N° d'ordre | 9 | 0 | 1 | 5 |

Langage de combinateurs pour XML
Conception
Typage
Implantation

Thèse

présentée et soutenue publiquement

le 7 Mai 2008

par

Kim NGUYỄN

pour l'obtention du titre de

Docteur de l'Université Paris-Sud 11
mention informatique

devant le jury composé de

Mme Véronique	Benzaken	directrice de thèse
M. Giuseppe	Castagna	co-directeur de thèse
M. Haruo	Hosoya	rapporteur
Mme Christine	Paulin-Mohring	
M. Peter	Thiemann	rapporteur
M. Philip	Wadler	

Remerciements

Je tiens à remercier en premier lieu mes directeurs de thèse, Véronique Benzaken et Giuseppe Castagna. Véronique en m'accueillant au LRI a su encadrer ma recherche avec la rigueur nécessaire, tout en me laissant une grande liberté de travail. Les expériences d'enseignement que j'ai eu a ses côté ont été un plaisir et une formation véritable et ont participé, au même titre que la recherche scientifique, à faire de ma thèse une expérience humaine et enrichissante. Giuseppe, quant à lui, a su prodiguer aide et conseils lorsque cela était nécessaire. C'est peu dire qu'il n'a pas ménagé sa peine lors de mon encadrement et en particulier pour la rédaction de cette thèse. Le travail d'encadrement accompli n'a d'égal que l'excellence du niveau scientifique dont il a pu me faire profiter. À tous les deux, je veux dire encore une fois merci. Le qualité scientifique, la grande humilité et honnêteté face à la recherche, le dévouement (il n'y a pas d'autre mot) et la chaleur humaine que vous avez témoignée lors de ma thèse sont à mes yeux sans prix. Je remercie également Alain Frisch, qui a encadré mon stage de DEA ainsi que le début de ma thèse. C'est une grande chance pour moi d'avoir pu poursuivre humblement les travaux qu'il a initiés avec sa thèse.

Je tiens également à remercier mes rapporteurs, Messieurs Haruo Hosoya et Peter Thiemann. Tous deux se sont acquittés de la lourde tâche de lecture du présent manuscrit. Ils l'ont effectuée avec une diligence que j'apprécie profondément.

Christine Paulin et Philip Wadler m'ont fait l'honneur de participer à mon jury, qu'ils en soient ici remerciés.

Je remercie mes collègues du LRI, de l'équipe BD ainsi que les membres de l'ACI Tralala, aux côtés desquels j'ai passé ces trois (bon ok, quatre) années de thèse. Parmi ceux avec lesquels j'ai collaboré, je tiens à remercier plus particulièrement Dario Colazzo, instigateur du travail sur les *type projectors*, avec lequel j'espère continuer à travailler dans une ambiance toujours aussi enjouée et amicale.

Je remercie finalement mes amis, Cédric, Sammy, Ignacy, Pierre-Yves, Alex, Émilie, Laurence, l'autre Laurence, Bertrand, Guillaume, Matthieu, ... Parmi les amis, l'un d'entre eux mérite une mention spéciale, mon vieux binôôôme Sylvain. Merci compère, pour l'amitié, la compétition et le cabernet d'Anjou.

Finalement cette thèse ne serait pas sans le soutien des très proches. Ma Laurette, son amour et son soutient sans faille (même pendant la rédaction, *surtout* pendant la rédaction). Mes parents pour leur affection, l'éducation qu'ils m'ont donnée, qui ont fait de moi ce que je suis et par la même de cette thèse ce qu'elle est; ainsi que pour leur confiance tranquille dans ce que je fais, même si je n'ai pas toujours pris le temps de leur expliquer (désolé !). Ma soeur Yohanna, à qui je dois entre autre d'avoir utilisé la fonte Dolly pour ce manuscrit (d'ailleurs si tu as une minute, on écrit « xml » ou « XML » ?) et qui, par sa réussite, me force à ne pas trop rester à la traîne.

À tous, un grand grand merci, vous m'êtes très chers.

Abstract

We study in this thesis the design of a programming language for XML. Many existing works tackle this problem: main-stream languages coupled with XML libraries (C, Java, ...), "standardized" languages for XML (XSLT, XQuery, XPath), specialized languages (XDuce, ℂDuce, XTatic, ...) and lastly, restricted formalisms (k-peeble tree transducers, macro-tree transducers, regular expression filters). There are many flaws in these current approaches : main-stream and standardized languages do not take XML types into account (they are very lightly typed at best), specialized languages suffers from a complex type-system (requiring the programmer to heavily annotate his/her code), do not feature type inference and provide poor code modularity. Restricted formalisms, which seem to best fulfill our expectations are neither practical (macro-tree transducers, k-peeble tree transducers) nor sufficiently expressive (regular expression filters cannot encode XPath expression for instance).

Our solution consists of a restricted language of combinators, dubbed *filters*, which features a precise typing policy, a type inference algorithm and are powerful enough to express complex XML transformations (XPath encoding, XSLT transformations, ...).

The first part of this thesis is devoted to the theoretical definition of the core language, its semantics and its type system as well as a type inference algorithm. In a second part, we study practical aspects of this language such as its embedding in a full-fleged language (ℂDuce) and give an encoding of a non-trivial fragment of XPath into filters. We also show how to use the typing informations to optimize the loading of an XML document.

Résumé

Nous étudions dans cette thèse la conception d'un langage de programmation pour XML. De nombreuses approches à ce problème existent déjà : langages généralistes avec des bibliothèques XML (C, Java, ...), langages « standards » pour XML (XSLT , XQuery, XPath), langages spécialisés (XDuce, ℂDuce, XTatic,...) et finalement, formalismes restreints (*k-peeble tree transducers*, *macro-tree transducers*, *regular expression filters*). Toutes ces approches souffrent de défauts : l'absence de discipline de typage « sérieuse » pour les langages généralistes et standards, un typage trop contraignant (nécessité pour le programmeur d'annoter lourdement le code, pas d'inférence de type, peu ou pas de modularité du code) pour les langages spécialisés. Les formalismes restreints, qui semblent répondre au problème sont soit difficilement utilisables en pratique (*k-peeble tree transducers*, *macro-tree transducers*) soit trop peu expressifs pour pouvoir exprimer des programmes intéressants (*regular expression filters*).

Notre solution consiste en un langage restreint de combinateurs, appelés *filtres*, qui possède à la fois une discipline de typage précise, de l'inférence de type et une expressivité suffisante pour exprimer des transformations XML complexes (encodage d'XPath, transformations XSLT , ...).

Une première partie de la thèse est consacrée à la définition théorique du langage, sa sémantique, son système de type ainsi que la définition d'un algorithme d'inférence de types. Dans une seconde partie, nous étudions des aspects pratiques, tels que le plongement de ce langage restreint dans un langage plus généraliste (ℂDuce) et donnons aussi un encodage d'un fragment non trivial du standard XPath dans nos filtres. Nous montrons aussi comment utiliser les informations de typage pour optimiser le chargement d'un document en mémoire.

Note : Afin d'en assurer une plus large diffusion, et en accord avec l'école doctorale d'informatique de Paris-Sud 11, cette thèse est rédigée majoritairement en anglais. Un résumé étendu en français des travaux se trouve en page 195.

