Università degli Studi di Bologna

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea in Informatica
Materia di Tesi: Linguaggi di programmazione

XRel: Analysis of the XDuce Data Types and Implementation of the Related Algorithms

Tesi di Laurea di:
Fabrizio Bisi

Relatore:
Chiar.mo Prof. Cosimo Laneve

Correlatore:
Dott. Lucian Wischik

Keywords: XML, XDuce, XRel, Tree Regular Expressions, Pattern Matching, Type Inference, Hipi

II Sessione
Anno Accademico 2002-2003

A mio padre Ivan e
A mia nonna Amedea

Contents

Chapter 1

Introduction

1.1 Web Services

Nowadays, many applications are based on network communication. Amongst such new kinds of applications are *Web Services*. A Web Service [Con02] is a service that resides at some remote location, that receives requests over the Internet, and that performs computation before giving an answer. In practical terms, Web Services look to become the foundation of e-commerce.

The first key enabling technology for Web Services is the use of a standard data format, *XML*, along with the use of *standard protocols* like HTTP [FGM+99] and SOAP [Con03]. These standards allow for the interaction of services written in different languages and executed in different environments.

The second key technology is *orchestration*. Orchestration is when a Web Service, as well as performing local computation, can also interact with other Web Services to prepare its answer. For instance, a travel-agent Web Service might interact with an airline Web Service and a hotel Web Service in order to complete its booking. Orchestration means that Web Services must be considered as *interacting processes* which exchange messages in complicated protocols, rather than just as remote functions which receive input and give back output. Orchestration standards have not yet been finalized; however, existing proposals [Cor, LM02, Bus, BIM+] are mostly based on the π–calculus formalism.

The π–*calculus* [MPW92, Mil99] is a widely studied formalism based on message passing for describing and reasoning about distributed and concurrent systems; born in early 90's, it had an immediate success in the academic world for its mobility features that potentially made suitable for many distributed applications, such as phone mobile networks.

Today also business world looks at it with interest. The evidence of this interest is in

the fact that it can be found together with XML-processing in new commercial products. Notable, they are found together in Microsoft Biztalk Server 2000 [Cor], a product used to create Web Services. The orchestration language used in Biztalk is called XLang, and it is based on the "asynchronous" version of the π–calculus [MS98]; all Biztalk messages are XML documents. XML and π–calculus are also found together in Highwire [Mer02, LM02], an ongoing project currently under development at Microsoft. Highwire is based on the "explicit fusion" version of the π–calculus [GW00]; it uses terms in linear logic as its data, uses cut–elimination for exchange of data, and argues that linear logic is a generalization of XML.

The "fusion" group at Bologna also wishes to integrate π–calculus and XML, to make a formalism of this integration, and to implement it. We have chosen to use a different model for XML — namely, the *tree regular expressions* of Hosoya's XDuce [HP03]. These tree regular expressions also form the foundation of XQuery [FMM$^+$03], CDuce [BCF03] and XTatic [GP03]. We believe that this approach will yield an "XML–π–calculus" which is easier for programmers to understand than Highwire, and which is easier to formalize and implement.

The fusion group's project has been carried out in two parallel dissertations. The first, by Paolo Milazzo, gives a Java implementation of the π–calculus but without any XML. My dissertation gives a Java implementation of tree regular expressions but without any π–calculus. An integration of the two implementations is currently in progress.

1.2 XDuce

One of the reasons for popularity of XML is the existence of a number of schema languages, including DTDs [XML98], XML-Schema [XS 00], DSD [KMS] and RELAX [CM01], that can be used to define "types" (or "schemas") describing structural constraints on data and thereby improving the safety of data processing and exchange.

However, as noted in XDuce [HVP00], the use of types in mainstream XML processing technology is often limited to checking only *data*, not *programs*. Typically, an XML processing program first reads an XML document and checks that it conforms to a given type using a *validating parser*. The program then uses either a generic tree manipulation library such as DOM [DOM] or a dedicated XML language such as XSLT [XSL99] or XML-QL [DFF$^+$]. Since these tools don't provide any systematic connection between the program and the types of the documents it manipulates, using them gives no compile-time guarantee that the documents produced by the program will always conform to an intended type.

XDuce proposes *tree regular expression types* as a foundation for statically typed processing of XML documents. Regular expression types capture (and generalize) the regular expression notations (`*`, `?`, `|`, etc.) commonly found in schema languages for XML, as well

as the hierarchical tree structure of XML.

Regular Expression Types

As a simple example of regular expression types, consider the definitions

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

corresponding to the following set of DTD declarations:

```
<!ELEMENT addrbook person*>
<!ELEMENT person (name,email*,tel?)>
<!ELEMENT name #PCDATA>
<!ELEMENT email #PCDATA>
<!ELEMENT tel #PCDATA>
```

This example comes from [HVP00]. Type constructors of the form `label[...]` classify tree nodes with the tag `label` (i.e. XML elements of the form `<label>...</label>`). Thus, the inhabitants of the types `Name`, `Email`, and `Tel` are all XML elements containing arbitrary strings with the appropriate identifying label. Types may also involve the regular expression operators `*` (repetition), `+` (repetition of one or more occurrences) and `?` (optional occurrence), as well as `|` (alternation). Thus, the type `Addrbook` describes a label `addrbook` whose content is zero or more repetitions of subtrees of type `Person`. Likewise, the type `Person` describes a label `person` whose content is a `Name` subtree, zero or more `Email` subtrees, and an optional `Tel` subtree. An instance of the type `Addrbook` is the following XML document:

```
<addrbook>
  <person>
    <name> Haruo Hosoya </name>
    <email> hahosoya@kyoto-u </email>
    <email> hahosoya@upenn </email>
  </person>
  <person>
    <name> Benjamin Pierce </name>
    <email> bcpierce@upenn </email>
    <tel> 123-456-789 </tel>
  </person>
</addrbook>
```

Regular Expression Pattern Matching

The other core feature of XDuce is *regular expression pattern matching*. This extends conventional ML-like pattern-matching with regular expression operators such as repetition (`*`), alternation (`|`), etc., and with the ability to specify (possibly recursive) types directly inside patterns. These features allow to write compact patterns that can match arbitrarily long sequences of subtrees and that can extract data from the middle of a complex sequence. This is often useful in XML processing, as we'll argue below.

To see why, consider again the example of the previous section. We can write a regular expression pattern match which, given a value `p` of type `Person`, checks whether `p` contains a `tel` field and if so extracts the contents of `name` and `tel`:

```
match p with
    person[name[val n], Email*, tel[val t]]
      -> (* do some stuff involving n and t *)
  | person[val c]
      -> (* do other stuff *)
```

Note how, apart the use of the regular expression patterns, the syntax is close to that of ML: the match construct consists of an input variable (above it's `p`) and a series of clauses (two in the example), each of which is a pair of a pattern and a body. If the value of the variable matches with a pattern, its corresponding body is run (or, using the terminology of XDuce, the term that constitutes the body is evaluated).

In the example the first pattern matches a node labelled `person` whose content is a sequence of a `name`, zero or more `emails`, and a `tel`. In this case, we bind the variable `n` to the `name`'s content and `t` to the `tel`'s content and then evaluate the body possibly containing `n` and `t`. The second case matches a label `person` with any content and binds `c` to the content. The second case is invoked only when the first case fails, i.e., when there is no `Tel` component.

Note how the first pattern uses the regular expression type `Email*` to "jump over" an arbitrary-length sequence and extract the `tel` node following it. That is, when the pattern matcher looks at the pattern `Email*`, no hint is available about how many emails there are, therefore the matcher must walk through the input value until it finds the end of the chain of emails. This iterative behavior, which is beyond ML pattern matching, enables matching of arbitrary length sequences. [Hos01], [HP02] argue that this is often quite useful in programming with XML, on the grounds that XML data structures often contain sequences where repetitive, optional, and fixed parts are mixed together; regular expression pattern matching allows direct access to the parts of such sequences.

For example, the pattern above is substantially more compact than explicitly writing a recursive function that traverses the sequence, as we would need to do if only ML-style matching of fixed-length sequences were supported.

The usefulness of matching against regular expression types is yet more evident in the following complex pattern, which extracts the subcomponents of an HTML table:

```
match t with
  table[cap as Caption?,
        col as (Col*|Colgroup*),
        hd as Thead,
        ft as Tfoot?,
        bd as (Tbody+|Tr+)]
    -> ...
```

An HTML table consists of several optional fields (`Caption?` and `Tfoot?`) and repetitive fields (`Col*`, `Colgroup*`, `Tbody+`, and `Tr+`) (we assume the types `Caption`, `Col`, etc., to be defined elsewhere). Again, by matching against regular expression patterns, we can directly pick out each subcomponent, whose position in the input sequence is statically unknown. Imagine equivalent code written only with simpler ML-like pattern matching.

1.3 Reviewing XDuce

The goal of writing tree regular expressions in Java seemed originally an easy one: merely a port of the XDuce code from ML into Java. In practice it turned out more difficult.

The source code of the last version of the tool (0.4.0) [XDu] is terse and sparsely commented, and contains substantial code duplication (e.g. automata are defined in both `patmatch.ml` and `typeop.ml`). Rather than porting XDuce source code, I found it better to write a Java implementation from scratch, based on the publications.

Beside the code, there are six other publications for the algorithms. The algorithms are spread throughout these publications.

1. "Regular Expression Types for XML" [Hos01], the PhD thesis of Hosoya written from the summer 1999 and completed in 3 January 2001, is a clear (but out of date) work

2. "Regular Expression Types for XML" [HVP00], is a paper written and published in 2000 that focuses on semantic checks

3. "Regular Expression Pattern Matching for XML" [HP02], is a paper written in 2001 and published in 2002 that focuses on type inference and pattern matching

4. "Regular Expression Pattern Matching — A Simpler Design —" [Hos03] is a recent work (February 2003) that describes most of the new algorithms (pattern matching, type inference and the ambiguity check)

5. "Validation and Boolean operations for Attribute-Element Constraints" [HM02b] is a paper written in October 2002, apparently focused on new attribute handling, but that also includes the formalization of one of the two subroutines needed for the new subtyping check (the difference between automata)

6. "Boolean Operations and Inclusion Test for Attribute-Element Constraints (Extended Abstract)" [HM02a] is a paper less or more written in the same period of the previous one and that has much overlapping material but also includes both the two subroutines for the new subtyping check (difference between automata and non-emptiness test)

The old articles are more complete (and I found them easier to understand) but alas theirs algorithms are often out of date.

Also, there were some problems with the difference algorithms as defined in [HM02a, HM02b]. Most seriously, it gives the wrong answers as compared with the intended meaning of "difference" and also with respect to the XDuce implementation itself — see Section 4.3.4, page 79. As written in [HM02a, HM02b] there are also some typographical errors with the algorithm. Finally, the explanation in section of [HM02b] of why there is the need to apply the subset construct in the algorithm is questionable[1].

I wish to thank Hosoya, the author of XDuce, for his availability and courtesy; everytime I asked to him clarifications by email, he always answered me readily with extreme friendliness. Also I would like to express my appreciation for the research work he made with XDuce.

1.4 Outline of the Thesis

The plan of this dissertation is as follows.

1. Chapter 2 provides an up-to-date snapshot of the entire language. I partly replicated the work of Hosoya here, especially in sections 2.1 and 2.2 that describe the whole language from scratch. However in section 2.3 I outline some aspects of XDuce which may not be apparent to the casual reader of the XDuce specifications. Also, Section 2.4 explains the changes that have been made in XDuce over the years, and their reasons.

2. Chapter 3 presents my final Java program, called XRel — "XML Regular Expressions Language". XRel is not a re–implementation of all of XDuce but essentially a test harness for tree regular expression algorithms. Despite this, all main algorithms of XDuce (pattern matching, subtyping, type inference and static typechecks) have been

[1] readers familiar with string automata techniques should notice here that, unlike string automata, deterministic top–down tree automata and non–deterministic top–down tree automata are not equally expressive, so we can't compute difference applying determinization

implemented, while for simplicity we preferred to omit some features that were not interesting for our purposes (filters [Hos04], attributes [HM02a, HM02b] and label classes [HP03]).

3. Chapter 4 is the core of the dissertation. It documents the algorithms and my Java implementation.

Appendix A contains the JavaCC / JJTree generated formal grammar for XRel.

Original Contributions

This thesis is mainly an analysis of the XDuce language and a re–implementation of tree regular expression algorithms.

It also introduces a novel and simpler ambiguity-check algorithm, for checking whether a tree regular expression would match some value in more than one way, that is based on the notion of weak ambiguity instead of the strong one used by Hosoya. Ambiguity is defined in Section 3.3.3 (page 50); my algorithm is presented in section 4.3.3 (page 50).

Another contribution is to draw attention to an error in the formalization of the difference algorithm given in [HM02a, HM02b] and to present a corrected algorithm.

A further contribution of this thesis is to provide an up-to-date snapshot of the language and especially of the algorithms, because — as indicated above — the work–in–progress condition of XDuce makes the documentation of the algorithms difficult.

1.5 Related Work

XDuce has made a significant impact in the XML world; in particular, it strongly influenced the type system of XQuery, the W3C standard query language for XML, as well as new schema languages such as TREX.

We outline here some of the main projects inspired from XDuce, specifically CDuce [BCF02, BCF03], Xtatic [GP03] and XQuery [GP03]. For an overview of other works and a thorough comparison of these with the XDuce approach we refer the reader to [HP00].

CDuce and XTatic CDuce ("seduce") is a general purpose functional programming language developed by Benzaken, Castagna, Frisch, whose design is targeted to XML applications. The work on CDuce started two years ago from an attempt to overtake some limitations of XDuce, as authors explain in [BCF03].

Xtatic is the "official" successor of XDuce developed at the University of Pennsylvania by B.C. Pierce, M.J. Levin, V. Gapeyev and A. Smith with the aim to integrate XDuce features in C#.

The goals for Xtatic and CDuce are similar in that they both try to integrate XDuce features in a larger design and a less XML-specific language. CDuce builds on the functional flavor of XDuce and extends it to a full-fledged functional language, whereas Xtatic goes toward OO principles and combines XDuce types with the class hierarchy of a host language, namely C#. The integration of XDuce and C# in Xtatic is smooth and elegant both on the language side (thanks to the introduction of a special `Seq` class, and a clever treatment of the concatenation) and on the implementation side (by resorting to an encoding technique reminiscent of Pizza's homogeneous translation [OW97]). Xtatic and CDuce both have to face the problem of combining XDuce's semantic definition of subtyping with a richer type algebra. Xtatic's solution is simpler as it relies on named typing for C# classes, where CDuce has to tackle classical issues when dealing with set-theoretic interpretation of arrow types. On a less theoretical side, CDuce and Xtatic share many design decisions, including syntactic choices and small decisions such as first-class XML tags and strings as sequences of characters. CDuce and Xtatic designs also have noticeable differences; among them is that CDuce avoids the stratification of the type algebra between XML types and non-XML types and this permits the use of pattern matching also for nonXML data structures (such as pairs or records). As for expressivity, their algorithm supports in addition ambiguous patterns (disambiguated with first match policy), non-linear capture variables (even under repetition operators), and XML attributes (implemented as records in CDuce).

XQuery XQuery is mainly aimed at performing queries on XML documents. Its type system took inspiration from that of XDuce. To perform queries, XQuery adds to XDuce a for loop as well as support for XPath, while it removes complex pattern matching with regular expression patterns and, recently, structural typing (replaced by named typing as in XML Schema [Con01], so as to avoid tree automata from having to check subtyping or validate documents).

Chapter 2

XDuce

XDuce (pronounced "transduce") is a tree transformation language, similar in spirit to mainstream functional languages but specialized to the domain of XML processing. Its novel features are regular expression types and a corresponding mechanism for regular expression pattern matching.

Regular expression types are a natural generalization of DTDs, describing, as DTDs do, structures in XML documents using regular expression operators (i.e. `*`, `?`, `|`, etc.). Moreover, regular expression types support a simple but powerful notion of subtyping, yielding a substantial degree of flexibility in programming.

Regular expression pattern matching is similar to ML pattern matching except that regular expression types can be embedded in patterns, which allows even more flexible matching.

In section 2.1 I introduce the language and its main novelties illustrating them with some simple examples, then I give a full detailed description of the language; a formalization of the language can be found in section 2.2. In section 2.3 I emphasize differences with the formalization of the author and I explain the reasons of the changes. Finally in section 2.4 I'll talk about some design changes of XDuce through the time and I'll make an attempt to motivate some author choices.

2.1 Highlights

In this section I give a detailed but somewhat informal explanation of the main aspects of the language. More demanding readers should read section 2.2 instead where I include a

formal definition of a core of the language[1]. Both the descriptions are very close but not exactly the same that author gives in [HP03]. Specifically formal description of the language has some non-trivial differences (see section 2.3 for a related discussion).

Below I will introduce basic features of XDuce: values, types, pattern matching and functions.

2.1.1 Values

Run-time values in XDuce roughly correspond to XML fragments[2]. To be precise, each value in XDuce is a sequence of nodes, each of which can be a base value (an integer, a float or a string), a labelled subtree or an empty sequence (note that a whole single-rooted XML document can be internally represented with a singleton sequence).

XDuce provides several operations for constructing such sequences. An expression of the form `l[e]` constructs a singleton sequence of a label `l` containing a sequence resulted from evaluating the expression `e`. Comma is a binary operator that concatenates two sequences (because it's associative, i.e. both `(e1,e2),e3` and `e1,(e2,e3)` produce the same sequence, we drop parentheses). String literals are enclosed in double-quotes, unlike XML. Finally the constructor `()` builds the empty sequence.

For example the XML document we've seen in section 1.2 can be represented with the following XDuce value:

```
addrbook[
  person[
    name["Haruo Hosoya"],
    email["hahosoya@kyoto-u"],
    email["hahosoya@upenn"]
  ],
  person[
    name["Benjamin Pierce"],
    email["bcpierce@upenn"],
    tel["123-456-789"]
  ]
]
```

Beside the use of sequence constructors we've seen above, XDuce implementation lets to load an external XML document from the file system with the built-in function "`load_xml(`$filename$`)`" and validate it at run-time against a type whose name is `X` with the built-in construct "`validate` v `with` X". Conversely, we can save a value `v` to an XML file with the built-in

[1] Actually I omit only attributes and unessential features (e.g. the `||` operator) or accessory features (e.g. built-in functions for typecasting, printing, ...). Concerning attributes author says in [HP03] that support for them is again on-going work

[2] see the description of XRel values in section **??** to see why there is not an exact correspondence

function "`save_xml(`$filename$`)(`v`)`"[3]. We'll give an example of the use of these statements in section 2.1.6.

2.1.2 Types

We've introduced types in section 1.2 when we've made the following example:

```
person[name[String], email[String]*, tel[String]?]
```

that describes values consisting of the single label `person` containing a sequence of a `name`, zero or more `email`s, and an optional `tel`, each containing a string.

XDuce types are descriptions of sets of structurally similar values. They're called regular expression types because they closely resemble ordinary string regular expressions, the only difference being that they describe sequences of tree nodes, whereas string regular expressions describe sequences of characters. As "atoms" we have labelled types like `label[T]` (which denotes the set of sequences containing a single subtree labelled label), the empty sequence type `()` and base types such as `String`, `Int` and `Float`[4].

Types can be composed by concatenation (comma), zero-or-more-times repetition (`*`), one-or-more-times repetition (`+`), optionality (`?`), and alternation (`|`, also called union).

Type expressions can be given names in XDuce programs by type definitions. For example in the followings:

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

the type named `Person` is defined to be an abbreviation for the type `person[Name,Email*,Tel?]`, which uses `Name`, `Email`, and `Tel` to refer to the types associated with these names. Type definitions are convenient for avoiding repetition of large type expressions in programs. More importantly, though, they may be (mutually) recursive; we will discuss this possibility further in Section 2.1.5.

Again, the XDuce implementation supports two ways of declaring types: use type definitions above or (statically) import existing DTDs from the external environment with the statement "`import_dtd` $filename$".

