Polymorphism and XDuce-style patterns

Jérôme Vouillon
CNRS and Université Paris 7
Jerome.Vouillon@pps.jussieu.fr

Abstract

We present an extension of XDuce, a programming language dedicated to the processing of XML documents, with polymorphism and abstract types, two crucial features for programming in the large. We show that this extension makes it possible to deal with first class functions and eases the interoperability with other languages. A key mechanism of XDuce is its powerful pattern matching construction and we mainly focus on this construction and its interaction with abstract types. Additionally, we present a novel type inference algorithm for XDuce patterns, which works directly on the syntax of patterns.

1. Introduction

XDuce [14] is a programming language dedicated to the processing of XML documents. It features a very powerful type system: types are regular tree expressions [15] which correspond closely to the schema languages used to specify the structure of XML documents. The subtyping relation is extremely flexible as it corresponds to the inclusion of tree automata. Another key feature is a pattern matching construction which extends the algebraic patterns popularized by functional languages by using regular tree expressions as patterns [13].

In this paper, we aim at extending in a seamless way the XDuce type system and pattern construction with ML-style prenex polymorphism and abstract types. These are indeed crucial features for programming in the large in a strongly typed programming language. In our extension, patterns are not allowed to break abstraction. This crucial property makes it possible to embed first class functions and foreign values in a natural way into XDuce values.

In another paper [21], we present a whole calculus dealing with polymorphism for regular tree types. Though most of the results in that paper (in particular, the results related to subtyping) can be fairly easily adapted for an extension of XDuce, a better treatment of patterns is necessary. Indeed, a straightforward application of the results would impose severe restrictions on patterns. For instance, binders and wildcards would be required to occur only in tail position. The present paper is therefore mostly focused on patterns and overcomes these limitations.

Additionally, we present a novel type inference algorithm for XDuce patterns, which works directly on the syntax of patterns, rather than relying on a prior translation to tree automata. This way, better type error messages can be provided, as the reported types are closer to the types written by the programmer. In particular, type abbreviations can be preserved, while they would be expanded by the translation into tree automata.

The paper is organized as follows. We introduce the XDuce type system (section 2) and present the extension (section 3). Then, we formalize patterns (section 4) and provide algorithms for checking patterns and performing type inference (section 5). Related works are presented in section 6.

2. A Taste of XDuce

XDuce values are *sequences* of elements, where an *element* is characterized by a *name* and a *contents*. (Elements may also contain *attributes*, both in XDuce and XML. We omit attributes here for the sake of simplicity.) This contents is itself a sequence of elements. These values corresponds closely to XML documents, such as this address book example.

```
<addrbook>
   <person>
      <name> Haruo Hosoya </name>
      <email> hosoya </email>
   </person>
   <person>
       <name> Jerome Vouillon </name>
      <tel> 123 </tel>
   </person>
</addrbook>
```

XDuce actually uses a more compact syntax, which we also adopt in this paper:

```
addrbook[
  person[name["Haruo Hosoya"], email["hosoya"]],
  person[name["Jerome Vouillon"], tel["123"]]]
```

The shape of values can be specified using *regular expression types*. A sequence of elements is described using a regular expression. Mutually recursive type definitions make it possible to deal with the nested nature of values. Here are the type definitions for address books.

```
type Addrbook = addrbook[Person*]
type Person   = person[Name,Email*,Tel?]
type Name     = name[String]
type Email    = email[String]
type Tel      = tel[String]
```

These type definitions can be read as follows. An `Addrbook` value is an element with name `addrbook` containing a sequence of any number of `Person` values. A `Person` value is an element with name `person` containing a `Name` value followed by a sequence of `Email` values and optionally a `Tel` value. Values of type `Name`, `Email`, and `Tel` are all composed of a single element containing a string of characters.

There is a close correspondence between regular expression types and tree automata [5]. As the inclusion problem between tree automata is decidable, the subtyping relation can be simply defined as language inclusion [15]. This subtyping relation is extremely powerful. It includes associativity of concatenation (type `A,(B,C)` is equivalent to type `(A,B),C`), distributivity rules (type `A,(B|C)` is equivalent to type `(A,B)|(A,C)`).

In order to present the next examples, we find it convenient to use the following parametric type definition for lists:

```
type List{X} = element[X]*
```

Parametric definitions are not currently implemented in XDuce, but are a natural extension and can be viewed as just syntactic sugar: all occurrences of `List{T}` (for any type `T`) can simply be replaced by the type `element[T]*` everywhere in the source code.

Another key feature of XDuce is *regular expression patterns*, a generalization of the algebraic patterns popularized by functional languages such as ML. These patterns are simply types annotated with *binders*. Consider for instance this function which extracts the names of a list of persons.

```
fun names (lst : Person*) : List{String} =
   match lst with
     () ->
        ()
   | person [name [nm : String], Email*, Tel?],
     rem : Person* ->
        element [nm], names (rem)
```

The function `names` takes an argument `lst` of type `Person*` and returns a value of type `List{String}`. The body of the function is a pattern matching construction. The value of the argument `lst` is matched against two patterns. If it is the empty sequence, then it will match the first pattern `()` (the type `()` is the type of the empty sequence `()`), and the function returns the empty sequence. Otherwise, the value must be a non-empty sequence of type `Person*`. Thus, it is an element of name `person` followed by a sequence of type `Person*`, and matches the second pattern. This second pattern contains two binders `nm` and `rem` which are bound to the corresponding part of the value.

Some *type inference* is performed on patterns: the type of the expression being matched is used to infer the type of the values that may be bound to a binder. By taking advantage of this, the function `names` can be rewritten more concisely using *wildcard* patterns[1] as follows. The type of the binders `nm` and `rem` are inferred to be respectively `String` and `Person*` by the compiler.

```
fun names (l : Person*) : List{String} =
   match l with
     () ->
        ()
   | person [name [nm : _], _], rem : _ ->
        element [nm], names (rem)
```

3. Basic Ideas

We want to extend regular expression types and patterns with ML-style polymorphism (with explicit type instantiation) and abstract types. Such an extension is interesting for numerous reasons. First, it makes it possible to describe XML documents in which arbitrary subdocuments can be plugged. A typical example is the SOAP envelop. Here is the type of SOAP messages and of a function that extracts the body of a SOAP message.

```
type Soap_message{X} =
   envelope[header[...], body[X]]
fun extract_body :
   forall{X}. Soap_message{X} -> X
```

A more important reason is that polymorphism is crucial for programming in the large. It is intensively used for collection datastructures. As an example, we present a generic map function over lists. This function has two type parameters `X` and `Y`.

[1] XDuce actually uses the pattern `Any` as a wildcard pattern.

```
fun map{X}{Y}
       (f : X -> Y)(l : List{X}) : List{Y} =
   match l with
     () ->
        ()
   | element[x : _], rem : _ ->
        element[f(x)], map{X}{Y}(f)(rem)
```

When using a polymorphic function, type arguments may have to be explicitly given, as shown in the following expression where the `map` function is applied to the identity function on integers and to the empty list:

```
map{Int}{Int} (fun (x : Int) -> x) ().
```

Indeed, it is possible to infer type arguments in simple cases, using an algorithm proposed by Hosoya, Frisch and Castagna [12], but not in general, as a best type argument does not necessarily exist: the problem is harder in our case due to function types which are contravariant on the left.

Abstract types facilitate interoperability with other languages. Indeed, we can consider any type from the foreign language as an abstract type as far as XDuce is concerned. For instance, the ML type[2] `int` can correspond to some XDuce type `Int`. This generalizes to parametric abstract types: to the ML type `int array` would correspond the polymorphic XDuce type `Array{Int}`. Furthermore, if the two languages share the same representation of functions, ML function types can be mapped to XDuce function types (and conversely). Thus, for instance, a function of type `int->int` can be written in either language and used directly in the other language without cumbersome conversion.

In order to preserve abstraction and to deal with foreign values that may not support any introspection, some patterns should be disallowed. For instance, this function should be rejected by the type checker as it tries to test whether a value `x` of some abstract type `Blob` is the empty sequence.

```
fun f (x : Blob) : Bool =
   match x with
     () -> true
   | _  -> false
```

Another restriction is that abstract types cannot be put directly in sequences. Indeed, it does not make sense to concatenate two values of the foreign language (two ML functions, for instance). In order to be put into a sequence, they must be wrapped in an element. As a type variable may be instantiated to an abstract type, and as we want to preserve abstraction for type variables too, the same restrictions apply to them: a pattern `a[],X,b[]` implicitly asserts that the variable `X` stands for a sequence, and thus would limit its polymorphism.