Contents

List of Figures

List of Tables

List of Symbols

Part I

Introduction

Chapter 1

Introduction

1.1 The XML standard

SINCE the rise of network communications in everyday computing, there has been a real need for easy data re-use and easy inter application exchange. For example, in the recent years, with the so-called Web 2.0 (which basically consists of web pages with per-user *customized* content, sometimes called *social* web), it is not uncommon to see *aggregator* sites. Such sites fetch data from various sources, sieve them in an user-defined way and present them to the user in the form of a web page. Such kind of applications, now pervasive on the Internet, are based amongst other things on the XML (eXtensible Markup Language, [XML]) standard. In its purest form, an XML document is just a text file, with a structured content. More precisely, it is a textual way to describe *tree*-organised data. An example of XML document is given in Figure 1.1. This document represents an addressbook. It is constituted of:

- raw text (as in `0123456789`)
- markups (for example `<contact group="Work" >` and `</addressbook>`)

In a markup such as `<contact group="Work" >`, `<contact>` is called a *tag* (or sometimes a label) and `group` is called an *attribute*. Furthermore, `<contact>` is an *opening* tag, while `</addressbook>` is a closing tag. The standard does not impose the names for tags and attributes: it is up to the user to name them and give them a proper semantics.

The specification enforces indeed very few constraints on a document. Aside from some light constraints on the encoding used for the document (XML was designed with multi-byte character sets in mind, such as Unicode), the only constraint is for a document to form a tree, that is:

- There must be one tag enclosing the whole document (e.g. `<addressbook> ...</addressbook>` in Figure 1.1) which is the *root* of the document,
- An opening tag must be closed by the corresponding closing tag, or equivalently, tags must be well parenthesized.

Such a specification has the advantage of separating the semantics of the document from its textual representation. Indeed, it is possible to write generic libraries

```
<addressbook>

  <contact group="Work">
    <name>
     <first>Giuseppe</first>
     <last>Castagna</last>
    </name>
    <phone>0123456789</phone>
    <email>gc@pps.jussieu.fr</email>
  </contact>

  <contact group="Family">
   <name>
    <first>Yohanna</first>
    <last>Nguyễn</last>
   </name>
   <phone>0987654321</phone>
   <email>yoh@yoh.org </email>
   <address>
    <nb>4</nb>
    <street>Rue du Yahourt</street>
    <zip>75000</zip><city>Paris</city>
   </address>
  </contact>

</addressbook>
```

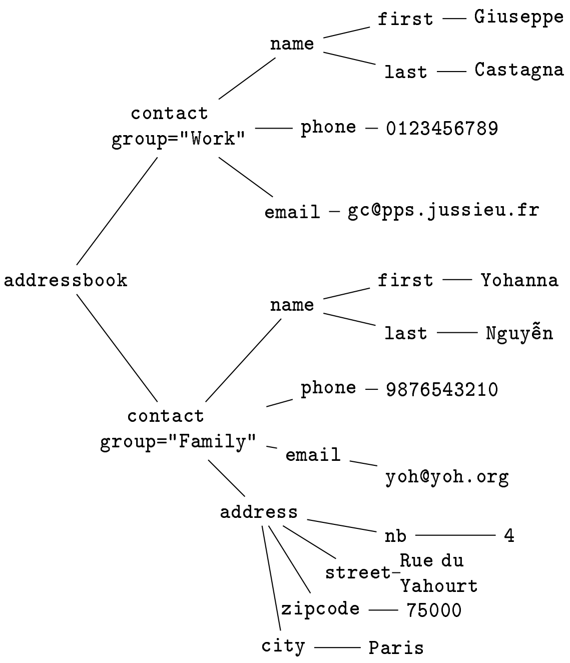


Figure 1.1: A simple XML document

to parse and print XML documents, without knowing anything about the meaning of the tags or the attributes. This feature has attracted many developers and formats based on XML are now common. For example, there is XHTML (an XML compliant version of HTML), the SVG file-format, used to represent vector graphics, the Open-Document format designed to store office suite files such as type-set documents, spreadsheets or presentations, but also many configuration file formats (the Apache and Tomcat web servers for example store their configuration in XML). Another use of XML is in the so called Web Services, that is, network applications which accept requests and send back their results encapsulated in XML. For example, many online selling sites publish a web service which can answer queries on their catalog. A client will make a request of the type “find all Jazz CDs that you sell”, and send it to the web service. The request is then processed by an underlying DBMS and the results are encapsulated in XML and sent back to the client. By using XML, all these applications delegate the parsing and printing of data to generic libraries.

There is of course a need to put some *semantic* constraints on a document. For example to specify the set of tags that can occur in a document. Fortunately, there exist many ways to describe such constraints. Additional pieces of information, such as DTD (Document Type Definitions, [DTD06]), Relax-NG ([Rel]) or XML-Schema ([XSc]) can be attached to an XML document. Such additional constraints can: restrain the set of possible tags, specify the order of appearance of tags, the content of a tag (e.g. a tag <a> must only contain elements of tag <b> or characters) and so on. It is then possible to check, via the use of a *validator* that a given document verifies its constraints. Figure 1.2 shows an example of such an XML type, namely, a DTD which accepts documents similar to the addressbook presented in Figure 1.1.

```
<!ELEMENT addressbook  (contact*)>
<!ELEMENT contact      (name, (phone|email)+,address?)>
<!ATTLIST contact      group CDATA #REQUIRED>
<!ELEMENT name         (first,last)>
<!ELEMENT phone        (#PCDATA)>
<!ELEMENT email        (#PCDATA)>
<!ELEMENT first        (#PCDATA)>
<!ELEMENT last         (#PCDATA)>
<!ELEMENT address      (nb,street,zip,city)>
<!ELEMENT nb           (#PCDATA)>
<!ELEMENT street       (#PCDATA)>
<!ELEMENT zip          (#PCDATA)>
<!ELEMENT city         (#PCDATA)>
```

Figure 1.2: An example of XML type (a DTD for addressbook documents)

This type consists of a set of definitions, one for each element that can occur in a document of that type. Definitions are conveniently represented as *regular expressions*. For example, we can see that an `<addressbook>` contains a possibly empty sequence of `<contact>` elements (denoted by the Kleene star `*`). The content of a `<contact>` element is itself a sequence of a `<name>` element, followed by a non-empty sequence (denoted by the `+`) of `<phone>` or `<email>` (alternation is noted "`|`") and an optional `<address>` (denoted by `?`). `CDATA` (in the attribute definition) and `#PCDATA` are raw text elements. People with knowledge of formal languages will immediately recognise DTDs as a syntax for *regular tree grammars*. And while other kinds of types such as XML-Schemas or Relax-NG are more expressive than DTDs, they all can be cast in the formalism of regular tree grammars (or equivalently of *regular expression types* which we will introduce later on). Checking that a given tree (a document) is a production of the grammar (a type) is then easy. For example it is shown in [SV02] that for some DTDs the document can be validated in linear time (with respect to the size of the document) and constant memory and that it can be done in the general case in linear time and logarithmic memory (by keeping a stack the size of which is at most the depth of the document).