Label Classes

The labelled types we have seen so far have the form `l[T]` and describe singleton sequences labelled exactly with `l`. XDuce actually generalizes such types to allow more complex forms

[3]Functions with more than one argument in XDuce take each argument enclosed in a pair of parentheses

[4]Actually also string, float and int literals are base types, see section 2.2.3

called "label classes" that represent sets of possible labels. (This idea is also present in other XML type systems such as RELAX NG [CM01]). The `l` in the form `l[T]` is a label class representing a singleton set. The label class "`~`" represents the set of all labels. Among other things we can use it to define a type `Any` that denotes the set of all values[5]:

```
type Any = (~[Any] | Int | Float | String)*
```

We also allow a label class of the form $(l_1|\ldots|l_n)$ representing the choice between several labels. Such label classes are useful for describing a labelled type that has multiple possible labels, all with the same content type. For example, HTML headings may be labelled `h1` through `h6`, all with the content type `Inline`[6]:

```
type Heading = (h1|h2|h3|h4|h5|h6)[Inline]
```

More, we have a difference operator $L_1 \backslash L_2$ that allows to specify a set of labels L_1 without another set of labels L_2 and that is mostly useful to specify *any label* except a finite set in this way: $\sim\backslash(l_1|\ldots|l_n)$. Such use is as much common that we have an abbreviation for it: a "negation" operator `^...` that stands for $\sim\backslash\ldots$. For example, we can use the negation operator in the following way:

```
match v with
  ^(h1|h2)[Inline]*, (h1|h2)[val c as Inline], Any -> ...
```

where we extract the content of the first `h1` or `h2` label in the given value, ignoring all the other labels prior to this.

Recursive Types

XDuce supports recursive types for describing arbitrarily nested structures. Consider the following definitions:

```
type Fld = Rcd*
type Rcd = name[String], folder[Fld]
         | ( name[String], url[String], (good[] | broken[]) )
```

The mutually recursive types `Fld` ("folder") and `Rec` ("record") define a simple template for storing structured lists of bookmarks, such as might be found in a web browser; in this example a folder is a list of records, while a record is either a named folder or a named URL plus a boolean indicating whether the link is good or broken. We'll use these types to build an example on recursive types in section 2.1.5.

[5] spurious empty nodes inside sequences of values are removed by the value constructors so that the given definition is a supertype for any possible value

[6] in current version (0.4.0) the line above won't work but we must write `~(h1|h2|h3|h4|h5|h6)[Inline]` instead. This is to make parsing easier (personal communication with the author). See also note in section 2.2.1

For theoretical reasons (to limit expressiveness of types to tree regular grammars instead of tree context-free grammars) we impose a constraint on the kind of recursion you can specify in a type. Specifically we require that type definitions don't have recursion "at the top-level" i.e. that either they don't have recursion at all or that if there is recursion this will be only inside subtrees. For example in the following:

```
type X = X | ()
```

`X` is not a well formed type because recursion is at top-level, while in the following:

```
type Y = a[ Y | () ]
```

`Y` is a well formed type (recursion is not at top-level).

Note that this doesn't exactly means that whenever we have a type name at top-level that causes a definition to be recursive we have a recursion "at top-level".

Actually the example on folders above has the following structure:

```
type F = R*
type R = ..., folder[F] | ...
```

In this example, that is well formed, the presence of `R` at top-level inside the definition of `F` leads to a recursion of `F` within a subtree; said in other words, `F` is recursively defined because it's possible to "reach" the type itself from its definition expanding type names you can find inside (`R` in the example), but it's not recursive *at top-level* because you can reach it from its definition only within subtrees and never at top-level. This notion of "reachability" is precisely formalized in section 2.2.3 where it's unsurprisingly used to define recursion at top-level.

2.1.3 Pattern Matching

So far we've focused on building values. We now turn our attention to decomposing existing values by pattern matching. We've already introduced pattern matching in section 1.2 but let us make another simple example.

Suppose to have a value whose label describes an Internet protocol name and consider you wish to create an URL string from it. You could use an expression like this (the binary operator `^` is a string concatenation):

```
match v with
    www[String as s] -> "http://" ^ s
  | email[String as s] -> "mailto:" ^ s
  | ftp[String as s] -> "ftp://" ^ s
```

This pattern match branches depending on the top label (`www`, `email` or `ftp`) of the input value and evaluates the corresponding body expression, which prepends the appropriate string to the variable `s`, which is bound to the content the label in the input value `v`.

In general, a pattern match expression takes an input value and a set of clauses of the form "*pattern* `->` *expression*". Given an input value, the pattern matcher finds the first clause whose pattern matches the value. It extracts the subtrees corresponding to bound variables in the pattern and then evaluates the corresponding body expression in an environment enriched with these bindings.

Patterns can be nested to test for the simultaneous presence of multiple labels and extract multiple subtrees, as in following pattern:

```
person[name[String as n], email[String as e]]
```

where the text content of the label `name` is bound to the variable `n` and the text content of the label `email` is bound to the variable `e`. Also, the logical-or can be expressed by the union operator:

```
email[String as s] | tel[String as s]
```

Indeed, XDuce patterns have exactly the same form as type expressions, except that they may include variable binders of the form "*pattern* `as` *x*"[7] (which checks if there is a match between the input value and *pattern* and if so "binds" the variable to the value — i.e. assigns the value to the variable). We demand that, for any input value, a pattern yields exactly one binding for each variable (we call this condition "linearity", see section 2.2.3 for a formal definition). Thus, a pattern like

```
email[String as e] | tel[String as t]
```

is forbidden because at least one of the variables will no be bound, no matter the input value is, whereas

```
email[String as e]*
```

is forbidden because variable `e` could have any number of bindings.

Since patterns are just types decorated with variable binders, we can even use patterns to perform dynamic typechecking. For example, the pattern

```
person[Name, Email+, Tel+]
```

matches the subset of elements of `Person` that contain a value of type `Name` followed by one or more values of type `Email` and then one or more values of type `Tel`. This capability is beyond the expressiveness of pattern matching facilities in conventional functional languages such as ML and Haskell.

[7] there are two equivalent syntaxes for bindings, the one presented in the text is the newest one but you can use as well the old syntax "`val` *x* `as` *pattern*". See section 2.4.1 for a discussion on the binders' syntax

First-Match Semantics

XDuce's pattern matching has a "first-match" semantics. That is, a pattern match expression tries its clauses from top to bottom and fires the first matching one. This semantics is particularly useful to write default cases. For example, in the following pattern match expression

```
match v with
    person[name[val n as String], Email+, Tel+] -> ...
  | person[name[val n as String], Any] -> ...
```

the first clause matches when the input person value contains both emails and tels, and the second clause matches otherwise (remember that `Any` is a type that matches any values, see section 2.1.2). If such overlapping patterns were not permitted, we would have to rewrite the second pattern so as to negate the first one, which would be quite cumbersome.

What if none of the clauses match? As we'll see in the next section, XDuce performs a static exhaustiveness check so that such a failure can never arise.

Exhaustiveness and irredundancy checks

A pattern is ambiguous if it yields multiple possible parses of some input value. For example, the following is ambiguous.

```
match v with
  (a[]|b[])*, (val x as b[]), (a[]|b[])* -> ...
```

Take the input value `b[],b[]`. There are two parses for this value. One assigns the first b to the leftmost `b[]` pattern and the second b to the middle `b[]` pattern. The other parse assigns the first `b` to the middle `b[]` pattern and the second `b` to the rightmost `b[]` pattern.

Usually, an ambiguous pattern signals a programming error. However, XDuce authors have found that, in some cases, writing ambiguous patterns is reasonable, therefore they decided to yield a warning for ambiguity rather than an error. In the case that the user writes an ambiguous pattern and ignores the warning, the semantics of pattern matching is to choose an arbitrary parse among multiple possibilities ("nondeterministic semantics").

Type Inference

The type annotations on pattern variables are normally redundant. For example, in the following pattern match taking values of type `Person*`:

```
match ps with
   person[name[String as n], Email*, Tel?], Person* as rest
     -> ...
 | ...
```

the type "`String`" of the variable `n` and the type "`Person*`" of the variable `rest` can be deduced from the input type and the shape of the patterns. XDuce supports a mechanism that automatically infers such type annotations. With type inference, the example above can be rewritten as follows.

```
match ps with
    person[name[val n], Email*, Tel?], val rest
      -> ...
  | ...
```

i.e. it's possible to use "bare" variables whose type will be computed from the compiler.

From the examples we have seen, it might appear that, whenever a pattern contains a binding of the form `(T as x)`, the inferred type for `x` is `T` itself. It is not always the case, however, as type inference may compute a more precise type than `T`.

Formally, the syntactic form `val x` is an abbreviation for `val x as Any`[8]. So, XDuce has not "true" bare variables but it requires type annotations on all pattern variables, only that type inference mechanism allow them to be larger than necessary. The actual types of the variables are inferred by combining the types given by the programmer with the types discovered by propagating the input type through the pattern. For example, the pattern

```
match v with
  (val head as ~[Any]), val tail -> ...
```

binds head to the first labelled value in the input sequence and tail to the rest of the sequence. The types inferred for head and tail depend on the input type. For example, if the input type is `(Email|Tel)*`, then we infer `(Email|Tel)` for head and `(Email|Tel)*` for tail. Indeed, if the input type is `(Email*,Tel)`, then we infer `(Email|Tel)` for head and `(Email*,Tel)?` for tail. This combination of declared and inferred structure is useful since it is often more concise to write a rough pattern (like the above `~[Any]` pattern) than a precise one. In addition, if the input type is changed later on, we may not have to change the pattern, since the type inference will recompute appropriate types for the variables.

2.1.4 Functions

Let us illustrate them with an example. Suppose we want to extract from a given value of type `Person*` (as it's defined in 2.1.2) all the entries with telephone numbers. We can use the following function:

[8] remember that `val x as Any` is the old syntax for the variable binder; author says that this old syntax remains for backward compatibility but there are many valid reasons to retain it — bare variables is one of these. See 2.4.1 for a discussion

```
(* conversion subroutine *)
fun make_tel_book (val ps as Person* ) : person[Name,Tel]* =
  match ps with
      person[name[val n], tel[val t], Email*],
             val rest as Person*
        -> person[name[n], tel[t]], make_tel_book(rest)
    | person[name[val n as String], Email*], val rest as Person*
        -> make_tel_book(rest)
    | ()
        -> ()
```

Note that the header shows that function takes a value `ps` of type `Person*` and returns a value of type `person[Name,Tel]*`.

The body of the function uses a pattern match to analyze `ps`. In the first case, the input sequence has a `person` label that contains a `tel` label; we pick out the `name` and `tel` components from the person, construct a new `person` label with them, and recursively call `make_tel_book` to process the remainder of the sequence. In the second case, the input sequence has a `person` label that does *not* contain a `tel` label; we simply ignore this person `label` and recursively call `make_tel_book`. In the last case, the input sequence is empty; we return the empty sequence itself.

Note the "functional flavor" of XDuce and specifically its ML inheritance (the style of comments, the syntax of both the match construct and the functions, the code blocks conceived as expressions): this is the reason why author asserts that is a functional language. However XDuce is a *first–order* and not an *high–order* functional language, so you can't pass functions as arguments as in ordinary functional languages.

You can also define functions with multiple arguments. The syntax is the following:

`fun f(`P_1`)(`P_2`)...(`P_n`): T = e`

For example XDuce uses functions with multiple arguments for basic mathematical operations. Here's an excerpt of the include file pervasive.q that contains declarations of several built-in functions:

```
...
extern iplus : (Int)(Int) -> Int
extern iminus : (Int)(Int) -> Int
extern imul : (Int)(Int) -> Int
extern idiv : (Int)(Int) -> Int
extern imod : (Int)(Int) -> Int
extern rplus : (Float)(Float) -> Float
extern rminus : (Float)(Float) -> Float
...
```

Finally a last note about the structure of a XDuce program. A XDuce program is a set of type definitions, (possibly unnamed) global variable definitions and function definitions, in any order. There is neither a "main" function nor a starting term with which to begin the execution. The execution of the code is a "side–effect" of the evaluation of the expressions to assign to the variables; evaluation is performed from the top to the bottom of the code; this will possibly invoke also code inside function bodies.

2.1.5 Functions and Output types

XDuce uses types for various purposes. The most important is in checking that the values that may be consumed and produced by each function definition are consistent with its explicitly declared argument and result types.

For example, consider the following function definition.

```
fun make_person (val nm as String)(val str as String) : Person =
  person[name[nm],
         (if looks_like_telnum(str) then tel[str] else email[str])]
```

The first line declares that the function `make_person` takes two parameters `nm` and `str` of type `String` and returns a value of type `Person`. (The `val` keyword in the function header is a signal that the following identifier is a bound variable.) In the body, we create a tree labelled person that contains two subtrees. The first is labelled name and contains the string `nm`. The second is labelled either `tel` or `email`, depending on whether the argument `str` looks like a telephone number or an email (according to some function looks like telnum defined elsewhere). As a result, the type of the whole body is this:

```
person[name[String], (email[String] | tel[String])]
```

Finally, we check that this type is a subtype of the annotated result type `Person`.

Another use of subtyping is in checking the type of an argument to a function call against the parameter type given by the programmer. For example, if we have the function definition

```
fun print_fields (val fs as (Name|Tel|Email)*) : () =
  ...
```

we can apply it to an argument of the following type:

```
Name,Email*,Tel?
```

Note, again, that, though they are syntactically quite different, the argument type and the parameter type are in the subtype relation (the ordering constraint in the argument type is lost in the parameter type, yielding a strictly larger set).

Folder Manipulation

Consider again the types described in section 2.1.2:

```
(* type of the input document *)
type Fld = Rcd*
type Rcd = name[String], folder[Fld]
         | ( name[String], url[String], (good[] | broken[]) )
```

Remember that a folder is a list of records, while a record is either a named folder or a named URL plus a boolean indicating whether the link is good or broken. We can define the following functions:

```
fun tidyFolder (val fl as Fld) : Fld =
  match fl with
      Rcd as record, Fld as folder
        -> tidyRecord(record), tidyFolder(folder)
    | ()
        -> ()

fun tidyRecord (val rc as Rcd) : Rcd? =
  match rc with
      name[String as nm], folder[Fld as fl]
        -> name[nm], folder[tidyFolder(fl)]
    | name[String as nm], url[String as s], good[]
       -> name[nm], url[s], good[]
    | name[String], url[String], broken[]
        -> ()
```

The functions `tidyFolder` and `tidyRecord` traverse a bookmark list recursively, preserving leaves with good links and dropping ones with bad links.

2.1.6 A Complete Example

Here's a small but complete example, taken by section 2.5 in [HP03]. The task of this program is to create, from an address book document, a telephone book document by extracting just the entries with telephone numbers.

After including the library for import/export from/to XML we specify the type definitions for input and output documents.

```
(* needed to use load_xml / save_xml *)
import "xml.q"

(* type of the input document *)
```

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

(* type of the output document *)
type TelBook = telbook[TelPerson*]
type TelPerson = person[Name,Tel]
```

After the declarations we have the input processing: we load an address book document from a file and validate it against the type `Addrbook`. Note that there can be a run-time error here if document doesn't conform to the type.

```
(* import and validate the document *)
let val doc = load_xml("mybook.xml")
let val valid_doc = validate doc with Addrbook
```

The "main" code extracts the content of the top label `addrbook`, send it to the function `make_tel_book` (defined below) and enclose the result with the label `telbook`.

```
(* out_doc = output document *)
let val out_doc =
  match valid_doc with
    addrbook[val persons as Person*] ->
      telbook[make_tel_book(persons)]
```

Finally, we show the result on the console and export it to an XML file. Note the use of a dummy variable "_" to simulate a procedure call.

```
(* show result *)
let val _ = display("Result:")
let val _ = print(out_doc)
(* and export to XML *)
let val _ = save_xml("output.xml")(out_doc)
```

At last we include the function defined in section 2.1.4.

```
(* conversion subroutine *)
fun make_tel_book (val ps as Person* ) : TelPerson* =
  match ps with
      person[name[val n], tel[val t], Email*],
             val rest as Person*
        -> person[name[n], tel[t]], make_tel_book(rest)
    | person[name[val n as String], Email*], val rest as Person*
```

```
      -> make_tel_book(rest)
  | ()
      -> ()
```

The example above is complete as it correctly works without modifies with the last version of the tool.

2.2 Core Language Definition

Here I give a formal definition of a subset of the surface language of XDuce as it has been implemented in the last version of the tool (0.4.0)[9].

Although this formalization is very close to that author gives in [HP03] there are some differences, mostly trivial but a few significative. Specifically there are two significative differences:

1. I remove from the language the explicit pattern definition statement because it's not implemented in the current release
2. I change the top-level definition of the language to make it closest to the implemented language

I wonder these differences as Hosoya works on the tool from at least three years updating it slowly but constantly, so I don't think these differences are to ascribe to an immaturity of the code. Furthermore, while discordance between the top-level of the implementation and that of formalization have no conceptual differences so that discrepancies are needless confusing, it's my opinion that explicit pattern definitions are not an useful feature in the language. I'll argue about these points in section 2.3.

Note that this formalization is more near to that I give for XRel so that it's easier to view syntactic differences between XRel and XDuce.

2.2.1 Labels and Label Classes

Assume a (possibly infinite) set `L` of *labels*, ranged over by `l`. *Label classes* are defined by the following syntax:

$L ::= l$	specific label
$\mid \sim$	wildcard label
$\mid L \mid L$	union
$\mid L \setminus L$	difference

[9] I remove from the implemented language attributes and some accessory features like the string concatenation operator "^" so to have a formalization more near to that given in [HP03] and similar to that of the XRel language

We write a negation `^L` as an abbreviation of `~\L`.

The semantics is that obvious (you can find its formalization in [HP03]).

Note that in the current version of the tool (0.4.0) you can write `l[...]` or `^L[...]` but you can't use directly unions and differences. To use them you have to write `~(`$\mathtt{L}_1$`|...|`$\mathtt{L}_n$`)[...]` and `~(`$\mathtt{L}_1$ \ $\mathtt{L}_2$`)[...]` (I've contacted the author by email and he said me that this is a trick to make parsing easier).

2.2.2 Values

A *value* `v` is a sequence of base values and labelled values, where a labelled value is a pair of a label and a value. We use the following syntax for values:

$v ::= ()$	empty sequence
$\mid l[v]$	tag with label l
$\mid v\ ,\ v$	sequence
$\mid s$	literal string (e.g. "hello")
$\mid i$	literal integer (e.g. 5 or -6)
$\mid f$	literal float (e.g. 5.0 or -6.2)

where s ranges over double quoted literal strings, i ranges over integers and f ranges over floating numbers (differently by the author I prefer to explicitly formalize base values, see section 2.3). Integer and float numbers are syntactically different so that there is no confusion between them (e.g. 5 is a integer, 5.0 or 5. is a float).

2.2.3 Types and Patterns

We first define patterns, then we will simply define types as the set of patterns which lack variable bindings.

We assume a countably infinite set of type names, ranged over by X, and a countably infinite set of variables, ranged over by x. Pattern expressions (or just *patterns*) are defined as follows:

$P ::= X$	type name
$\mid L\ [P]$	labelling
$\mid ()$	empty pattern
$\mid P\ ,\ P$	concatenation
$\mid P \mid P$	union
$\mid P*$	repetition (zero or more)
$\mid P$ `as` x	variable binder
$\mid$ `val` x `as` P	variable binder (old syntax)

`| String`	string type
`| Int`	integer type
`| Float`	float type
$\mid s$	literal string (e.g. "hello")
$\mid n$	literal number (e.g. 5.0 or -6)

It's possible to use `L[]` as an abbreviation of `L[()]`. Also we can define the other regular expression operators as syntactic sugar:

```
P? = P | ()
P+ = P, P*
```

Furthermore, `val x` can be used to mean `val x as Any`, where `Any` is defined below. Finally we assume the following fixed type definition[10]:

```
type Any = ( ~[Any | Int | Float | String] )*
```

Linearity constraints on patterns

Let `BV(P)` be the set of variables bound in the pattern `P`. We say that `P` is linear iff for any subphrase `P'` of `P` the following conditions hold:

$x \notin BV(P_1)$	if $\begin{cases} P' = P_1 \texttt{ as } x \\ P' = \texttt{val } x \texttt{ as } P_1 \end{cases}$
$BV(P_1) \cap BV(P_2) = \emptyset$	if $P' = P_1 \texttt{ , } P_2$
$BV(P_1) = BV(P_2)$	if $P' = P_1 \mid P_2$
$BV(P_1) = \emptyset$	if $P' = P_1 *$

Types

Type expressions T (or just *types*) are patterns that have no bound variables, i.e. such that $BV(T) = \emptyset$. Types are ranged over `S`, `T`, `U`, etc.

Well-formedness of types

Types are well-formed when they don't have recursion "at the top-level" of theirs definitions, i.e. when they have not a recursive definition or there is recursion only inside subtrees. For example in the following

[10] Implemented language has other built-in type definitions as `AnyElm` but they're not interesting for our purposes

```
type X = X | ()
```

`X` is not a well formed type, while in the following

```
type Y = a[ Y | () ]
```

`Y` is a well formed type (recursion is not at top-level).

Formally consider a mapping E from type names to theirs bodies. This means that when we have a declaration as the following:

