter just flattens the tail. If we apply the filter `flatten` to an expression `mytree` of type `Recipe` defined in Figure 1, then we need to suggest an approximation:

```haskell
apply flatten to [mytree]
where flatten = { [ ItemRecipe|Enumerate|Itemize|Char] }
```

The type algorithm checks that the result has the type specified in the annotation.

Parametric filters: As we have previously seen with the map example, filters allow one to iterate a transformation over a given input. Usually, the transformation is given by a set of branches, that is an alternation of filters of the form:

```
px_1 -> x_1 | ... | px_n -> x_n
```

The programmer here is left with two choices. If one encapsulates\(^7\) the code of the transformation into a function, as we did with the map example, one has to annotate this function. On the other hand, one can choose to inline the transformation in the map filter, thus relying on the type inference algorithm to infer a precise type. However, this is clearly bad for modularity, as the iterating part of the map filter has to be duplicated over and over for every new map defined. To solve this issue, we introduce, parametric filters. These are filters which take other filters as argument and allow one, for instance, to define iterators taking a transformation as parameter.

For instance:

```haskell
let filter map =
    | [] -> []
    | (f , map );;
    ;;
    apply (map g) to [1 2 3 ];
    let filter g = x -> x+1
    apply (map h) to [1 ’true’ ];
```

Such filter should not be seen as true “higher-order filters” since they merely consist in replacing the place-holder names in the definition (f in our example) with the body of the argument filters (g and h) and typing the resulting filter as a whole. In this respect, they are reminiscent of Walders higher-order macros, which were introduced to encode higher-order transformation in the context of deforestation (see [29]).

XML idioms: We can now show how to encode (descending) XPATH expressions into filters. In a nutshell, XPATH expressions select nodes in an XML tree. For instance, the expression “//a/b” selects all the nodes tagged `<b>` which are below a node tagged `<a>`, which can itself be at any depth in the input document. This XPATH expression is composed of two steps, //a and //b. // represents the descendant-or-self axis and / stands for child axis. A constraint given by the XPATH specification [31] is that the result is an ordered set, meaning that nodes must occur in document order and without any duplicates. This means that a compositional semantics of steps is not sufficient to account for the XPATH specification. Let us consider the following simple document:

```
<xml>
</xml>
```

and the path expression:

```
//a/a
```

If we try to apply the XPATH expression, step by step, we obtain:


The final result does not comply with the XPATH specification since the sub-tree `<a>[2]` is duplicated. To circumvent this issue, the specification recommends having a unique node id for every node in a document, thus allowing one to filter out the results so as to eliminate duplicates and order the result set according to the node id. Unfortunately, this requires the host language to take node ids into account in its low-level representation of XML data. This is not the case of e.g. XDupe, CDuexe but also the SYB+XPATH experiment ([19]). Indeed, these implementations rely on much simpler —persistent— data-structures to represent XML documents. Consequently, these languages provide some kind of hard-coded combinators, such as single / and //, but do not allow one to write fully XPATH compliant expressions. Furthermore, their typing is far from being precise. In the case of CDuexe, the output type will be for instance, `[tag[Any]*]` and in the case of SYB, the XML type will be encoded by using Haskell type classes, with the shortcomings we discussed in our Introduction. On the contrary, filters allows us to encode a non trivial subset of forward XPATH
expressions multiple steps composed of the axes: self, child, descendant-or-self and descendant. The use of Cduce patterns instead of simple tag name test allows us to encode a wide range of XPath predicates. Finally, an ad-hoc typing algorithm allows us to compute a precise approximation of the output type automatically, thus relieving the programmer from writing any type annotation. Since we do not have enough space to detail the full encoding and typing algorithm, we illustrate them with a short example. Let us consider the XPath expression "//a/b". The idea is to see this path expression as a regular expression on paths in the input document and encode the corresponding word automaton into a set of mutually recursive filters. The resulting set of filters is given in Figure 6 together with the corresponding DFA.

While this set of filters seems complicated, it is nothing but the deterministic finite word automaton recognizing the regular expression a*b. The filter's filters correspond to the states of the DFA. The transitions are encoded by the branches of each filter. Finally, each filter is a simple auxiliary filters which iterates a given filter on a sequence of XML values and concatenates the intermediate results (we only gave the code for filter, the others being similar). As we see in filter, if the input tag is an a, then we evaluate the filter I_f12 which iterates filter I_f2 on every child of the input value. Likewise, if the input tag is a b, then we iterate filter I_f3 on every child of the current input. However, since this transition leads to the accept state f13, we memorize the current input in the variable res which we return as part of the result. This encoding allows us to ensure both the ordering and the unicity of the element in the result set. Another point of interest is that, despite the fact that the recursive calls to the filter's and filter's appear on the left-hand side of a composition and this must be annotated, they all require the same annotation. The complete description of the annotation inference process, in the particular case of XPath, can be found in Chapter 7 of the second author’s PhD thesis. Let us wrap it up in a final example:

```
type doc = <a>[ (doc | <b>[ <c>[] | ] ) * ]
let d:doc = <a>[ <b>[ <c>[] | ] ]<a>[ <b>[ <c>[] | ]<c>[] | ]]
apply html to d;
val d : [ <b>[<c>[] | ] * ] = [ <b>[<c>[] | ]<b>[<c>[] | ]<c>[] | ] ]
```

As we can see, the order and unicity of the elements in the result are respected and the typing is more precise than a generic <b>[Any*].