1.2 Programming with XML

Specifying documents is one thing, but the core of the XML technology is the *manipulation* of such documents. Indeed, as XML documents are but a way to organise pieces of information, one needs to *query* the document to extract information, *process* the information to produce a *result* which then need to be *published*.

The simplest way to do so is to consider the XML document in its roughest form: a text file. This technique shows rapidly its limits as hand-writing a parser can be error prone and nullify the advantages of a generic format. Besides, document con-

types\values	−	+
−	C, Java, C♯, OCaml, Haskell, PHP,...	XQuery, XSLT, XPath, ...
+	OCaml (Ocsigen), JAXB, Scala, ...	XDuce, Xtatic, ℂDuce, XHaskell, OCamlDuce,XAct...

Table 1.1: Four categories of XML programming

straints (which we will from now on call the *type* of the document) are not considered, let alone enforced by such a low level of programming. It is doubtful that any serious XML-based program can be written in such way.

If we consider XML programming at a higher level then, in the same way as XML documents are twofolds —the brute content and its type— we can consider two aspects of a programming language, with respect to XML: whether or not it provides syntactical support to process XML documents (value side) and whether or not it can enforce document constraints (type side).

Both the type-system and the semantics of a programming language can be either of strong or weak flavour with respect to XML programming, as shown in Table 1.1.

The first category, is the one of generic programming languages. On the value side, those languages do not have any specific support for XML. They therefore rely on external libraries to perform the parsing, the manipulation and the printing of XML documents. Note that if it is possible to write any XML transformation in those languages, the best that can be done to ensure the validity with respect to a set of constraints is to use a *validating parser* to load a document and to validate every generated document afterwards, to ensure its correctness with respect to its type. Such development might lead to tedious debugging efforts as it is difficult in general to link an error in a generated document to the corresponding faulty line of code in the program.

Amongst the members of the second category, one finds for example the standards specified by the W3C. As surprising as it might seems, while the specifications for both types (DTD, XML-Schema,...) and languages (XPath, XQuery, ...) are very detailed, no typing policy is given for those languages which are often very lightly typed. However, as they provide syntactic supports for XML data as well as highly declarative constructs inherited from database programming languages, they offer a more comfortable way to code XML transformations.

The third category may puzzle the reader. How can a language which was not designed for XML (a generic language) have an XML aware type-system? The answer is that many languages have a type-system expressive enough to encode *some* XML type constraints thus giving added safety to the use of the previously mentioned XML libraries. For example, the Ocsigen project [Bal06, OCS] is a framework for developing web pages written in OCaml. Within this framework, XHTML type constraints are encoded into OCaml's polymorphic variants (via the clever use of phantom types: types that carry extra information unrelated to the values of this types to guide the type inference algorithm and to help it enforce extra constraints). Other tools (such as JAXB [JAX] or Castor [CAS] for example), create a *mapping* from a DTD (or a Schema) to a Java class. In a nutshell, they automatically generate a Java class hierarchy. This permits to use the type system of the language to enforce XML type constraints. However such technique (known as *data-mapping*) is limited, first in the sense that it lacks some flexibility (one has to regenerate a new class when the corresponding XML type is modified) and more importantly a type system that wasn't designed for XML might not be able to provide the necessary precision (it is a "better than nothing" approach).

Finally the last (but not least) category, is the one of statically typed languages designed for XML processing. As this category is the foundation on which we base our work, we present it more thoroughly.

1.2.1 Statically typed languages for XML

The safest (and in our opinion most elegant) way to program with XML is to "take types seriously". Indeed, in a type-system which is fully XML-aware, if one can check that a transformation (a function) has type $t \to s$ then it is guaranteed that the output of the transformation will always have type s, for any input of type t. In the case where s is an XML type (say the XHTML DTD for example), then the type-system *ensures* that the program will *always* output a valid XHTML document, without the need of further validation. This is the trend that has been initiated with Haruo Hosoya's work ([Hos00, HP01]) which led to the development of the XDuce language. In this work, Hosoya explains that DTD, Relax-NG or XML-Schema are all some concrete representations of *regular expression types*. Regular expression types are nothing else than regular tree languages and as such, have nice properties of closure under Boolean connectives, decidability of emptiness, membership and all the operations required to forge a type system (see for example [CDG$^+$97] for a complete survey on tree languages processing and recognition).

In XDuce, documents are first class values and types are regular expression types. We can see in Figure 1.3 a sample of code, defining a type and a document of that type (the same we used to introduce the syntax for XML documents and DTDs)[1]. The only novelty with respect to XML documents in this example is the use of the square-bracket notation to denote sequences of elements and the absence of closing tag (the end of an element being marked by the closing square bracket). With respect to types, we see that the *name* of the type can be different from the *tag* of the element thus allowing two different type definitions to share the same tag (with different contents). This distinction is absent from DTDs but is present in XML-Schemas for which XDuce types can be viewed as a formal representation.

In this setting, the first major contribution of XDuce, is the definition of *regular pattern matching*, a generalisation of *pattern-matching* as found in many (but mostly functional) programming languages, such as SML, OCaml or Haskell, which selects efficiently sub-parts of an input document in a precisely typed way. The other contribution of paramount importance is the definition of the so-called semantic sub-

[1] This code is actually ℂDuce code, the syntax of which is close to the XDuce's one and which we use throughout the rest of this manuscript.

```
type city = <city>[ Char* ]
type zip = <zip>[ Char* ]
type street = <street>[ Char* ]
type number = <nb>[ Char* ]
type address = <address>[ number street zip city]
type email = <email>[ Char* ]
type first = <first>[ Char* ]
type last = <last>[ Char * ]
type name = <name>[ first last ]
type data = <contact group=String>[ name (phone|email)+ address ? ]
type addressbook = <addressbook>[ data* ]

<addressbook>[ <contact group="Work">[
                 <name>[
                        <first>[ ’Giuseppe’ ]
                        <last>[ ’Castagna’ ]
                       ]
                 <phone>[ ’0123456789’ ]
                 <email>[ ’gc@pps.jussieu.fr’ ]
               ]
               <contact group="Family">[
                 <name>[
                        <first>[ ’Yohanna’ ]
                        <last>[ ’Nguyễn’ ]
                       ]
                 <email>[ ’yoh@yoh.org’ ]
                 <address>[
                        <nb>[ ’4’ ]
                        <street>[ ’Rue du Yahourt’ ]
                        <zip>[ ’75000’ ]
                        <city>[ ’Paris’ ]
                         ]
               ]
]
```

Figure 1.3: ℂDuce definitions for the `addressbook` type and a sample document.

typing, which is uniquely suited to finely check constraints on XML documents (we formalize both notions in Chapter 2).

XDuce has been further extended, in many different directions. First of all, one that is singularly apart from the others in that it embeds XDuce features in an *object oriented* language is the work by Benjamin Pierce *et al.* on Xtatic, which consists of the addition of XDuce regular expression types and regular expression patterns to C♯ (see [GLPS05]). While being of great benefit for a C♯ programmer and having led to several works on efficient implementation of pattern matching, the merge between XDuce notion of subtyping and C♯ presents no conceptual difficulty, since C♯ subtyping relation is rather poor (inheritance) and defined by the programmer; when a class inherits from another, both classes are added to the subtype relation.