```
type X = T
```

we write $E(\texttt{X})$ to intend the right-hand side of the declaration (`T` in the example above). For a given type `T`, we define the set S(T) of type names reachable from T at the top level as the smallest set satisfying the following

$$S(T) = \begin{cases} S(E(X)) \cup \{X\} & \text{if T=X} \\ S(T_1) & \text{if } T = T_1^* \\ S(T_1) \cup S(T_2) & \text{if } T = T_1, T_2 \text{ or } T = T_1 \mid T_2 \\ \emptyset & \text{otherwise} \end{cases}$$

We then require that every type is not reachable at the top level from itself. Formally we require that the set E of type definitions satisfies

$$\texttt{X} \notin \texttt{S(}E\texttt{(X))} \qquad \forall\ \texttt{X} \in dom(\texttt{E})$$

Note that differently from the author we only need to apply this check to types because other patterns can't be recursive. Apart this, formalization above is the same author gives.

Semantics

Semantics is almost the same of the author, the only significative difference is that we can merge the matching semantics of types with that of patterns, as types are considered a subset of patterns.

I rewrite below the semantics of patterns as there are some trivial differences with that formalized in [HP03] (specifically the new binding syntax, the explicit semantics of base values and the rule EP-Var adapted to expand type names only).

The semantics of patterns is given by the relation $v \in P \Rightarrow V$ read "value v is matched by P, yielding environment V", where an environment V is a finite mapping from variables to values (written $x_1 : v_1, \ldots, x_n : v_v$). The concatenation of environments binding distinct

variables is written with a comma.

$$\frac{v \in P \Rightarrow V}{v \in (P \texttt{ as } x) \Rightarrow x : v, V} \quad \text{(EP-As1)}$$

$$\frac{v \in P \Rightarrow V}{v \in (\texttt{val } x \texttt{ as } P) \Rightarrow x : v, V} \quad \text{(EP-As2)}$$

$$() \in () \Rightarrow \emptyset \quad \text{(EP-Emp)}$$

$$\frac{E(X) = P \qquad v \in P \Rightarrow V}{v \in X \Rightarrow V} \quad \text{(EP-Var)}$$

$$\frac{v \in P \Rightarrow V \qquad l \in L}{l[v] \in L[P] \Rightarrow V} \quad \text{(EP-Lab)}$$

$$\frac{v_1 \in P_1 \Rightarrow V_1 \qquad v_2 \in P_2 \Rightarrow V_2}{v_1, v_2 \in P_1, P_2 \Rightarrow V_1, V_2} \quad \text{(EP-Cat)}$$

$$\frac{v \in P_1 \Rightarrow V}{v \in P_1 \mid P_2 \Rightarrow V} \quad \text{(EP-Or1)}$$

$$\frac{v \in P_2 \Rightarrow V}{v \in P_1 \mid P_2 \Rightarrow V} \quad \text{(EP-Or2)}$$

$$\frac{v_i \in P \Rightarrow V_i \quad \forall i}{v_1, \ldots, v_n \in P* \Rightarrow V_1, \ldots, V_n} \quad \text{(EP-Rep)}$$

$$i \in i \Rightarrow \emptyset \quad \text{(EP-LitInt)}$$

$$i \in \texttt{Int} \Rightarrow \emptyset \quad \text{(EP-Int)}$$

$$f \in f \Rightarrow \emptyset \quad \text{(EP-LitFloat)}$$

$$f \in \texttt{Float} \Rightarrow \emptyset \quad \text{(EP-Flt)}$$

$$s \in s \Rightarrow \emptyset \quad \text{(EP-LitStr)}$$

$$s \in \texttt{String} \Rightarrow \emptyset \quad \text{(EP-Str)}$$

Note that linearity ensures that environments that are concatenated in the conclusions of rules EP-As1, EP-As2, EP-Cat and EP-Rep have different domains (e.g. $x \notin dom(V)$ always holds in rules EP-As1 and EP-As2).

Note also that the application of the rule EP-VAR will always yield an empty environment V, because X refers to a type name, and types have no binders. The use of this rule is a little trick to avoid to define an apposite relation $v \in T$.

Finally consider the match of base values: the value 5 will match with itself or `Int` but it will won't match with 5.0 or `Float`; similarly 5.0 matches with itself and with `Float` but it will won't match with 5 or `Int` and "5" won't match neither with `Int` or `Float`. Full language has built-in functions for explicit typecasting, but implicit typecasting is never allowed.

I skip all static semantics because differences with that of [HP03] are trivial and similar to those I've made above on pattern matching semantics.

2.2.4 Terms

We assume given a countably infinite set of function names, ranged over by f (we'll see function definitions in the next section).

The syntax of terms e is defined by the following grammar:

$e ::= l\ [e]$	tag with label l
$\mid e\ ,\ e$	concatenation
$\mid x$	program variable
$\mid v$	value
$\mid f(e)$	application
$\mid$ `match` e `with` $\bar{P}$ `->` $\bar{e}$	pattern match

where we write "$\bar{P}$ `->` $\bar{e}$" as an abbreviation for the n-ary form "$\mathtt{P}_1$ `->` $\mathtt{e}_1$ `|` ... `|` $\mathtt{P}_n$ `->` $\mathtt{e}_n$".

We also allow the following shorthands:

$$\begin{aligned} \texttt{let}\ P = e_1\ \texttt{in}\ e_2 &\equiv \texttt{match}\ e_1\ \texttt{with}\ P\ \texttt{->}\ e_2 \\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 &\equiv \texttt{match}\ e_1\ \texttt{with}\ \texttt{True}[\,]\ \texttt{->}\ e_2, \texttt{False}[\,]\ \texttt{->}\ e_3 \\ e_1\ ;\ e_2 &\equiv \texttt{match}\ e_1\ \texttt{with}\ \mathit{Any}\ \texttt{->}\ e_2 \end{aligned}$$

A few considerations on constructs above:

- the `let` statement is that used inside terms, for example inside function bodies, and it's different from that you can use at top-level to define a "global" variable
- $\mathtt{e}_1\mathtt{;}\mathtt{e}_2$ is useful when you want to discard the value of $\mathtt{e}_1$ retaining only that of $\mathtt{e}_2$; this operator can be useful for example when $\mathtt{e}_1$ is a function call that has an useful "side-effect" (e.g. printing)
- Note that the "else" part of the `if-then-else` construct is not optional

Again, we won't replicate static semantics here because differences with [HP03] are trivial.

2.2.5 Top-level language

I propose here a formalization of the top-level of the language completely different from that given by the author in [HP03] but closest to the implementation.

$$
\begin{aligned}
Program &::= Definition* \\
Definition &::= TypeDef \\
&\quad\ \ ValDef \\
&\quad\ \ FunDef \\
TypeDef &::= \texttt{type}\ X = T \\
ValDef &::= \texttt{let}\ P = e \\
FunDef &::= \texttt{fun}\ f(P) \colon T = e
\end{aligned}
$$

Note that in the full implemented language there are other things at top-level such as the construct "`import_dtd`" to statically import DTD schemas into XDuce types, a support for XML namespaces and other built-in constructs we'll ignore here as we limit our formal description to the core features[11]. More, for brevity I formalize (as author) only one-argument functions because the extension to multi-argument functions is straightforward.

In [HP03] there are the evaluation rules for terms and the typing rules both for terms and function definitions, but Hosoya never talks about the semantics of variable definitions at top-level. However it's straightforward to give an encoding from the top-level variable definitions of XDuce implementation to the top level with an only starting term formalized by Hosoya. For example, having a program with the following variable definitions:

```
let P1 = e1
let P2 = e2
...
let Pn = en
```

we have only to encode them in the following stand–alone term:

```
let P1 = e1 in let P2 = e2 in ... let Pn = en in ()
```

2.3 Differences with the Formalization of the Author

You can see an updated formalization of the language in [HP03]. Differences between formalization you find in this paper and my formalization are the following:

[11] To see a full list of them take at look at the source file parser.mly [XDu]

- I remove "`pat Y = P`" because it's not implemented in the current version of XDuce (0.4.0). It was implemented in an old version? Author wish implement it in future? I don't know (see discussion below)

- for the modify above the formalization of linearity constraints on patterns becomes simpler. We have not to "reach" bound variables inside patterns as Hosoya does; the only names we can find inside pattern expressions are type names, but types have no bindings so that we don't need to look inside them.

- top-level definition of the language is different: Hosoya says that a program is a set of type definitions, a set of pattern definitions, a set of function definitions and a starting term. Actually this is a very good top-level design and it's completely equivalent to that really implemented so... why he has not implemented it? why to "invent" a different syntax for global variable definitions? I can't figure why

- I define types as patterns without bindings. This is actually what author does in a more recent paper [Hos03] but here he describes only internal representation of patterns; my contribution is to apply this simple idea to the surface language so to make its description easier to read (both for improved conciseness and for the highlighting of the strong relationship between the structure of the types and that of the patterns)

- I formalize base types and values: why author doesn't? he says that he omits them from the formalization because the modify is straightforward but my opinion is that explicit formalization make things neater. For example one of my doubts when I begun to study XDuce was if it's possible to insert literal values inside types and patterns. See discussion below for a list of reasons I think it's better to explicitly handle base types/values in formalization

Let us see main points above in details now.

2.3.1 Removal of Explicit Pattern Definitions

There is an essential difference between the implementation of XDuce (0.4.0) and its formalization as found in [HP03] or in other papers of Hosoya: the implementation lacks the capability to define explicitly patterns with the instruction `pat`. In the papers there is a declaration

```
pat X = P
```

which declares pattern variables, similar to the existing type variable introduced with `type`. This difference is not trivial because from its presence we have or less the ability to write recursive patterns and ultimately to make bindings at arbitrary depths.

Because in [HP03] Hosoya talks about `pat` statement only in the formal definition of the core language (and not in the informal description of the surface language), one could suspect that its presence is only a trick he uses to make things simpler in formalization, and that he never thought to use it for the "true" language.

However, in the following excerpts taken by [HP02], he claims that it's possible write recursive patterns in XDuce and gives an example.

> Nothing prevents us from writing a single pattern that traverses a tree to an arbitrary depth. For example, consider the following recursively defined type for binary trees, with two forms of leaves, `b[]` and `c[]`, and internal nodes labeled a,
>
> ```
> type T = a[T],T | b[] | c[]
> ```
>
> and the match expression
>
> ```
> match t with
>   P -> ...
> ```
>
> where `P` is recursively defined as follows:
>
> ```
> pat P = a[P],T | a[T],P | x as b[]
> ```
>
> The pattern `P` matches a tree that has at least one `b[]`, and yields exactly one binding of the variable `x` to one of the `b[]`s. Since `P` has the choice of patterns `a[P],T` and `a[T],P` in this order, the first-match policy ensures that the variable `x` is bound to the first `b[]` in depth-first order.[12]

Why this discordance between theory and practice? We can only wonder this discordance also because the same author in the continuation of the excerpt above says that this feature is unessential:

> Although this "deep" matching is somewhat attractive, we are not sure about its usefulness because, after obtaining the first `b[]` as above, it is not clear what to do to get the next one, or more generally to iterate through all the `b[]`s in the input tree. (By contrast, this sort of deep matching would be more clearly useful if we had chosen the "all-matches" semantics instead.)

This opinion of the author and the fact he's developing XDuce for three years make we think that this discrepancy is not to ascribe to an "immaturity" of the code. We remark also that other languages based on XDuce (namely CDuce and Xtatic) claim to allow recursive patterns.

[12] Note two old things of this excerpt: bindings syntax is not updated and he talks about first match policy inside a pattern

2.3.2 Top-Level Language

There are some differences of the top–level language as it's formalized with respect to as it's implemented. Actually I needed to perform several tests with the tool before to understand, between other things, the followings:

- the syntax of variable definition at top-level is different from that of local variable definitions inside functions
- type declarations, variable definitions and function definitions can be mixed (all examples in papers put types before other things). Mixing them could be useful to improve code readability
- functions accept as formal parameters arbitrary patterns, not only sequences of variables with theirs types

To see why it could be useful to specify arbitrarily complex patterns as formal parameters instead of sequences of binders consider the following example, where instead to specify a match construct with one only clause as in this too much verbose code:

```
fun process(val v): ... =
  match v with
    root[a[String as s1], b[String as s2]] ->
      (* ... use s1 and s2 ... *)
```

we can simply use the whole pattern as formal parameter as in this neater code:

```
fun process(root[a[String as s1], b[String as s2]]): ... =
  (* ... use s1 and s2 ... *)
```

2.3.3 Base Types and Base Values

We draw attention to some aspects that are implicit in formalization, but which we initially found confusing.

- it's possible to insert literal values inside types and patterns. For instance, the type `3, int*` denotes all lists whose first element is `3`
- XDuce has no implicit typecasting mechanism (as I've discussed in section 2.2.3)
- `Int` and `Float` types never match against XML documents; instead, one must use explicit tpyecasting functions from XML `String`s to numeric types
- a pattern `String,String` could never match with a XML document, as there is no pattern matching mechanism on strings (only on trees)

2.4 Design Changes

In this section we will see some design changes and will see how authors motivated them.

2.4.1 Changes of the Binding Syntax

Table 2.1 reports the evolution of bindings' syntax in XDuce through the time. Note how syntax is changed three times taking 4 different syntaxes through the years.

	Syntax	**Papers**	**Still alive**	**From**
1	x[:P]	[HVP00],[HP00]	no	1999?
2	x [as P]	[Hos01],[HP02]	no	2000-1?
3	val x [as P]	[HP03]	yes	2002?
4	P as x	[Hos03],[Hos04]	yes	2003 (0.4.0)

Table 2.1: Evolution of bindings' syntax in XDuce

Why author changed its mind so many times? My opinion is that, although syntax is just an aesthetic problem, it is not easy to solve it in an optimal way.

For example the first choice ("`x:P`") is elegant and similar to Pascal, but conflicts with namespaces — consider `x:uri:label[...]`. The second author choice were to replace the semicolon with the `as` operator obtaining "`x as P`" (or simply `x` for bare variables) – note this is the actual syntax of XQuery. Why he changed its mind again? I guess he would intended to avoid some subtle errors: a misspelled type name would be considered by the compiler the name of a bare variable, so that an "undefined identifier error" would be never signalled. So he added the keyword `val`. In this way the attempt of using a bare variable is immediately identifiable from the presence of the keyword and wrong type names can be detected as well.

Up to this point reasons behind author's choices are simple to imagine, also without he never talked about. But the last change is a little more demanding for our imagination: why he changed a good syntax like "`val x as P`" with an equivalent one like "`P as x`"?

Because I don't know real author motivations, I can only advance suppositions. The first thing one can imagine is that he thought that the use of the `val` keyword made syntax less readable as before, so that he would be intentioned to revaluate its last choice and return to a syntax more similar to the old one. Note how the problem of bare variables was solved: forcing the user to explicitly write "`Any as x`". But actually also if definition of "normal" typed variables is less verbose, the use of bare variables has become awkward, not so much comfortable as before. This take me to a second supposition, that is my actual opinion: that he changed its mind to suggest a programming style more type-oriented, and less abusing of

"magic" untyped variables. In fact in the file "`CHANGES`" of the distribution of XDuce0.4.0 [XDu] author wrote that "the old syntax remains for backward compatibility".

Note how the usefulness of the old syntax is as far to be obsolete. For example when we declare a variable with the `let` statement in such a way:

```
let val x = ...
```

we are using it because the syntax of the let statement is "`let P = ...`" so that without this "old" syntax we would need to write instead:

```
let Any as x = ...
```

2.4.2 Longest Match vs Arbitrary Match Policy

The match policy in pattern matching has undergone substantial changes.

There are two kinds of ambiguity in pattern matching. The first is when more than one clause matches the same input. This has been resolved in all versions of XDuce by the "first–match" policy, that is the first matching clause is taken.

The second kind of ambiguity is when the pattern of a single clause can match the same value in different ways. In older versions of XDuce this was to resolved with the "longest–match" policy, that earlier subpatterns of a sequence will "greedily" match as much as possible. In newer versions of XDuce the policy is abandoned, the compiler gives an "ambiguous match" warning, and at run–time it is not defined which match will be taken.

We adopt the newer solution of giving a compiler-warning for ambiguous patterns. Note, incidentally, that the ambiguity check has been disabled in the most recent versions of XDuce because it conflicted with the handling of attributes.

In this section I report both the reasons, the original one about why the longest match is good and the last one about why the longest match is bad. Note how both motivations are valid and how neither of one contradicts with the other.

Advantages of longest match policy

Old approach can be found in [HP02]. Here's an excerpt.

> The second form of ambiguity occurs when a single pattern can match a given value in different ways, giving rise to different bindings for the pattern variables. [..] For example, in the pattern

```
match e with
  e1 as Email*, e2 as Email*
    -> ...
```

> which splits a sequence of emails into two, it is ambiguous how many emails the variable e1 should take. We resolve this ambiguity by adopting a "longest match" policy where patterns appearing earlier have higher priority. In the example, e1 is bound to the whole input sequence, e2 to the empty sequence. Again, an alternative design choice would be to disallow such ambiguity. However, the longest-match policy can make patterns more concise.

In this approach longest match policy is derived from first match policy, as explained in this other excerpt:

> The longest-match policy actually arises from these abbreviations and the first-match policy. For example, Email* is defined as a pattern name Y that is recursively defined as
>
> ```
> pat Y = Email,Y | ()
> ```
>
> and, with the first-match policy, the first branch (Email,Y) is taken as often as possible, which accounts for the longest-match policy. The same argument is applied to the other operators + and ?. Notice that the order of union clauses in the definitions of the abbreviations matters for the semantics of pattern matching.

The longest-match policy can make patterns more concise. Consider the contents of an HTML dl (description list), which is a sequence of type (Dt|Dd)*, where Dt (term) and Dd (description) are defined as dt[...] and dd[...], respectively (the content types "..." are not important here). Suppose we want to format this sequence in such a way that each term is associated with all the following descriptions before the next term (if any). We may write an iteration for scanning the sequence.

