# 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, even simple ones, that modify these features may affect the types of documents. 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. Xtatic [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 sieve 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 the tag of an element and imagine that we iterate it on an XML document of a given DTD. In order to obtain the necessary type precision, the type system must be able to deduce that the iteration will produce a document whose type is exactly the DTD of the input document, whatever it is, where all the tags have been capitalised (XML types are case-sensitive). Thus the iterator at issue must be $(i)$ highly polymorphic, since it can be applied to any DTD and $(ii)$ must return a very precise output type calculated by performing an abstract execution of the iterator on the input type.

Such a kind of polymorphism is well beyond what currently exists. It cannot be handled by *parametric* polymorphism (either implicit *à la* ML or explicit *à la* System F) because it is precisely the opposite of parametricity which leaves polymorphic elements untouched. It cannot be handled by *subtyping* polymorphism because the least upper bound of transformations such as those at issue is the completely uninformative type of all XML documents. This kind of polymorphism resembles very much to the application of an overloaded function (since to different and possibly unrelated input types correspond different and precisely defined output types), the so-called *ad hoc* polymorphism. However, since such polymorphism must be able to cope with a potentially infinite set of different input contexts, it is out of reach of the *ad hoc* polymorphism, even when coupled (as in Haskell) with the parametric one.

The only solution used so far in XML processing programming languages is to hard-code these iterators in the language so as to make it possible to define specific typing rules for them that, in practice, *compute the output type of an iterator by executing it on the (input) type of its argument*. This is what is done in languages such as Xtatic, ℂDuce, and XDuce which all provide several built-in iterators for sequences and XML-trees. But this approach soon shows its limits: while for an operation as simple as changing 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 Xtatic,...), 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

* A preliminary version of this work was presented at the *ACM SIGPLAN Workshop on Programming Language Techniques for XML* (PLAN-X) 2008 (no formal proceedings).

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-explorations, 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` that 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>[]    type B = <b>[]    type T = [] | [ A T B ]
```

which define three types: `A` and `B` which type single elements of tag `<a>` and `<b>` 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-trees 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 $\{\texttt{[A}^n\texttt{B}^n\texttt{]} \mid n \geq 0\}$, which is not regular. Of course there are regular approximations of this type such as `[(A*) (B*)]`, or `[]|[A(A*) (B*)]`, 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 $\{\texttt{[A}^n\texttt{B}^n\texttt{]} \mid n \geq 0\}$). 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 ℂDuce'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 ℂDuce as host language (the implementation uses the same language used for the ℂDuce 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 commentating 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/`, 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, filters are applied to an input value to iterate over and transform it into another value. As a first approximation, one can consider pattern matching as found in Haskell as well as in various dialects of ML such SML and OCaml. There, the basic pattern matching construction has the form $p$ `->` $e$ (even though it is never found standalone) where $p$ is a pattern and $e$ an expression. This construction can be "applied" to an expression: in such a case the expression is evaluated into a value; the pattern $p$ is matched against this value and this results into an environment that associates the "capture variables" (the variables occurring in the pattern) to the sub-parts of the value they match; finally, the environment is used to evaluate the expression $e$.

Usually, several of these constructions can be composed by using a vertical bar `|` (a customary syntax for pattern matching is `match` $e$ `with` $p_1$`->`$e_1$ `|`...`|` $p_n$`->`$e_n$) denoting alternation, which is used according a first-match or a any-match policy. In [14] Haruo Hosoya adds to alternation the Kleene star `*` and juxtaposition. These allow the program to iterate (sequences of) pattern matching over sequences and thus to define `map`-like iterations.[1]

[1] We borrowed our terminology from Hosoya who calls his iterators *filters*.

What we propose is to generalize such a technique by transforming the constructions of the form $p$ -> $e$ 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:

```
let filter concat = (x , y) -> (x ;
                                let filter aux =
                                 'nil -> y
                                | (head->head,aux) 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 concat introduces most of the syntax of filters. A filter is: either a *pattern filter* of the form $p$->$f$ (e.g. the body of 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 aux) that succeeds only if it is applied to a pair of values, it applies each filter $f_i$ 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 aux) which applies $f_2$ on the argument value only if the application of $f_1$ failed; or a *composition filter* $f_1;f_2$ (e.g. the right hand filter of the body of 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 *head*), or a possibly recursive filter declaration `let filter` $\underline{x}=f$.

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

```
let concat (x , y) =
   let rec aux z = match z with
               'nil -> y
              | (head , tail ) -> (head , aux tail )
   in aux x
```

The definition of the filter 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 ($x$ and $y$) via the pattern $(x, y)$. Then, a second recursive filter aux is defined and applied (via the composition operator "`;`") to the result of the *expression* "$x$", that is to the first list. Note the similarity between `(x;let filter aux = ...)` and `let rec aux z =...in aux x`. The recursive filter aux is the union ("`|`") of two filters, playing the same role as two branches of a pattern matching. If the argument is the constant `'nil`, (this case is handled by the pattern filter "`'nil` -> $y$"), then the second list $y$ is returned. If the argument is a pair, (handled by the pair filter "`(,)`"), then the first component is left unchanged (by the use of an identity filter: *head*->*head*) while the second component, which is the tail of the list, is recursively iterated over by the 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: $\forall\alpha.\ \alpha\ \mathit{list} \times \alpha\ \mathit{list} \to \alpha\ \mathit{list}$. 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 subtyping, 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: `[Any*]` $\times$ `[Any*]` $\longrightarrow$ `[Any*]` (`Any` being the super type of all types). Again, precision is lost because while any lists is accepted as argument (thanks to subtyping its type is "up-casted" to `[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 concat. It is easy to see that the subfilter aux will not fail as long as it is applied to a list. Therefore the composition in concat will work if $x$ is bound to a list. From this we deduce that 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 concat.