Another extension of interest, in the context of which this thesis is placed, is the

ℂDuce language defined by Alain Frisch in his thesis [Fri04b]. In his work, Alain Frisch extends the notion of (semantic) subtyping to arrow types, and add (amongst other things), higher-order and overloaded functions to XDuce (in which functions were not first class values). He also generalized the notion of regular pattern matching, exposing all Boolean connectives to the programmer (intersection and negation of patterns and types, while XDuce only provides union to the user). Furthermore, he defined an efficient execution model for patterns, based on special kind of automata (dubbed NUA, for Non-Uniform Automata) enriched with type information [Fri04a]. In particular, these automata compute the matching of a value by a pattern *without backtracking* and by doing as few operations as possible.

All these languages achieved their goal to provide a way to statically and precisely type XML transformations. One may then wonder: is there anything left to do? Yes, there is.

1.2.2 Statically and precisely typed are not enough!

With the exception of Xtatic, all the previously cited languages are functional. For example, a seasoned ML programmer will use ℂDuce with ease... except in one aspect: the lack of polymorphism. Indeed, while some degree of polymorphism can be achieved via the use of overloaded functions (*ad-hoc* polymorphism) a more general notion of polymorphism is completely missing from ℂDuce. This is also true for XDuce[2] or even Xtatic for which the interaction between regular expression types and the kind of parametric polymorphism introduced by *generics* is yet to be studied. The lack of polymorphism leads to an heavily annotated code: every function has to be manually typed by the programmer. While this is certainly a cumbersome task, a greater issue is that it has a great impact on modularity and code reuse. However, before blaming the creators of these languages, we must ask ourselves, what is polymorphism, what kind of polymorphism do we need in the context of XML? Let us illustrate it with a paradigmatic example: list concatenation. Consider a `concat` function which concatenates two lists. In a type-system with parametric polymorphism, such as the one used in ML, such a function usually has type:

$$\texttt{concat} : \alpha\ \texttt{list} \longrightarrow\ \alpha\ \texttt{list} \longrightarrow\ \alpha\ \texttt{list}$$

where the α `list` type is inductively defined[3] as:

$$\texttt{type}\ \alpha\ \texttt{list} = (\alpha \times \alpha\ \texttt{list}) | \texttt{[]}$$

As usual, these types are universally quantified over the type variable[4] α , meaning that a list is either the empty list `[]` or the pair of an element of type α and α list. Such type variables can later (in the program) be instantiated with a type e.g. lists of integers with the type expression `int list`. While this definition of lists is satisfactory in the context of general programming, it is rather inadequate for the XML

[2] Or rather was true when XDuce first started. We present briefly in Section 1.4 the latest work on XDuce with polymorphism.

[3] We use the popular OCaml syntax, which is shared by many functional languages.

[4] We recall that such a type is to be red, $\forall\alpha.(\alpha \times \alpha\ \texttt{list})|\texttt{[]}$, with the so called *prenex quantification*.

type of x	type of y	type of `concat` x y
`[ Any* ]`	`[ Any* ]`	`[ Any* ]`
`[ Int* ]`	`[ Char* ]`	`[ Int * Char* ]`
`[ Int* ]`	`[ Int Bool? ]`	`[ Int+ Bool? ]`
...	...	...

Table 1.2: appropriate types for list concatenation

programmer. In the context of XML, one wants indeed to have heterogeneous sequences, containing elements of different types. For example, in the type definition of Figure 1.3, we see that the content of a `<contact>` element is the heterogeneous sequence `[name (phone|email)+ address ?]`. Unfortunately, in the case of ML, the α variable forces all elements to be of the same type. Such polymorphism is then too restrictive to be used for typing (even simple) XML operations. Of course a more general form of polymorphism coupled with subtyping (System F with subtyping for instance), could provide a more general type schema but for such systems type-checking is known to be undecidable ([Wel99]).

The situation for regular expression types is hardly better. Indeed, in the absence of polymorphism, one has to type the `concat` function as such:

$$\texttt{concat} : \texttt{[Any*]} \rightarrow \texttt{[Any*]} \rightarrow \texttt{[Any*]}$$

In the above type expression, `Any` is the upper bound of all types (with respect to the subtyping relation). A basic property is that any value has type `Any`. While heterogeneous sequences are available in XDuce, ℂDuce, ... the only way to write a generic concatenation function is to assume that the type of every element of the input lists is `Any`. This allows the programmer to pass any pair of lists as argument to `concat` but has the rather annoying effect of losing all type information, giving a result of type `[ Any* ]`. The behaviour of the type system which would really be wanted in our example is illustrated in Table 1.2. The first line in this table represents the instance where both arguments are generic, of type `[Any*]`. Of course in that case, the only sensible type for the concatenation is `[Any*]`. More interestingly, if we consider the second line, we see that in the case where the first argument is of type `[Int*]` and the second one of type `[Char*]`, then the most precise result is the set of lists that contain first a sequence of integers and then a sequence of characters and only that. This set is exactly the type: `[Int* Char*]`. The third example illustrate what kind of precision we expect from our type-system. In the case of concatenating a sequence of integers with a sequence of exactly one integer followed by an optional Boolean, then the result is a *non-empty* list of integers with the trailing optional Boolean.

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

A knowledgeable reader could however argue that ℂDuce or XDuce have proposed a precisely typed concatenation since the beginning. ℂDuce does so by providing an *hard-coded* and specifically typed `@` (read "concat") *operator*, which is not a function and has no meaning *per se*. An expression of the shape *x@y* is nevertheless very precisely typed (with the behaviour sketched in Table 1.2) by *evaluating the operator on the type of its inputs*, thus reflecting in its output type the precise computation that will be done at run-time. This is the key observation and the main basis of our work: when the semantics of an operator is known, it can be typed very precisely and the typing can be tailored for every specific input type. So to say, it performs an *abstract execution* of the operator on the input *type*.

This is what is done in languages such as Xtatic, ℂDuce, and XDuce which all provide several built-in iterators for sequences and XML-trees. But this approach soon shows its limits: while for an operation as simple as changing a tag, a predefined operator that iterates a given expression on an XML tree is available (e.g. `xtransform` in ℂDuce, `map` in XDuce, `foreach` and `iterate` in Xtatic,...), for slightly more complex —but fairly standard— manipulations (e.g. context sensitive document pruning, or the cleaning of XHTML documents to cope with "XHTML-deprecated" elements) this is not the case. The programmer is then left with two choices. Either ask the language maintainers to add more of these iterators, which at one point, the language maintainers will refuse to do for obvious reasons[5], or hand-write its own iterators as recursive functions, hence typing them explicitly and thus losing in modularity and code reuse. For what is worst, XML types can sometimes be huge, and writing down such a type might not be an easy task. If we consider the transformation which removes every hypertext link from an XHTML page, we see that even if a programmer wishes to type this function by hand, (s)he finds itself in the situation where (s)he must:

- first create the type corresponding to the XHTML DTD. Granted that it is a fairly common type, we can assume that the type is already defined somewhere in a library;

- secondly, the programmer must create a type XHTML_MINUS_A which is the same type as XHTML without any occurrence of the <a> element. As this type is not very common, the programmer has no choice but to enter it by hand (the XHTML DTD defines around 300 different elements).