```
match l with
  dt[t], d as Dd*, rest
    -> (* display term t with d, and do rest *)
 |
  () -> (* finish *)
```

Here, the first case matches a sequence beginning with dt, where we extract the content of the dt and take as many as possible of the following dds, using the longest match. Note that, without the longest match, it is ambiguous how many dds are taken by each of the consecutive patterns (d as Dd*) and rest. If we rewrite this pattern to an unambiguous one, the variable rest must be restricted not to match a sequence that begins with dd, resulting in a somewhat more cumbersome pattern:

```
dt[t], d as Dd*, rest as ((Dt,(Dd|Dt)*) | ())
```

Disadvantages of longest-match policy

We can find why longest-match policy has been dropped in the following excerpt by [Hos03].

> Consider the following type definition
>
> ```
> type Person = person[Name,Email*,Tel]
> ```
>
> we can write the following pattern match for extracting the tel node from a given person.
>
> ```
> match person with
>     person[Name, Email*, (Tel as x)]
>       -> ... (* do something with x *)
> ```
>
> This matches any values of type Person and binds x to the tel node. This pattern happens to have the structure as the input type, but it is not necessary in general. For example, we can replace the pattern Name,Email* in the above with the Any pattern, which matches any sequences.
>
> ```
> match person with
>     person[Any, (Tel as x)]
>       -> ...
> ```
>
> Using Any is convenient since it allows us to just ignore the parts that we do not care about. However, this idiom sometimes causes patterns to behave rather strangely in the first-match semantics. Let us consider a slight variation. Suppose that the type Person is actually defined as follows
>
> ```
> type Person = person[Name,Email*,Tel?]
> ```
>
> where the Tel part is now optional. Accordingly, the pattern for extracting this part would become:
>
> ```
> match person with
>     person[Any, (Tel? as x)]
>       -> ...
> ```
>
> However, in the first-match semantics, this pattern always binds x to the empty sequence. Understanding why is slightly complicated. First, the Any pattern tries to capture the longest possible sequence. This is because Any is actually recursively defined as
>
> ```
> type Any = ~[Any],Any | ()
> ```
>
> (where ~ matches any label and () is the empty sequence type) and the first choice is taken as often as possible. Likewise, the Tel? pattern also tries to capture the longest possible sequence since it is actually an abbreviation of
>
> ```
> Tel | ()
> ```

> and the first choice is taken first. Finally, when these two patterns are concatenated, the first one is given higher priority. Consequently, the Any pattern always captures the whole sequence, leaving the Tel? pattern only the empty sequence.
>
> In our new design, when a given pattern can parse an input value in multiple ways, then the system blames it as ambiguous. For example, the pattern in the last example is ambiguous since any person's content that ends with a tel can be parsed in two ways. To make it unambiguous, we need to be more explicit, e.g.,
>
> ```
> match person with
>     person[^tel[Any]*, (Tel? as x)]
>       -> ...
> ```
>
> (where ^tel matches any label except tel). The pattern is now slightly more verbose, but its meaning is clearer.

In `TODO` file authors commented to reconsider the ambiguity condition: "too strong right now"

2.4.3 Changes of Type Inference and of Pattern Constraints

Type Inference For Non-tail Position Variables

From version 0.2.0 type inference for patterns supports inference on non-tail pattern variables. Here's an excerpt with a description of this improvement (footnote at page 15 of [HP03])

> The power of the type inference scheme has been improved from the one described in our previous paper. In the previous one, we were able to infer precise types only for variables in tail positions. For example, in the pattern
>
> ```
> match v with
>   (val head as ~[Any]), val tail
> ```
>
> we could infer a precise type for tail but not for head (we simply extracted the type [Any] directly from the pattern, which is less precise). This was due to a naiveness of the inference algorithm that we used at that time. However, since then, we have developed a new algorithm that overcomes this limitation and have incorporated it in the current XDuce.

Top-level recursion forbidden

As of version 0.2.2 of XDuce top-level recursion has been completely forbidden (before it was allowed for tail variables). For example, we can no longer write

```
type X = a[],X | ()
```

or:

```
type X = a[],Y | ()
type Y = b[],X | c[]
```

2.4.4 Dropped subtagging

Subtagging is a feature to abstract over groups of labels. It has been dropped as of version 0.3.0 of XDuce, because it conflicts with the handling of attributes.

Hosoya describes subtagging in Section 2.5 of [HVP00]:

> In XML processing, we sometimes encounter situations where we have to handle a large number of labels and it is convenient to organize them in a hierarchy, in the style of object-oriented languages. This leads us to support a notion of "subtagging" in our type system, allowing subtyping between types with different labels. This feature goes beyond the expressive power of DTDs, but a similar mechanism called "substitution groups" can be found in XML-Schema. The subtagging relation is a reflexive and transitive relation on labels. We declare subtagging explicitly with a set of global subtag forms. For example, the following declares that the tags i (italic) and b (bold) are subtags of fontstyle:
>
> ```
> subtag i < : fontstyle subtag b < : fontstyle
> ```
>
> In the presence of these declarations, we have the subtype relations
>
> ```
> i[T] < : fontstyle[T] b[T] < : fontstyle[T]
> ```
>
> for all T. These relations allow us to collapse two case branches for i and b into one for fontstyle, when both branches behave the same. This use of subtagging is similar to the common technique in object-oriented programming of defining an abstract class fontstyle with subclasses i and b.

Chapter 3

XRel

This chapter describes a stand–alone tool I've written to study the algorithms of XDuce; I've called it XRel (XML Regular Expression Language).

XRel incorporates the two main features of XDuce — tree regular expressions types and tree regular expression pattern matching — to make pattern matching with XML documents.

XRel is a test–harness for the XDuce algorithms, and a porting of these algorithms from ML to Java. Note that it lacks recursive functions: the reason is that it's our intention to discard it as a stand-alone tool in favor of an integration with Hipi. Hipi is a concurrent language based upon π–calculus and developed as master thesis by Paolo Milazzo at the university of Bologna [Mil03]. Because Hipi has functions and all the other basic constructs of a programming language these features are not repeated in XRel.

Despite its minimality (it has not much more than a `typeswitch` statement[1] with which it can make pattern matching) all the algorithms of XDuce are still implemented: it has subtyping, type inference, static checks on types and the pattern matching algorithm.

In this chapter we describe the syntax of the language and discuss its functionalities.

3.1 Usage

XRel can be downloaded from my home page at the following address:

```
http://www.cs.unibo.it/~bisi/thesis
```

It can be run either with the .class files present in the bin directory or with the jar file XRel.jar in the main folder. All that it requires to work is a JVM version 1.3 or later and

[1] It's a lucky case that after we've chosen the name "typeswitch" for our pattern matching construct inspiring to Modula 3 we discovered that also XQuery designers made the same choice :-)

two jar files that are part of the Xerces[2] distribution: `xercesImpl.jar` and `xml-apis.jar`. The most convenient method to allow the program to locate these two files is to copy them in the main folder of the distribution so that from the main folder you can run XRel with the line:

```
java -cp bin;xercesImpl.jar;xml-apis.jar xrel.XRel ...
```

or simply with:

```
java -jar XRel.jar ...
```

Note that in this last case it's required that you copy `xml-apis.jar` and `xercesImpl.jar` in the current working directory, while in the first case you can specify the path of these files as you wish[3].

XRel can be run with one or two arguments: if only the name of a program is specified, it will be compiled and correctness checks will be applied to it. If both a program and an XML document are specified, the program will be compiled, statically verified and passed to the run-time pattern matching algorithm that will check the match of the document against the patterns of the program.

In the folder "examples" you can find a few ready examples. Note that XRel program files have the extension ".xre".

For instance to execute "`FirstExample.xre`" you need to write from the main folder either:

```
java -cp bin;xercesImpl.jar;xml-apis.jar xrel.XRel
examples\FirstExample.xre examples\mybook.xml
```

or simply:

```
java -jar XRel.jar examples\FirstExample.xre examples\mybook.xml
```

3.2 Our First Example

Let us see a simple example that will immediately reveal the syntax of the language. The following is adapted from the classical XDuce example where we need to convert an address book in a telephone book. However, since XRel lacks functions, we need to make it simpler. Suppose then we have an address book (actually a database of peoples whose records have a name, an optional telephone number and one or more e-mails) and that we want to extract the name and the telephone from the *first record* that contains a telephone number.

We expect to have in output an XML fragment like:

[2]Xerces is an XML parser written by the Apache Foundation. See [Fou] for more information

[3]for advanced users only: if you want to place these two files in some special place and then again use the jar file you can change the meta-inf\MANIFEST.MF inside XRel.jar to correctly locate them

```
<first><name>...</name><telephone>...</telephone></first>
```

or the string `"telephone not found"` if no record exists with a telephone number. Here's the code:

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

import addrbook[Person* as P];

typeswitch(P)
{
  // this pattern filters out the first record with a telephone
  case
    person[Name,Email+]*,
    person[name[String as firstName],
           tel[String as firstTel],
           Email+],
    Person* :
      printf(first[name[firstName],telephone[firstTel]]);
  // no existing telephones
  default: printf("no telephone found");
}
```

The first rows of the program are type declarations. Note how syntax of regular expression types is exactly the same as that of XDuce, while the syntax of the statement itself is slightly retouched to make it more C–like ("`typedef x = P;`" instead of "`type x = P`"). Comments follow C++/Java conventions.

The `import` statement declares the pattern against which to match the input value: when the program loads the XML document, it checks whether it matches with the pattern "`addrbook[Person* as P]`", i.e. if it belongs to the type denoted by the regular expression `addrbook[Person*]`. If so, the list of persons inside the root tag are bound to the variable `P` that is passed to the `typeswitch` construct (the equivalent of the `match` construct of XDuce); if not, the execution aborts.

The `typeswitch` construct above has two clauses: the first one handles documents with a telephone, the second one handles all the remaining cases (documents that don't have a telephone).

The pattern inside the first clause has the following structure: the first part `person[Name,Email+]*` skips all `person`s without any telephone, the second part `person[...]` selects the first per-

son that *has* a telephone and the third part `Person*` skips all remaining `person`s. When the central part of this pattern matches with a `person` it binds the content of the `name` tag with the variable `firstName` and the content of the `tel` tag with the variable `firstTel`.

3.3 Formal Definition

This section describes the syntax and the semantics of the language formally[4].

Each XRel program is a sequence of zero or more type declarations, followed by a mandatory `import` instruction, an optional `export` instruction and an optional `typeswitch` statement, in that order.

As in XDuce, type declarations are usually used both to describe the type of the XML document in input (its schema) or parts of it and parts of the output type to which expressions of the program need to conform. The `import` instruction performs a first match with the document in input; the pattern we put here should contain at least the bound variable to use with the `typeswitch`, if present. The `export` instruction gives a way to specify a "supertype" to which all the expressions of the program need to conform. Finally the `typeswitch` statement let us make complex pattern matching which produces different outputs on different alternatives.

Here's the syntax of the top–level of the language:

$$
\begin{aligned}
Program &::= TypeDeclaration* \ \ Import \ Export? \ TypeSwitch? \\
TypeDeclaration &::= \texttt{typedef}\ X = T; \\
Import &::= \texttt{import}\ P; \\
Export &::= \texttt{export}\ T; \\
\\
TypeSwitch &::= \texttt{typeswitch}\ (x)\ \{\ Clause*\ \ DefaultClause?\ \} \\
Clause &::= \texttt{case}\ P\texttt{:}\ CodeBlock \\
DefaultClause &::= \texttt{default}\texttt{:}\ CodeBlock \\
\\
CodeBlock &::= \ ; \\
&\quad\ \ Printf \\
&\quad\ \ \{\ Printf*\ \} \\
Printf &::= \texttt{printf}(e); \\
\end{aligned}
$$

where `X` ranges over type names, `T` ranges over type expressions, `P` ranges over patterns, `x` ranges over variables names and `e` ranges over expressions.

[4] see also the appendix A for a tool-generated grammar

Note that as discussed in the introduction of the chapter the language is reduced to a minimal core (for example we have only a printf statement in code blocks inside clauses).

Type names and program variables are identifiers and their syntax is defined as usual. In the next paragraphs we'll see the formal definitions for each of the other categories.

3.3.1 Types and Patterns

As for XDuce we first define patterns, and then define types as patterns containing no variable binder. Pattern expressions (or just *patterns*) are as follows:

$P ::= X$	type name
$\mid P$ `as` x	variable binder
$\mid$ `l` $[P]$	tag with label `l`
$\mid \sim[P]$	tag with any label
$\mid ()$	empty pattern
$\mid P \, , \, P$	concatenation
$\mid P \mid P$	union
$\mid P*$	repetition (zero or more)

Here X ranges over type names and x ranges over variable names.

Operators are as usual in regular expressions: "`P,P`" is concatenation, "`P|P`" is choice, "`P*`" is zero or more repetitions, "`P+`" is one or more repetitions and "`P?`" is zero or one occurrence. Additionally to the standard syntax for string regular expressions we have a syntax for tagged subtrees `l[P]` and a symbol "$\sim$" meaning *any label* which allows one to specify subtrees with generic label but known content. Our base type is the empty pattern "`()`". Finally we can bind a variable `x` to a fragment `P` of the pattern with the "`as`" operator, with "`P as x`" (note how we retain only the updated binding syntax of XDuce 2.4.1).

It's possible to use `l[]` or $\sim$`[]` as abbreviations of l[()] and $\sim$[()]. We can define the other regular expression operators as syntactic sugar:

```
P? = P | ()
P+ = P, P*
```

We introduce also two other base types with syntactic sugar: the "`String`" type and the literal strings, using pseudo–tags for them

```
String = #String[()]
"..." = #"..."[()]
```

The `String` is mainly used to code the `#PCDATA` sections of XML documents, while string literals let us to match against specific text contents. Note that the names for these

pseudo–labels are chosen in such a way that they don't clash with normal XML labels, as XML labels never begin with the "`#`" symbol.

We don't support numeric base types because we want to make our base types as simple as possible and also to avoid issues concerning the choice of an appropriate typecasting policy.

Furthermore we assume the following fixed type definition

```
typedef Any = ~[Any | String]*;
```

Finally define type expressions T (or just *types*) as the set of patterns which lack the "`as`" operator.

Well Formed Types

XRel types need to be non–recursive at top–level. We give the same formalization we've seen in the previous chapter for XDuce (2.2.3). We assume a mapping E from type names to theirs bodies. This means that when we have a declaration as the following:

```
typedef X = T;
```

we write E(`X`) to intend the right-hand side of the declaration (`T` in the example). For a given type `T`, we define the set S(T) of type names reachable from T at the top level as the smallest set satisfying the following

$$S(T) = \begin{cases} S(E(X)) \cup \{X\} & \text{if T=X} \\ S(T_1) & \text{if } T = T_1* \\ S(T_1) \cup S(T_2) & \text{if } T = T_1, T_2 \text{ or } T = T_1 \mid T_2 \\ \emptyset & \text{otherwise} \end{cases}$$

We then require that every type is not reachable at the top level from itself. Formally we require that the set E of type definitions satisfies

$$\texttt{X} \notin \texttt{S}(E(\texttt{X})) \qquad \forall\ \texttt{X} \in dom(\texttt{E})$$

Linear Patterns

Let `BV(P)` be the set of variables bound in the pattern `P`. For example in the following pattern P:

$$\texttt{l1[P}_1\ \texttt{as x], (l2[] as y | l3[P}_2\ \texttt{as y])}$$

`BV(P)` is {`x,y`}. We say that `P` is linear iff for any subphrase `P'` of `P` the following conditions hold:

$x \notin BV(P_1)$	if $P' = P_1$ `as` x
$BV(P_1) \cap BV(P_2) = \emptyset$	if $P' = P_1$, P_2
$BV(P_1) = BV(P_2)$	if $P' = P_1 \mid P_2$
$BV(P_1) = \emptyset$	if $P' = P_1*$

All the patterns in the program are required to be linear.

3.3.2 Expressions and Values

Parameters inside `printf` statements are expressions. Expressions e are defined as follows:

$e ::= l\ [e]$	tag with label l
$\mid e\ ,\ e$	sequence
$\mid x$	program variable
$\mid (\,)$	empty sequence
$\mid s$	double quoted literal string (e.g. “hello”)

For conciseness we write `l[]` as an abbreviation of `l[()]`.

Values v are expressions which contain no variables.

Values and XML documents

Our values roughly correspond to XML fragments without attributes. For example we can rewrite the following:

```
<person>
  <name>
    <first>Fabrizio</first>
    <last>Bisi</last>
  </name>
  <email>bisi@cs.unibo.it</email>
  <email>bisif@tin.it</email>
  <telephone/>
</person>
```

as an XRel value in the following way:

```
person[
  name[
    first["Fabrizio"],
    last["Bisi"]
  ],
  email["bisi@cs.unibo.it"],
  email["bisif@tin.it"],
  telephone[]
]
```

Note that we need an empty sequence in values not only to be able to specify empty documents but especially to specify empty subtrees that correspond to XML tag elements with an empty content (`<l></l>` or `<l/>`).

The correspondence between our values and XML documents is not exact as it appears. There are three reasons for this:

1. XRel doesn't handle XML attributes
2. an XRel value can be a sequence of values, whereas XML documents need to have a unique root
3. XRel values with adjacent string nodes don't have a correspondent in XML trees built from DOM parsers

To clarify the last point we show another example of XML code. While the example above plainly shows the tree nature of XML documents we must not forget that in XML it's possible to write things like the following:

```
<h1>Welcome to my home page</h1>
This page describes...
<h2>Interests</h2>
My hobbies...
```

where flat text and tag elements are mixed together so that it's not immediate to see a tree structure for the document. We can see this code as a tree if we consider adjacent parts of text as leaf nodes (so that we have actually two types of leaf nodes: empty nodes and string nodes). This is exactly what DOM parsers usually do[5].

So the piece of XML code above can be seen as the following XRel value:

[5] the Xerces DOM parser has exactly this behavior; however CDuce authors say that XML parsers do not usually guarantee that text nodes represent maximal textual portions of the documents, so that they removed the `String` type replacing it with `Char*` — see the related discussion in [BCF03]

```
h1["Welcome to my home page"],
"This page describes...",
h2["Interests"],
"My hobbies..."
```

While it's possible convert univocally each XML fragment without attributes into an XRel value, it's not clear to understand what is the correspondent XML fragment for XRel values that have adjacent string nodes. For example the correspondent XML code for the following value:

```
h1[...],"This page ","describes",h2[...],...
```

should be

```
<h1>...</h1>This page describes<h2>...</h2>...
```

but if we read again this XML with a DOM parser we do not get back our original value. While we can write sequences with adjacent String nodes a DOM parser will never produce a such tree. This means that if we try to match a value against a such pattern the match will always fail.

Label Classes

Whether include label classes or not was a design issue for XRel. [HP03] argue that label classes are useful for the following reasons:

1. differences and unions of labels are useful
2. in the algorithm that computes difference between automata difference between labels is however needed whereas union between labels can be inserted for optimization purposes (in difference algorithm I can merge `L1[A]` with `L2[A]` in `(L1|L2)[A]`)

However we made the choice to omit them to try to retain the language design as simple as possible.

3.3.3 Type Checking

In this section I give the definitions and the semantics for static type–checks applied to the patterns. Note that, with the exception of the weak ambiguity check, they are the same of XDuce (see sections 5.4 and 5.5 in [HP03]).

Definitions of exhaustiveness, irredundancy, non-ambiguity, and type inference for pattern-match expressions are all made with respect to an "input type" T describing the set of values that may be presented to the expression at run time.

Typechecks on the `typeswitch` construct

Patterns inside `typeswitch` construct are required to be *exhaustive* and *irredundant* with respect to the input type T. Let us give definitions for exhaustiveness and irredundancy.

Exhaustiveness A list $\texttt{P}_1$ `|` ... `|` $\texttt{P}_n$ of patterns is *exhaustive* with respect to T, iff, for all v ∈ T implies v ∈ P_i ⇒ V for some P_i and V.

Irredundancy A list $\texttt{P}_1$ `|` ... `|` $\texttt{P}_n$ of patterns is *irredundant* with respect to T, iff, for all P_i, there is a value v ∈ T such that v ∉ P_j for 1 ≤ j ≤ i-1 and v ∈ P_i ⇒ V for some V.

Furthermore, the type of each expression is checked to be a subtype of the declared output type and all patterns are checked to be weakly unambiguous. We explain in the next sections.

Expressions' type correctness

If an XRel program includes an `export` clause, all the expressions e are checked to be correctly typed, i.e. we check that $T_e \subseteq T_{out}$, where $e \in T_e$ and T_{out} is the declared output type. To evaluate T_e we use type inference for variables. Let us define semantics for type inference and subtyping.

Type inference The goal of pattern type inference is to "compute" the range of a pattern, defined as follows. A type environment Γ describes the range of a pattern P with respect to type T iff, for all variables x and values v, we have

> v ∈ Γ(x) iff there exists a value u ∈ T such that u ∈ P ⇒ V for some V with V(x) = v

Subtyping A type T_1 is a subtype of another type T_2, written T_1 `<:` T_2, iff $v \in T_1$ implies $v \in T_2$ for all v.

Weak unambiguity

All the patterns in a XRel program are required to be *weakly* unambiguous.

Hosoya uses a definition of non-ambiguity similar to *strong* non-ambiguity for string regular expressions [SSS88]; differences are that XDuce treats sequences of trees rather than strings, and that non-ambiguity is considered for a given restricted set of input values rather than for all input values (section 5.4.6 in [HP03]).

However I've found a simpler algorithm that checks *weak* non-ambiguity instead. Notions of weak and strong non-ambiguity for regular expressions are formally defined in [BK91] and

in [SSS88]. [BK91] gives the following definition of weak ambiguity for automata and regular expressions(definition 4.1, page 13):

1. an ϵ-NFA M is unambiguous if for each value v there is at most one path from the initial state to the final state that recognizes v

2. a regular expression T is weakly unambiguous if and only if the NFA M_T is unambiguous

where ϵ-NFAs are NFAs with ϵ-transitions that are built with a procedure that preserves strong ambiguity (as that Hosoya uses).

So a regular expression T is *weakly unambiguous* if each value v can be traced uniquely with a path through T, whereas T is *strongly unambiguous* if each value can be uniquely decomposed into subvalues according to the syntactic structure of T [SSS88] (refer to 5.4.6 page 19 in [HP03] for an operational semantics for *strong* unambiguity).

To give an example, the expression `(a*|b*)*` is trivially weakly unambiguous because each symbol occurs only once. Thus, any symbol in a value can be matched by exactly one position in the expression. In contrast, the value `a,a` is denoted by `(a*|b*)*` as a single application of the outer star and a twofold application of the inner one, or, alternatively, as a twofold application of the outer star and two single applications of the inner one. Thus, `(a*|b*)*` is not strongly unambiguous.

Bruggemann and Klein [BK91] also give a mathematical framework and discuss the relation between the two notions of unambiguity; in this framework, they give an inductive definition of weak unambiguity (lemma 4.7, page 14).