Precision of type inference is achieved by using for concat the same typing policy as the one used for the hard-coded concatenation operator `@` of XDuce or ℂDuce. That is, instead of typing the operator, one types each single application of the operator. In terms of the filter concat this corresponds to type the application 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 `[String Bool*]` and $l_2$ has type `[Bool Int?]` , then the type system infers for the result the type `[String Bool+ Int?]` which, in this case, is the most precise one. If 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 `<`$f_1$ $f_2$`>`$f_3$ 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 of tag `<itemize>`, `<enumerate>` or `<item>`, then the tag is changed to `<ul>`, `<ol>` or `<li>`, respectively. If the tag is something else, then it is just copied. The content is then recursively iterated by the map_list filter. If the argument is not an XML element (a 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 `<recipe>` to `<html>`, extracts the content of the root tag (by the pattern `x ->...`), and rebuilds a new element `<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:

```
Content of the file recipe.xml:
<recipe>
 <itemize>
  <item> 400g of chocolate</item>
  <item> 3 eggs </item>
  ...
 </itemize>
 <enumerate>
  <item> Melt the chocolate </item>
  <item> Separate the white from the yolk </item>
  <item> Beat the whites to foam </item>
  ...
 </enumerate>
</recipe>

The types of the data:

type Item = <item>[ Char* ]
type Enumerate = <enumerate>[ Item+ ]
type Itemize = <itemize> [ Item+ ]
type Recipe = <recipe> [ Itemize Enumerate ]

Definition of auxiliary filters:

let filter map_elem =
                <( itemize->ul
                  |enumerate ->ol
                  |item->li
                  | x ->x) >map_list
               | x -> x
and map_list =  ([] -> []) | (map_elem,map_list)

Call of the filter in the host language:

apply (<(recipe ->html)>(x->
              <body>(apply map_list to x)))
to (load_xml "recipe .xml")
```

**Figure 1.** XML document and a transformation into XHTML

`<head>[<body>[<ul>[<li>[Char*]+] <ol>[<li>[Char*]+]]]`

Had we applied the filter to more a complex expression in which itemize and enumerate elements were nested and/or interleaved with other elements, then we would have found all of them (and at the right position) in the type of the result. In Section 6, right after the formal development, we will show more advanced uses of filters.

## 3. Syntax and dynamic semantics

Our filters are deeply inspired by ℂDuce patterns. These patterns are nothing but types in which capture variables may occur. Therefore we start our presentation with a brief overview of ℂDuce types and patterns as defined in [5] and [10], followed by the definition of filters and of their operational semantics.

### 3.1 Types and patterns

**Definition 1** (Types [5])**.** A *type* is a possibly infinite term produced by the following grammar:

$$t \;::=\; b \mid (t_1,t_2) \mid t_1|t_2 \mid t_1\&t_2 \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

with two additional requirements:

1. (regularity) the term must be a regular tree (it has only a finite number of distinct sub-terms);
2. (contractivity) every infinite branch must contain an infinite number of pair nodes $(\_,\_)$.

We use $b$ to range over basic types, while `Empty` and `Any` respectively denote the empty type and the type of all values. Besides, there are product $(t_1,t_2)$, union $(t_1|t_2)$, intersection $(t_1\&t_2)$, and negation $(\neg t)$ types. Infiniteness of types accounts for recursive types, and regularity implies that they are finitely representable, e.g. by recursive equations or by explicit $\mu$-notation (as we do in Section 5). The contractivity condition rules out meaningless terms such as $X = \neg X$ (that is, $\mu\alpha.\neg\alpha$, an infinite unary tree where all nodes are labelled by $\neg$). Both conditions are standard when dealing with recursive types (e.g. see [1]).

These types are enough to encode XML types which, we remind, are given by specifying tags that label sequences of elements whose content is described by regular expressions on types. For instance:

`type Book = <book>[ Title (Author+|Editor+) Price? ]`

defines a type `Book` that types elements tagged by `<book>` and that contain a title followed by either a non-empty list of authors or a non-empty list of editors and possibly ended by an optional price (`Title`, `Author`, `Editor` and `Price` being types defined in other declarations, here left unspecified).

Sequences can be encoded *à la* Lisp by nested pairs. Pairs can also be used to encode element types, while regular expression types are encoded by recursive types. Therefore the declaration of `Book` above can be considered as syntactic sugar for `Book=(`book,(Title,X|Y))`, `X=(Author,X)|(Author,Z)`, — which is the non-empty list of authors— `Y=(Editor,Y)|(Editor,Z)` —the non empty list of editors— and `Z=(Price,`nil)|`nil` — the optional price — and where `` `book`` and `` `nil`` are singleton (basic) types.

Patterns are just types in which capture variables may occur in a controlled way:

**Definition 2** (Patterns [10])**.** A *pattern* is a (possibly infinite) term produced by the following grammar

$$p \;::=\; x \mid t \mid (p_1,p_2) \mid p_1|p_2 \mid p_1\&p_2\,,$$

that is regular, contractive (as in Definition 1), and in which every subtree of the form $p_1\&p_2$ or $p_1'|p_2'$ satisfies $\mathit{Var}(p_1)\cap\mathit{Var}(p_2)=\varnothing$ and $\mathit{Var}(p_1') = \mathit{Var}(p_2')$, respectively (where $\mathit{Var}(p)$ is the set of capture variables occurring in $p$).

The semantics of both types and patterns is expressed in terms of *values*. In the framework of XML processing languages, values are XML documents and, following Hosoya *et al.* [18], an XML type is (interpreted as) the set of XML documents that have that type. In this paper we do not fix a particular set of values (since it depends on the host language filters are used in) but we rather suppose its existence and implicitly assume that it contains all XML documents. Then we consider a type as the set of values that have that type, the union, intersection, and negation types as the corresponding set-theoretic operations, and the subtyping relation, noted $\leq$, as set-containment. Since the use of subsumption makes two equivalent types (that is, two types denoting the same set of values) operationally indistinguishable, then we will always work up to type equivalence and consider, e.g. $t|t$, $\texttt{Any}\&t$ and $t$ as the same type.