There are different ways to deal with type variables and abstract types occurring syntactically in patterns. The simplest possibility is not to allow them. Instead, one can use wildcards and rely on type inference to assign polymorphic types to binders. This approach is taken in the related work by Hosoya, Frisch and Castagna [12]. Another possibility is to consider that type variables should behave as the actual types they are instantiated to at runtime. This is a natural approach, but this implies that patterns do not preserve abstraction. It is also challenging to implement this efficiently, though it may be possible to get good results by performing pattern compilation (and optimization) at run-time. Finally, it is not clear in this case how abstract types should behave in patterns. We propose a middle-ground, by restricting patterns so that their behaviors do not depend on what type variables are instantiated to, and on what abstract

[2] We consider here ML as the foreign language, as XDuce is currently implemented in OCaml. But this would apply equally well to other languages.

types stand for. In other words, patterns are not allowed to break abstraction. As a consequence, type variables can be compiled as wildcards. In other words, type variables and abstract types occurring in patterns can be considered as annotations which are checked at compile time but have no effect at run-time. We indeed feel it is interesting to allow type variables and abstract types in patterns. A first reason is that it is natural to use patterns to specify the parameters of a functions. And we definitively want to put full types there. For instance, we should be able to write such a function:

```
fun apply{X}{Y}
    (funct[f : X --> Y], arg[x : X]) : Y = f(x)
```

Another reason is that one may want to reuse a large type definition containing abstract types in a pattern, and it would be inconvenient to have to duplicate this definition, replacing abstract types with wildcards. Finally, the check can be implemented easily: the type inference algorithm can be used to find the type of the values that may be matched against any of the type variables occurring in the pattern, so one just has to check that this type is a subtype of the type variable (this usually means that the type is either empty or equal to the type variable, but some more complex relations are possible, as we will see in section 4.3).

4. Specifications

We now specify our pattern matching construction, starting from the data model, continuing with types and patterns, before finally dealing with the whole construction.

4.1 Values

We assume given a set of *names* l and a set of *foreign values* e. A *value* v is either a foreign value or a *sequence* f of *elements* $l[v]$ (with name l and *contents* v).

v	$::=$	e	foreign value
		f	sequence
f	$::=$	$l[v], \ldots, l[v]$	

We write ϵ for the *empty sequence*, and f, f' for the *concatenation* of two sequences f and f'.

Note that strings of characters can be embedded in this syntax by representing each character `c` as an element whose name is this very character and whose contents is empty: $\mathtt{c}[\epsilon]$. This encoding was introduced by Gapeyev and Pierce [9].

4.2 Patterns

We start by two comments clarifying the specification of patterns. First, in all the examples given up to now, in a pattern element `L[T]`, the construction `L` stands for a single name. It actually corresponds in general to a set of names. This turns out to be extremely convenient in practice. For instance, this can be used to define character sets (remember that characters are encoded as names). Second, abstract types and type variables are very close notions. Essentially, the distinction is a difference of scope: an abstract type stands for a type which is unknown to the whole piece of code considered, while a type variable has a local scope (typically, the scope of a function). Thus, for patterns, we can unify both notions. Parametric abstract types can be handled by considering each of their instances as a distinct type variable. Thus, the two types `Array{Int}` and `Array{Bool}` correspond each to a distinct type variable in our formalization of patterns. Similarly, each function type `T2-->T1` corresponds to a distinct type variable. We explain in section 4.3 how subtyping can be expressed for these types.

As a running example, we consider the pattern matching code in function `map`:

```
match l with
  () --> ...
| element[x : _], rem : List{X} --> ...
```

where `l` has type `List{X}`.

Such a grammar-based syntax of patterns is convenient for writing patterns but typically does not reflect their internal representation in a compiler. For instance, it assumes a notion of pattern names (such as `List{X}` or `Name`) which may be expanded away at an early stage by the compiler. Binders may also be represented in a different way. Finally, this notation is not precise about subpattern identity: for instance, in the pattern `a[_]|b[_]`, it in not clear whether one should consider the two occurrences of the wildcard pattern `_` as two different subpatterns, or as a single subpattern. The distinction matters as a compiler usually does not identify expressions which are structurally equal. In particular, one should be careful not to use any termination argument that relies on structural equality. Another reason is that we need to be able to specify precisely how a value is matched by a pattern. This is especially important for type inference (section 4.9), where we get a different result depending on whether we infer a single type for both occurrences of the wildcard pattern or a distinct type for each occurrence.

Thus, we define a more abstract representation of patterns which provides more latitude for actual implementations. A pattern is a rooted labeled directed graph. Intuitively, this graph can be understood as an in-memory representation of a pattern: nodes stands for memory locations and edges specify the contents of each memory location. To be more accurate, a pattern is actually a hypergraph, as edges may connect a node to zero, one or several nodes: for instance, for a pattern `()`, there is an (hyper)edge with source the location of the whole pattern and with no target, while for a pattern `P,Q`, there is an (hyper)edge connecting the location of the whole pattern to the location of subpatterns `P` and `Q`.

We assume given a family of *name sets* L, a set of *type variables* X and a set $\mathcal{X}$ of *binders* x. Formally, a pattern is a quadruple $(\Pi, \phi, \pi_0, \mathcal{B})$ of

- a finite set Π of *pattern locations* π;
- a mapping $\phi : \Pi \to \mathcal{C}(\Pi)$ from pattern locations to *pattern components* $p \in \mathcal{C}(\Pi)$, defined below;
- a *root* pattern location $\pi_0 \in \Pi$.
- a relation $\mathcal{B} \subseteq \mathcal{X} \times \Pi$ between binders and pattern locations.

Pattern components $\mathcal{C}(\Pi)$ are defined by the following grammar, parameterized over the set Π of pattern locations.

p	$::=$	$L[\pi]$	element pattern
		ϵ	empty sequence pattern
		π, π	pattern concatenation
		$\pi \cup \ldots \cup \pi$	pattern union
		$\pi*$	pattern repetition
		$\square$	wildcard
		X	type variable

Binders do not appear directly in patterns. Instead they are specified by a relation between binder names and pattern locations. This allows us to simplify significantly the presentation of the different algorithms on patterns. Indeed, most of them simply ignore binders.

As an example, the two patterns:

`()` and `element[x:_],rem:List{X}`

can be formally specified respectively as:

$$(\Pi, \phi, 1, \emptyset) \quad \text{and} \quad (\Pi, \phi, 2, \{(\mathtt{x}, 4), (\mathtt{rem}, 5)\})$$

where the set of pattern locations is:

$$\Pi = \{1, 2, 3, 4, 5, 6, 7\}$$

$$\begin{tikzcd}
\textcircled{1} \arrow[r, dash, "\epsilon"] & \vdash & & & \\
\textcircled{2} \arrow[r, "\_,\_"] \arrow[dr, bend right] & 3 \arrow[r, "\texttt{element}[\_]"] & 4 \arrow[r, dash, "\square"] & \vdash & \\
 & 5 \arrow[r, "\_*"] & 6 \arrow[r, "\texttt{element}[\_]"] & 7 \arrow[r, dash, "\texttt{X}"] & \vdash
\end{tikzcd}$$

Figure 1. Graphical Depiction of Two Patterns

$$\frac{\phi(\pi) = \pi', \pi''}{\pi' \text{ seq}} \qquad \frac{\phi(\pi) = \pi', \pi''}{\pi'' \text{ seq}} \qquad \frac{\phi(\pi) = \pi'*}{\pi' \text{ seq}}$$

$$\frac{\pi \text{ seq} \qquad \phi(\pi) = \pi_1 \cup \ldots \cup \pi_n}{\pi_i \text{ seq}}$$

Figure 2. Locations in a Sequence π seq

$$\frac{\phi(\pi) = L[\pi']}{\pi \text{ wf}} \qquad \frac{\phi(\pi) = \epsilon}{\pi \text{ wf}} \qquad \frac{\begin{array}{c}\pi' \text{ wf} \qquad \pi'' \text{ wf}\\ \phi(\pi) = \pi', \pi''\end{array}}{\pi \text{ wf}}$$

$$\frac{\begin{array}{c}\pi_1 \text{ wf} \qquad \ldots \qquad \pi_n \text{ wf}\\ \phi(\pi) = \pi_1 \cup \ldots \cup \pi_n\end{array}}{\pi \text{ wf}} \qquad \frac{\phi(\pi) = \square}{\pi \text{ wf}}$$

$$\frac{\neg(\pi \text{ seq}) \qquad \phi(\pi) = X}{\pi \text{ wf}} \qquad \frac{\pi' \text{ wf} \qquad \phi(\pi) = \pi'*}{\pi \text{ wf}}$$

Figure 3. Location Well-Formedness π wf

and the mapping from pattern locations to pattern components is the function ϕ defined by:

$$\begin{array}{lll}
\phi(1) = \epsilon & & \\
\phi(2) = 3, 5 & \phi(3) = \texttt{element}[4] & \phi(4) = \square \\
\phi(5) = 6* & \phi(6) = \texttt{element}[7] & \phi(7) = \texttt{X}
\end{array}$$

(We write `element` for the name set containing only the name `element`.) A graphical depiction of the formal representation of the two patterns is given in figure 1. The two root locations 1 and 2 are circled. Edges are labeled with the corresponding component. One can see three kind of edges on this picture: the edges with labels ϵ, $\square$ and `X` have no target; one edge with label $\_,\_$ has two targets 3 and 5 and corresponds to the component 3, 5; some edges with label $\_*$ or `element[_]` has a single target. Note that the locations 5 to 7 correspond to the expansion of type `List{X}`.