My intuition for why we only need weak ambiguity, and not strong ambiguity, is that our variable binders need to be linked to the label, so it doesn't matter whether `(a*)*` is ambiguous because anyway we can't bind `a` for linearity constraints; instead it's important that `a,b | a,b` is ambiguous — consider `(a as x,b as y) | (a as y, b as x)`.

I discuss my algorithm in section 4.3.3. I talked with Hosoya by e–mail and he said that although he doesn't have a proof, he can believe that my algorithm checks weak-ambiguity. Also he doesn't know whether strong ambiguity is better than weak one or not in practice, but he has an impression that weak one is better.

Hosoya cites in section 7 of [Hos03] an algorithm developed by Kawaguchi that detects weak ambiguity for tree grammars [Koh]; in this paper he says that he believes that it's possible to obtain an algorithm for checking weak ambiguity by a slight modification of its algorithm.

Chapter 4

Implementation

In this chapter we analyze the algorithms of XRel. Remember that XRel has a static (compile-time) phase and a dynamic (run-time) phase.

In its static phase it loads a program, builds a syntax tree for it, checks for syntactic errors, adds predefined types and finally calls the semantic analyzer. The semantic analyzer checks that there are no type errors and produces in output a set of automata, each corresponding to a pattern of the program.

The dynamic phase (if present) loads the specified XML document and converts the tree produced by the DOM parser into an internal tree (i.e. into the internal representation of an XRel value), then it calls the pattern matching algorithm and generates the output.

We omit to describe third–party code (the program parser generated by a JavaCC / JJTree grammar and the DOM parser written by Apache Foundation) and accessory code (like the syntax error checks) and we focus instead on the more interesting parts of the program: the semantic analyzer and the run-time pattern matching algorithm.

The semantic analyzer does the followings:

1. it checks that types are well formed
2. it checks that patterns are linear
3. it converts the types and the patterns of the language to an internal normalized format
4. it builds automata
5. finally it performs the other type error checks, i.e. the ambiguity check on the `import` pattern and the checks on the `typeswitch` statement (exhaustivity of the clauses, unambiguity and irredundancy of each clause, adherence of the expressions to the declared output type).

4.1 Symbol Table

The code that builds the syntax tree also fills the symbol table with all the information it collects from the input source code. This data structure will be continuously updated in each phase of the static part of the program and then used in its dynamic part.

When the parser finishes its run, the symbol table contains four kinds of information: the list of all the types declared in the program (comprehensive of the output type), a list of all the patterns (the import pattern and the clauses inside the `typeswitch` construct), a list of all the variable names together with the declared type and theirs scope, and a list of all the expressions inside `printf` statements along with theirs scope.

These four lists are implemented with `Vector`s of `SymElement` (symbol table elements) objects. `SymElement` is a general purpose class that can storage information for a type, a pattern, a variable or a program expression. For example it has the field `name` to store the name for variables and types[1] and an integer field `scope` to record scoping information for variables and expressions (the scope is 0 for variables declared inside the import statement, the number of the clause starting from 1 for variables and expressions inside the `typeswitch` statement). `SymElement` objects of whichever kind use the field `node` to refer to the root of the subtree in the abstract syntax tree that represents the related expression (type expressions for types and variables, pattern expressions for patterns, the expressions themselves for program expressions). The class also has an integer field `idCode` that uniquely identifies objects inside their category; as we'll see below, this number is used to build the internal names for patterns and expressions.

For this document, let us formalize these four vectors with the following mappings E,F,G,H

$E : X \to T$	types
$F : Y \to P$	patterns
$G : V \to T$	declared variable types
$H : W \to e$	expressions

where `X` ranges over type names, `T` ranges over type expressions, `Y` ranges over (internal) pattern names, `P` ranges over pattern expressions, `V` ranges over variable names, `W` ranges over internal expression names and `e` ranges over program expressions.

Internal pattern names are generated by concatenating an underscore with the `idCode` (e.g. `_0` is always the import pattern), while expressions are generated by concatenating two underscores with the `idCode` (`__0`, `__1`, ...). In the real code, internal names are also generated to identify variables, as a variable name is unique only inside its scope while the `idCode` identifies univocally the variable in each point of the program. For the sake of

[1] Actually this field is used also for patterns and expressions to store the internal generated name, as explained below

clarity, in this chapter we'll use the source code names to identify variables assuming that there are no conflicts.

Consider the following input example:

```
typedef T1 = name1[String],name2[String];
typedef T2 = name[String]+;

import root[ (T1 | T2) as x ];

typeswitch(x)
{
  // this clause matches values of type T1
  case name1[String as n1],name2[String as n2]:
    { printf(first[n1]); printf(last[n2]); }
  // this clause matches values of type T2
  // when there are at least two names
  case name[String as fst],name[String]*,name[String as lst]:
    printf(first[fst], last[lst]);
  // one only name case
  case name[String as nm]:
    printf(first[nm], last[nm]);
}
```

the parser generates the following symbol table

```
E(T1) = name1[String],name2[String]
E(T2) = name[String]+

F(_0) = root[ (T1 | T2) as x ]
F(_1) = name1[String as n1],name2[String as n2]
F(_2) = name[String as fst],name[String]*,name[String as lst]
F(_3) = name[String as nm]

G(x)   = (T1 | T2)
G(n1)  = String
G(n2)  = String
G(fst) = String
G(lst) = String
G(nm)  = String

H(__0) = first[n1]
H(__1) = last[n2]
H(__2) = first[fst],first[lst]
H(__3) = first[nm],last[nm]
```

Expressions to the right–side of the equations are actually *trees* whose nodes can belong to one of the following classes: ASTComma, ASTPar (`|`), ASTIdentifier (variables and type names), ASTEmpty (empty sequence), ASTAs (binders), ASTOperator, ASTString, ASTStringLiteral (`"..."`) and ASTTag. All these classes implement the interface *Node* that has the usual fields for tree nodes: the field `parent` points to the parent node, `children` is an array with pointers to children, etc. More, some classes add specific fields: ASTIdentifier and ASTTag have a `name` field (the latter one uses it to store labels), ASTAs has a `varName` field to store the name of the variable to bind, ASTOperator has an `operator` field which can assume only the values "`*`", "`+`" or "`?`", ASTStringLiteral has a `value` field, and so on.

As an example of tree, recall that `T1` is `name1[String],name2[String]`. Hence `E(T1)` is a tree with a ASTComma root that has 2 children, both of type ASTTag; the tags have one child, each of type ASTString.

4.1.1 Well–Formedness Check

Remember that we require that every type is not recursive at the top level. In section 3.3.1 we defined "reachability at the top–level" and we asserted that a type is well–formed if it's not reachable at top–level by its expansion.

To check that a given type with name X is not reachable at the top-level from inside a type expression T we define a "*disconnectedness*" judgment of the form "$\sigma \vdash T : \mathtt{dc}(X)$", where σ is a set of variables. It should be read "T is disconnected from X (i.e. X does not occur in the top level of T), assuming that all bodies of variables in σ are disconnected from X". This judgment is defined by the following rules (where $\mathrm{X} \neq \mathrm{X}_1$)[2]:

$$
\begin{array}{ll}
\sigma \vdash T :\ dc(X) & \text{for } T = () \text{ or } \mathrm{l}[T'] \text{ or base type} \\
\sigma \vdash X_1 :\ dc(X) & \text{if} \quad X_1 \in \sigma \\
\sigma \vdash X_1 :\ dc(X) & \text{if} \quad X_1 \notin \sigma \quad \text{and} \quad \sigma \cup \{X\} \vdash E(X_1) : dc(X) \\
\sigma \vdash T_1 \mid T_2 :\ dc(X) & \text{if} \quad \sigma \vdash T_1 : dc(X) \quad \text{and} \quad \sigma \vdash T_2 : dc(X) \\
\sigma \vdash T_1, T_2 :\ dc(X) & \text{if} \quad \sigma \vdash T_1 : dc(X) \quad \text{and} \quad \sigma \vdash T_2 : dc(X) \\
\sigma \vdash T\star :\ dc(X) & \text{if} \quad \sigma \vdash T : dc(X) \quad \text{where } \star \in \{*, ?, +\}
\end{array}
$$

We now explain this definition. The empty sequence, a label or a base type (`String` or a string literal) are disconnected from X. When there is a concatenation we check that all concatenated subtypes are disconnected from X; analogously for choice. Note that rules for concatenation and choice are defined in a binary mode for conciseness: in XRel the concatenation and choice nodes have an arbitrary number of children. $T*$, $T?$ and $T+$ are disconnected from X when T is disconnected from X.

[2]This formalization is given by Hosoya and the other XDuce authors in [HVP00] to describe a subroutine of the old right–linearity check. This old subroutine applied to the types of the program is all we need to check the non-reachability at top–level

To ensure termination, we keep track in σ of variables that have already been found to be disconnected. For any type name $X_1 \neq X$ we encounter, if $X_1 \in \sigma$ we immediately return true, otherwise we recursively check $E(X_1)$. If we encounter X none of the rules above applies, so that we can assert that X is not disconnected from itself.

Now the set of type definitions E is said to be well–formed if each type X is disconnected from its body expansion E(X), i.e. if

$$\emptyset \vdash E(X) : dc(X) \qquad \forall X \in dom(E)$$

A function in Java pseudo–code for the judgment above is the following:

```
boolean dc(Symtab symtab, HashSet sigma, Node T, String X) {
  switch (T) {
    case T1| ..| Tn:
    case T1, .., Tn:
      // T is disconnected from X iff T1,..,Tn are disconnected from X
      return dc(symtab,sigma,T1,X) && .. && dc(symtab,sigma,Tn,X);
    case T'*,T'+,T'?:
      return dc(symtab,sigma,T',X);
    case (),l[T'],literal,String:
      return true;         // base case 1 (success)
    case X1: // type name
      if (X1 == X)
        return false;      // base case 2 (fails!)
      else if (X1 is in sigma)
        // X1 has already been found disconnected
        return true;
      else
        return dc(symtab,sigma + {X1},symtab.E(X1),X);
  }
}
```

It takes four arguments:

1. the symbol table `symtab` to access to the list E of type names in the program (used for the case $X_1 \neq X$)

2. the set of variables `sigma` (σ) already checked (HashSet is the Java built-in class for sets)

3. the root `T` of the (sub)type expression to recursively check

4. the type name `X` to check

and returns true if `X` is disconnected from `T` (false otherwise).

The function is invoked by the semantic analyzer on each type X of the program with the following call:

$$\texttt{dc}(\text{symtab},\varnothing,\text{E(X)},\text{X});$$

4.1.2 Linear Patterns

Recall from discussion in 3.3.1 that we require all the patterns in the program to be linear, and that a pattern `P` is defined to be linear if for every subphrase `P'` the following conditions hold:

$$\begin{array}{ll} x \notin BV(P_1) & \text{if } P' = P_1 \texttt{ as } x \\ BV(P_1) \cap BV(P_2) = \emptyset & \text{if } P' = P_1 \texttt{ , } P_2 \\ BV(P_1) = BV(P_2) & \text{if } P' = P_1 \texttt{ | } P_2 \\ BV(P_1) = \emptyset & \text{if } P' = P_1 * \end{array}$$

These rules show us how to build a recursive function *bv* that takes in input a pattern expression and returns the set of bound variables inside it.

```
HashSet bv(Node P) throws LinBrokenException {
  switch (P) {
    case (),literal,String,Y:
      return ∅; // base case
    case l[P']:
      return bv(P'); // recurse on P'
    case P1,...,Pn:
      // P is linear if P1,...,Pn are linear and
      // bv1=bv(P1),...,bvn=bv(Pn) are disjoint
      // S1 ⊎ S2 throws a LinBrokenException if
      // S1 ∩ S2 ≠ ∅ (if sets are not disjoint)
      return bv(P1) ⊎ ... ⊎ bv(Pn);
    case P1|...|Pn:
      // P is linear if P1,...,Pn are linear and
      // bv1=bv(P1),...,bvn=bv(Pn) are equal
      HashSet bv1=bv(P1); ...; HashSet bvn=bv(Pn);
      if (!(bv1==...==bvn)) throw new LinBrokenException();
      return bv1;
    case P'*,P'+,P'?:
      if (bv(P') ≠ ∅) throw new LinBrokenException();
      return ∅;
```

```
    case P' as x:
      return bv(P') ⊎ {x};
  }
}
```

Note that we used a subroutine $\uplus$ (called `unionDisjoint` in the implementation) that merges two sets while checking that they are disjoint (it throws a LinBrokenException if they are not).

We can use the function bv above to build a predicate $isLinear(P)$ that says if a pattern is linear in the following way

```
boolean isLinear(Node P) {
  try {
    bv(P);
  }
  catch (LinBrokenException e) {
    return false;
  }
  return true;
}
```

4.1.3 Translation to Internal Patterns

We obtain internal patterns from the types and the patterns of the surface language by removing the syntactic sugar from all the types and patterns in the symbol table, then following Hosoya [Hos03] we reduce the resulting patterns to a simplified normal form used for the conversion to automata. Henceforth we talk of patterns referring both to the types and to the patterns of the program (remember that the type expressions are a subset of the pattern expressions).

Removing syntactic sugar from a pattern expression means to perform the following simple transformations to it:

- convert `String` into the pseudo-tag `#String[]` and convert every string literal "literal" into the pseudo-tag `#"literal"[]` (these names are guaranteed never to clash with other labels).

- remove the operators "`+`" and "`?`" with the following substitutions:

  - `P+ = P,P*`
  - `P? = P|()`

This desugaring is straightforward — indeed the normalization function `ts` below can easily be modified to do also the desugaring work — so I omit it for conciseness[3]. "Desugared" patterns as used in XRel source code were defined previously in section 3.3.1 as

$$P \ ::= \ X \ \mid \ P \ \texttt{as} \ x \ \mid \ \ell[P] \ \mid \ \sim[P] \ \mid \ () \ \mid \ P, P \ \mid \ P|P \ \mid \ P*$$

The simplified "normal form" has the following syntax

$$Q \ ::= \ Q \ \texttt{as} \ x \ \mid \ \ell[Y] \ \mid \ \sim[Y] \ \mid \ () \ \mid \ Q, Q \ \mid \ Q|Q \ \mid \ Q*$$

where Y ranges both over pattern names and type names. Note that direct nesting of labels is forbidden and that pattern names are required to occur only inside labels.

Hosoya describes the arbitrary patterns in [HP03] but only defines translation from the normal forms to automata [Hos03]: he never describes formally translation from arbitrary patterns to the normal form above[4]. Below we describe our conversion algorithm, whose work is to

1. eliminate top-level type names by replacing them with their related expressions (e.g. `l[],X` where E(X) = `m[]` becomes `l[],m[]`)

2. convert into a reference every pattern expression inside a tag (e.g. `l[m[],n[]]` becomes `m[Y]` with F`(Y) = m[],n[]`)

Note that transformations at point 2 can create many new patterns so we need to be careful that the algorithm terminates. In the next section we discuss this problem.

Algorithm

We give the pseudo–code for the algorithm for conversion to normal form. The main code calls a function *ts* for all type expression `E(X)` and for all pattern expression `F(Y)` in the source. The function *ts* takes in input the symbol table and a tree that represents the pattern, and returns the normalized tree; during its execution, it can possibly add new patterns to the vector `F` in the symbol table .

```
Node ts(SymTab symtab, Node P) {
  switch (P) {
    case P1| ..| Pn:
      // Recursion on subexpressions
      return ts(P1) |...| ts(Pn);
```

[3]In the implementation I've separated the desugaring phase from the normalization phase for debug purposes

[4]In its thesis [Hos01] and in papers [HP02, HVP00] he gives a translation from external to internal patterns, but the algorithm described in these old works is out by date, as the old internal patterns are very different by the current ones

```
    case P1, .., Pn:
      // Recursion on subexpressions
      return ts(P1), ..., ts(Pn);
    case ():
      return ();                // nothing to do
    case P' as x:
      return ts(P') as x;       // recursion on P'
    case P'*:
      return ts(P')*;           // recursion on P'
    case l[P']:
      if (P' is a type name) {
        return l[P'];           // it's already normal
      }
      else {
        // create new pattern Y' in symtab.F
        symtab.F.append(Y' -> P');
        // replace P' with Y' inside l
        return l[Y'];
      }
    case X: // P is a type name
      return ts( symtab.E(X) );  // recursion on type expansion
  }
}
```

Note that both subtypes in E and proper subpatterns in F are added to the vector F as new patterns. For example a type $E(\mathrm{X}) =$ `a[b[]]` is normalized as

```
E(X) = a[_1]
F(_1) = b[]
```

Note also that new generated patterns are not immediately normalized by ts. For instance, a type $E(\mathrm{X}) =$ `a[b[],Y,...]` is normalized by the function above as

```
E(X) = a[_1]
F(_1) = b[],Y,...
```

However, because they're appended at the tail of the list of patterns of the program (`symtab.F` in the code), they're normalized afterwards when they're reached by the `while` loop that calls `ts` on each element of the list.

For proof of termination, `ts` always terminates: each case decreases the size of its input, except for the final case; but it is only called finitely many times due to disconnectedness (Section 4.1.1). As for the while loop in the main code that call `ts`, key to its termination is the fact that children $Y' \to P'$ are *appended* to the list `symtab.F`. We illustrate with two examples.

Consider this first example:

```
typedef A = a[b,A,c] | ();
```

When `ts` is applied on the body expression of A, it creates a new pattern for the content of the tag labelled `a`, giving:

```
E(A) = a[_1] | ()
F(_1) = b,A,c
```

Afterwards, the main code proceeds by applying the `ts` function to the next pattern, that is `_1`; `ts` replaces `A` with its expression obtaining:

```
E(A) = a[_1] | ()
F(_1) = b, (a[_1]|()), c
```

Supposing there are no other patterns, the procedure terminates because all patterns in the program (`A` and `_1`) are normalized.

To see why we need to normalize the freshly created subpatterns *after* the original patterns, let us show a more complex example:

```
typedef X = l[b,Y,c];
typedef Y = m[d,Y,e] | ();
```

Suppose that `ts(E(X))` is invoked first. The content of the tag labelled `l` is put in a new subpattern `_1`

```
E(X) = l[_1]
E(Y) = m[d,Y,e] | ()
F(_1) = b,Y,c
```

after which `ts(E(Y))` (and *not* `ts(F(_1))`) is called. The normalization of `Y` leads to the following:

```
E(X) = l[_1]
E(Y) = m[_2] | ()
F(_1) = b,Y,c
F(_2) = d,Y,e
```

Note that it's not important the order with which `ts(F(_1))` and `ts(F(_2))` are applied; we always arrive at the following situation:

```
E(X) = l[_1]
E(Y) = m[_2] | ()
F(_1) = b, (m[_2]|()), c
F(_2) = d, (m[_2]|()), e
```

that is final, because all patterns in the program are normalized.

If we don't process the new patterns after the original ones we could have infinite loops. For instance in the situation above

```
E(X) = l[_1]
E(Y) = m[d,Y,e] | ()
F(_1) = b,Y,c
```

if `ts(F(_1))` is applied before `ts(E(X))`, we have

```
E(X) = l[_1]
E(Y) = m[d,Y,e] | ()
F(_1) = b,m[d,m[d,m[d,...
```

where the algorithm gets stuck on the infinite loop because `Y` is infinitely replaced with its expression.

4.2 Tree Automata

My automata are almost identical to those Hosoya defines in [Hos03]. The only differences are

- I've limited label classes to only basic cases (for automata derived from patterns there are only single labels ℓ and the *any* label ˜)
- I chose to represent tag contents with subautomata, applying to automata an idea that author gives in his newest papers for grammars [HM02a, HM02b][5]

The aim of the normalization given in the previous section was to prepare patterns for translation into automata. For conciseness in formalization we suppose from now to have copied all the types in E into F. Recall that an internal pattern is a regular expression whose atoms are of the form `()`, ℓ`[Y]` or ˜`[Y]` (below I refer to () with ϵ and to the two last cases collectively as L`[Y]`), where `Y` is a pattern name or a type name. From each internal pattern, we can construct an automaton whose transition labels are also of the form ϵ or L`[Y]`.

Formally, a tree automaton A is a tuple (Q, Q^{init}, Q^{fin}, δ) where

- Q is a finite set of *states*
- $\mathrm{Q}^{init} \subseteq$ Q is a set of *initial states*
- $\mathrm{Q}^{fin} \subseteq$ Q is a set of *final states*
- δ is a set of *transition rules* of the form $\mathrm{q} \xrightarrow{\mathcal{S}:\, L[X]} \mathrm{q}$ or $\mathrm{q} \xrightarrow{\epsilon} \mathrm{q}$

[5] This seems merely to be difference in formalization — looking at the code of XDuce 0.4.0 it seems that Hosoya also uses subautomata to represent subtrees

We also write these components as $\mathbf{st}(\mathrm{A})$, $\mathbf{init}(\mathrm{A})$, $\mathbf{fin}(\mathrm{A})$ and $\mathbf{tr}(\mathrm{A})$, respectively. These automata are different from usual ones [CDG+97] in that each transition is associated with a set of variables $\mathcal{S}$. We omit "$\mathcal{S}$:" from a transition rule when $\mathcal{S}$ does not matter.