[5] Indeed, adding a new iterator means interfering with all parts of the language: its syntax (adding new keywords to a language is always an hassle as it might break existing code), obviously the type-checker which must be extended with the new specific typing rules and of course the code generation process, in which the addition of new features can have intricate consequences on previous optimizations, the whole thing transforming a compiler into a maintenance nightmare.

The programmer is then left with two bad options. (S)he can give its function the type XHTML →XHTML, which is simple to write but will not reflect in the type of the function the fact that the output document is <a>-free. If the code of the function is buggy and leaves at some place an <a> tag, the type-system will not detect it. The programmer can also give the function the type: XHTML →XHTML_MINUS_A, in which case (s)he must write by hand the output type, which can be error-prone and not realistic if one considers larger types (the DocBook DTD for example).

1.3 A solution

A possible solution, along the lines of which we present this thesis, is to define a small language of combinators, expressive enough to write complex operations over XML documents but simple enough to type them precisely. Differently said, a language of *non first-class* operators which are not typed (or are just lightly typed) at their definition but, rather, are very precisely typed at the places of their application. Such technique has been explored by Haruo Hosoya by adding *regular expression filters* ([Hos04]) to XDuce . In such a framework, the combinator language is parametrized by the expressions and the patterns of the host language. This allows a programmer to iterate a particular expression, in a customized way, and to capture variables during the process which can be further used to compute the result.

While such a solution might seem simple and natural, designing such a language is not. Indeed, the language must meet the following conditions:

1. It must be able to call any expression of the host language and therefore its design must be independent from a particular host language.
2. It must be statically typed. This has two consequences on the type system which must be able (i) to associate a domain type to each iterator, that is a set of expressions for which the iterator will not fail (so that, say, an iterator for lists cannot be applied to an XML tree) and (ii) it must be able to deduce a precise type for the output by *running the iterator on the type of the input*.
3. A consequence of (ii) in the previous point is that the language must define only iterators that always terminate even when applied on (infinite) types. More precisely, the abstract execution of any iterator on an (input) type —thus the type checking phase— is required to terminate (therefore the application of an iterator to some data may diverge only either because it called a diverging expression of the host language or because it was applied to infinite data).
4. It must be expressive enough to define common sequence and tree operators such as concatenation, reversal, map-functions, various tree-explorations, XPath expressions, and so on.
5. It must not come at the cost of modularity and code reuse.

Of course there is a clear tension between requirements 3 and 4: expressive power and termination are contrasting requirements, therefore a trade-off must be found between them. Indeed between simple *map*-like iterators (that is applying a local transformation to every node of a tree or a list) and Turing-completeness there are a whole lot of possible class of transformations to choose from. Considering that *map*-like iterators were insufficient to deal with XML, we state that the minimal requirement for such a language is to be able to flatten a tree, that is return the list of all its content. With this minimal requirement, we must however accept that forward type-checking cannot be *exact*. Indeed, it is a well known property of tree languages (i.e. our types) that they are not closed by homomorphism (see [CDG^{+}97]). As an example, let us consider the following recursive type:

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

This type is a sum (or union, noted `|`) of either the singleton `[]` or a list of three elements, the first being the singleton `<a>[]`, the third the singleton `<b>[]` and the second T itself. Such a type is regular. However, when applying the flattening operation to a value of this type, the exact set of values that contains all the possible results (and nothing else) is the set: $\texttt{S} = \{\texttt{[<a>[]}^{\texttt{n}}\,\texttt{<b>[]}^{\texttt{n}}\texttt{]} | n \geq 0\}$ which is not regular. Also of interest is the fact that there is not, in general, a *better* regular approximation for such types. For example, the previously mentioned set of values can be approximated by the $\texttt{S}_i$ types hereafter where $\texttt{S}_{i+1}$ is strictly more precise than $\texttt{S}_i$:

$$\begin{array}{lcl}
\texttt{type S}_0 & = & \texttt{[<a>[]* <b>[]*]} \\
\texttt{type S}_1 & = & \texttt{[] | [<a>[] <b>[]] | [<a>[] <a>[]+ <b>[] <b>[]+]} \\
 & \vdots & \\
\texttt{type S}_n & = & \texttt{[] | [<a>[] <b>[]]| ...| [<a>[]}^{n}\texttt{<a>[]+ <b>[]}^{n}\texttt{<b>[]+]} \\
 & \vdots &
\end{array}$$

As a side note, even the use of pattern matching cannot be *exactly* typed in the presence of set-theoretic subtyping . Indeed, while patterns themselves can be exactly typed (as in the case of ℂDuce's patterns), the whole `match with` constructs yield an approximation. Consider the expression:

```
match e with
x -> (x,x)
```

which returns a pair where both component are the input expression `e`. If `e` has type `Int`, then the compiler will compute an *exact* type for the pattern `x` which is `Int` but will compute the type (`Int` × `Int`) for the expression `(x,x)`, which is only an approximation of the exact set of values returned by this expression. The exact *output type* in that case would be $\{(n,n)|n \in \texttt{Int}\}$ which is not a regular set (it is not a regular tree language). This approximation is however sufficient in practice.

1.4 State of the art

In this thesis, we define a small language of combinators, dubbed *filters*[6] which are expressive enough to write complex transformations but simple enough to allow precise type inference. Filters are then grafted into the ℂDuce language —pretty

[6] We borrowed the term "filter" from Hosoya, and the similarities do not end here.

much like Hosoya's filters are embedded into XDuce—[7], and allow the programmer to define its own polymorphic and precisely typed iterators. It should be noted that comparison between our work and Hosoya's work on regular expression filters are numerous in this thesis, as it is the formalism closest to ours.

In order to better describe the contributions of our thesis (see Section 1.5), let us first quickly overview existing formalisms for XML whose design targets expressiveness, modularity, and type precision. The solutions proposed in the literature to mediate among these characteristics can be roughly divided in four categories.

1. Languages with polymorphic types for XML
2. Languages with hard-coded operators
3. Languages with generic XML iterators
4. Languages based on tree automata theory

1.4.1 Polymorphic type-systems for XML

There exist various attempts to mix XML types and parametric polymorphism. The parametric polymorphism currently available in XDuce mixes explicit type annotations with well-localised type reconstruction [HFC05]. Indeed, the key point in mixing parametric polymorphism with regular expression types is the interaction between pattern-matching and polymorphic values. The approach taken by [HFC05] relies on *value marking*. In a nutshell, if a value inhabits a certain polymorphic type, the sub-parts of the value corresponding to the type variables are *marked* with those type variables. Pattern-matching and more generally the language semantics are defined so as to preserve the marking of values. The type-checking of a function (with polymorphic input and output types) can then be performed as follows: verify that given a value marked according to the input, the body of the function would result in a value whose marking corresponds to the polymorphic output type. This technique nicely fits the semantics of XDuce for which pattern matching can perform run-time type checks. Here, a polymorphic value can be visited via pattern-matching provided that it is sufficiently annotated.

Of similar flavour, but following a completely different approach, is the work by Jérôme Vouillon [Vou06] where explicit polymorphism is designed so that pattern matching does not break parametricity. While it seems more restrictive than the previous approach, as one cannot inspect a polymorphic value, it ensures that pattern matching can be compiled statically, without requiring any run-time type-check. It also impacts the internal representation of data, as in this system the *boxing* of values (marking of a value so as to know its type at run-time) is unnecessary thus yielding an efficient data model.