Not all patterns are correct. The most important restriction is that cycles are not allowed except when going through an element $L[\pi']$: for instance, the pattern

```
Balanced = a[], Balanced, b[]
```

should be rejected, while the pattern

```
Tree = leaf[] | node[Tree, Tree]
```

is accepted. This restriction ensures that the set of values matching a given pattern is a regular tree language[3]. The other restriction is that pattern variables should not occur in sequences. For instance, the patterns `a[],X` and `X*`, where `X` is a pattern variable, are rejected. Indeed, the semantics of a pattern variable may contain foreign values, which cannot be concatenated. These two restrictions are formally specified using a well-formedness condition. First, we define when a pattern location is in a sequence (figure 2). Then, we define the well-formedness condition for pattern locations (figure 3). There is one rule per pattern component. For all rules but one, in order to deduce that a pattern location is well-formed, one must first show that its subpatterns are themselves well-formed. This ensures that there is no cycle. The exception is the rule for element patterns $L[\pi']$, hence cycles going through elements are allowed. The rule for type variables X additionally requires that the pattern location is not in a sequence. Finally, a pattern is *well-formed* if all its locations are. These restrictions could also have been enforced syntactically [11, 17], but we prefer to keep the syntax as simple and uniform as possible. In the remainder of this paper, all patterns are implicitly assumed to be well-formed.

4.3 Typing Environments

In order to provide a semantics to patterns, we assume given a class of binary relations between values and types variables, which we call typing environments. Equivalently, we can consider a typing environment as a function from type variables to their semantics which is a set of values). We have two motivations for restricting ourselves to a class of such relations rather than allowing all relations. First, some type variables may have a fixed semantics, identical in all typing environments. This makes it possible to define the type of a function `T2—>T1` (assuming that `T1` and `T2` are pure regular expression types, without type variables). The semantics of some type variables may also be correlated to the semantics of other type variables. For instance, the semantics of the type `Array{X}` depends on the semantics of the type variable `X`. Second, for semantic reasons, the semantics of any type, and thus the semantics of type variables, may be required to satisfy some closure properties. This is the case for instance in the ideal model [16].

4.4 Pattern Matching

In order for the algorithms presented in this paper to be implementable, the family of name sets L should be chosen so that the following predicates are decidable:

- the inclusion of a name in a name set: $l \vartriangleleft L$;
- the non-emptiness of a name set: $\vartriangleleft L$ (that is, there exists a name l such that $l \vartriangleleft L$);
- the non-disjointness of two name sets: $L_1 \bowtie L_2$ (that is, there exists a name l such that $l \vartriangleleft L_1$ and $l \vartriangleleft L_2$).

Furthermore, for technical reasons (see section 4.7), there must be a name set $\top$ containing all names.

The semantics of a pattern $(\Pi, \phi, \pi_0, \mathcal{B})$ is given in figure 4 using inductive rules. It it parameterized over a typing environment, that is a relation $v \vartriangleleft X$ which provides a semantics to each type variable. We define simultaneously the relation $v \vartriangleleft \pi$ (the value v matches the pattern location π) and a relation $f \vartriangleleft_* \pi$ (the sequence f matches a repetition of the pattern location π). Then, a value v matches a whole pattern if it matches its root location, that is, $v \vartriangleleft \pi_0$.

A *match* of a value v against a location π is a derivation of $v \vartriangleleft \pi$. Given such a match, we define the *submatches* as the set of assertions $v' \vartriangleleft \pi'$ which occur in the derivation. These submatches indicate precisely which parts of the value is associated to each

[3] Actually this is not quite accurate due to type variables. In order to state the regularity property precisely, the semantics of patterns should be defined in two steps. The first step would be a semantics in which values contain variables matching the variables in the pattern. With this initial semantics, the denotation of a pattern would indeed be a regular tree language. The second step would correspond in substituting values for the type variables.

MATCH-ELEMENT
$$\frac{l \lhd L \qquad v \lhd \pi' \qquad \phi(\pi) = L[\pi']}{l[v] \lhd \pi}$$

MATCH-EPS
$$\frac{\phi(\pi) = \epsilon}{\epsilon \lhd \pi}$$

MATCH-CONCAT
$$\frac{f' \lhd \pi' \qquad f'' \lhd \pi'' \qquad \phi(\pi) = \pi', \pi''}{f', f'' \lhd \pi}$$

MATCH-UNION
$$\frac{v \lhd \pi_i \qquad \phi(\pi) = \pi_1 \cup \ldots \cup \pi_n}{v \lhd \pi}$$

MATCH-WILD
$$\frac{\phi(\pi) = \Box}{v \lhd \pi}$$

MATCH-ABSTRACT
$$\frac{v \lhd X \qquad \phi(\pi) = X}{v \lhd \pi}$$

MATCH-REP
$$\frac{f \lhd_* \pi' \qquad \phi(\pi) = \pi' *}{f \lhd \pi}$$

MATCH-REP-EPS
$$\epsilon \lhd_* \pi$$

MATCH-REP-ONCE
$$\frac{f \lhd \pi}{f \lhd_* \pi}$$

MATCH-REP-CONCAT
$$\frac{f \lhd_* \pi \qquad f' \lhd_* \pi}{f, f' \lhd_* \pi}$$

Figure 4. Matching a Value against a Pattern $v \lhd \pi$

location in the pattern. They can thus be used to define which value to associate to each binder during pattern matching.

We choose to use a non-deterministic semantics: there may be several ways to match a value against a given pattern. The reasons are twofold. First, this yields much simpler specifications and algorithms. Second, we don't want to commit ourselves to a particular semantics. Indeed, we may imagine that the programmer is allowed to choose between different semantics, such as a first-match policy (Perl style) or a longest match policy (Posix style). Our algorithms will be sound in both cases, without any adaptation needed.

4.5 Types

A pattern specifies a set of values: the set of values which matches this pattern. So, patterns can be used as types. More precisely, we define a *type* as a pattern $(\Pi, \phi, \pi_0, \mathcal{B})$ with no wildcard $\Box$ (that is, $\phi(\pi)$ is different from the wildcard $\Box$ for all locations $\pi \in \Pi$) and no binder (the relation $\mathcal{B}$ is empty).

The wildcard has a somewhat ambiguous status: it stands for any value when not in a sequence, but only stands for sequence values when it occurs inside a sequence. For instance, the values accepted by a pattern `_,P` are not the concatenations of the values accepted by pattern `_` and pattern `P`, as some values in pattern `_` cannot be concatenated. Due to this ambiguous status, type inference would be more complicated if the wildcard pattern was allowed in types.

4.6 Subtyping

We define a subtyping relation $<:$ in a semantic way on the locations $\pi_1 \in \Pi_1$ and $\pi_2 \in \Pi_2$ of two patterns $P_1 = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$ and $P_2 = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2)$ by $\pi_1 <: \pi_2$ if and only if, for all typing environments and for all values v, the assertion $v \lhd \pi_1$ implies the assertion $v \lhd \pi_2$. Two patterns are in a subtype relation, written $P_1 <: P_2$, if their root locations are. The actual algorithmic subtyping relation used for type checking does not have to be as precise as this semantics subtyping relation. This will simply result in a loss of precision.

4.7 Bidirectional Automata and Disallowed Matchings

In the previous section, the semantics of patterns is specified in a declarative way. In order to clarify the operational semantics

$$l[v], f \xrightarrow{\mathsf{l}} l,v,f \qquad\qquad f, l[v] \xrightarrow{\mathsf{r}} l,v,f$$

Figure 5. Value Decomposition $v \longrightarrow^{\delta} l,v,v$

LABEL-TRANS
$$\frac{\sigma \xrightarrow{\delta} L,\sigma_1,\sigma_2 \qquad v \xrightarrow{\delta} l,v_1,v_2 \qquad l \lhd L \qquad v_1 \in \sigma_1 \qquad v_2 \in \sigma_2}{v \in \sigma}$$

EPS-TRANS
$$\frac{\sigma \rightsquigarrow \sigma' \qquad v \in \sigma'}{v \in \sigma}$$

ACCEPT
$$\frac{\sigma \downarrow v}{v \in \sigma}$$

Figure 6. Automaton Semantics $v \in \sigma$

of patterns, we now define a notion of tree automata, which we call *bidirectional automata*. These automata are used in particular to specify which patterns should be rejected. They capture the idea that a value is matched from its root and that a sequence is matched one element at a time from its extremities. Still, some freedom is left over the implementation. In particular, the automata do not mandate any specific strategy (such as left to right) for the traversal of sequences. This is achieved thanks to an original feature of the automata: at each step of their execution, the matched sequence may be consumed from either side. This symmetry in the definition of automata results in symmetric restrictions on patterns: if a pattern is disallowed, then the pattern obtained by reversing the order of all elements in all sequences is also disallowed. We believe this is easier to understand for a programmer. Additionally, this feature is a key ingredient for our type inference algorithm.