The set of variables $\mathcal{S}$ associated to a labelled transition represent the variables to "update" when the transition is followed during pattern matching. For each variable there is a "sequence accumulator" that collects the elements that match with that variable in the input value; while the automaton runs, if a transition $q_1 \xrightarrow{\{x\}:\ell[Y]} q_2$ is followed, the element $\ell[Y]$ is added to the accumulator of the variable x[6]. When the execution stops because it has successfully found a path that matches the input value, all sequence accumulators contain the values to bind with their respective variables.

To each transition is associated a set of variables (rather than a single variable) since variable binders can be nested. For example the automaton in fig. 4.1 represents the pattern `(a[], (b[]* as y)) as x`.

Figure 4.1: Automaton for "`(a[], (b[]* as y)) as x`" ($Y_{()}$ is a pattern that matches with the empty sequence)

4.2.1 Building Automata

Following Hosoya [Hos03] we use the standard Thompson's algorithm adapted to build tree automata with sequence accumulators.

Given a pattern P we create an automaton $A_P = A_{P,\emptyset}$, where $A_{P,\mathcal{S}} = (Q_{P,\mathcal{S}}, Q^{init}_{P,\mathcal{S}}, Q^{fin}_{P,\mathcal{S}}, \delta_{P,\mathcal{S}})$ is defined inductively on the structure of P:

$$
\begin{aligned}
A_{(),\mathcal{S}} &= (\{q_1\}, \quad \{q_1\}, \quad \{q_1\}, \quad \emptyset) \\
A_{L[Y],\mathcal{S}} &= (\{q_1, q_2\}, \quad \{q_1\}, \quad \{q_2\}, \quad \{q_1 \xrightarrow{\mathcal{S}:\, L[Y]} q_2\}) \\
A_{(P_1,P_2),\mathcal{S}} &= (Q_{P_1,\mathcal{S}} \cup Q_{P_2,\mathcal{S}}, \quad Q^{init}_{P_1,\mathcal{S}}, \quad Q^{fin}_{P_2,\mathcal{S}}, \quad \delta_{P_1,\mathcal{S}} \cup \delta_{P_2,\mathcal{S}} \cup (Q^{fin}_{P_1,\mathcal{S}} \xrightarrow{\epsilon} Q^{init}_{P_2,\mathcal{S}})) \\
A_{(P_1|P_2),\mathcal{S}} &= (Q_{P_1,\mathcal{S}} \cup Q_{P_2,\mathcal{S}}, \quad Q^{init}_{P_1,\mathcal{S}} \cup Q^{init}_{P_2,\mathcal{S}}, \quad Q^{fin}_{P_1,\mathcal{S}} \cup Q^{fin}_{P_2,\mathcal{S}}, \quad \delta_{P_1,\mathcal{S}} \cup \delta_{P_2,\mathcal{S}}) \\
A_{P*,\mathcal{S}} &= (Q_{P,\mathcal{S}} \uplus \{q_1, q_2\}, \quad \{q_1\}, \quad \{q_2\}, \\
&\qquad \delta_{P,\mathcal{S}} \cup ((Q^{fin}_{P,\mathcal{S}} \cup \{q_1\}) \xrightarrow{\epsilon} (Q^{init}_{P,\mathcal{S}} \cup \{q_2\}))) \\
A_{(P \texttt{ as } x),\mathcal{S}} &= A_{P,\mathcal{S}\cup\{x\}}
\end{aligned}
$$

[6] Following Hosoya we chose to implement a simpler algorithm in two phases, see 4.2.2

The notation $Q_1 \xrightarrow{l} Q_2$ means $\{q_1 \xrightarrow{l} q_2 \mid q_1 \in Q_1, q_2 \in Q_2\}$ (l is either "$\mathcal{S} : L[Y]$" or ϵ).

In this phase the semantic analyzer invokes the function `buildTA` on all the types and the patterns of the program; this implements the algorithm above, adding to the symbol table a new mapping $M : Y \to A$ from pattern names to the built automata. I omit pseudo–code for this function because it's long and not very interesting (it recursively builds and merges subautomata by following the rules described in the formalization above).

Again, note that there are two differences with the algorithm that Hosoya describes in appendix A of [Hos03]:

- my label classes L lack complex cases ($\mathtt{L}_1 \mid \mathtt{L}_2$ and $\mathtt{L}_1 \backslash \mathtt{L}_2$)

- the automata formalization in [Hos03] creates a big automaton $A^{\mathbf{all}}_{P_0,\emptyset}$ that includes also the subautomata corresponding to the subpatterns inside the tags of P_0. Here I only build the main automaton corresponding to the top-level of the given pattern P_0 and then I write inside labelled transitions pattern names instead of states as Hosoya does — the appropriate subautomaton can be correctly localized during pattern matching with the M mapping explained above

To complete the building of an automaton we use two further steps:

1. the removal of ϵ-transitions

2. the removal of useless states, i.e. states that are not reachable from the initial states and that are not matched by any value

Removal of ϵ-Transitions

Differently by Hosoya, I always remove ϵ-transitions from the automata; he has instead to retain them in the ambiguity check algorithm, because ϵ-transition operation doesn't preserve strong ambiguity (see section 4.3.3).

I've used the same algorithm as Hosoya, that is actually the standard algorithm used also for string automata (see [HU97]).

Following Hosoya[7] we first define the function $\epsilon-\mathbf{clos}$ to compute the set of states reachable by ϵ-transitions from a given state:

$$\epsilon-\mathbf{clos}_A(q_1) = \{q_2 \mid q_1 \xrightarrow{\epsilon}{}^{*} q_2 \in \mathbf{tr}(A)\}$$

($q_1 \xrightarrow{\epsilon}{}^{*} q_n \in \mathbf{tr}(A)$ means $q_i \xrightarrow{\epsilon} q_{i+1} \in \mathbf{tr}(A) \quad \forall i = 1, \ldots, n-1, n \geq 1$). Then we define

[7] material here is replicated in section 4.3 of [Hos03]

B = $\epsilon-\mathbf{elim}(A)$ as follows.

$$\begin{aligned}
\mathbf{st}(B) &= \mathbf{st}(A) \\
\mathbf{init}(B) &= \mathbf{init}(A) \\
\mathbf{fin}(B) &= \{q_1 \in \mathbf{st}(A) \mid \epsilon-\mathbf{clos}_A(q_1) \cap \mathbf{fin}(A) \neq \emptyset\} \\
\mathbf{tr}(B) &= \left\{ q_1 \xrightarrow{\mathcal{S}:l[Y]} q_2 \;\middle|\; q_3 \in \epsilon-\mathbf{clos}_A(q_1), \quad q_3 \xrightarrow{\mathcal{S}:l[Y]} q_2 \in \mathbf{tr}(A) \right\}
\end{aligned}$$

In practice to eliminate the ϵ-transitions from an automaton we have to

1. add each state that has a final state in its epsilon closure to the set of final states
2. add to each state the exiting transitions that exit from all the states in its epsilon closure
3. lastly remove ϵ-transitions from the automaton

For instance given the automaton in fig. 4.2 representing the pattern "`a[],b[]*`", $\epsilon-\mathbf{elim}$ transforms it in the automaton in fig. 4.3.

Figure 4.2: Automaton for `a[],b[]*` with ϵ-transitions

Figure 4.3: Automaton for `a[],b[]*` without ϵ-transitions

Remove Useless States

This operation removes, from a given automaton, all the states that are not reachable from the initial states and those that are not matched by any value. Following Hosoya[8], we define

[8]This definition is taken by Section 4.2 in [Hos03]

$\mathbf{reach}_A(q)$ as the set of states reachable from q and $\mathrm{B} = \emptyset{-}\mathbf{elim}(A)$ as follows.

$$\begin{aligned}
\mathbf{st}(B) &= \{q \mid q \in \mathbf{reach}_A(\mathbf{init}(A)), \quad \exists v.\ A \vdash v \in q\} \\
\mathbf{init}(B) &= \mathbf{init}(A) \cap \mathbf{st}(B) \\
\mathbf{fin}(B) &= \mathbf{fin}(A) \cap \mathbf{st}(B) \\
\mathbf{tr}(B) &= \left\{ q_1 \xrightarrow{\mathcal{S}:l[Y]} q_2 \mid q_1, q_2 \in \mathbf{st}(B) \right\}
\end{aligned}$$

where $A \vdash v \in q$ is the pattern matching relation we'll see in section 4.2.2 (intuitively it means that there exists a path starting from q in A that recognizes v).

The definition above gives no operative algorithm; however we can use the well–known algorithms on string automata. Actually, there are two distinct algorithms in [HU97]: the former removes unreachable states (states that could possibly match a value but that can't be reached from the initial states), the latter removes "unmatchable" states (states for which there doesn't exist any starting path that will recognize a value, in practice states for which there doesn't exist a path that arrives in a final state).

To remove unreachable states we use the following:

1. assign the initial states to the set of *reachable states*
2. add to the set of reachable states all the states reachable in a step that are not yet in the set
3. repeat step 2 until there are no more states to add to the set
4. discard the remaining states and update the set of final states; discard transitions from/to deleted states states

To remove "unmatchable" states we use the following:

1. assign the final states to the set of *matchable states*
2. add to the set of matchable states all the states that go in a matchable state in a step and that are not yet in the set
3. repeat step 2 until there are no more states to add to the set
4. discard the remaining states and update the set of initial states; discard transitions from/to deleted states

Both these algorithms work on automata without ϵ-transitions. Note that they are similar: while the former proceeds forward from initial states searching for final states, the latter proceeds backward from final states searching for initial states: both algorithms terminate when a "fixed point" is reached, i.e when it's not possible to add further new

states to the set of processed states. Finally both algorithms delete unprocessed states and transitions that begin from / end to a deleted state.

After removing useless states the automaton in fig. 4.3 becomes as that in fig. 4.4. Note

Figure 4.4: Automaton for `a[],b[]*` without ϵ-transitions and without useless states

that this is the same automaton we've seen in figure 4.1, but without bindings.

N.B.: Hosoya claims in section 4.2 of [Hos03] that there is a well known linear-time algorithm for useless states elimination and cites [CDG+97]. I've looked at [CDG+97] but I've found a polynomial algorithm (RED at page 23) equivalent to that I'm using here. I don't know what Hosoya refers to (algorithms proposed below are polynomial as well).

4.2.2 Pattern Matching

Let me describe a formalization of the behavior of tree automata with sequence accumulators informally described above (this formalization is taken and adapted from that given in Section 4.1 of [Hos03]). Rather than directly dealing with sequence accumulation, we take two steps for ease of reasoning. First, an automaton annotates each node of the input value with a set of variables while matching the value. Then we extract a value for each variable by erasing the nodes whose annotated sets do not contain that variable.

Annotated values ρ are values where each label is annotated with a set of variables $\mathcal{S}$ and defined by the following syntax:

$\rho ::=$	$a[\rho]^{\mathcal{S}}$	annotated label
$\mid$	ρ, ρ	concatenation
$\mid$	ϵ	empty sequence

We omit $\mathcal{S}$ from an annotated label in contexts where it does not matter. Now, the semantics is described by the matching relation "$\mathtt{A} \vdash \mathtt{v} \in \mathtt{q} \Rightarrow \rho$", meaning "in automaton A the value v is accepted by state q and yields an annotated value ρ", by the following set of

rules.

$$\frac{q \in \mathbf{fin}(A)}{A \vdash \epsilon \in q \Rightarrow \epsilon} \quad \text{(T-Fin)}$$

$$\frac{q_1 \xrightarrow{\mathcal{S}:\, L[Y]} q_2 \in \mathbf{tr}(A) \qquad \ell \in L \qquad M(Y) \vdash v_1 \Rightarrow \rho_1 \qquad A \vdash v_2 \in q_2 \Rightarrow \rho_2}{A \vdash \ell[v_1], v_2 \in q_1 \Rightarrow \ell[\rho_1]^X, \rho_2} \quad \text{(T-Lab)}$$

We write $A \vdash v \in \rho$ when $A \vdash v \in q_0 \Rightarrow \rho$ with $q_0 \in \mathbf{init}(A)$ and write $A \vdash v$ when $A \vdash v \Rightarrow \rho$ for some ρ.

Observe the differences between my and Hosoya's formalization:

- I've removed the T-Eps rule because I never need to match values against automata with epsilon transitions
- the handling of the subtrees in the T-Lab rule is different: while Hosoya can simply move to the initial state of the subautomaton that recognizes the subtree, I need to search my separate subautomaton through the M mapping and explicitly "run" it (however I guess this is only a difference in formalization and that Hosoya's algorithm is similar to mine).

Again, recall that our label classes L have only the cases ℓ and $\tilde{}$, so that $\ell \in L$ means either $L = \ell$ or $L = \tilde{}$.

As a second step, we define the function `env`[9] that takes in input an annotated value and a variable x and returns the bound value for x.

$$\begin{array}{lcll}
\mathbf{env}(\epsilon)(x) & = & \epsilon & \\
\mathbf{env}(a[\rho_1]^{\mathcal{S}}, \rho_2)(x) & = & a[\mathbf{erase}(\rho_1)], \mathbf{env}(\rho_2)(x) & \text{if } x \in \mathcal{S} \\
\mathbf{env}(a[\rho_1]^{\mathcal{S}}, \rho_2)(x) & = & \mathbf{env}(\rho_1)(x) & \text{if } x \notin \mathcal{S},\ \mathbf{env}(\rho_1)(x) \neq \epsilon \\
\mathbf{env}(a[\rho_1]^{\mathcal{S}}, \rho_2)(x) & = & \mathbf{env}(\rho_2)(x) & \text{if } x \notin \mathcal{S},\ \mathbf{env}(\rho_1)(x) = \epsilon
\end{array}$$

where **erase** removes all the variable-set annotations from a given annotated value:

$$\begin{array}{lcl}
\mathbf{erase}(l[\rho]^{\mathcal{S}}) & = & l[\mathbf{erase}(\rho)] \\
\mathbf{erase}(\rho_1, \rho_2) & = & \mathbf{erase}(\rho_1), \mathbf{erase}(\rho_2) \\
\mathbf{erase}(\epsilon) & = & \epsilon
\end{array}$$

To understand how the function `env` works, consider what it happens when it visits a node in the input value. There are three cases. If x is in the node's variable set, it retains the node (by erasing all the annotations in its content ρ_1) and continues to process the rest of the sequence. Otherwise, if x is not in the variable set, the function searches for x

[9] defined at page 12 of [Hos03]; I fixed a misspelling error, here — in second row original paper had $a[\mathbf{erase}(\rho_1)], \rho_2(x)$ instead of $a[\mathbf{erase}(\rho_1)], \mathbf{env}(\rho_2)(x)$

inside the subtree ρ_1, recursively calling itself. If the result of this recursive call is not the empty sequence, that is if it finds x inside ρ_1, it returns the value and discard the rest of the sequence; the reason of this behavior is that annotations for a variable will always stay in subsequent nodes and never at different depths. Conversely, if it can't find x inside the subtree, it continues to search it in the rest ρ_2 of the sequence.

To understand why annotations for a variable are always in subsequent nodes recall that automata are generated from linear patterns. For example an automaton generated from the pattern "`(a[b[]],c[]) as x`" will annotate the value `a[b[]],c[]` in the following way:

$$\texttt{a[b[]}^{\emptyset}\ \texttt{]}^{\{x\}}\texttt{, c[]}^{\{x\}}$$

The only way to have a variable to be annotated on different depths, for instance as in $\texttt{a[b[]}^{\{x\}}\ \texttt{]}^{\emptyset}\texttt{, c[]}^{\{x\}}$, would be to match the value against an automaton generated from a pattern like "`a[b[] as x],c[] as x`". However this pattern is not linear[10].

Algorithm

Now I give an algorithm for pattern matching that is directly inspired by the semantics above (annotate/extract). I've found that not only the formalization but also the implementation is simpler taking the two steps described above[11].

Pattern matching is performed by two methods of the class `TA` (Tree Automaton), `matchTree` and its subroutine `matchSequence`. The fields of this class store the structure of the automaton, so we have the four `Vector` fields `states`, `initialStates`, `finalStates` for states and `transitions` for labelled transitions (there is also a fifth vector for epsilon transitions but we don't care about it here); states are represented by objects of the class `TAState` while labelled transitions are represented by objects of the class `TATransition` (recall that we won't match against automata with ϵ-transitions). As in the formal definition, states referenced inside `initialStates` and `finalStates` vectors are also referenced inside the `states` vector (i.e. the set of the initial states and the set of the final states are both subsets of the states' set of the automaton). The class `TATransition` has the following fields (in the code below we use most of them):

`String label`:	the label (it could be also "~")
`TAState src`:	the starting state
`TAState dst`:	the arriving state
`String content`:	the name of the pattern representing the subtree
`HashSet variables`:	the set of variables $\mathcal{S}$

[10] this discussion gives only an intuitive understanding, but see [Hos03] for a more formal discussion

[11] Although it's possible to implement an algorithm that directly treats with variable sequence accumulators and that is equivalent to that I give here (maybe ever a little faster), I had many technical difficulties when I attempted to implement it, so I'm returned to the algorithm suggested by the Hosoya's formalization

`matchTree` handles only the top level of the execution of the automaton: the big part of the work is done by the subroutine `matchSequence`. `matchTree` calls it on each initial state of the automaton, passing the first tag of the value or null if the current value is the empty sequence. `matchSequence` processes the whole sequence in the value, calling itself recursively.

Here's the main code.

```
boolean matchTree(Node v) {
  for each state q in initialStates {
    if (matchSequence(firstTag(v),q) == true)
      return true; // match!
  }
  return false; // it doesn't match
}

// Recursively checks the rest of the sequence in v
boolean matchSequence(Node v, TAState q) {
  if (v == null) {
    // base case: the sequence is empty or terminated,
    // return true if we are in a final state
    return q.isFinal;
  }
  else {
    for each transition t exiting from the state q
        such that v.label matches with t.label {
      // A = subautomaton for the pattern specified in t
      TA A = symtab.M(t.content);
      // call matchTree with the first tag of the sequence
      if ((A.matchTree(v.jjtGetChild(0)) == true) &&
          (matchSequence(nextTag(v),t.dst) == true)) {
        // success! annotate current node
        v.annotateValue(t.variables);
        return true;
      }
      else {
        // failure, we backtrack!
        // (delete the annotations in the subtree)
        v.eraseSubtreeAnnotations();
      }
    }
    return false;
  }
}
```

In the code above we used the following:

`boolean q.isFinal`	is a flag that says if q is a final state
`Node firstTag(v)`	is a method of the class TA that returns the first tag in v or null
`Node nextTag(v)`	is a method of the class TA that returns the next tag in v or null
`String v.label`	the label name in a tag node
`Node v.jjtGetChild(i)`	this method of the *Node* interface returns the i^{th} child of the node
`v.annotateValue(vars)`	annotates `v` with `vars` set
`v.eraseSubtreeAnnotations()`	deletes annotations from the subtree (used for backtracking)

Notice the similarities between the T-FIN/T-LAB rules of section 4.2.2 and the code of `matchSequence` function[12]: the external `if-then-else` construct choose which of the two rules apply, while the code corresponding to the T-LAB rule checks if there exists a transition $t = q \xrightarrow{\mathcal{S}: L[Y]} q_{dst}$ such that:

- it starts from current state q
- its label class L (`t.label`) matches with the label of the value ℓ (`v.label`)
- its subpattern Y (`t.content`) recognizes the subtree v_1 ($M(Y) \vdash v_1 \in \rho_1$ is the same as `A.matchTree(v.jjtGetChild(0))` with `A = M(t.content)`)
- it arrives in a state q_{dst} (`t.dst`) that recognizes the rest of the sequence v_2 in the value ($A \vdash v_2 \in q_{dst} \Rightarrow \rho_2$ is the same as `matchSequence(nextTag(v),t.dst)`)

The implementation of the `env` function is straightforward. I've written a method `bindAll` shared by all Node classes[13] that recursively calls itself on its subtree. Before recursing, it reads the annotations on the node on which is invoked (annotations are inside a `Vars` field) and updates the sequence accumulators corresponding to the variables which are in the `Vars` set.

```
public void bindAll() {
    for each x in Vars {
    // retrieves the accumulator for the variable x
```

[12] careful readers could have noticed that in section 4.1 I've said that tag nodes have a `name` field for labels whereas in the code above I've used a field `label` for it — actually `v.label` is pseudo–code for "`(ASTTag)v.name`"

[13] In real code all Node classes inherit from a SimpleNode class that implements the *Node* interface

```
    Node xAcc = getAccumulator(x);
    // append this node in tail to the sequence accumulator
    xAcc = value , this;
  }
  // recur on children
  for (int i = 0; i < children.length; i++) {
    children[i].bindAll();
  }
}
```

`getAccumulator(x)` above is a pseudo–code that retrieves the sequence accumulator corresponding to the variable `x` (in the real code the sequence accumulator is a field of the `SymElement` object representing `x` and is easily accessible because annotations on the nodes of the input value are actually lists of `SymElement` references instead of sets of variable names).