[7] The key point is that, to be usable in practice, XML document processing must be done in a *full-fledged language*. This implies that, besides being Turing complete, the language must provide many facilities such as pretty-printing, error detection, access to external libraries, access to input/output devices and network, etc. This is why our filter language is meant to be grafted on a host language rather than used stand-alone.

A different approach consisting in the coexistence or juxtaposition of both XML and ML type systems in a same language [Fri06] is available and actively maintained for OCaml. While this eases the writing of polymorphic functions on XML values, this solution does not solve the problem of writing precisely typed operators. Indeed, both type systems (ML and XDuce) are kept apart, and a value is either seen as on the ML side—and can then be polymorphic—or on the XDuce side—and can then be precisely typed (with XDuce pattern matching for example). Considering the example of the `concat` function presented in Section 1.2.2, in OCamlDuce, one can specify either that it has type α `list` → α `list` → α `list`, thus being on the OCaml side, or type `[ Any∗ ]` → `[ Any∗ ]` → `[ Any∗ ]` and therefore on the XDuce side. Simply put, it does not allow the programmer to express more than either XDuce or OCaml but only regroups both of them in a coherent framework.

Finally, in the same spirit of combining two type systems, a more general approach was defined by Martin Sulzmann and Kevin Zhuo Ming Lu [SL06a] for Haskell where the authors mix Haskell type classes with XDuce regular expression types into a system called XHaskell [SL06b]. They provide a semantics via a type-directed rewriting of the language into System F. While the decidability of the general version is not clear, some restrictions make it tractable and lead to an implementation of this work using the GHC Haskell compiler as a back-end. Type safety is granted, but the programmer is required to heavily annotate the code: in particular, every polymorphic variable that is instantiated with a *regular expression type* has to be explicitly annotated. A common trait in all those approaches is that a polymorphic value either is never visited (through pattern matching for example) and so is never precisely typed, or if it is visited then it loses its polymorphic nature and becomes monomorphic and precisely typed. While this eases the writing of generic function over XML values it does not address the problem we study here, that is to have both precision and polymorphism.

1.4.2 Hard-coded constructs

As previously mentioned, ℂDuce, XDuce, and Xtatic rely on hard-coded constructs to perform XML transformations. We briefly review the generic iterators found in these languages, explain their semantics, and show why they are insufficient. First of all, an operation of interest, pervasive in the functional world as well as in the XML world is *list mapping*, which applies a transformation to every elements of a list and returns the list of results. To that end, ℂDuce, XDuce, and Xtatic all provide an iterator over sequences, named `map` (or `foreach` in the case of Xtatic). This iterator provide a very precise typing of sequence transformations, in particular with respect to their size. For example, in ℂDuce, the following expression

```
map l with
| x -> string_of x
```

applies the `string_of` function to every element of the list `l`. The `string_of` function is an universal conversion function which transforms any value into its string representation. If `l` has type `[Int∗]` then the output type for the transformation will be `[String∗]`, but if the `l` has type `[Int∗ Bool?  Char]`, then the output type will be `[String+]` (since there is at least one element in `l`, the one with type `Char`). The counterpart of this precise typing is that elements of the input sequence cannot be "ignored" and discarded while the transformation is iterated over the input sequence. This operation, known as list filtering, is however quite important. ℂDuce proposes another construct, `transform`, which allows one to discard elements of the input sequence if none of the patterns in the transformation matches an element. The inconvenience with `transform` is that, while its semantics is quite close to the one of `map`, it needs to be typed a bit differently, requiring the addition of a specific typing rule in the language.

Programming with XML also requires iterators that can explore a document (a tree) *in depth*. ℂDuce generalizes the notion of `transform` to XML trees with the `xtransform` iterator. As with `map` and `transform`, `xtransform` iterates on an XML document a transformation given by a list of *branches* of the form $p \rightarrow e$ where p is a pattern and e an expression. There are two cases encountered during the evaluation of an `xtransform`. Either a pattern matches an element which is transformed according to the corresponding expression, or the element is not matched, in which case, the transformation is recursively applied to its children. While this iterator can be of certain use, it fails to cover many interesting cases because the recursion stops at the first match. For example, the following expression:

```
xtransform [ <a>[ <a> [] ] ] with
| <a>x    -> [ <b>x ]
```

transforms any `<a>` into a `<b>` but stop at the first match (in depth) hence giving the result `[ <b>[ <a>[] ] ]`. Writing a transformation which would give the expected result `[ <b>[ <b>[] ] ]` is not possible in ℂDuce, without resorting to a recursive function and hence having to type it explicitly.

Of particular interest, in this respect, is the `iterate` construct of Xtatic. This construct is based on an *ambiguous* interpretation of XDuce's patterns. Usually, in order to bind a variable to an exact sub-tree of the input, pattern matching is defined so as to be *unambiguous*. The observation made by Vladimir Gapeyev and Benjamin Pierce in [GP04] is that, instead of a *single-match* policy where exactly one sub-tree is associated to a variable, one can consider a *collect-all-match* policy where all the possible matchings for a given variable are accumulated (in a sequence). This makes it possible to provide an encoding of XPath expressions into patterns and to use them in a typed context. An example of Xtatic expression using the `iterate` construct is for example:

```
iterate (doc) matching x at path [[a//b/c]] {
  print(x);
}
```

This code binds the variable `x`, in turn, to any sub-tree `<c>` in `doc` that matches the XPath expression `a//b/c`. While this powerful construct allows one to precisely type an XPath expression, which is a most wanted feature, its semantics is to *collect* elements and then operate on them, not to *map* a tree into another.

To conclude with hard-coded operators, it should also be noted that they do not cope well with code reuse. Consider the following ℂDuce expression:

type of l	output type
[Int+]	[Bool+]
[Bool∗]	[(0|1)∗]
...	...

Table 1.3: Possible input and output types for map.

```
map l with
| (x&Int) -> x != 0
| ‘true -> 1
| ‘false -> 0
```

which, when applied to a sequence of either integer or Boolean values, replaces any non-null integer by ‘true, zero by ‘false and conversely replaces ‘true by one and ‘false by zero. Table 1.3 describes the possible types for l and for this expression. If this transformation has to be used at several places in the code, then it must be duplicated every time. Indeed, it cannot be encapsulated into a function, which would have to be typed explicitly, with e.g. the type [(Int|Bool)∗] → [(0|1|Bool)∗]. Code written this way is clearly harder to maintain and debug. This is one of the main downsides of hard-coded iterators, making them an unsatisfactory solution for large pieces of code.