Formally, a bidirectional automaton is composed of

- a finite set Σ of *states* σ;
- an *initial state* $\sigma_0 \in \Sigma$;
- a set of *labeled transitions* $\sigma \longrightarrow^{\delta} L,\sigma,\sigma$;
- a set of *epsilon transitions* $\sigma \rightsquigarrow \sigma$;
- an *immediate acceptance relation* $\sigma \downarrow v$.

The transitions are annotated by a tag $\delta \in \{\mathsf{l}, \mathsf{r}\}$ which indicates on which side of the matched sequence they take place: either on the left (tag l) or on the right (tag r). The semantics of automata is given in figure 6: the relation $v \in \sigma$ specifies when a value v is *accepted* by a state σ of the automaton. A value is accepted by a whole automaton if it is accepted by its initial state σ_0. The rule LABEL-TRANS states that, starting from a goal $v \in \sigma$, a labeled transition $\sigma \longrightarrow^{\delta} L,\sigma_1,\sigma_2$ may be performed provided that the value v decomposes itself on side δ into a element with name l and contents v_1 followed by a value v_2 (value decomposition is specified in figure 5). The name l must furthermore be included in the name set L. One then gets two subgoals $v_1 \in \sigma_1$ and $v_2 \in \sigma_2$. The rule EPS-TRANS moves to another state of the automaton while remaining on the same part of the value. Usually, automata have a set of accepting states, which all accept the empty sequence ϵ. Here, we use an accepting relation, so that a state may accept whole values at once (rule ACCEPT). This is necessary to deal with type variables `X` that match a possibly non-regular set of values and with foreign values e which are not sequences. The use of an epsilon transition relation simplify the translation from patterns to automata. It also keeps the automata smaller. Indeed, eliminating epsilon transitions may make an automaton quadratically larger. Note that our automata are non-deterministic.

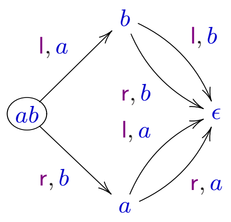

Figure 7. Example of Bidirectional Automaton

Not all patterns could be translated into deterministic automata as top down deterministic tree automata are strictly less powerful than non-deterministic ones [5].

An example of bidirectional automaton is depicted in figure 7. This automaton recognizes the sequence a[],b[]. It has four states $\Sigma = \{ab, a, b, \epsilon\}$. The initial state ab is circled. The labeled transitions are all of the form $\sigma \longrightarrow^{\delta} L,\epsilon,\sigma'$ and are represented by an arrow from state σ to state σ' with label the pair δ, L. There is no epsilon transition. The acceptance relation, not represented, is reduced to $\epsilon \downarrow \epsilon$. In order to recognize the sequence a[],b[], one can first consume a[] from the left and then the remaining part b[] from either side, or consume the part b[] before the part a[].

The automata we build below satisfy some commutation properties, which ensure that the strategy used to match a value is not important. For instance, one can choose to consume values only from the left, or only from the right, or any combination of these two strategies. In all cases, the set of accepted values remain the same. We do not state these properties.

We now specify the translation of a pattern $(\Pi, \phi, \pi_0, \mathcal{B})$ into an automaton. This translation is inspired by some algorithms by Hopkins [10] and Antimirov [2] for building a non-deterministic automaton using *partial derivatives* of a regular expression. The way we apply the same operations symmetrically on both sides of a pattern is inspired by Conway's *factors* [6, 18]. At the root of all these works is Brzozowski's notion of regular expression *derivatives* [3].

The key idea for the translation is that each state corresponds to a regular expression that exactly matches what is accepted by the state, and a transition corresponds to a syntactic transformation of a regular expression into the regular expression of the next state. In our case, one may have expected pattern locations to take the role of regular expression. As they are not flexible enough, we actually use finite sequences of pattern locations (plus some non-binding variants). Thus, a state σ of the automaton is defined by the following grammar.

s	$::=$	π	single pattern
		$*\pi$	non-binding pattern repetition
		$\diamond$	non-binding wildcard
σ	$::=$	$[s; \ldots; s]$	pattern sequence

We write $[\,]$ for the empty pattern sequence, and $\sigma; \sigma'$ for the concatenation of the pattern sequences σ and σ'. The intuition behind non-binding variants is the following. Suppose we match a value `a[],a[],a[]` against a pattern `A*`. As we will see, this pattern reduces to something akin to `A,A*` by epsilon transitions. According to the semantics of patterns, the beginning of the value `a[]` is indeed bound to the location of subpattern `A`, but the remaining part `a[],a[]` is not bound to any location. Thus, the subpattern `A*` does not correspond to a pattern location, but rather to a repetition of the location of the subpattern `A`.

The initial state of the automaton is the sequence $[\pi_0]$ containing only the root π_0 of the pattern. The epsilon transitions, the labeled transitions and the immediate acceptance relation are respectively

DEC-EPS
$$\frac{\phi(\pi) = \epsilon}{[\pi] \overset{\delta}{\rightsquigarrow} [\,]}$$

DEC-CONCAT
$$\frac{\phi(\pi) = \pi', \pi''}{[\pi] \overset{\delta}{\rightsquigarrow} [\pi'; \pi'']}$$

DEC-UNION
$$\frac{\phi(\pi) = \pi_1 \cup \ldots \cup \pi_n}{[\pi] \overset{\delta}{\rightsquigarrow} [\pi_i]}$$

DEC-REP
$$\frac{\phi(\pi) = \pi'*}{[\pi] \overset{\delta}{\rightsquigarrow} [*\pi']}$$

DEC-WILDCARD
$$\frac{\phi(\pi) = \square}{[\pi] \overset{\delta}{\rightsquigarrow} [\diamond]}$$

DEC-REP-LEFT
$$[*\pi] \overset{\mathsf{l}}{\rightsquigarrow} [\pi; *\pi]$$

DEC-REP-RIGHT
$$[*\pi] \overset{\mathsf{r}}{\rightsquigarrow} [*\pi; \pi]$$

DEC-REP-EPS
$$[*\pi] \overset{\delta}{\rightsquigarrow} [\,]$$

DEC-WILDCARD-EPS
$$[\diamond] \overset{\delta}{\rightsquigarrow} [\,]$$

DEC-LEFT
$$\frac{\sigma \overset{\mathsf{l}}{\rightsquigarrow} \sigma'}{\sigma; \sigma'' \overset{\mathsf{l}}{\rightsquigarrow} \sigma'; \sigma''}$$

DEC-RIGHT
$$\frac{\sigma \overset{\mathsf{r}}{\rightsquigarrow} \sigma'}{\sigma''; \sigma \overset{\mathsf{r}}{\rightsquigarrow} \sigma''; \sigma'}$$

Figure 8. Epsilon Transitions $\sigma \rightsquigarrow^{\delta} \sigma$

$$\frac{\phi(\pi) = L[\pi']}{[\pi] \xrightarrow{\delta} L,[\pi'],[\,]} \qquad [\diamond] \xrightarrow{\delta} \top,[\diamond],[\diamond]$$

$$\frac{\sigma \xrightarrow{\mathsf{l}} L,\sigma_1,\sigma_2}{\sigma; \sigma' \xrightarrow{\mathsf{l}} L,\sigma_1,(\sigma_2; \sigma')} \qquad \frac{\sigma \xrightarrow{\mathsf{r}} L,\sigma_1,\sigma_2}{\sigma'; \sigma \xrightarrow{\mathsf{r}} L,\sigma_1,(\sigma'; \sigma_2)}$$

Figure 9. Labeled Transitions $\sigma \longrightarrow^{\delta} L,\sigma,\sigma$

$$[\,] \downarrow \epsilon \qquad [\diamond] \downarrow e \qquad \frac{v \lhd X \qquad \phi(\pi) = X}{[\pi] \downarrow v}$$

Figure 10. Immediate Acceptance Relation $\sigma \downarrow v$

defined in figure 8, 9, and 10. The assertion $\sigma \rightsquigarrow \sigma$ holds when either assertion $\sigma \rightsquigarrow^{\mathsf{l}} \sigma$ or $\sigma \rightsquigarrow^{\mathsf{r}} \sigma$ holds. Note that the definition of the immediate acceptance relation depends on the typing environment.

As there is an infinite number of sequences σ, we define the finite set of states Σ of the automaton as the set of sequences reachable from the initial state $[\pi_0]$ of the automaton through the transitions. The following lemma states that we define this way a finite automaton.

LEMMA 1 (Finite Automaton). *The number of states σ reachable from the initial state $[\pi_0]$ through the transition relations is finite.*

The number of states can however be exponential in the size of the pattern due to sharing. A typical example is the type definitions below.

```
type T = a[],a[] and U = T,T and V = U,U
```

We expect the state of the automata to be reasonable in practice. Indeed, for patterns without sharing of locations, the bound is much better: it is quadratic in the size of the pattern.