Note that, differently from `env`, `bindAll` updates all the variables at the same time so it doesn't need to take a variable as argument (and obviously it doesn't need an annotated value in input because it's invoked as method of an annotated value). Ultimately, given a tree value t and a pattern represented by an automaton A, the pattern matcher runs a code similar to the following

```
if A.matchTree(t) {
  t.bindAll();
  ...
} else { ... }
```

4.3 Static Typechecks

Suppose the following XRel source code:

```
...
import P0; // import pattern
export Tout; // output type
typeswitch(x) {
  case P1:  ...
  ...
  case Pn:  ...
}
```

where $\mathtt{P}_0$ is the import pattern, $\mathtt{T}_{out}$ is the declared output type, $\mathtt{x} \in \mathbf{bv}(\mathtt{P}_0)$ is the variable used for pattern matching, $\mathtt{P}_1$, ..., $\mathtt{P}_n$ are the patterns inside the clauses of the `typeswitch` construct ($n \geq 0$). Further, suppose that x ∈ T.

As we've seen in the previous chapter, several static typechecks are applied to the source code. Below we write $tyof(P)$ for the type obtained by erasing all variable binders from P.

Exhaustivity check: patterns in `typeswitch` construct must be exhaustive with respect to the input type. Formally we require
$T \subseteq tyof(P_1) \mid \ldots \mid tyof(P_n)$

Irredundancy check: for all pattern P_i in `typeswitch` construct, check that P_i is not redundant with respect to the input type and the previous patterns. Formally
$(tyof(P_i) \ \cap \ T \backslash (tyof(P_1) \mid \ldots \mid tyof(P_{i-1})) \ \neq \ \emptyset)$

Ambiguity check: all patterns in the program (P_0,..., P_n) must be weakly unambiguous. A pattern P described by an automaton M_P is defined to be *weakly unambiguous* if and only if for each value v there is at most one path from the initial state to a final state in M_P that recognizes v (definition 4.1 in [BK91])

Output type check: If an XRel program includes an `export` clause, all the expressions e are checked to be correctly typed, i.e. we check that $T_e \subseteq T_{out}$, where T_e is the inferred type for e and T_{out} is the declared output type.

For all checks above I've used the following algorithms.

Union of regular expressions $T_1 \cup T_2$ (or equivalently $\mathrm{T}_1 \,|\, \mathrm{T}_2$) uses merging of automata. As we've seen in section 4.2, merging of automata is a basic operation in Thompson's construction algorithm, so we have only to use a subroutine of that algorithm (note that union is used both for exhaustivity and irredundancy checks).

Intersection between regular expressions $T_1 \cap T_2$ uses product between automata. From classical theory [HU97, CDG+97], we know that $L(T_1 \cap T_2) = L(M_{T_1} \times M_{T_2})$, where M_T is the automaton generated by the regular expression T, $L(T)$ is the language denoted by T, and $L(M)$ is the language recognized by the automaton M. Note that intersection is used by the irredundancy check (and we'll see below that also type inference and the ambiguity check use product).

Subtyping To check T_1 `<:` T_2 (or equivalently $T_1 \subseteq T_2$), there is a well-known algorithm in string automata theory. First, remember from set theory that

$$T_1 \subseteq T_2 \quad \Leftrightarrow \quad T_1 \backslash T_2 = \emptyset \quad \Leftrightarrow \quad T_1 \cap \overline{T}_2 = \emptyset$$

The classical algorithm [HU97] uses automata and splits the problem into three steps:

1. compute $\overline{T}_2$ (complement M_{T_2})
2. compute $T_1 \cap \overline{T}_2$ (compute $M_{T_1} \times M_{T_2}$)
3. check non-emptiness of the result

Unfortunately we can't use this algorithm in a naive way, as we have problems at point 1. Although the complementation of an automaton is easy by itself (it merely consists of the inversion between final and non-final states), it can only be applied to deterministic automata. To apply it also to non-deterministic automata like that built by the Thompson's algorithm, a "determinization" algorithm must be used to translate the NFA into a DFA. However, top–down non-deterministic tree automata and top–down deterministic tree automata are *not* equally expressive, as their counterparts on string automata, so we can't use classical determinization algorithm based upon subset construction.

Our solution is to give a difference algorithm and then a non emptiness test on automata. The difference algorithm includes an unusual subset construction different by the usual subset construction used in standard string automata theory; it also uses a label normalization algorithm that handles matching between labels and wildcard labels (see section 4.3.4).

Evaluation of the expressions To compute the type T_e of an expression we have only to replace in the tree representing the expression the variable names with their inferred types. So the non trivial part of the algorithm is the **type inference** for variables. We'll talk about it in section 4.3.2.

Finally, note that $T = tyof(P)$ is really not a function but a formalism (when we use the automaton M_P generated from the pattern P we only have to ignore variable sets on labelled transitions).

4.3.1 Product Automata

Given two automata A and B representing patterns, we define the product A × B as follows

$$\begin{aligned}
\mathbf{st}(A \times B) &= \mathbf{st}(A) \times \mathbf{st}(B) \\
\mathbf{init}(A \times B) &= \mathbf{init}(A) \times \mathbf{init}(B) \\
\mathbf{fin}(A \times B) &= \mathbf{fin}(A) \times \mathbf{fin}(B) \\
\mathbf{tr}(A \times B) &= \left\{ \langle p_1, q_1 \rangle \xrightarrow{\mathcal{S}:\,(L_1 \cap L_2)[\langle Y_1, Y_2 \rangle]} \langle p_2, q_2 \rangle \;\middle|\; \begin{array}{l} L_1 \cap L_2 \neq \emptyset \\ p_1 \xrightarrow{L_1[Y_1]} p_2 \in \mathbf{tr}(A) \\ q_1 \xrightarrow{\mathcal{S}:\,L_2[Y_2]} q_2 \in \mathbf{tr}(B) \end{array} \right\}
\end{aligned}$$

Note that this product definition "discards" the variable sets on the transitions of the automaton A, retaining the binding behavior of the automaton B. The reason is that we'll

use it either to compute the intersection between a type and pattern (in type inference) or between a pattern with itself (in the ambiguity check).

Note also that the intersection between label classes is extremely simplified in my case (I only need to handle the three cases "$\ell \cap \ell = \ell$", "$\sim \cap \ell = \ell$" and "$\sim \cap \sim = \sim$")

Differently from Hosoya, I define the product only between automata without ϵ-transitions. Hosoya instead needs to use automata with ϵ-transitions in the product because $\epsilon-\mathbf{elim}(A)$ operation we've seen in 4.2.1 doesn't preserve strong ambiguity (refer to the section **??** for a related discussion).

I don't give pseudo–code for the product as it merely follows the formalization above. Indeed refer to the difference algorithm in the section 4.3.4 for an example of a compositional algorithm on the automata (actually the product algorithm could be actually "extracted" from the difference algorithm as an its subset).

4.3.2 Type inference

Given an automaton A that represent an input type and an automaton B that represent a pattern, the inference algorithm works in two steps.

First, it specializes B with respect to A such that the resulting automaton C behaves exactly the same as B except that it accepts only values from A. For this it uses the product construction above that preserves the variable binding behavior in B.

Second, it obtains for each variable x an automaton D_x such that D_x accepts a value v if and only if C accepts some value from A and yields a binding of x to v. It computes this automaton from C by retaining all transitions with the annotated set containing x and eliminating all the other.

Formally, given ϵ-free automata A and B, we first take the product of them and then eliminate useless states (note that ϵ-elimination preserves the binding behavior).

$$\mathrm{C} = \emptyset-\mathbf{elim}(\mathrm{A} \times \mathrm{B})$$

My algorithm works in the following way: collect all the subautomata of a pattern (included the pattern itself), then update the inferred types of the variables encountered.

We then compute the following automaton D_x for each variable x.

$$\begin{aligned}
\mathbf{st}(D_x) &= \mathbf{st}(C) \\
\mathbf{init}(D_x) &= \mathbf{init}(C) \\
\mathbf{fin}(D_x) &= \mathbf{fin}(C) \\
\mathbf{tr}(D_x) &= \left\{ q_1 \xrightarrow{l[Y]} q_2 \;\middle|\; q_1 \xrightarrow{\mathcal{S}:\, l[Y]} q_2 \in \mathbf{tr}(C) \quad x \in \mathcal{S} \right\} \\
&\cup \left\{ q_1 \xrightarrow{\epsilon} q_2 \;\middle|\; q_1 \xrightarrow{\mathcal{S}:\, l[Y]} q_2 \in \mathbf{tr}(C) \quad x \notin \mathcal{S} \right\}
\end{aligned}$$

Despite the bigger complexity of the formalization of Hosoya in section 5 of [Hos03] I think he uses the same algorithm and that this bigger complexity is only due to the choice to include subautomata in the main automaton.

4.3.3 Ambiguity Check

Given an automaton A representing the input type and an automaton B representing a pattern, consider the following procedure in four steps to check if B is weakly ambiguous at top–level w.r.t A

1. compute the product C = B × B
2. eliminate the useless states from C
3. compute the product D between A and the resulting automaton
4. check that in D there are no states $\langle p, \langle q, r\rangle\rangle$ such that $q = r$

Here top–level means that we check if there is ambiguity between labels at top–level, without searching for ambiguity inside subautomata.

To see why it's correct let us consider a more simplified setting, removing the input type so that resulting algorithm is given by the steps 1, 2 and 4 of the algorithm above. Note that this is the same example Hosoya does in section 6 of [HP03] where it limits reasoning to string automata (that is the same we do above talking about top–level ambiguity of tree automata), with no ϵ-transition (we don't need them because we check for weak ambiguity instead of strong ambiguity) and to no input type.

The given algorithm check whether there are two different accepting paths (i.e. a path from an initial state to a final state) that spell out the same string. Indeed, if there are two different accepting paths in B

$$q_1 \dots q_n \qquad q'_1 \dots q'_n$$

both spelling out a string s, then there is an accepting path

$$< q_1, q'_1 > \dots < q_n, q'_n >$$

spelling out s where $q_k \neq q'_k$ for some k, and vice versa. Note that $< q_k, q'_k >$ is not discarded by the step 2 of the algorithm, because it is in an accepting path. But then we've finished, because an automaton is weakly ambiguous if and only if each value v can be traced uniquely with a path through the automaton.

Formally, given ϵ-free automata A and B, we first compute the automaton:

$$\mathrm{D} = \emptyset\text{−}\mathbf{elim}(\mathrm{A} \times (\mathrm{B} \times \mathrm{B}))$$

Then we check than for all $\langle p, \langle q, r\rangle\rangle \in D$ we have that $q \neq r$.

Note that D has the binding behavior of B (the rightmost operand in the series of products).

The complete algorithm applies the procedure above to the top–level automaton. Then if it doesn't find ambiguity, it recursively calls itself applying the procedure to immediately descendent subautomata, and so on. Recursion is avoided by storing the list of automata that have already been checked and not checking them again.

4.3.4 Difference Automata

The construction of the difference automata is similar but not the same as the usual subset construction.

For s a set $\{s_1, \ldots, s_k\}$, write $\widetilde{Y}_s$ for $\{Y_{s_1}, \ldots, Y_{s_n}\}$ and similarly $\widetilde{p_s}$.

Given two element automata $M_i = (Q_i, Q_i^{init}, Q_i^{fin}, \delta_i)$ (where $i = 1, 2$) the difference between M_1 and M_2 is an automaton $M = (Q, Q^{init}, Q^{fin}, \delta)$ as follows.

$$
\begin{aligned}
Q &= Q_1 \times \mathcal{P}(Q_2) \\
Q^{init} &= \{\langle q_1, \{Q_2^{init}\}\rangle \mid q_1 \in Q_1^{init}\} \\
Q^{fin} &= \{\langle q_1, P\rangle \mid q_1 \in Q_1^{fin} \text{ and } P \subseteq Q_2 \text{ and } P \cap Q_2^{fin} = \emptyset\} \\
\delta &= \{(\langle q_1, \widetilde{P_N}\rangle, l[Y \backslash \widetilde{Y}_s], \langle q_1', \widetilde{P}_{N\backslash s}\rangle \mid (q_1, l[Y], q_1') \in \delta_1 \\
&\qquad \text{and let } N \text{ be the least set containing all } n\text{: } p_n \xrightarrow{l[Y_n]} p_n' \\
&\qquad \text{and } s \text{ ranges over all subsets of } N\}
\end{aligned}
$$

Highlights of the algorithm

To understand why we need to use the strange subset construction above, consider the following

$$a[x], b[x] \backslash (a[y], b[y] \mid a[z], b[z])$$

We proceed the difference by first subtracting a[y],b[y] from a[x],b[x]. This yields

$$(a[x] \backslash a[y]), b[x] \;\mid\; a[x], (b[x] \backslash b[y])$$

That is, we obtain the union of two expressions, one resulting from subtracting the first component and the other resulting from subtracting the second. This can be understood by observing that "a value $a[v], b[w]$ being not in $a[y], b[y]$" means "either $a[v]$ being not in $a[y]$ or $b[w]$ being not in $b[y]$". Now, back to the original difference calculation, we next subtract the second clause $a[z], b[z]$ from the above result. Performing a similar subtraction,

we obtain:

$$\begin{array}{ll} (a[x]\backslash(a[y] \mid a[z])), & b[x] \\ \mid (a[x]\backslash a[y]), & (b[x] \mid b[z]) \\ \mid (a[x]\backslash a[z]), & (b[x] \mid b[y]) \\ \mid a[x], & (b[x]\backslash(b[y] \mid b[z])) \end{array}$$

Thus, the original goal of difference has reduced to the combination of the subgoals of difference. Let us consider only the first difference $(a[x]\backslash(a[y] \mid a[z]))$ since the other are similar. Since all expressions appearing here are single-elements with a[], the result is obviously a single-element with a[]. What is less obvious is that its content is the difference between the variable x and the "union" of the variables y and z. In general, given two grammars, we have to compute the difference between a variable from one grammar and a set of variables from the other grammar. This is why the algorithm involves a sort subset construction (note however that we don't standard subset construction).

Wrong formalization

Hosoya gives both in [HM02b] and in [HM02a] the following definition.

Given two element automata $M_i = (Q_i, Q_i^{init}, Q_i^{fin}, \delta_i)$ (where $i = 1, 2$) the difference between M_1 and M_2 is an automaton $M = (Q, Q^{init}, Q^{fin}, \delta)$ where[14]:

$$\begin{aligned} Q &= Q_1 \times \mathcal{P}(Q_2) \\ Q^{init} &= \langle Q_1^{init}, \{Q_2^{init}\}\rangle \\ Q^{fin} &= \{\langle q_1, P\rangle \mid q_1 \in Q_1^{fin} \text{ and } P \subseteq Q_2 \text{ and } P \cap Q_2^{fin} \neq \emptyset\} \\ \delta &= \{(\langle q_1, \{p_1, \ldots, p_k\}\rangle, l[\langle Y, \{Y_1, \ldots, Y_k\}\rangle], \langle q_1', \{p_1', \ldots, p_k'\}\rangle) \mid \\ &\qquad (q_1, l[Y], q_1') \in \delta_1 \text{ and } \{(p_1, l[Y_1], p_1'), \ldots, (p_k, l[Y_k], p_k')\} \subseteq \delta_2\} \end{aligned}$$

This algorithm (that is actually the *standard* subset construction) is wrong. Consider first that it doesn't handle properly the case $\epsilon\backslash\epsilon$ (it returns ϵ instead of $\emptyset$).

However the wrong handling of base cases (ϵ) is not the only problem. For example, algorithm above would compute wrongly `a[x],b[x] \ a[y],b[y] = a[x\y], b[x\y]` instead of `a[x], b[x\y] | a[x\y], b[x]`.

To see that this is wrong consider for instance `x = l1 | l2` and `y = l2`. We would have as an answer `a[l1],b[l1]`, instead of the correct answer `a[l1],b[l1] | a[l1],b[l2] | a[l2],b[l1]`.

For example in the following XRel source code

[14] I fixed several typographical errors here

```
import a[l1|l2],b[l1|l2];

typeswitch {
  case a[l1],b[l1]: ...
  case a[l1],b[l2]: ...
  case a[l2],b[l1]: ...
  case a[l2],b[l2]: ...
}
```

that is correct, the algorithm above asserts that the second clause is redundant. The irredundancy check for the second clause is `a[l1],b[l2]` $\cap$ `(a[l1|l2],b[l1|l2]` $\setminus$ `a[l1],b[l1])` $\neq \emptyset$. Because the algorithm wrongly asserts that `(a[l1|l2],b[l1|l2]` $\setminus$ `a[l1],b[l1])` = `a[l2],b[l2])`, the irredundancy wrongly claims that the second case is redundant.

Pseudo–Code

`A.buildDiff(A1,A2)` computes the difference `A1` $\setminus$`A2` between `A1` and `A2`; it's invoked as a method on the difference automaton to build.

```
void buildDiff(TA A1, TA A2) {
  // create initial states
  // (note that the automaton has as many initial states as A1)
  for each state s1 in A1.initialStates {
    // create the subset state that collects states of A2
    new s = <s1,A2.initialStates>;
    states = states ∪ s;
    initialStates = initialStates ∪ s;
    // final iff s1 is final and there are no final states in
    // A2.initialStates (remember we compute the difference:
    // A1\A2 recognizes the empty string if and only if A1
    // recognizes the empty string but A2 don't recognize it)
    if (s1.isFinal && (A2.initialStates ∩ A2.finalStates) = ∅) {
      finalStates = finalStates ∪ s;
    }
  }
  // Then iterate adding transitions and the other states
  for each s in states {
    processState(A1,A2,s); // s = <s1,s2List>
  }
}
```

Note that states of a difference automaton are different from those of a normal automaton derived by a pattern because they have the form <state, states' set> (e.g. $\texttt{<p}_1\texttt{,\{q}_1\texttt{,} \ldots \texttt{q}_n\texttt{\}>}$).

Also, transitions `content`s have the form `<Y \(Y`$_1$`|...|Y`$_n$`)>` where `Y,Y`$_1$`,...Y`$_n$ are pattern names referencing subautomata.

The subroutine `processState` takes in input a state `s = <s1,s2List>` and builds two sets:

1. `srcTransitions` is the set of transitions starting from `s1` that have been normalized (we'll see in the next section what it means).
2. `dstTransitions` is the set of all the transitions exiting from states in `s2List`

I use `srcTransitions` instead of directly to use `s1.transitions` because I won't to modify transitions of automaton `A1` (normalization can possibly add new transitions and modify existing ones).

Then, `processState` invokes the subroutine `processMatchingTransitions` on each transition `t1` in `srcTransitions` passing to it:

1. the transition `t1`
2. the set of transitions in `dstTransitions` that label–match with `t1` (two transitions label–match if their labels match)
3. the state `s = <s1,s2List>`

The subroutine `processMatchingTransitions` will take the source state `s`, the transition `t1` exiting from `s1` and the transitions `dstTransitions` exiting from the states in `s2List` and will build the new transitions for the difference automaton that exit from `s` along with the new arriving states.

Here's the code for `processState`.

```
void processState(TA A1, TA A2,TADifferenceState s) {
  // build the list of transitions exiting from the
  // difference states (difference states are states in s2List,
  // where s = <s1,s2lList>)
  HashSet dstTransitions = ∅;
  for each state slEl in s2List such that s = <s1,s2lList> {
    dstTransitions = dstTransitions ∪ slEl.transitions;
  }

  // Label normalization
  HashSet srcTransitions =
    normalizeLabels(s.s1.transitions,slTransitions);

  // now loops through transitions
```

```
  for each transition t1 in srcTransitions {
    // t2List is the list of transitions label-matching with t1
    HashSet t2List = ∅;
    for each transition t2 in dstTransitions {
      if (t1.label matches with t2.label)
        tList = tList ∪ t2;
    }
    // finally call the code matching transitions
    processMatchingTransitions(t1,t2List,s);
  }
}
```

Finally the subroutine `processMatchingTransitions` contains the more interesting part of the code. It takes in input a transition `t1` of the automaton `A1`, a set `t2List` of transitions of `A2` label–matching with `t1` and a starting state `s` of the difference automaton and it adds to the difference automaton the transitions exiting from s and the arrival states (if they're new).