1.4.3 Iterator languages

Our solution to propose a sub-language that can be precisely typed and merged with a generic language (ℂDuce in our implementation), raises the question of what sub-language to use. Indeed, there are many specialized sub-languages already available in the XML field. First of all, XPath ([XPa]), is the standard language to navigate in an XML tree. However XPath is merely a query language and does not allow one to transform a document into another, but only to extract part of it. It can be seen as the counter-part of XDuce's patterns. Another, more powerful language is XQuery ([XQu]), based on the FLWR (For Let Where Return) paradigm. XQuery provides basic constructs to select sub-parts of a document (by the use of XPath expressions), iterate through them (by using the for construct) and construct new values based on the extracted data. As such it seems to be a good candidate for a transformation language. However XQuery is also very tied to the XML data model, which for example specifies that each node in a document has a unique ID. It also allows (through XPath) to navigate from a node to its parent, siblings, descendants and so on, making the internal representation of the XML document a cyclic structure. These features do not fit the data model used in functional languages such as ℂDuce or XDuce, where XML documents are trees (and not graphs) and in which only forward navigation is allowed (from a node to its children). The concept of a query language grafted in a functional language has nevertheless been studied, particularly by Cédric Miachon who proposed ℂQL in his thesis ([BCM05, Mia06]). ℂQL couples the *select-from-where* iterator (popular SQL idiom, similar in spirit to XQuery's *for-*

let-where-return) with ℂDuce pattern matching (as opposed to XPath expressions in XQuery). ℂQL also provides syntactical support for XPath like expressions, which are rewritten into patterns. While providing a highly declarative and optimized (by the intensive use of ℂDuce's efficient patterns) construct to manipulate XML data, ℂQL suffers some flaws. First of all, while fulfilling its duty as a "query" language, it is still a very strict and hard-coded construct. The programmer cannot change the order of traversal of the document, nor can (s)he use it to perform non-local transformations like swapping two unrelated sub-trees of a given document. Secondly, the XPath encoding is limited to one-step paths while general XPath expressions are composed of many steps. It does not also strictly respect the semantics of XPath (e.g. the same sub-tree can be returned twice in the result set, which is forbidden by the XPath specification).

1.4.4 Tree-transducers, backward type inference

Drifting away from the standards, we find a quite rich literature of combinators for XML, mainly from the tree automata theory. Most of them have the appealing property to be *exactly* typable by the so-called *backward* type-checking technique. In this technique, given a transformation f and an *output* type s, one computes the largest (in the sense of inclusion) type t such that (with informal notations):

$$f(t) \subseteq s$$

In practice, this consists in computing:

$$t = f^{-1}(s)$$

As tree languages are closed under inverse homomorphism, it is guaranteed that such an *input* type is always regular (thus solving the "flattening" problem described in Section 1.3). This technique was introduced, in the context of XML, with the k-pebble tree transducers, for which Tova Milo *et al.* exhibited a type-cheking algorithm [MSV03]. This effort was later pursued by Akihiko Tozawa [Toz01] who proposed a backward type-checking algorithm for XSLT0, a subset of XSLT without XPath expression nor reference (variables). More recently, Macro Tree Transducers (MTT), have been of particular interest in the context of XML processing. Introduced by Joost Engelfriet ([EV85]), MTTs have proved to be a suitable formalism to encode XML transformations. Indeed, Helmut Seidl *et al.* have developed a backward type checking algorithm for MTTs ([MBPS05]). The appealing side of MTTs is that, although they are not Turing-Complete (as they are *tree-transducers* —tree automata that not only recognise a tree but also output a value— which are not Turing-Complete) they are quite expressive due to their use of *accumulators*. The common trait in all these techniques is that they are exact and do not require type annotations (except for the original output type which must be given). The downside is the high complexity of the type-checking process, which makes it impossible to use in practice. Some progress has however been made recently. Of notable interest are the contributions of Helmut Seidl, Thomas Perst and Sebastian Maneth ([MPS07]) where MTTs are shown to be typable in polynomial time provided some (rather heavy) restrictions on their accumulators. The recent work of Alain Frisch and Haruo Hosoya [FH07] provides a more general enhancement as well as an efficient implementation allowing to type-check *small* MTTs on *real life* types (such as the XHTML DTD). However, practical implementations that allow to tackle realistic transformations (such as complex XPath expressions or full sized XSLT style sheets) are yet to be seen. Another problem, in the context of backward type inference is the integration in a generic language, the latter being typed in the "forward" direction. The interaction of both types of inference is unclear.

One may then wonder (again), why a new algebra? Why didn't we used forward type-checking on MTTs or XSLT0? Well, we did actually. While our *filters* are derived and were designed as an extension of ℂDuce's pattern matching operators, they are nothing but *top-down* tree transducers, and provide, as we will see in the presentation of the semantics, the same standard actions as MTT, XSLT0 terms or Hosoya's regular expression filters, that is:

- the ability to reconstruct a value,
- the ability to conditionally execute an action based on the tag of the input,
- the ability to iterate through the children of an element,
- the ability to perform recursive calls.

The main difference between our approach and the others being that in our case, the lack of accumulators (as those found in MTTs) is balanced by the presence of both:

- the use of ℂDuce pattern matching (allowing in-depth capture of sub-trees, testing for complex condition on a sub-tree and so on)
- a *controlled* use of *composition*, which allows us to encode XPath expressions as well as flattening of trees or non-local transformations (which XSLT0 or regular expression filters cannot perform)

We also provide a precise (while unfortunately not exact) way to type our *filters* and an implementation in the ℂDuce compiler.

1.5 Contributions

The thesis consists in the study of the filters sketched at the end of the previous section, and is organized as follows. The rest of Part I contains a chapter devoted to notations. Part II presents the formal algebra of filters, their semantics as well as a type-system and a type inference algorithm. Part III presents the implementation as well as some interesting extensions to the core algebra. Finally, Part IV concludes the study with a discussion on possible extensions and future work. The contents of Part II and Part III are sketched hereafter.

1.5.1 Filters and their semantics (Chapter 3)

Starting from ℂDuce patterns [FCB02] (which, as previously mentioned, are an extension of XDuce patterns defined in [HP01]), we generalize them to combinators, thus allowing the programmer to iterate patterns and expressions over documents. More precisely, as pattern are used to decompose and *capture* parts of a value, we design filters to decompose and *transform* a value. If we think of patterns as tree automata (which is indeed the compilation target for patterns), then filters are nothing but *tree transducers*: they are used both to recognize (iterate) over an input and to transform it into a new value.

The calculus of combinators we devise is expressive enough to perform: generic operations on sequences and XML trees (reversal, concatenation, flattening, mapping,...) but also to encode a forward fragment of XPath (i.e. with only descendant and child axes), as well as XSLT like transformations. This is achieved by the use of a *composition* operator. Such an operator is completely absent from Hosoya's filters which can thus only express *map-like* transformations. To see exactly where we are headed, let us give a flavour of filters, namely the concatenation filter:

Example 1.1
The concatenation filter:

$$\begin{array}{rcl} \texttt{concat} & = & (x,y) \to (x;f) \\ f & = & \texttt{`nil} \to y \\ & | & (z \to z,f) \end{array}$$

is equivalent to the following OCaml function:

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

While the filter expression for `concat` may seem a little cryptic, it is best understood in the light of the behaviorally equivalent OCaml function. Filters are applied to a unique input value. The first filter, $(x,y) \to (x;f)$, is a pattern filter for which the left-hand side of the arrow is a ℂDuce pattern, here (x,y), and the right-hand side a filter. It should be noted that the scope of the variables in the left-hand side pattern extends to the whole right-hand side, hence x and y are available in f. Equivalently in the OCaml code, the function `f` is nested in the definition of `concat`, and consequently `x` and `y` are visible in `f`. The right-hand side of the pattern filter is the *composition*, denoted by ";", of two filters. Let us break this down into several steps. The composition of two filters, $f_1;f_2$, applied to a value v, merely consists in the computation of $f_2(f_1(v))$. Although it is a natural construct to think of, we will see that composition must be handled with care so as to ensure termination of filters. Back to our examples, the two composed filters are an expression filter, x

and the filter f, which we are recursively defining. An expression filter is simply an expression of the host language, used to *output* a value. The whole composition here is equivalent to the application `f x` found at the end of the OCaml function. As for the recursive filter f, it consists of the union "|" of two filters; this pretty much resembles the two branches of the pattern matching found in the OCaml version of the function. The first branch of the union is the base case of the recursion: if the input is the empty list `'nil`, then it returns the second list y. If the input is a pair (CDuce sequences are encoded as nested pairs), then $z \rightarrow z$ is applied to its first component (the head of the list) , f is recursively applied to the second component. $z \rightarrow z$ is just the identity. At the end, the pair of the two results is returned.