The semantics of patterns is defined in terms of *matching*. Informally, the *matching* of a value $v$ against a pattern $p$, that we note $v/p$, is either a failure (noted $\Omega$) or a substitution from the capture variables of $p$ to values. The substitution is then used as an environment in which some expression is evaluated. If the pattern is a type, then the matching fails if and only if the pattern is matched against a value that has not that type. If it is a variable, then the matching always succeeds and returns the substitution that assigns the matched value to the variable. The pair pattern $(p_1,p_2)$ succeeds if and only if it is matched against a pair of values and each sub-pattern succeeds on the corresponding projection of the value (the union of the two substitutions is then returned). An intersection pattern $p_1\&p_2$ succeeds if and only if both patterns succeed (the union of the two substitutions is then returned). The union pattern $p_1|p_2$ first tries to match the pattern $p_1$ and if it fails it tries the pattern $p_2$.

For instance the pattern $(\texttt{Int}\&x,y)$ succeeds only if the matched value is a pair of values $(v_1, v_2)$ in which $v_1$ is an integer —in which case it returns the substitution $\{x{:=}v_1, y{:=}v_2\}$— and fails otherwise.

$$\textbf{(e-expr)}\quad \frac{}{\gamma \vdash_e e(v) \rightsquigarrow r} \quad r = \mathtt{eval}(\gamma, e)$$

$$\textbf{(e-prod-ok)}\quad \frac{\gamma \vdash_e f_1(v_1) \rightsquigarrow r_1 \qquad \gamma \vdash_e f_2(v_2) \rightsquigarrow r_2}{\gamma \vdash_e \boldsymbol{(}f_1{,}f_2\boldsymbol{)}(v_1, v_2) \rightsquigarrow (r_1, r_2)} \quad \text{if } r_1 \neq \Omega \text{ and } r_2 \neq \Omega$$

$$\textbf{(e-prod-err1)}\quad \frac{\gamma \vdash_e f_1(v_1) \rightsquigarrow r_1 \qquad \gamma \vdash_e f_2(v_2) \rightsquigarrow r_2}{\gamma \vdash_e \boldsymbol{(}f_1{,}f_2\boldsymbol{)}(v_1, v_2) \rightsquigarrow \Omega} \quad \text{if } r_1 = \Omega \text{ or } r_2 = \Omega$$

$$\textbf{(e-prod-err2)}\quad \frac{}{\gamma \vdash_e \boldsymbol{(}f_1{,}f_2\boldsymbol{)}(v) \rightsquigarrow \Omega} \quad \text{if } v \not\equiv (v_1, v_2)$$

$$\textbf{(e-patt-ok)}\quad \frac{\gamma \cup v/p \vdash_e f(v) \rightsquigarrow r}{\gamma \vdash_e (p \rightarrow f)(v) \rightsquigarrow r} \quad \text{if } v/p \neq \Omega$$

$$\textbf{(e-patt-err)}\quad \frac{}{\gamma \vdash_e (p \rightarrow f)(v) \rightsquigarrow \Omega} \quad \text{if } v/p = \Omega$$

$$\textbf{(e-comp-ok)}\quad \frac{\gamma \vdash_e f_1(v) \rightsquigarrow r_1 \qquad \gamma \vdash_e f_2(r_1) \rightsquigarrow r_2}{\gamma \vdash_e f_1;f_2(v) \rightsquigarrow r_2} \quad \text{if } r_1 \neq \Omega$$

$$\textbf{(e-comp-err)}\quad \frac{\gamma \vdash_e f_1(v) \rightsquigarrow \Omega}{\gamma \vdash_e (f_1;f_2)(v) \rightsquigarrow \Omega}$$

$$\textbf{(e-union1)}\quad \frac{\gamma \vdash_e f_1(v) \rightsquigarrow r_1}{\gamma \vdash_e (f_1|f_2)(v) \rightsquigarrow r_1} \quad \text{if } r_1 \neq \Omega$$

$$\textbf{(e-union2)}\quad \frac{\gamma \vdash_e f_1(v) \rightsquigarrow \Omega \qquad \gamma \vdash_e f_2(v) \rightsquigarrow r_2}{\gamma \vdash_e (f_1|f_2)(v) \rightsquigarrow r_2}$$

**Figure 2.** Dynamic semantics of filters

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 $\lbag p \rbag$.*

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 $\lbag p \rbag$, 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 $t \leq \lbag p \rbag$ returns a type environment $t/p \in Var(p) \rightarrow Types$ 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)

$$\begin{array}{rcll} f & ::= & e & \text{expression} \\ & | & p \rightarrow f & \text{pattern, } p \in \mathbb{P} \\ & | & f;f & \text{composition} \\ & | & \boldsymbol{(}f{,}f\boldsymbol{)} & \text{product} \\ & | & f|f & \text{union} \end{array}$$

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 semicolons ";". For example, $f{=}\boldsymbol{(}f{,}f\boldsymbol{)};g$ with $g{=}\boldsymbol{(}g{,}g\boldsymbol{)}|x \rightarrow x$ is permitted, while $f{=}(x \rightarrow (x,x));\boldsymbol{(}f{,}f\boldsymbol{)}$ is not. Basically, this restriction prevents our filters from diverging when they are applied to a (possibly infinite) type. This ensures the termination both of the type inference algorithm (as we explain in Section 5) and of the execution of a filter on a finite value. Henceforward we use $\mathbb{F}$ to denote the set of (well-formed) filters.

We formally define the sets of free and capture variables for filters, as an extension of free and capture variables for expressions and patterns of the host language.

**Definition 4** (Capture variables)**.** We define the set of *capture variables* of a filter $f$, $Var(f)$ as:
$Var(e) = \varnothing$
$Var(f_1;f_2) = Var(f_1|f_2) = Var(\boldsymbol{(}f_1{,}f_2\boldsymbol{)}) = Var(f_1) \cup Var(f_2)$
$Var(p \rightarrow f) = Var(p) \cup Var(f)$

**Definition 5** (Free variables)**.** We define the set of *free variables* of a filter $f$, $FV(f)$ as:
$FV(f_1;f_2) = FV(f_1|f_2) = FV(\boldsymbol{(}f_1{,}f_2\boldsymbol{)}) = FV(f_1) \cup FV(f_2)$
$FV(p \rightarrow f) = FV(f) \setminus Var(p)$

We suppose that $FV(e)$ is defined (provided by the host language).