An example of translation is given in figure 11. The pattern `a[],b[]` is represented using the same notation as in figure 1. For the sake of simplicity, we do not represent the part of the automa-

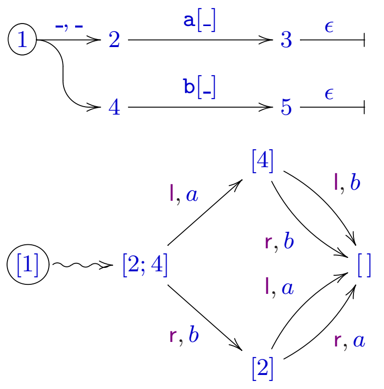

Figure 11. Pattern and its Translation (Simplified)

$$\frac{\phi(\pi) \text{ is a test}}{e \mathrel{↯} [\pi]} \qquad \frac{\sigma \rightsquigarrow \sigma' \qquad v \mathrel{↯} \sigma'}{v \mathrel{↯} \sigma}$$

$$\frac{\sigma \xrightarrow{\delta} L,\sigma_1,\sigma_2 \qquad v \xrightarrow{\delta} l,v_1,v_2 \qquad l \triangleleft L \qquad v_1 \mathrel{↯} \sigma_1}{v \mathrel{↯} \sigma}$$

$$\frac{\sigma \xrightarrow{\delta} L,\sigma_1,\sigma_2 \qquad v \xrightarrow{\delta} l,v_1,v_2 \qquad l \triangleleft L \qquad v_2 \mathrel{↯} \sigma_2}{v \mathrel{↯} \sigma}$$

Figure 12. Disallowed Matching $v \mathrel{↯} \sigma$

MATCH-SEQ-EPS
$$\epsilon \triangleleft [\,]$$

MATCH-SEQ-SINGLE
$$\frac{v \triangleleft \pi}{v \triangleleft [\pi]}$$

MATCH-SEQ-STAR
$$\frac{f \triangleleft_* \pi}{f \triangleleft [*\pi]}$$

MATCH-SEQ-WILCARD
$$v \triangleleft [\diamond]$$

MATCH-SEQ-CONCAT
$$\frac{f \triangleleft \sigma \qquad f' \triangleleft \sigma'}{f, f' \triangleleft \sigma; \sigma'}$$

Figure 13. Matching a Value against a Pattern Sequence $v \triangleleft \sigma$

ton corresponding to the element contents. The initial state of the automaton is the sequence $[1]$. An epsilon transition yields from this state to the state $[2; 4]$. Then, values can either be consumed from the left or from the right. The first case correspond to a transition $[2; 4] \xrightarrow{\mathsf{l}} a,[3],[4]$, depicted as an arrow from state $[2; 4]$ to state $[4]$ with label l, a.

Some matchings of a value against a pattern should not be allowed, either because they are not implementable, or because they would break abstraction. As the automata describe the operational semantics of patterns, they are the right tool to specify which matchings should be rejected. This *disallowed matching* relation $v \mathrel{↯} \sigma$ is defined in figure 12. Automaton matching can be viewed as a dynamic process: for matching a value v against a pattern sequence σ, we start from the assertion $v \in \sigma$ and try to consume the whole value by applying repeatedly the rules in figure 6. We should never arrive in a position where a test needs to be performed on an external value. Therefore, in the definition of the disallowed matching relation, there is one rule corresponding to epsilon transitions and two rules corresponding to labeled transitions, depending on whether the failure occurs in the element contents or in the sequence but outside this element. The last rule corresponds to an immediate failure, where a test is performed on an external value. The following pattern components are *tests*: $L[\pi]$, ϵ, (π, π), and $\pi*$. Basically, a test is a pattern component that only accepts sequences. For this last rule, we only need to consider the case when the pattern sequence contains a single pattern location. Indeed, one can easily show that the only way to arrive to a sequence which is not of this form is through epsilon transitions, starting from a sequence of this form. This specification of disallowed matchings is quite algorithmic. Still, we are confident it can be understood intuitively by a programmer.

We now relate the semantics of a pattern to the semantics of its translation into an automaton. It is convenient to first extend the semantics of pattern locations to pattern sequences (figure 13). We then have the following result.

LEMMA 2. *A value is matched by a pattern if and only if it is matched by the corresponding automaton, as long as the matching is allowed: if $v \mathrel{↯} [\pi_0]$ does not hold, then $v \in [\pi_0]$ if and only if $v \triangleleft \pi_0$. More generally, for any value v and any state σ such that $v \mathrel{↯} \sigma$ does not hold, we have $v \in \sigma$ if and only if $v \triangleleft \sigma$.*

The restriction to allowed matchings is important. Indeed, consider the pattern `(),_`. It matches only sequences but it is translated into an automaton that matches everything, as the empty sequence is eliminated by epsilon transitions (rule DEC-CONCAT followed by rule DEC-EPS together with rule DEC-LEFT).

Our automata are actually designed for analyzing patterns rather than for being executed. They make it possible to focus on a particular part of a pattern by consuming subpatterns from both sides. For instance, if we have a pattern `A,B,C`, we can focus on `B` by consuming `A` on the left and `C` on the right. Thus, type inference can be performed by consuming a type and a pattern in a synchronized way in order to find out which parts of the type corresponds to which parts of the pattern. For instance, if we have a type `a[],T,b[]` and a pattern `a[],(x : _),b[]`, we can compute that the type of the variable `x` is `T` by simultaneously consuming the elements `a[]` and `b[]` of the type and the pattern. For this to work, it must be possible to associate a state to each part of a value matched by a pattern. As a consequence, there is a slight mismatch between our definition and what should be an actual implementation of patterns. First, the rule for type variables in the definition of the acceptance relation is important for analyzing patterns but would not be used in an actual implementation, where matching against a type variable should always succeed. Second, when in state $[\diamond]$, only foreign values are immediately accepted while sequence values are progressively decomposed. Thus is crucial for type inference but cannot be implemented: foreign values cannot be tested and thus an implementation cannot adopt a different behavior depending on whether a value is a sequence or a foreign value. A simple change is sufficient to adapt the automaton: make the state $[\diamond]$ accept any value and remove any transition from this state. Note that this change does not affect the disallowed matching relation.

4.8 Pattern Matching Construction

We can now complete our specification of pattern matching. We are only interested in how a value is matched in a pattern matching construction: which branch is selected and which values are associated to the binders in this branch. We do not consider what happens afterwards. Thus, we can ignore the body of each branch of the construction and can formalize a pattern matching construction as a list of patterns. It turns out to be convenient to share between all patterns a set of pattern locations and a mapping from pattern location to pattern components. Therefore, a pattern construction is characterized by:

- a set of pattern locations Π;
- a mapping $\phi : \Pi \to \mathcal{C}(\Pi)$;
- a family (π_i) of root pattern locations ($\pi_i \in \Pi$);
- a family $(\mathcal{B}_i)$ of binder relations ($\mathcal{B}_i \subseteq \mathcal{X} \times \Pi$)

The i-th pattern is defined as $P_i = (\Pi, \phi, \pi_i, \mathcal{B}_i)$. For instance, the pattern construction in the body of the function `map` presented in section 4.2 can be specified by reusing the corresponding definitions of the set Π and mapping ϕ and defining (π_i) and $(\mathcal{B}_i)$ by

$$\begin{array}{ll} \pi_1 = 1 & \mathcal{B}_1 = \emptyset \\ \pi_2 = 2 & \mathcal{B}_2 = \{(\mathtt{x}, 4), (\mathtt{rem}, 5)\} \end{array}$$

In order to type-check a pattern construction, the type T of the values that may be matched by the pattern must be known. In our example, this type is `List{X}`, which can be represented as a pattern $(R, \psi, 1, \emptyset)$ with

$$R = \{1, 2, 3\}$$

and

$$\psi(1) = 2* \qquad \psi(2) = \mathtt{element}[3] \qquad \psi(3) = \mathtt{X}.$$

The semantics of pattern matching is as follows. Given a value v_0 belonging to the input type T, a pattern P_i is chosen such that the value v_0 matches the root location π_i of the pattern, that is, so that there exists a derivation of $v_0 \triangleleft \pi_i$. We then consider all submatches, that is, all assertions $v \triangleleft \pi$ which occur in this derivation. This defines a relation $\mathcal{M}$ between locations and values. The composition $\mathcal{M} \circ \mathcal{B} = \{(x, v) \mid \exists \pi.(x, \pi) \in \mathcal{B} \wedge (\pi, v) \in \mathcal{M}\}$ of this relation with the binder relation $\mathcal{B}$ is then expected to be a total function from the set of binders of the pattern to parts of value v. This function indicates which part of the value v is bound to each binder x.

In order to ensure that this matching process succeeds for any value of the input type T, the following checks must be performed:

- *exhaustiveness*: for all typing environments and for all values v in the input type T, there must exists a pattern P_i such that the value v matches the root location π_i of this pattern;
- *linearity*: for all typing environments, for all values v in the input type T and for all derivations $v \triangleleft \pi_i$ where π_i is the root location of one of the patterns P_i, the composition $\mathcal{M} \circ \mathcal{X}$ defined above must be a function.