Here's the code.

```
void processMatchingTransitions(TATransition t1,
          HashSet t2List,TADifferenceState srcState) {
  // Given n = tList.size() we need to build 2^n new transitions
  // that will start from srcState and will go into 2^n distinct
  // (possibly new) states, all with "l" as label.
  int n = tList.size;
  int subsetNum = 2^n;
  /* loop through the first 2^n naturals (c stands for counter).
  * c is used both as counter and as subset specifier of tList:
  * the first n digits of the counter represent the elements
  * of tList (1 is in the subset, 0 is not).
  * For each loop of this cycle we create a new transition and
  * possibly a new state */
  for each c in {1,...,2^n} {
    /* before to create the new transitions we need to compute
    * the current subset of subtrees to differentiate and the
    * current subset of states to differentiate.
    * The two subsets are one the complement of the other.
    * For example if n=3 and c=2 (010 in binary) I take my subtrees
    * to differentiate from the first and the third transition
    * in tList (because the first and third digits of c are 0),
    * whereas my set of states to differentiate is the singleton
    * given by the destination state of the second transition in
```

```
* tList (because the second digit of c is 0).
* For example given tList as follows
* tList[0] = "q3 = a[x] --> q4"
* tList[1] = "q5 = a[y] --> q6"
* tList[2] = "q7 = a[z] --> q8"
* if c=5 I take diffTrees = {x,z} and diffStates = {q6}
*/
HashSet diffTrees = ∅; // set of strings (names of automata)
HashSet diffStates = ∅; // set of (destination) states
if (n > 0) {
  for (int i=0; i<n; i++) {
    int bitmask = 1 << i;
    if ((c & bitmask) != 0) {
      diffStates = diffStates ∪ tList[i].dst;
    }
    else {
      diffTrees = diffTrees ∪ tList[i].content;
    }
  }
}
// find the destination state
// if destination state doesn't exist create it
if (<t1.dst, diffStates> doesn't exist) {
  new destState = <t1.dst,diffStates>;
  states = states ∪ destState;
  /* final if the destination in t1 is final but there are
  * not final destination states in the list of states to
  * subtract */
  if (t1.dst.isFinal &&
      (finalStates ∩ diffStates = ∅) {
    finalStates = finalStates ∪ destState;
  }
}
// else use that existent one
else destState = <t1.dst,diffStates>;

// create the new transition
content = <t1.content, diffTrees>;
// Note that bindings are discarded (I put variables to null)
new nt = TATransition(l,srcState,destState,content,∅);
// diffLabels = l_1 | l_n are labels removed from a
// wildcard label as in (~\diffLabels)[...]
```

```
    // (see Label Normalization section below)
    nt.diffLabels = nt.diffLabels ∪ t1.diffLabels;
    transitions = transitions ∪ nt;
  }
}
```

This code takes a transition from the automaton $\texttt{A}_1$ and n transitions from automaton $\texttt{A}_2$ and builds 2^n transitions in the difference automaton, each corresponding to a subset of the transitions in $\texttt{A}_2$ (this is for what is a kind of subset construction). However, observe that the building is different from that of the usual subset construction for string automata. In fact, we need to "revert" links: this means that if we have the transition $t = q_1 \xrightarrow{a[Y]} q'_1 \in tr(A_1)$ and the transitions $t_1 = p_1 \xrightarrow{a[Y_1]} p'_1, \ldots, t_n = p_n \xrightarrow{a[Y_n]} p'_n \in tr(A_2)$ then, for each subset $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$, (k $\leq$ n), we have the transition $\langle q_1, \{p_1, \ldots, p_n\}\rangle \xrightarrow{a[Y \setminus (Y_{i_1}, Y_{i_k})]} \langle q'_1, \{p'_{i_{k+1}}, p'_{i_n}\}\rangle$

For example if n=3 and we take the subset {1,2} (k=2) then we have the transition $\langle q_1, \{p_1, p_2, p_3\}\rangle \xrightarrow{a[Y \setminus (Y_1, Y_2)]} \langle q'_1, \{p'_3\}\rangle$.

I've deployed this algorithm adapting the old subtyping algorithm on grammars I found in [Hos01].

Label normalization

As we've seen above the function `normalizeLabels(a1TransSet,a2TransSet)` takes in input a set `a1TransSet` of transitions from automaton `A1` and a set `a2TransSet` of transitions from automaton `A2` and returns the set `a1TransSet` normalized.

The normalization of the transitions in `a1TransSet` fixes a problem with the algorithm above. Specifically suppose to have $\texttt{~[Y}_1\texttt{]} \setminus \texttt{a[Y}_2\texttt{]}$: what is the result? well, to write $\texttt{~[Y}_1\texttt{]} \setminus \texttt{a[Y}_2\texttt{]}$ is the same as to write $\texttt{(~\a)[Y}_1\texttt{]} \texttt{ | a[Y}_1\texttt{]} \setminus \texttt{a[Y}_2\texttt{]}$ so the result is $\texttt{(~\a)[Y}_1\texttt{]| a[Y}_1\texttt{\Y}_2\texttt{]}$ — consider for example $\texttt{~[l}_1\texttt{,l}_2\texttt{]} \setminus \texttt{a[l}_1\texttt{]}$ whose result is $\texttt{(~\a)[l}_1\texttt{,l}_2\texttt{] | a[l}_2\texttt{]}$.

Actually the algorithm in `processMatchingTransitions` requires that label classes are disjoint, so we need to split transitions with wildcard labels as in the example I shown above. Note that not only `~[...]` but also `#String[...]` is a wildcard label — we could replace the examples above with `#String[...]  \ #"hello"[...]`.

This is what label-normalization algorithm does: it splits the transitions in `a1TransSet` with wildcard labels so that they're equal or disjoint with the labels in `a2TransSet`. I omit the code because it's straightforward.

So label classes L in difference automata also need to have the case $\texttt{~\(l}_1\texttt{ |...| l}_n\texttt{)}$. This is a reason for which label classes were a design issue in XRel (see Section 3.3.2, page 49).

4.3.5 Emptiness Test On Automata

Hosoya gives in [HM02a] a non–emptiness test for compound regular expressions, i.e. for regular expressions with elements and attributes. Below I report an adaptation for element-only expressions and for my framework.

Given a mapping F from pattern names Y to patterns $F(Y)$, we compute a series of (total) functions ϕ_0,ϕ_1,... from the names in dom(F) to booleans, as defined below.

$$\phi_0(Y) = \mathbf{false}$$
$$\phi_i(Y) = \mathbf{nemp}(F(Y))\phi_{i-1}$$

where the function **nemp** is inductively defined as follows.

$$\begin{array}{lcl}
\mathbf{nemp}(l[Y])\phi & = & \phi(Y) \\
\mathbf{nemp}(())\phi & = & \mathbf{true} \\
\mathbf{nemp}(P_1, P_2)\phi & = & \mathbf{nemp}(P_1)\phi \wedge \mathbf{nemp}(P_2)\phi \\
\mathbf{nemp}(P_1 \mid P_2)\phi & = & \mathbf{nemp}(P_1)\phi \vee \mathbf{nemp}(P_2)\phi \\
\mathbf{nemp}(P*)\phi & = & \mathbf{nemp}(P)\phi \\
\mathbf{nemp}(P \texttt{ as } x)\phi & = & \mathbf{nemp}(P)\phi
\end{array}$$

This algorithm was somewhat unsatisfactory for me as I needed an algorithm on tree automata, so I've re–written it for automata. Below I give my pseudo–code.

Non–emptiness test on automata

```
// check if an automaton is empty
public boolean isNull() {
  // take the list of all the subautomata and include myself
  HashSet saList = getDescendantAutomata();
  saList.add(name);
  /* In the fixpoint algorithm defined below the following mapping
   * from automata names to boolean values will contain the partial
   * emptiness values of the automata.*/
  HashMap map = new HashMap(); // map: automata -> booleans
  // Initialize mapping from automata to emptiness values
  // in the following way. Given a subautomaton sa:
  // - if sa recognizes the empty string then set map[sa] = false
  // - else map[sa] = true
  // The algorithm would work well also filling all emptiness values
  // with true (the shown inizialization is an optimization)
  for each string saName in saList {
    TA sa = symtab.M(saName);
    if (sa.recognizesEmptyString()) map(saName) = false;
    else map(saName) = true;
```

```
  }
  // now let us begin with the fixpoint algorithm
  boolean fixpoint; // we have reached the fixpoint?
  do {
    fixpoint = true;
    for each saName in  dom(map) {
      TA sa = symtab.M(saName);
      // fixpoint iteration
      if ((map(saName) == true) && (sa.isNullYet(map) == false)) {
        /* the empty value for sa is changed! update the map and
         * force another loop */
        map(saName) = false;
        fixpoint = false;
      }
    }
  } while (!fixpoint);
  return map(this);
}
```

We give also subroutines. `recognizesEmptyString()` simply returns true if the current automaton recognizes the empty sequence, while `isNullYet()` is the equivalent of the *nemp* function formalized above (actually is !*nemp*).

```
private boolean recognizesEmptyString() {
  for each s in initialStates {
    if (s.isFinal()) return true;
  }
  return false;
}

private boolean isNullYet(HashMap map) {
  /* the valid transitions are the transitions with
   * a non-empty content automaton */
  Vector validTransitions = new Vector();
  for each transition t in transitions {
    if (map(t.content) == false)
      validTransitions.add(t);
  }
  // validStates are states reachable from the the initial ones
  Vector validStates;
  // add before all the initial states
  validStates = initialStates;
  // then add all the states reachable with valid transitions
```

```
  boolean changed;
  do {
    changed = false;
    for each transitions vt in validTransitions {
      if (validStates.contains(vt.src)) {
        changed = true;
        validTransitions.remove(vt);
        // also add the destination state
        if(!validStates.contains(vt.dst)) validStates.add(vt.dst);
      }
    }
  } while(changed);
  // finally check if it exists at least a reachable final state
  for each fs in finalStates {
    if (validStates.contains(fs)) return false;
  }
  // I've not found any reachable final state, is null yet
  return true;
}
```

Appendix A

BNF grammar of XRel

This appendix contains a formal BNF grammar generated from the tool JJDoc[1].

A.1 Non-Terminals

```
Program    ::= TypeDecl* ImportSt ExportTp? SwitchCase?
TypeDecl   ::= <TYPE> Identifier "=" TypeExpr ";"
ImportSt   ::= <IMPORT> TypeExpr ";"
ExportTp   ::= <EXPORT> TypeExpr ";"
SwitchCase ::= <SWITCH> "(" Identifier ")"
               "{" Case* ( <CASEDEFAULT> ":" StatementList )? "}"
Case       ::= <CASE> TypeExpr ":" StatementList
StatementList ::= "{" Statement* "}"
                | ";"
                | Statement
Statement  ::= <PRINT> "(" ValueExpr ")" ";"
ValueExpr  ::= ValueTerm ( "," ValueTerm )*
ValueTerm  ::= Label "[" "]"
             | Label "[" ValueExpr "]"
             | Identifier
             | "(" ValueExpr ")"
             | "()"
             | Literal
TypeExpr   ::= CommaSeq ( "|" CommaSeq )*
CommaSeq   ::= UnaryOp ( "," UnaryOp )*
```

[1] I've removed useless parentheses and hand-formatted the grammar generated from JJDoc, the original code is in the file doc\XRel_grammar.html

```
UnaryOp    ::= TermExpr ( "*"  | "+" | "?" )? ( <AS> Identifier )?
TermExpr   ::= Label "[" "]"
             | Label "[" TypeExpr "]"
             | Identifier
             | "(" TypeExpr ")"
             | "()"
             | PrimitiveType
             | Literal
Literal    ::= <STRING_LITERAL>
PrimitiveType ::= <STRING>
Label      ::= <ANYTAG> | <IDENTIFIER>
Identifier ::= <IDENTIFIER>
```

A.2 Terminals

```
ANYTAG ::= "~"
AS ::= "as"
CASE ::= "case"
CASEDEFAULT ::= "default"
IDENTIFIER ::= <LETTER> ( <LETTER> | <DIGIT> | <UNDERSCORE> )*
IMPORT ::= "import"
EXPORT ::= "export"
PRINT ::= "printf"
STRING ::= "String"
STRING_LITERAL ::= ... double quoted strings ...
SWITCH ::= "typeswitch"
TYPE ::= "typedef"
```

Bibliography

[BCF02] Veronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: a white paper. Available on: `http://www.cduce.org`, November 2002.

[BCF03] Veronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: An XML-centric general-purpose language. Available on: `http://www.cduce.org`, August 2003.

[BIM+] BEA, IBM, Microsoft, SAP, and Siebel. Business process execution language for web services (BPEL4WS). http:// ifr.sap.com/ bpel4ws/.

[BK91] Anne Brueggemann-Klein. Regular expressions into finite automata. Technical report, 1991.

[Bus] Business Process Management Initiative. Business process modelling notation (BPML). http:// www.bpmi.org/.

[CDG+97] 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.

[CM01] James Clark and Makoto Murata. RELAX NG. `http://www.relaxng.org`, 2001.

[Con01] W3C World Wide Web Consortium. XML Schema Part 0: Primer, May 2001. http://www.w3.org/TR/xmlschema-0/.

[Con02] W3C World Wide Web Consortium. Web services activity web site, 2002. http://www.w3.org/2002/ws/.

[Con03] W3C World Wide Web Consortium. SOAP Version 1.2 Part 0: Primer, June 2003. http://www.w3.org/TR/soap12-part0/.

[Cor] Microsoft Corporation. Biztalk server. http:// www.microsoft.com/ biztalk/.

[DFF+] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. XML-QL: A query language for XML. `http://www.w3.org/TR/NOTE-xml-ql`.

[DOM] Document object model (DOM). `http://www.w3.org/DOM/`.

[FGM+99] Robert Fielding, Jim Gettys, Jeff Mogul, Henrik Frystyk, L. Masinter, Paul Leach, and Tim Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2616, June 1999. http://www.ietf.org/rfc/rfc2616.txt.

[FMM+03] M. Fernández, A. Malhotra, J. Marsh, D. Florescu, J. Siméon, and J. Robie. XQuery 1.0: An XML query language. `http://www.w3.org/TR/2003/WD-query-20030822`, 2003.

[Fou] The Apache Software Foundation. Xerces: XML parser for Java. http://xml.apache.org.

[GP03] V. Gapeyev and B. Pierce. Regular object types, 2003.

[GW00] Philippa Gardner and Lucian Wischik. Explicit fusions. In Mogens Nielsen and Branislav Rovan, editors, *MFCS 2000*, volume 1893 of *Lecture Notes in Computer Science*, pages 373–382. Springer-Verlag, 2000. Full version to appear in TCS.

[HM02a] H. Hosoya and M. Murata. Boolean operations and inclusion test for attribute-element constraints (extended abstract). 2002.

[HM02b] Haruo Hosoya and Makoto Murata. Validation and boolean operations for attribute-element constraints. In *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2002.

[Hos01] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, Graduate School of University of Tokyo, Japan, 1 2001.

[Hos03] H. Hosoya. Regular expression pattern matching - a simpler design. Technical report, 2 2003.

[Hos04] H. Hosoya. Regular expression filters for XML. *Programming Languages Technologies for XML (PLAN-X), 2004*, 2004. To appear.

[HP00] H. Hosoya and B. C. Pierce. "XDuce: A Typed XML Processing Language (Preliminary Report)". In *Int'l Workshop on the Web and Databases (WebDB)*, Dallas, TX, 2000.

[HP02] H. Hosoya and B. Pierce. Regular expression pattern matching for XML. 2002.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, May 2003.

[HU97] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison Wesley, 1997.

[HVP00] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.

[KMS] Nils Klarlund, Anders Møller, and Michael I. Schwartzbach. DSD: A schema language for XML. `http://www.brics.dk/DSD/`.

[Koh] KAWAGUCHI Kohsuke. Ambiguity detection of RELAX grammars. `www.kohsuke.org/relaxng/ambiguity/ AmbiguousGrammarDetection.pdf`.

[LM02] David Richter L.G Meredith, Steve Bjorg. Highwire language specification version 1.0. Microsoft Corp., 2002.

[Mer02] L.G Meredith. Highwire: Protocol object oriented programming. Microsoft Corp., 2002.

[Mil99] Robin Milner. *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press, Cambridge, England, 1999.

[Mil03] Paolo Milazzo. Implementazione di un linguaggio di programmazione distribuito. Master's thesis, Universitá di Bologna, October 2003. Masters Thesis Home Page — http://www.cs.unibo.it/∼milazzo.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I + II. *Information and Computation*, 100:1–40, 41–77, 1992.

[MS98] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. *Lecture Notes in Computer Science*, 1443:856–872, 1998.

[OW97] M. Ordersky and P. Wadler. Pizza into java: Translating theory into practice. In *24th ACM POPL*, 1997.

[SSS88] Seppo Sippu and Eljas Soisalon-Soininen. *Parsing theory. Vol. 1: languages and parsing*. Springer-Verlag New York, Inc., 1988.

[XDu] H. Hosoya, XDuce project. `http://sourceforge.net/projects/xduce/`.

[XML98] Extensible Markup Language (XML™), February 1998. XML 1.0, W3C Recommendation, `http://www.w3.org/XML/`.

[XS 00] XML Schema Part 0: Primer, W3C Working Draft. http://www.w3.org/TR/xmlschema-0/, 2000.

[XSL99] XSL Transformations (XSLT), 1999. http://www.w3.org/TR/xslt.

Ringraziamenti

Desidero innanzitutto ringraziare Lucian Wischik, che mi ha costantemente seguito durante tutto l'arco di tempo che mi ha visto impegnato nella realizzazione di questo lavoro e senza il cui prezioso aiuto questa tesi sarebbe stata certamente molto peggiore. Un sentito ringraziamento anche a Paolo Milazzo, con cui mi sono trovato a condividere la maggior parte del mio tempo in questi ultimi sei faticosissimi mesi, e che si é dimostrato essere un valido compagno di fatiche condividendo con me gioie e frustrazioni tipiche di chi affronta un lavoro di questa portata. La sua intelligenza viva e il suo continuo interesse verso gli aspetti piú problematici della mia tesi mi hanno spesso portato a uscire da imbarazzanti empasse, per questo non gli saró mai abbastanza riconoscente.

Ringrazio il Prof. Cosimo Laneve, Samuele Carpineti e Manuel Mazzara per le preziose discussioni che ho avuto con loro e che mi hanno permesso di mettere meglio a fuoco alcune problematiche relative a XDuce.

Desidero anche ringraziare Haruo Hosoya, l'autore di XDuce, che si é sempre mostrato estremamente disponibile con me rispondendo prontamente a ogni mia mail con cortesia e simpatia, e naturalmente lo ringrazio anche per l'ottimo lavoro che ha svolto per XDuce e per le spiegazioni puntuali che ho trovato nelle sue pubblicazioni e che mi hanno permesso di portare a termine questo compito.

Ringrazio tutta la mia famiglia, mia nonna Vanna, mia zia Filomena e i miei cugini Fabio e Daniele, per essermi stati sempre vicini in questi anni e in particolar modo mia madre Germana, la cui fiducia quasi cieca nelle mie capacitá ha permesso l'avvento di questo fatidico evento.

Ringrazio i miei amici, che hanno avuto la tenacia e la pazienza di sopportare il mio brutto carattere nell'arco di tutti questi anni senza mai cedere alla tentazione di mandarmi a quel paese. Cito i primi nomi che mi vengono in mente, Luana e Massimo, Ombretta (quando parleremo ancora di politica e religione?), M. Mazzara, Piero (grazie per i bei film visti insieme), Riccardo. Chi non si trova nella lista non se ne abbia a male, una lista esaustiva di tutte le persone che conosco e che vorrei ringraziare sarebbe troppo lunga per questa pagina dei ringraziamenti.

Il mio cuore va infine alle persone che non ci sono piú e che non possono condividere questo importante momento della mia vita. Dedico questa tesi a mia nonna Amedea e a mio padre, spero che ovunque siano mi stiano guardando e che possano essere orgogliosi di me.

A tutte le persone che ho nel cuore e a cui sono nel cuore. Ai passati e ai presenti. Grazie