### 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 judgement $\gamma \vdash_e f(v) \rightsquigarrow r$ in Figure 2 and describes when the evaluation of the application of filter $f$ on a value $v$ in an environment yields an object $r$ where $r$ is either a value or $\Omega$. The latter is a special value which represents a runtime error: it is raised either because a filter did not match the form of its argument (**e-prod-err2**) or because some pattern matching failed (**e-patt-err**). It is easy to read the rules of Figure 2 in the light of the informal semantics we gave in Section 2. Filter application is defined on values, which are returned by the host language to evaluate an `apply to` expression. The *expression* filter discards its input and evaluates (rather, asks the host language to evaluate) the expression $e$ in the current environment (**e-expr**). The *product* filter expects a pair as input (we use $\equiv$ to denote syntactic equivalence), applies its filters componentwise and returns the pair of the results. The *pattern* filter first matches its pattern $p$ against the input value $v$; if it fails it raises an error (**e-patt-err**), otherwise it evaluates its sub-filter in the environment augmented by the substitution $v/p$ (**e-patt-ok**). The *alternative* filter follows the standard first match policy. Finally, *composition* allows us to pass the result of $f_1$ as input to $f_2$. As stated before, the condition on filter composition ensures termination:

**Theorem 3** (Termination of filtering)**.** *Let $v$ be a finite value of the language and $f$ a well-formed filter, in which every expression subterm terminates for all well-typed substitution. Then the evaluation of $f(v)$ terminates.*

## 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 $f \in \mathbb{F}$ we define the type $\llparenthesis f \rrparenthesis$ as follows:

$$
\begin{array}{lcl}
\llparenthesis e \rrparenthesis & = & \texttt{Any} \\
\llparenthesis (f_1,f_2) \rrparenthesis & = & (\llparenthesis f_1 \rrparenthesis , \llparenthesis f_2 \rrparenthesis) \\
\llparenthesis f_1 | f_2 \rrparenthesis & = & \llparenthesis f_1 \rrparenthesis \,|\, \llparenthesis f_2 \rrparenthesis
\end{array}
\qquad
\begin{array}{lcl}
\llparenthesis p \rightarrow f \rrparenthesis & = & \llparenthesis p \rrparenthesis \,\&\, \llparenthesis f \rrparenthesis \\
\llparenthesis f_1 ; f_2 \rrparenthesis & = & \llparenthesis f_1 \rrparenthesis
\end{array}
$$

Input inclusion in the accepted type of a filter is a necessary condition for filter application to succeed: $\forall v \notin \llparenthesis f \rrparenthesis, f(v) \rightsquigarrow \Omega$. Unfortunately it is not also sufficient since, for instance, the accepted type of $\texttt{Any} \rightarrow \texttt{3}\ ;\ (\_,\_) \rightarrow \texttt{5}$ is `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 $t_1 | \cdots | t_n$.

$$
\textbf{(t-expr)}\quad \frac{\texttt{type}(\Gamma, e) = s}{\Gamma \vdash e(t) = s}
$$

$$
\textbf{(t-prod)}\quad \frac{\begin{array}{l} i = 1..\texttt{rank}(t) \\ j = 1..2 \end{array} \quad \Gamma \vdash f_j(\mho^i_j(t)) = s^i_j}{\Gamma \vdash (f_1,f_2)(t) = \bigvee_i (s^i_1,s^i_2)}
$$

$$
\textbf{(t-patt)}\quad \frac{\Gamma \cup t/p \vdash f(t) = s \quad t \leq \llparenthesis p \rrparenthesis \,\&\, \llparenthesis f \rrparenthesis}{\Gamma \vdash (p \rightarrow f)(t) = s}
$$

$$
\textbf{(t-union)}\quad \frac{\begin{array}{l} t \leq \llparenthesis f_1 \rrparenthesis \,|\, \llparenthesis f_2 \rrparenthesis \\ t_1 = t \,\&\, \llparenthesis f_1 \rrparenthesis \\ t_2 = t \,\&\neg\, \llparenthesis f_1 \rrparenthesis \end{array} \quad \begin{array}{l} \Gamma \vdash f_1(t_1) = s_1 \\ \Gamma \vdash f_2(t_2) = s_2 \end{array}}{\Gamma \vdash (f_1|f_2)(t) = \bigvee_{\{i | t_i \neq \mathsf{Empty}\}} s_i}
$$

$$
\textbf{(t-comp)}\quad \frac{\Gamma \vdash f_1(t) = s_1 \quad \Gamma \vdash f_2(s_1) = s_2 \quad \begin{array}{l} t \leq \llparenthesis f_1 \rrparenthesis \\ s_1 \leq \llparenthesis f_2 \rrparenthesis \end{array}}{\Gamma \vdash (f_1;f_2)(t) = s_2}
$$

$$
\textbf{(t-subs)}\quad \frac{\Gamma \vdash f(t) = s' \quad s' \leq s}{\Gamma \vdash f(t) = s}
$$