7. Related work

There exist various attempts to mix XML types and parametric polymorphism. The parametric polymorphism currently available in Xduce, mixes explicit type annotations with well-localised type reconstruction [15]. Of similar flavour, but following a completely different approach, is the work by Vouillon [28] where explicit polymorphism is designed so as that pattern matching does not break parametricity. A different approach consisting in the “coexistence” or juxtaposition of both XML and ML type systems in a same language [9] is available and actively maintained for OCaml. While this eases the writing of polymorphic functions on XML values, this solution does not solve the problem of writing precisely typed operators. Indeed, both type systems (ML and Xduce) are kept apart, and a value is either seen as on the ML side — and can then be polymorphic — or on the Xduce side — and can then be precisely typed (with Xduce pattern matching for example) —. Finally, in the same spirit of combining two type systems, a more general approach was defined by Sulzmann and Lu [26] for Haskell where the authors mix Haskell type classes with Xduce regular expression types into a system called XHaskell [27]. They provide a semantics via a type-directed rewriting of the language into System F. While the decidability of the general version is not clear, some restrictions make it tractable and lead to an implementation of this work using the GHC Haskell compiler as backend. Type safety is granted, but the programmer is required to heavily annotate the code: in particular, every polymorphic variable that is instantiated with a regular expression type has to be explicitly annotated.

A common trait in all these approaches is that a polymorphic value either is never visited (through pattern matching for example) and so is never precisely typed, or if it is visited then it loses its polymorphic nature and becomes monomorphic and precisely typed. While this eases the writing of generic function over XML values it does not address the problem we study here, that is to have both precision and polymorphism.

For what concerns restricted iterators for XML, the literature is quite rich. In the framework of our work the most interesting techniques appear to be the k-pebble tree transducers [23], the macro tree transducers (MTTs) [21], and Hosoya’s regular expression filters [14]. For macro tree transducers (and k-pebble tree transducers) the general approach is to use the so-called backward type inference, in which the output type of the operators is given by the programmer, and the biggest valid input type is deduced by the system. This clearly has the advantage of solving the issue of non-regular results, since the inverse image of a regular tree language by an homomorphism is a regular tree language (while the direct image in general is not regular). However, the good theoretical properties of backward type checking are mitigated by some challenging issues. First of all, the complexity of backward type checking is still a concern. Some advances have been made on this topic, most notably in [22], where the complexity is reduced by only allowing a limited number of copies of the input, and in [11] where an efficient implementation coupled with algorithmic optimizations make it possible to type-check small transformations on real life types (such as XHTML) in a reasonable time. It is nevertheless still unclear how a backward type inference language can be integrated into a more expressive language, which is needed if one wants to provide a full-fledged language with precise XML typing. Regular expression filters, instead, provide quite a natural way of writing XML transformations and are implemented in the current XDuce distribution, but they are restricted to map-like operators. In particular, they cannot express XPath-like expressions nor fold-like functions over sequences, nor can they perform non-local transformations. Moreover, even in the cases they can handle, we saw in Section 6.2 that while they enjoy a property of local pre-
cision (as defined by Hosoya) they still remain imprecise for some common transformations.

We would also like to emphasize that, to the best of our knowledge, this work is the first to provide precise typing of such XML transformations, that was tested on real life types (such as the XHTML or DocBook DTDs) and non-trivial programs (hundreds of lines of code with heavy use of filters). Indeed, Hosoya’s regular expression filters, which are implemented in XDuce, do no match the expressiveness and typing precision of our filters and MTT-based solutions, while theoretically appealing, still lack an usable implementation.

8. Conclusion

In this paper we presented a small language of combinators we dubbed filters. This specific set of combinators allows us to write and type many XML transformations and, more generally, to precisely type the application of highly polymorphic iterators over complex data structures. While type inference is not completely automated in some cases (some of which, we admit, are truly of use for XML transformations), we have precisely pointed out the set of filters for which annotations may become necessary and verified that, in practice, those annotations were very light. We believe our language constitutes a good compilation target for higher-level and more declarative idioms such as XPath, fragments of XSLT, or more functional iterators such as map and fold. This small algebra gives us a broad range of perspectives for future work. First of all, at the typing level some work is yet to be done. Heuristics can be used to guess the annotations automatically, based on the context of the filter for example, or by giving a regular approximation to non-regular equations. In this perspective the work of Nederhof [24] constitutes an important base to start from. Formalising such heuristics seems however challenging.

Changing the target language and pattern algebra has a direct impact on the typing of the filters. Embedding our combinators in an object-oriented language, for example, should constitute an interesting and potentially fruitful extension, the mainstream languages being known for lacking such polymorphic features (see the recently added generics in Java and C#). Of course, if we aim to have filters be used for XML processing in production code, then efficient compilation of filters is mandatory. Our algebra permits to study algebraic optimisations via term rewriting such as, say, interleaving a filter and a pattern to avoid two traversals of a data structure. Furthermore in this area we can surely benefit from the impressive amount of previous work done on the compilation of tree automata and tree-transducers in general.

While we focussed our presentation on a simple “core” language, we want to stress that several useful extra features can be found in the prototype, which are obtained either by new syntactic sugar or by minimal extensions to the core algebra. In addition to parametric filters and the XPath encoding we presented in Section 6, our implementation supports, for instance, regular-expression-like syntax (à la Hosoya) to provide a filter such as \[(x -> x+1)*\] as well as some typing extensions which allow one to type simple, yet useful, composed filters without annotations.

References