These two checks are standard [13]. In our case, two additional checks must be performed. Indeed, some matchings are not allowed in order to preserve abstraction and for the patterns to be implementable. Furthermore, patterns are not implemented directly but only after erasure. We define the *erasure* of a pattern $(\Pi, \phi, \pi_0, \mathcal{B})$ as the pattern $(\Pi, \phi', \pi_0, \mathcal{B})$ where:

$$\phi'(\pi) = \begin{cases} \square & \text{if } \phi(\pi) \text{ is a type variable } X \\ \phi(\pi) & \text{otherwise.} \end{cases}$$

An *erased* pattern is a pattern containing no pattern variable (that is, $\phi(\pi)$ is different from any variable X for all locations π in the pattern). The semantics remain the one given above, but applied to the erased patterns. We thus have these two additional checks:

- *allowed patterns*: the erasure of each pattern P_i should be *allowed with respect to* the input type T, that is, we must not have $v \not\triangleleft \pi_i$ for any value v in the input type T, any erasure of pattern P_i, and any typing environment.
- *preservation of the semantics*: for all typing environments and for all values v in T, the value v is matched the same way by each pattern P_i and its erasure;

By "matched the same way", we mean that, if there is a derivation of $v \triangleleft \pi_i$ in one of the patterns P_i, then there must be an identical derivation in the erasure of pattern P_i (except for applications of rule MATCH-ABSTRACT which should be replaced by applications of rule MATCH-WILD), and conversely. Algorithms for performing all these checks are presented section 5. The linearity check algorithm is actually omitted as it is standard and its presentation is long.

4.9 Type Inference

An additional operation we are interested in is type inference: we want to compute for each binder a type which approximates the set of values that may be bound to it. From the semantics of the pattern matching construction above, we can derive the following characterization of this set of values. Consider an input type $(\Pi_1, \phi_1, \pi_1^0, \emptyset)$ and a pattern $(\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2)$. Then, a value v may be bound to a binder x if there exists a value v_0 and a location $\pi \in \Pi_2$ such that:

- $v_0 \triangleleft \pi_1^0$ (the value v_0 belongs to the input type);
- $v_0 \triangleleft \pi_2^0$ (the value is matched by the pattern);
- $(x, \pi) \in \mathcal{B}_2$ (the binder x is at location π in the pattern);
- there exists a derivation of $v_0 \triangleleft \pi_2^0$ containing an occurrence of the assertion $v \triangleleft \pi$ (the assertion $v \triangleleft \pi$ is a submatch).

Several algorithms for *precise* type inference have been proposed [7, 13, 20]. These algorithms are tuned to a particular matching policy (such as the first-match policy). With these algorithms, the semantics of the type computed for a binder is exactly the set of values that may be bound to it. (As binders are considered independently, any correlation between them is lost, though.) For instance, let us consider the following function.

```
fun f (x : (a[] | b[] | c[])) =
   match x with
     b[]                      → ...
   | y : (a[] | b[] | d[])  → ...
   | _                        → ...
```

A precise type algorithm infers the type `a[]` for the binder `y`. Indeed, values of type `b[]` are matched by the first line of the pattern. Therefore, only values of type the difference between type `a[]|b[]|c[]` and type `b[]`, that is, type `a[]|c[]` may be matched by the second pattern. Finally, the values matching the second pattern must also have type `a[]|b[]|d[]`, hence their type is the intersection of `a[]|c[]` and `a[]|b[]|d[]`, that is, `a[]`. Such a type algorithm is implemented in CDuce and was initially implemented in XDuce.

Difference is costly to implement. Besides, though this is not apparent in the example above, difference operations may need to be performed at many places in the pattern, especially when binders are deeply nested. Hosoya proposed a simpler design [11], remarking that with a non-deterministic semantics (in other words, when the matching policy is left unspecified) no difference operation needs to be performed. An intersection operation still needs to be performed, but only once per occurrence of a binder. So, in our example, the second line of the pattern still matches values of type `b[]`. Therefore, the type of `y` is the intersection of the initial type `a[]|b[]|c[]` and the type `a[]|b[]`, that is, `a[]|b[]`.

In our case, even the intersection operations must be avoided. Indeed, our types are not closed under intersection: for instance, there is no type that corresponds to the intersection of two type variables. Xtatic has the same issue [8, section 5.3]. The current implementation of Xtatic thus computes an approximation of the intersection. Another reason to avoid intersection is that it is not a syntactic operation on types in XDuce. Thus, in order to compute an intersection, types must first be translated to automata and the intersection must be translated back from an automaton to a type. In the process, the type may become more complex. In the worst case, the size of the intersection of two automata is quadratic in the size of these automata. Also, some type abbreviations may be lost during the successive translations.

What we propose is to infer types not for binders but for wildcards `_` and compute the type of binders by substitutions. The key idea is that the intersection of a type with a wildcard is the type itself. Thus, no intersection is actually needed. Consider for instance the function below.

```
  fun g (x : (a[],b[])) =
     match x with
       y : (_,(b[]|c[])) → ...
```

The type inferred for the wildcard is `a[]`. Thus, by substitution, the type inferred for the binder `y` is `a[],(b[]|c[])`. We deliberately gave an example for which the inferred type is not precise, in order to emphasize the difference with other specifications of type inference. We expect this weaker form of type inference to perform well in practice. In particular, type inference is still precise for wildcards (assuming a non-deterministic semantics). When needed, the programmer can provide explicitly a more precise type. We experimented with the examples provided with the XDuce distribution. Only some small changes were necessary to get them to compile. What we actually had to do was to replace by wildcards some explicit types which were not precise enough.

More formally, we define the *semantics of a location* π of the pattern as the set of values v such that there exists a value v_0 such that the assertions $v_0 \lhd \pi_1^0$ and $v_0 \lhd \pi_2^0$ holds and the assertion $v \lhd \pi$ is a submatch of a derivation of $v_0 \lhd \pi_2^0$. The type inference algorithm then consists in computing for each location corresponding to a wildcard a type whose semantics is the semantics of this location and substituting this type in place of the wildcard. The substitution may not preserve pattern well-formedness. In this case, the type checking fails. But we believe this is unlikely to occur in practice, as this can only happen when a wildcard location is shared in two different contexts. For instance, consider the type `a[X]` and the pattern `a[Q],Q` where `Q` $= \_$. The type inferred for the wildcard is `X|()` and substituting this type does not preserve well-formedness. If the resulting pattern is well-formed, then it is a type: it does not contain any wildcard. The type of a binder is the type corresponding to the union of the locations the binder is associated to.

5. Algorithms on Patterns

We define a number of algorithms for type checking and type inference for patterns. Each of these algorithms is specified in an abstract way, by defining a relation over a finite domain using inductive definitions. Actually implementing them is a constraint solving issue. Standard techniques can be used, such as search pruning (when an assertion is either obviously true or obviously false), memoization (so as not to perform the same computation several times), and lazy computation (in order not to compute unused parts of the relation).

The size of the finite domain provides a bound on the complexity of the algorithm. We don't study the precise complexity of these algorithms, as we believe this would not be really meaningful. In particular, the complexity of all these algorithms is polynomial in the sizes of the automata associated to the patterns it operates on, but these sizes can be exponential in the size of the patterns. Our experience on the subject leads us to believe that the algorithms should perform well in practice.

ROOT
$$[\pi_1^0] \smile [\pi_2^0]$$

DEC-LEFT
$$\frac{\sigma_1 \smile \sigma_2 \qquad \sigma_1 \leadsto \sigma_1'}{\sigma_1' \smile \sigma_2}$$

DEC-RIGHT
$$\frac{\sigma_1 \smile \sigma_2 \qquad \sigma_2 \leadsto \sigma_2'}{\sigma_1 \smile \sigma_2'}$$

ENTER
$$\frac{\begin{array}{c}\sigma_1 \smile \sigma_2 \\ L_1 \bowtie L_2 \qquad \sigma_1 \xrightarrow{\delta} L_1,\sigma_1',\sigma_1'' \\ \sigma_1'' \leftrightarrow \sigma_2'' \qquad \sigma_2 \xrightarrow{\delta} L_2,\sigma_2',\sigma_2''\end{array}}{\sigma_1' \smile \sigma_2'}$$

SHIFT
$$\frac{\begin{array}{c}\sigma_1 \smile \sigma_2 \\ L_1 \bowtie L_2 \qquad \sigma_1 \xrightarrow{\delta} L_1,\sigma_1',\sigma_1'' \\ \sigma_1' \leftrightarrow \sigma_2' \qquad \sigma_2 \xrightarrow{\delta} L_2,\sigma_2',\sigma_2''\end{array}}{\sigma_1'' \smile \sigma_2''}$$

Figure 14. Type Propagation $\sigma \smile \sigma$

5.1 Exhaustiveness

The input of the algorithm is the input type $T = (R, \psi, \pi, \emptyset)$ and the different patterns $P_i = (\Pi, \phi, \pi_i, \mathcal{B}_i)$ of the pattern construction. We define the union of the patterns P_i by $P = (\Pi \cup \{\star\}, \phi', \star, \emptyset)$ where the location $\star$ is assumed not to be in Π and the mapping ϕ' is such that $\phi'(\star) = \pi_1 \cup \ldots \cup \pi_n$ (the union of all root locations) and $\phi'(\pi) = \phi(\pi)$ for $\pi \in \Pi$ One can easily prove that the semantics of the pattern P is the union of the semantics of the patterns P_i. Then, the pattern is exhaustive if and only if $T <: P$.