**Figure 3.** Deduction system associated with $\mathscr{F}$

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 $\mathscr{F}$ 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 univoquely 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-expr**) calls the type system of the host language with the current environment. Typing of the union pattern (**t-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 `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-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-comp)** resembles a logical cut since it introduces an intermediary type $s_1$ (which is the other reason why the type system is non-algorithmic). The standard example is the leaves filter and type `T` informally discussed in Section 1. There is an infinite number of regular derivations for $\Gamma \vdash \underline{\text{leaves}}(\texttt{T}) = s$, each one giving a different $s$ with no lower bound (the limit being the context free language: $\{\, \texttt{[A}^n\texttt{B}^n\texttt{]} \mid n \geq 0 \}$ ). As for the rule for products, it is not as 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 $(\texttt{Any},\texttt{Any})$. 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-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 aspects is the decomposition itself. Indeed, while every subtype of $(\texttt{Any},\texttt{Any})$ 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 $\mho$ (pronounced *[pi:]* as for « pea ») that has better properties (with respect to subtyping). 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 $i|i{+}1|\cdots|j$):

$$f_1 = \mathsf{0..4} \rightarrow \texttt{A} | \mathsf{5..8} \rightarrow \texttt{B} \qquad f_2 = \mathsf{0..3} \rightarrow \texttt{C} | \mathsf{0..7} \rightarrow \texttt{D} \qquad f = (f_1,f_2)$$

and the types $t$ and $s$:

$$t = (\mathsf{0..4},\mathsf{0..3}) | (\mathsf{5..8},\mathsf{0..7}) \qquad s = (\mathsf{4..6},\mathsf{1..2})$$

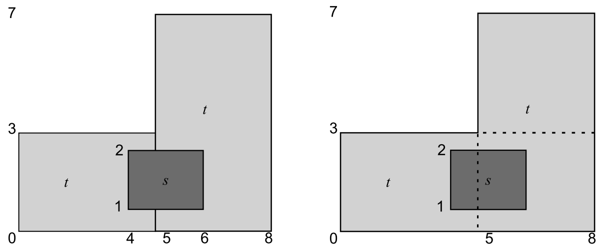
It is clear that $s \leq t$: by drawing all intervals on a plane, as in Figure 4, it is easy to check that the rectangle $s$ is contained in the "$\lrcorner$"-shaped $t$. However, $s$ overlaps the two rectangles which form $t$. If we decompose naively (i.e., syntactically) both types and compute the result type of $f$ by separately applying the filter on each component of the obtained decomposition, then we have:

$$\varnothing \vdash f(t) = (\mathtt{A},\mathtt{C})|(\mathtt{B},\mathtt{D})$$

but also:

$$\varnothing \vdash f(s) = (\mathtt{A}|\mathtt{B}\ ,\ \mathtt{C}|\mathtt{D})$$

the latter being a supertype of the former. Indeed, in $f_1$, a value




Two disjoint components: (0..4,0..3) and (5..8,0..7). $s$ overlaps both.

Two non-disjoint components: (0..8,0..3) and (5..8,0..7). $s$ is included in (0..8,0..3).

**Figure 4.** Syntactic and maximal product decompositions

in 4..6 can match either 0..4 or 5..8 (and likewise for $f_2$), hence the necessity, in this naïve approach, of returning the union of the output type of the two branches, reflecting in the type the fact that at run-time either branch can be taken. Therefore, we would have $s \leq t$ but $f(s) \not\leq f(t)$. The problem is solved by choosing a decomposition that is stable with respect to the subtyping relation. One such decomposition is the *maximal product decomposition*, noted $\mathcal{U}^{\kappa}$, which we define as follows:

**Definition 7** (Maximal product decomposition)**.** Let $t$ be a type such that $t \leq (\mathtt{Any},\mathtt{Any})$. Then, there exists $n \in \mathbb{N}$ such that:

$$t \simeq \bigvee_{i \in 1 \ldots n} (t_1^i,t_2^i)$$

and that:

i. $\forall s_1, s_2, (s_1,s_2) \leq t \implies \exists i{\in}\{1,..,n\}, (s_1,s_2) \leq (t_1^i,t_2^i)$
ii. $\forall i \in \{1,..,n\}, \forall k \in \{1,..,n\}, i \neq k \implies (t_1^i,t_2^i) \not\leq (t_1^k,t_2^k)$

Given $t$, $n$, and $t_j^i$'s as above, we note:

- $\mathcal{U}^{\kappa}(t) = \{(t_1^1,t_2^1), \ldots, (t_1^n,t_2^n)\}$
- $\mathcal{U}^{\kappa}_j{}^i(t) = t_j^i$
- $\mathtt{rank}(t) = n$.

Although the definition above is not immediate, the intuition it formalises is quite simple: the maximal decomposition of a type is the one formed *only* by (possibly overlapping) rectangles that are as large as possible. The right-hand side of Figure 4 shows the maximal decomposition of $t$, the one of $s$ being $s$ itself. Formally, the maximality of the components is specified by condition $(i.)$: every rectangle contained in $t$ is contained in a rectangle of its maximal decomposition (in our example, $s$ is a subtype of the (0..8,0..3) component of $t$). Condition $(ii.)$ instead ensures that *only* maximal components are used, by ruling out redundant ones (in our example $\{$(0..8,0..3), (5..8,0..7), (5..8,0..3)$\}$, which satisfies $(i.)$, would not be a maximal decomposition because of the extra (5..8,0..3)). The key property of our maximal decomposition is that if one product type is smaller than another, then every component of the maximal decomposition of the former is contained in at least one component of the maximal decomposition of the latter. It should be noted the resulting products in the decomposition may be non disjoint (in our example, the rectangle (5..8,0..3) is part of the two component of the decomposition). Since (**t-prod**) transforms maximal decompositions component-wise, this property is the keystone of the proof that the typing of filters is stable with respect to subtyping:

**Lemma 1** (Stability of filtering)**.** *For every filter $f$, types $s$ and $t$, and type environment $\Gamma$, if $s \leq t$ and $\Gamma \vdash f(t) = t'$, then $\Gamma \vdash f(s) = s'$ and $s' \leq t'$.*

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:

**Theorem 4** (Subject reduction)**.** *Let $f$, $\Gamma$, $t$ and $s$ be such that $\Gamma \vdash f(t) = s$. For every $\gamma : \Gamma$ and $v : t$, we have that $\gamma \vdash_e f(v) \leadsto v'$ implies $v' : s$*

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.

# 5. Typing algorithm

## 5.1 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 $\mathscr{F}$. 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 best 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 "$\mu$" notation to explicit the recursive binder. $\mu$-types are inductively generated by the following grammar:

$$\tau \ ::= \ \mu\alpha.\tau \mid \alpha \mid b \mid (\tau_1,\tau_2) \mid \tau_1|\tau_2 \mid \tau_1\&\tau_2 \mid \neg\tau \mid \mathtt{Empty} \mid \mathtt{Any}$$

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 $\alpha$, $\beta$,.... 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. [12, 6]). It is well-known that every recursive term ("$\mu$-term") represents a recursive tree and, conversely, we can choose a canonical $\mu$-term that represents a regular tree.

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

**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 $[\,]_\mu$ and $[\,]_\infty$ 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:

**Definition 10** (Annotated filters)**.**

$$\begin{array}{lcll} f & ::= & e \mid p \to f \mid f;f \mid (f,f) \mid f|f & \text{unchanged} \\ & \mid & f_{\mathbb{E}} & \text{annotation} \end{array}$$

**Structural rules**

**(a-expr)**
$$\frac{\texttt{type}_{\mathscr{A}}(\Gamma, \texttt{e}) = \sigma}{\Gamma, \Delta \vdash_{\mathscr{A}} e(\tau) = \sigma}$$

**(a-prod)**
$$\frac{\begin{array}{l} i = 1..\texttt{rank}(\tau) \\ j = 1..2 \end{array} \quad \Gamma, \Delta \vdash_{\mathscr{A}} f_j(\mathcal{U}^i_j(\tau)) = \sigma^i_j}{\Gamma, \Delta \vdash_{\mathscr{A}} (f_1,f_2)(\tau) = \bigvee_i (\sigma^i_1,\sigma^i_2)}$$

**(a-patt)**
$$\frac{\Gamma \cup \tau/p, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma \quad \tau \leq \wr p \wr \,\&\, \wr f \wr}{\Gamma, \Delta \vdash_{\mathscr{A}} (p \to f)(\tau) = \sigma}$$

**(a-union)**
$$\frac{\begin{array}{l} \tau \leq \wr f_1 \wr \,|\, \wr f_2 \wr \\ \tau_1 = \tau \,\&\, \wr f_1 \wr \\ \tau_2 = \tau \,\&\neg\, \wr f_1 \wr \end{array} \quad \begin{array}{l} \Gamma, \Delta \vdash_{\mathscr{A}} f_1(\tau_1) = \sigma_1 \\ \Gamma, \Delta \vdash_{\mathscr{A}} f_2(\tau_2) = \sigma_2 \end{array}}{\Gamma, \Delta \vdash_{\mathscr{A}} (f_1|f_2)(\tau) = \bigvee_{\{i|\tau_i \neq \mathsf{Empty}\}} \sigma_i}$$

**(a-comp)**
$$\frac{\begin{array}{ll} \Gamma, \Delta \vdash_{\mathscr{A}} f_1(\tau) = \sigma_1 & \tau \leq \wr f_1 \wr \\ \Gamma, \Delta \vdash_{\mathscr{A}} f_2(\sigma_1) = \sigma_2 & \sigma_1 \leq \wr f_2 \wr \end{array}}{\Gamma, \Delta \vdash_{\mathscr{A}} (f_1;f_2)(\tau) = \sigma_2}$$

**Memoization rules**

**(a-base-rec)**
$$\frac{(f, \tau, \sigma) \in \Delta}{\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma}$$

**(a-unfold-rec)**
$$\frac{\begin{array}{c} (f, \mu\alpha.\tau, \beta) \notin \Delta \text{ and } \beta \text{ fresh var.} \\ \Gamma, \Delta \cup \{(f, \mu\alpha.\tau, \beta)\} \vdash_{\mathscr{A}} f(\tau[\alpha \leftarrow \mu\alpha.\tau]) = \sigma \end{array}}{\Gamma, \Delta \vdash_{\mathscr{A}} f(\mu\alpha.\tau) = \mu\beta.\sigma}$$

**(a-unfold-non-rec)**
$$\frac{\begin{array}{c} (f, \tau, \alpha) \notin \Delta \text{ and } \alpha \text{ fresh var.} \\ \Gamma, \Delta \cup \{(f, \tau, \alpha)\} \vdash_{\mathscr{A}} f(\tau) = \sigma \end{array}}{\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \mu\alpha.\sigma}$$

**(a-annot)**
$$\frac{\begin{array}{c} \sigma = \mathit{choose}(\mathbb{E}) \text{ and } (f, \tau, \sigma) \notin \Delta \\ \Gamma, \Delta \cup \{(f, \tau, \sigma)\} \vdash_{\mathscr{A}} f(\tau) = \sigma' \;\; \sigma' \leq \sigma \end{array}}{\Gamma, \Delta \vdash_{\mathscr{A}} f_{\mathbb{E}}(\tau) = \sigma}$$

**Figure 5.** Deduction system associated with $\mathscr{F}_{\mathscr{A}}$

An annotation is a set $\mathbb{E}$ 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 $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$, 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 $(f, \tau, \sigma)$ where $f$ 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 the rules is the following: one must apply a memoization rule (if possible) before a structural rule, and a memoization rule must be followed by a structural rule (if the rule is not terminal). This order of evaluation is important, since it ensures the termination of the algorithm.

The only rule which requires some attention, since it does not derive directly from the non-algorithmic type system is the **(a-annot)** rule which cope with all the cases that made the type system behave non-algorithmically. Recall that there are two such cases. The first is the subsumption rule **(t-subs)**, 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 algorithmic type system, every instance of the rule **(t-subs)** can be erased, except for those occuring 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)**.** *For all filters $f$ and types $\tau$, the typing algorithm for $f(\tau)$ terminates.*

**Theorem 6** (Soundness of the typing algorithm)**.** *For all $\Gamma, \Delta, f, \tau,$ and $\sigma$, if $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$, 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: $[\Gamma]_\mu, \varnothing \vdash_{\mathscr{A}} f([t]_\mu) = [s]_\mu$ (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:*