Note that the union construction above can be applied to any finite set of patterns sharing a common mapping ϕ. This construction is also used for type inference (section 5.6).

5.2 Type Propagation

This algorithm propagates type information in a pattern. It is used both for checking whether a pattern is allowed (section 5.4) and for type inference (section 5.6). The input of the algorithm is composed of two patterns $P_1 = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$ and $P_2 = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2)$ and a relation $\sigma_1 \leftrightarrow \sigma_2$ (where the sequences σ_1 and σ_2 range over the states of the automata associated respectively to P_1 and P_2). The relation controls when the type information is propagated across an element. The algorithm is defined in figure 14 as a relation $\sigma_1 \smile \sigma_2$. The roots of the two patterns are related (rule ROOT). The relation is preserved by epsilon transition (rules DEC-LEFT and DEC-RIGHT). The rules ENTER and SHIFT specify how the relation is propagated to the contents of an element and aside an element.

Though the rules are symmetric, the algorithm is used in an asymmetric way. One of the pattern is actually always a type and the algorithm can be read as propagating type information derived from this type into the other pattern. Besides, we are not interested in computing the whole relation $\sigma_1 \smile \sigma_2$. Rather, for some given sequences σ_1, the set of pattern sequences σ_2 such that $\sigma_1 \smile \sigma_2$ must be computed.

As the algorithm is defined as a binary relation $\sigma_1 \smile \sigma_2$ over the states of the automata associated to the patterns P_1 and P_2, it is quadratic in the size of these automata.

Pattern

$$\frac{\phi(\pi) = L[\pi'] \qquad \lhd L \qquad \lhd \pi'}{\lhd \pi} \qquad\qquad \frac{\phi(\pi) = \epsilon}{\lhd \pi}$$

$$\frac{\phi(\pi) = \pi_1, \pi_2 \qquad \lhd \pi_1 \qquad \lhd \pi_2}{\lhd \pi}$$

$$\frac{\phi(\pi) = \pi_1 \cup \ldots \cup \pi_n \qquad \lhd \pi_i}{\lhd \pi} \qquad\qquad \frac{\phi(\pi) = \pi' *}{\lhd \pi}$$

$$\frac{\phi(\pi) = \square}{\lhd \pi} \qquad\qquad \frac{\phi(\pi) = X}{\lhd \pi}$$

Pattern sequence

$$\lhd [\,] \qquad \lhd [*\pi] \qquad \lhd [\Diamond] \qquad \frac{\lhd \pi}{\lhd [\pi]} \qquad \frac{\lhd \sigma_1 \qquad \lhd \sigma_2}{\lhd \sigma_1 ; \sigma_2}$$

Figure 15. Non-Emptiness $\lhd \pi$ and $\lhd \sigma$

5.3 Type Non-Emptiness

In order to check whether a pattern is allowed (section 5.4), it turns out that we need an algorithm to decide whether, given a pattern $P = (\Pi, \phi, \pi_0, \mathcal{B})$, the semantics of a pattern location π or a pattern sequence σ (belonging to the set of states of the automaton associated to pattern P) is empty, that is, whether there exists a value v such that $v \lhd \pi$ or $v \lhd \sigma$. These algorithms are defined in figure 15 as two relations $\lhd \pi$ and $\lhd \sigma$. Their properties can be stated as follows.

LEMMA 3. *Let π be a location in pattern P and σ be a state of the automaton associated to pattern P. If there exists a value v such that $v \lhd \pi$, then $\lhd \pi$. Likewise, if there exists a value v such that $v \lhd \sigma$, then $\lhd \sigma$. The converse holds in any typing environment such that for all type variables X there exists at least a value v such that $v \lhd X$.*

The proof of the lemma is straightforward. The reason for the restriction in the converse case can be seen on the last rule in figure 15: if $\phi(\pi) = X$, then we have $\lhd \pi$. We thus need to ensure that there exists a value v such that $v \lhd X$.

The inference rules define a relation $\lhd \pi$ over the finite set of pattern locations Π in pattern P. Each rule can be implemented in constant time. Hence, computing the relation for all locations in a pattern can be done in linear time in the size of the pattern P. Likewise, the relation $\lhd \sigma$ can be computed in linear time in the size of the automaton associated to the pattern P.

5.4 Disallowed Pattern

The algorithm checking whether a pattern $P = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2)$ is allowed with respect to an input type $T = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$ is based on an instance of the type propagation algorithm (section 5.2) applied to the type T and the pattern P. For this instance, we take $\sigma_1 \leftrightarrow \sigma_2$ iff $\lhd \sigma_1$. Intuitively, if we have a type `L[T1],T2` and a pattern `L'[P1],P2` such that the sets `L` and `L'` are not disjoint, the type information `T1` should be propagated in the pattern `P1`, but only if there is indeed a value of type `L[T1],T2`, thus in particular only if the semantics of type `T2` is not empty. On the other hand, as the implementation of the automaton may try to match a value against the subpattern `P1` before considering the subpattern `P2`, nothing should be assumed about `P2`. The algorithm relies on the following theorem.

$$\frac{\sigma_1' \bowtie \sigma_2 \qquad \sigma_1 \overset{\mathsf{I}}{\rightsquigarrow} \sigma_1'}{\sigma_1 \bowtie \sigma_2} \qquad\qquad \frac{\sigma_1 \bowtie \sigma_2' \qquad \sigma_2 \overset{\mathsf{I}}{\rightsquigarrow} \sigma_2'}{\sigma_1 \bowtie \sigma_2}$$

$$\frac{\begin{array}{cc} \sigma_1' \bowtie \sigma_2' & \sigma_1 \xrightarrow{\mathsf{I}} L_1, \sigma_1', \sigma_1'' \\ \sigma_1'' \bowtie \sigma_2'' & \sigma_2 \xrightarrow{\mathsf{I}} L_2, \sigma_2', \sigma_2'' \\ \multicolumn{2}{c}{L_1 \bowtie L_2} \end{array}}{\sigma_1 \bowtie \sigma_2}$$

$$[\,] \bowtie [\,] \qquad\qquad \frac{\phi(\pi) = X}{[\pi] \bowtie [\Diamond]}$$

Figure 16. Non-Disjointness $\sigma \bowtie \sigma$

THEOREM 4. *If the pattern P is disallowed with respect to the type T, then there exists two locations $\pi_1 \in \Pi_1$ and $\pi_2 \in \Pi_2$ such that $\phi(\pi_1)$ is a type variable X, the location π_2 is a test (as defined in section 4.7), and $[\pi_1] \smile [\pi_2]$. The converse holds in any typing environment such that for all type variables X there exists a foreign value e such that $e \lhd X$.*

The restriction in the converse case ensures that the relation $\lhd \sigma$ really coincide with the non-emptiness of the sequence σ. It also ensures that if $\phi(\pi_1) = X$ and π_2 is a test, then there exists a foreign value e such that $e \lightning \pi_2$. From this and $[\pi_1] \smile [\pi_2]$, one can then show that the whole pattern is disallowed.

A naïve implementation of the algorithm would compute all pairs of locations π_1 and π_2 such that $[\pi_1] \smile [\pi_2]$ and then checks whether there exists a pair for which both $\phi(\pi_1)$ is a type variable X and the location π_2 is a test. This implementation would have the same complexity as the type propagation algorithm. An immediate optimization is to stop propagating type information whenever a type location π_1 with no type variable below it is reached.

5.5 Non-Disjointness

The type inference algorithm relies on an algorithm that, given a type

$$T = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$$

and an erased pattern

$$P = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2),$$

decides the non-disjointness of the semantics of two pattern sequences σ_1 and σ_2 belonging to the states of the automata associated respectively to the type T and the pattern P. The pattern P is assumed to be allowed with respect to type T. The algorithm is defined in figure 16 as a relation $\sigma_1 \bowtie \sigma_2$. It is based on a standard algorithm for checking non-disjointness of tree automata, with a special case for type variables X. Note that only transitions with tag I are used. The relation would remain unchanged if this restriction was removed.

The intended semantics of the algorithm is that $\sigma_1 \bowtie \sigma_2$ if and only if there exists a value v such that $v \lhd \sigma_1$ and $v \lhd \sigma_2$. However this does not hold for arbitrary sequences σ_1 and σ_2. The completeness of the algorithm can be stated as follows.