- *i. 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).*
- *ii. One of the meta four operations $\tau/p$, or $\mathcal{U}(\tau)$, or testing for equality, or subtyping is applied to a type $\tau$ such that $FV(\tau) \neq \varnothing$.*
- *iii. The choice operator cannot find a suitable type amongst the given annotations for a certain filter $f_{\mathbb{E}}$.*

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 $\mathbb{A}_f$, 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 prevent 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 $f_1;f_2 \in \mathbb{A}_f$, $f_1 \equiv f'_{1\mathbb{E}}$ for some $\mathbb{E}$. For all type $\tau'$ occurring in the derivation of $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$, if $FV(\tau') \neq \varnothing$, then $\tau'$ 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 $\mathbb{A}_f = \{f_1^1;f_2^1, \ldots, f_1^n;f_2^n\}$, we call $\mathbb{E}_i$ the set of all the output types for $f_1^i$ in this derivation. A *t-labelling* of $f$, noted $[\![f]\!]_t$, is obtained from the filter $f$ by replacing all its $f_1^i;f_2^i$ sub-terms in $\mathbb{A}_f$ by the corresponding $f^i_{1\mathbb{E}_i};f_2^i$.

It is important to note that thanks to the regularity of the derivation of $\Gamma \vdash f(t) = s$, all the sets $\mathbb{E}_i$ 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 $\mathscr{F}_{\mathscr{A}}$ is complete with respect to the type system $\mathscr{F}$, that is: if $\Gamma \vdash f(t) = s$, then $\Gamma, \varnothing \vdash_{\mathscr{A}} [\![f]\!]_t([t]_\mu) = [s]_\mu$.*

## 6. Concrete language

### 6.1 Syntax

We have implemented our language into the ℂDuce 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[2]. In this section we give various examples of filters, to show how they can be used to implement and type common iterators on heterogeneous data-structures. We also present some useful extensions to the core filter algebra presented in Section 3.2, namely parametric filters and an XPATH encoding. For this purpose we introduce a more declarative and concrete syntax.

**Definition 13** (Concrete syntax)**.**

| | | | |
|---|---|---|---|
| $f$ | ::= | $e$ \| $p$ `->` $f$ \| $f$ `;` $f$ \| `(` $f$ `,` $f$ `)` \| $f$ `\|` $f$ | unchanged |
| | \| | `<` $f$ $f$ `>` $f$ | xml |
| | \| | `let filter` $\underline{x}$ `=` $f$ [ `and` $\underline{x}$ `=` $f$ ...] | binding |
| | \| | $\underline{x}$ | variable |
| $e$ | ::= | ... \| `apply` $f$ `to` $e$ [ `where` $a$ ] | application |
| $a$ | ::= | $\underline{x}$ `={` $\tau_1$ `,` ... `,` $\tau_n$ `}` [ `and` $a$ ] | annotation |

We use the same typesetting conventions as in Section 2 where we also explained most of the constructions above. In particular recall that the *xml* filter is just syntactic sugar over particular products. The same syntactic sugar will also be used in patterns which, in practice, turn out to be ℂDuce's patterns. What is new here with respect to the presentation of Section 2 is the construct `apply_to` which can now specify an optional (as indicated by the BNF brackets) annotation environment which associates filter variables with sets of types (the same variable must appear at most once). There is of course a connection between this syntax and the formal calculus of annotated filters:

`let filter` $\underline{f}$ `=` $f$
`apply` $\underline{g}$ `to` $v$ `where` $\underline{f}$ `=` $\{t_1, \ldots, t_n\}$

is equivalent to: $g[f \leftarrow f_{\{t_1,\ldots,t_n\}}](v)$.

### 6.2 Examples

We now describe some examples of increasing complexity.
**Pattern matching:** the customary `match with` (or Haskell's `case of`) construct can be thought of as syntactic sugar for an equivalent filter. For instance the pattern on the left-hand side becomes the filter on the right-hand side:

```
match x with                      let filter mymatch =
| [ Int* ] & y -> length y          x -> ( [Int*] &y -> length y
| Int -> string_of x                       | Int -> string_of x )
                                  apply mymatch to x
```

Type precision is retained: for both expressions the output type of `mymatch` applied to e.g. `[Int *]|Int` is `Int|String`. Note however that the output type of `mymatch` applied to a value of type `Int` is just `String`; such a precision is possible thanks to the rule **a-union** (or **t-union**) which discards output types corresponding to empty input types (here, if the input is an `Int`, then the `[Int *]` branch is never chosen, hence the type of its result is not taken into account whilst computing the output type).
**Map:** We have already seen in Section 2 the concatenation filter. It is also possible to define a generic map filter over lists like this:

```
let filter map =(f, l) -> (l;      let convert (Int ->String ;
  let filter aux_map =              String ->Int) =
   [] -> []                           | Int & i  -> string_of_int i
 | (x -> f x , aux_map)               | s        -> int_of_string s
)
apply map to (convert , [ 1 2 "17" "1" ])
val - : [ String String Int Int ]=[ "1" "2" 17 1 ]
```

*convert* is defined in ℂDuce as an overloaded function that transforms integers into strings and vice versa. The map filter behaves like the `map` function available in many functional languages but, unlike these, our map can be applied to heterogeneous lists and the overloaded behaviour of the "mapped" function is taken into account, as shown by the static type returned by the system (the line starting with "`val - :`" is the output of the interactive version of our prototype and consists of the *statically computed* type followed by the result).