LEMMA 5 (Completeness). *Let σ_1 and σ_2 be two states of the automata associated respectively to the type T and the pattern P. We assume that:*

- *the pattern P is allowed with respect to type T;*
- *$\sigma_1 \smile \sigma_2$ (where this relation is the instance defined in section 5.4 for checking for disallowed patterns);*

- *the typing environment is such that for all type variables X there exists at least a value v such that $v \lhd X$.*

Then, if $\sigma_1 \bowtie \sigma_2$, there exists a value v such that $v \lhd \sigma_1$ and $v \lhd \sigma_2$.

The condition $\sigma_1 \smile \sigma_2$ together with the allowed pattern condition ensures that, for instance, one never considers the type sequence $[\pi]$ with $\phi(\pi) = X$ against the pattern sequence $[\Diamond; \Diamond]$ for which the algorithm may give a wrong answer: one has $[\pi] \bowtie [\Diamond; \Diamond]$ as the sequence $[\Diamond; \Diamond]$ reduces by epsilon transition to the sequence $[\Diamond]$, but there is not reason for the sequences $[\pi]$ and $[\Diamond; \Diamond]$ to share a common value. The constraint on typing environments can be easily understood by looking at the rule concerning type variables.

The soundness property is hard to state. Rather than defining precisely when it holds, which would involve defining an additional complex relation, we use the following somewhat imprecise statement.

LEMMA 6 (Soundness). *Let σ_1 and σ_2 be two sequences of the automaton respectively associated to the input type and the input pattern. In any position where the relation $\sigma_1 \bowtie \sigma_2$ is used in the type inference algorithm below (section 5.6), if there exists a value v such that $v \lhd \sigma_1$ and $v \lhd \sigma_2$, then $\sigma_1 \bowtie \sigma_2$.*

The algorithm is quadratic in the size of the automata associated to the type T and the pattern P.

5.6 Type Inference

The input of the type inference algorithm is a type

$$T = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$$

and an erased pattern

$$P = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2).$$

The pattern P is assumed to be allowed with respect to the type T. The algorithm is based on an instance of the type propagation algorithm (section 5.2) applied to the type T and the pattern P. For this instance, we take $\sigma_1 \leftrightarrow \sigma_2$ iff $\sigma_1 \bowtie \sigma_2$. Intuitively, if we have a type `L[T1],T2` and a pattern `L'[P1],P2`, the type information `T1` should be propagated in the pattern `P1`, but only if there is indeed a value of the whole type matched by the whole pattern. In particular, there should be a value shared by the type `T2` and the pattern `P2`. We rely on the following result.

THEOREM 7. *If a value v is included in the semantics of a location π_2 of the pattern, then there exists a sequence σ_1 such that $\sigma_1 \smile [\pi_2]$. The converse holds in any typing environment such that for all type variables X there exists at least a value v such that $v \lhd X$.*

The type inferred for a subpattern π_2 should thus corresponds to the union of the sequences σ_1 such that $\sigma_1 \smile [\pi_2]$. One can show that for any such sequence σ_1, as it belongs to the set of states of an automaton associated to a type, one can build a type T_1 with the same semantics. Then, the type inferred for the subpattern π_2 can be build by taking the union of these types, as defined in section 5.1.

The algorithm is complete only when all type variables have a non-empty semantics. It may be possible to get a stronger result by extending our type system with *conditional types* [1]. But we believe this would unnecessarily complicate the type system.

An interesting feature of this algorithm is that it works directly on the syntax of patterns and types. In particular, the inferred type is build from the input type using only simple operations (concatenation and union).

As in the case of the "disallowed pattern" check (section 5.4), a naïve implementation which would compute the whole relation $\sigma_1 \smile \sigma_2$ can be improved by stopping the propagation of type information whenever one reach a pattern sequence σ_2 with no location whose type needs to be inferred below it.

5.7 Preservation of the Semantics

The input of the type inference algorithm is a type

$$T = (\Pi_1, \phi_1, \pi_1^0, \mathcal{B}_1)$$

and a pattern

$$P = (\Pi_2, \phi_2, \pi_2^0, \mathcal{B}_2).$$

The algorithm is as follows. For each location π in the pattern P corresponding to a type variable X (that is, $\phi_1(\pi) = X$), a type T' is computed using the type inference algorithm on the erasure of pattern P. We then compare this type with the part of the pattern P corresponding to location π, that is, the pattern $P' = (\Pi_2, \phi_2, \pi, \mathcal{B}_2)$. The semantics is preserved if for all such locations π we have $T' <: P'$. The soundness of the algorithm relies on the following theorem.

THEOREM 8. *The semantics of the pattern is preserved by erasure if and only if for any pattern location π_2 such that $\phi(\pi_2) = X$ (for any type variable X) and any type location σ_1 such that $\sigma_1 \smile \pi_2$ (by type inference on the erasure of pattern P), we have $\sigma_1 <: [\pi_2]$.*

6. Related Works

As mentioned in the introduction, we presented in a previous paper [21] a calculus dealing with polymorphism for regular tree types. Values are binary trees rather than sequences of elements. It is straightforward to translate sequences into binary trees by representing an element contents as a node whose first child is its contents and second child is its right sibling. A similar translation can be defined to some extent for patterns. But there is a number of restrictions. In particular, wildcards and binders should only occur in tail position. The present paper deals directly with sequences, which makes it possible to avoid these restrictions. Additionally, we specify patterns in a more precise way: we believe very few extensions to patterns, besides support for XML attributes, would be necessary for a realistic implementation.

Hosoya, Frisch and Castagna have also proposed an extension of XDuce with polymorphism [12], now implemented in the latest release. In their work, type variables range over sets of basic XDuce values rather than over sets of arbitrary values. This results in design decisions which are drastically different from ours. For instance, they consider that pattern matching on values whose type is a type variable is possible (as the structure of all such values can be explored by pattern matching), while we consider that this would break abstraction. They can deal with bounded quantification. On the other hand, it is not clear how to extend their work to deal with foreign types and higher-order functions.

Sulzmann and Lu propose to use a structured representation of XDuce values [19] and interpret subtyping as a runtime coercion. As types reflect the structure of values, they do not have the issue of concatenating foreign values: the values of type `A,B` are pairs, rather than concatenations of values of type `A` and type `B`. However, they may need to use algorithms similar to ours in order to ensure that pattern matching interact well with polymorphism.

Several type inference algorithms have been proposed for regular expression types. The first one [13], by Hosoya and Pierce, is precise (assuming a first-match policy) but can infer a type only for binders in tail position in the pattern. Hosoya later proposed a simpler design [11], corresponding to a non-deterministic semantics for patterns, where this restriction was removed. Both algorithms use a translation of types and patterns into tree automata. Several algorithms for precise type inference for different match policies have also been presented by Vansummeren [20]. They work directly on the syntax of patterns but require complex operations on types such as intersection and difference.

CDuce [4] has some extensive support for importing functions from OCaml. Contrary to what we propose in section 3, their extension relies on a runtime translation of ML values into CDuce values according to their types.

References

[1] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 163–173, New York, NY, USA, 1994. ACM Press.

[2] V. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[3] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[4] CDuce Development Team. *CDuce Programming Language User's Manual*. Available from `http://www.cduce.org/documentation.html`.

[5] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 1997. release October, 1rst 2002.

[6] J. H. Conway. *Regular Algebra and Finite Machines*. William Clowes and Sons, 1971.

[7] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping. In *17th IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[8] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, Jan. 2005. University of Pennsylvania Technical Report MS-CIS-04-24, Oct 2004.

[9] V. Gapeyev and B. C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[10] M. W. Hopkins. Converting regular expressions to non-deterministic finite automata, May 1992. Newsgroup message on comp.theory.

[11] H. Hosoya. Regular expression pattern matching — a simpler design. Technical Report 1397, RIMS, Kyoto University, 2003.

[12] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages*, pages 50–62. ACM Press, 2005.

[13] H. Hosoya and B. C. Pierce. Regular expression pattern matching. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), London, England*, 2001. Full version in *Journal of Functional Programming*, 13(6), Nov. 2003, pp. 961–1004.

[14] H. Hosoya and B. C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[15] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2000.

[16] D. MacQueen, G. Plotkin, and R. Sethi. An ideal model for recursive polymorphic types. *Information and Control*, 71(1-2):95–130, 1986.

[17] M. Murata. Hedge automata: a formal model for XML schemata. `http://www.xml.gr.jp/relax/hedge_nice.html`, 2000.

[18] G. Sittampalam, O. de Moor, and K. F. Larsen. Incremental execution of transformation specifications. *SIGPLAN Not.*, 39(1):26–38, 2004.

[19] M. Sulzmann and K. Z. M. Lu. A type-safe embedding of xduce into ml. In *ACM SIGPLAN Workshop on ML*, informal proceedings, Sept. 2005.

[20] S. Vansummeren. Type inference for unique pattern matching. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2003. To appear.

[21] J. Vouillon. Polymorphic regular tree types and patterns. In *Proceedings of the 33th ACM Conference on Principles of Programming Languages*, Charleston, USA, Jan. 2006. To appear. Available from `http://www.pps.jussieu.fr/~vouillon/publi/`.