[2] It should be noted that this early prototype makes use of the product decomposition operator already available in ℂDuce, 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 typing 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:

```
let filter replace_a = [] -> []
                     | ( (<a>[] -> <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>[] <b>[] <c>[] <d>[]]`, then $x$ can be bound to values of type `<b>[]`, `<c>[]`, or `<d>[]`, which in Hosoya's system yields the output type `[<d>[] (<b|c|d>[]) (<b|c|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>[] <b>[] <c>[] <d>[]]`. Hosoya justifies his typing policy by stressing that in some cases there is no lower bound to the output type[3] 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 $x \longrightarrow x$ 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:

```
let filter flatten =
      [] -> []
    | (x , _) -> ((<id id> flatten , flatten );
                  ((<_>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:

```
apply flatten to [mytree]
where flatten = { [(Item|Recipe|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:

$$p_1\text{->}e_1 | \ldots | p_n\text{->}e_n$$

The programmer here is left with two choices. If one encapsulates 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:

```
let filter map f =              let filter h = x&Int -> -x
   [] -> []                                  | x&Bool -> not x
   | (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 Wadlers *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`. `//` represent 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:

`<a>[ 1 <a>[2] ]`

and the path expression:

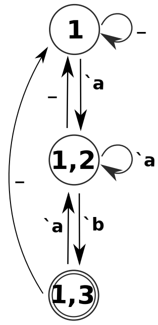
`//a//a`

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

1. `//a` applied to `<a>[ 1 <a>[2] ]` returns `[<a>[ 1 <a>[2] ] <a>[2]]`. Indeed, the first node has tag <a> and so does its only child, hence both appear in the result.
2. To this intermediary result, we apply `//a` again and obtain: `[<a>[ 1 <a>[2] ] <a>[2] <a>[2]]`, the first two elements coming from `//a` applied to `<a>[ 1 <a>[2] ]` and the third one coming from `//a` applied to `<a>[2]`.

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. XDuce, ℂDuce 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 ℂDuce, the output type will be for instance, `[ <a>[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

[3] Even the simple filter $x \longrightarrow (x, x)$ cannot be typed precisely when applied to `Int` since its most precise type is the infinite union of all pairs $(n, n)$ for $n \in \mathbb{N}$ (our system instead deduces `(Int,Int)`)




```
let filter f1 =
     <(‘a ->‘a)> I_f12;<_>x ->x
   | <( x ->x )> I_f1;<_>x ->x
   | _ -> []

and filter f12 =
 <(‘a ->‘a)> I_f12; <_> x -> x
 |  res ->
       <(‘b ->‘b)>I_f13; <_> x -> (res , x)
 | <(x ->x)> I_f1; <_> x ->x
 | _ -> []

and filter f13 =
     <(‘a ->‘a)> I_f12; <_> x -> x
   | <(x ->x)> I_f1; <_> x ->x
   | _ -> []

and filter I_f1 =
   [] -> []
   | (f1 , I_f1);concat
...
```

**Figure 6.** DFA encoding the `//a/b` expression

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 *fi*'s filters correspond to the states of the DFA. The transitions are encoded by the branches of each *fi* filter. Finally, each I_*fi* is a simple auxiliary filters which iterates a given *fi* on a sequence of XML values and concatenates the intermediate results (we only gave the code for I_f1, the others being similar). As we see in f1, if the input tag is an `<a>`, then the we evaluate the filter I_f12 which iterates f12 on every child of the input value. Likewise, in f12, if the input tag is a `<b>`, then we iterate f13 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 *fi*'s and I_*fi*'s appear on the left-hand side of a composition and thus 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>[] <b>[<c>[]] <a>[<b>[<c>[] <c>[]]]]

apply //a/b 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 automatised 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


[1] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. on Programming Languages*, 15(4):575–631, 1993.

[2] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyễn. Type-based XML projection. In *VLDB 2006*, pages 271–282, 2006.

[3] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03*, pages 51–63, 2003.

[4] V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL 05*, number 3350 in LNCS, pages 235–252, 2005.

[5] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proc. of *PPDP '05* (full version) and *ICALP '05,* LNCS n. 3580, (summary), 2005. Joint ICALP-PPDP keynote talk.

[6] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[7] W3C: DTD specifications. http://www.w3.org/TR/REC-xml/#dt-doctype, 2006.

[8] A. Frisch. Regular tree language recognition with static information. In *Proc. IFIP Conf. on Theor. Comput. Sci. (TCS)*. Kleuwer, 2004.

[9] A. Frisch. OCaml + XDuce. *SIGPLAN Not.*, 41(9):192–200, 2006.

[10] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02*, pages 137–146. IEEE Computer Society Press, 2002.

[11] Alain Frisch and Haruo Hosoya. Towards practial typechecking for macro tree transducers. In *DBPL*, 2007.

[12] V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2003.

[13] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *PLAN-X*, 2005.

[14] H. Hosoya. Regular expression filters for XML. In *Programming Languages Technologies for XML (PLAN-X)*, pages 13–27, 2004.

[15] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *POPL '05*, pages 50–62, 2005.

[16] H. Hosoya and B.C. Pierce. Regular expression pattern matching for XML. In *POPL '01*, 2001.

[17] H. Hosoya and B.C. Pierce. XDuce: A typed XML processing language. In *ACM Trans. on Internet Tech.*, pages 117–148, 2003.

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

[19] R. Lämmel. Scrap your boilerplate with XPath-like combinators. In *POPL'07, Proceedings*. ACM Press, January 2007.

[20] M.Y. Levin and B.C. Pierce. Type-based optimization for regular patterns. In *DBPL '05*, August 2005.

[21] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML Type checking with macro tree transducers. In *ACM PODS*, pages 283–294, 2005.

[22] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *ICDT*, pages 254–268, 2007.

[23] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comput. Syst. Sci.*, 66(1), 2003.

[24] M.-J. Nederhof. Practical experiments with regular approximation of context-free languages. *Computat. Linguistics*, 26(1):17–44, 2000.

[25] OASIS Committee Specification: Relax-NG. http://relaxng.org/spec-20011203.html, 2001.

[26] M. Sulzmann and K. Zhuo Ming Lu. A type-safe embedding of XDuce into ML. *El. Notes Theor. Comp. Sci.*, 148(2):239–264, 2006.

[27] M. Sulzmann and K. Zhuo Ming Lu. XHaskell. In *PLAN-X*, 2006.

[28] J. Vouillon. Polymorphic regular tree types and patterns. In *POPL*, pages 103–114, 2006.

[29] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

[30] W3C: XML Schema. http://www.w3.org/XML/Schema, 2004.

[31] W3C: XML Path Language Ver. 1.0. http://www.w3.org/TR/xpath, 1999.