This small example presents the whole algebra of filters: ground expressions, pattern capture, pair deconstruction and reconstruction, composition, alternation and regularity (recursive call).

1.5.2 A type-system for filters (Chapter 4)

As sketched previously, the typing policy for filters is quite uncommon. Rather than typing the definition of a filter, as one would do with a function, we type them at the *place of their application*. Besides, the precise type of their output is computed by an *abstract evaluation* (in the sense of abstract interpretation [CC77]) of the filter on the type of its input. The main problem with this approach is that, for some filters, the exact set of output values is not a regular type. Instead of choosing a particular approximation, we first present a type-system which, given an filter and an input type, infers all the possible regular approximations of the output type. More precisely the type-system is given as a set of inference rules that prove judgments of the form:

$$\Gamma \vdash f(t) = s$$

stating that in a type environment Γ, the filter f applied to an input of type t returns a result of type s. If we take the example of the concatenation given in the previous section, our system is able to infer:

$$\varnothing \vdash \texttt{concat}\big((\texttt{[Int}*\texttt{]} \times \texttt{[Bool}*\texttt{]})\big) = \texttt{[Int} * \texttt{Bool}*\texttt{]}$$

We show the type safety of filters equiped with this type-system, that is, that a well typed application of a filters to a value never fails. We also show how some variations of the type system impact on the precision of the inferred output types. In particular we show that not only our filters are more expressive than Hosoya's regular expression filters, but also that they are typed more precisely.

1.5.3 Type inference algorithm (Chapter 5)

The downside of the precise type-system we devised for filters is that it cannot be directly turned into an algorithm. Indeed, as mentioned at the end of Section 1.3, for some filters an input type, there might not exist a *best* regular approximation. While in our type-system all of these regular approximations can be derived, an algorithm would have to pick one (that is in practice, we want to return one output type for a given input).

We address this problem by adding *type annotations* to filters, that is we decorate the syntactic tree of the filter with type expressions. For instance, to specify that the output of the `concat` filter must be of type `[Int∗]`:

Example 1.2

$$\begin{array}{rcl} \texttt{concat} & = & (x,y) \to (x;f_{\{\texttt{[Int*]}\}}) \\ f & = & \texttt{`nil} \to y \\ & | & (z \to z,f) \end{array}$$

Of course, in this case, an annotation is not needed as the algorithm can infer the exact output type, but when it cannot, we let the programmer specify which approximation to use, by the mean of a type annotation. We present an algorithm which given an input type and an annotated filter computes the output type. We show that it is sound, with respect to the type-system and complete *up-to annotation*, that is if the type-system is able to infer a type and if we annotate the filter accordingly, then the algorithm will find the exact same type.

We also pinpoint precisely where annotations are needed so as to keep them as minimal as possible. We show that in practice, annotations are very light and needed only in a few particular (alas interesting) cases.

1.5.4 Concrete language (Chapter 6)

The motivation for the formal calculus of filter is ultimately, to provide a practical language to deal with XML transformation. To achieve this, we study (and provide a prototype of) the integration of filters into ℂDuce. First of all, we provide a concrete syntax for filters. The following example illustrates how the `concat` filter is written in ℂDuce:

Example 1.3

```
let filter concat =
 $  (x,y)  -> ( ~{x} ; (
                            let filter f =
                             $  []  -> ~{ y }
                           | (   $  z  -> ~{ z }, f )
                           in f ))
```

Besides the syntactic delimiters, `$` and `~`, used only to disambiguate patterns, filters and expressions, the novelty is a `let filter` binding, used to define recursive filters in a very natural way (pretty much like a recursive function).

Of course one of the key point is the specification of the annotations. While decorating the syntactical tree of the filter is acceptable for the formal study, it is not acceptable in the concrete language. Indeed, while annotations are (in our opinion) very light, marking the *code* of the filter itself with type annotation would break the modularity and reusability we aim at, since different input type might require different annotations. Our idea is to perform "late annotations" in the same sense as we perform "late typing": annotations (and typing) are performed at *the place* of the application. A sample of ℂDuce+𝔽ilter code is given in Figure 1.4. Even if not clearly understandable right now, it allows us to shed some light on the matter at hand.

```
  type T = [] | [ <a>[] T <b>[] ]
  type S = [<c>[] <d>[] ] | [ <c>[] S <d>[] ]


let filter flatten =
  $ [] -> ~{ [] }
     | $ ([ Any* ] ,_) -> (( flatten , flatten ) ; concat)
     | ( $ x & (Any ∖ [Any*]) ->~{ x } , flatten )


let v1 = (∗ some value of type T ∗)
let v2 = (∗ some value of type S ∗)
(∗
 ... other definitions
∗)

let r1 = apply flatten to v1 where {| flatten = [ <a>[]∗ <b>[]∗ ] |}
let r2 = apply flatten to v2 where {| flatten = [ <c>[]+ <d>[]+ ] |}
```

Figure 1.4: A sample of ℂDuce+𝔽ilter code

At the beginning of the code, one sees the type `T` presented in Section 1.3, and a variation `S`. Below the type definition, one finds the concrete syntax for the `concat` filter, introduced in Section 1.5.1. Next, the problematic flatten filter is defined. The latter is an example of filter for which annotations are needed. Surprisingly, we can see that its definition is annotation-free. If the filter was annotated to deal with inputs of type, say `T` then it would not be possible to use it on a value of type `S` and *vice versa*. The solution we propose is to specify the annotations at the place of the application, *i.e.* at the place of the `apply_to` construct. More precisely, the clause `where` provides a mean to bind a filter name to a type annotation. In our example, the flattening of a value of type `T` is annotated by `[<a>[]∗ <b>[]∗ ]` while the flattening of a value of type `S`, which cannot be the empty sequence, is annotated by `[<a>[]+ <d>[]+ ]`. As we see, two completely different annotations can be given without code duplication.

Rather than simply implementing our core filter algebra into ℂDuce, we extend it and use it as a mean of studying many language design issues. For instance, to enhance code modularity, we introduce macros filters or filters taking other filters as argument (without nevertheless providing true higher-order filters with respect to typing). We also introduce regular expression filters à la Hosoya and add some more "atomic" filters, such as the first and second projection of a pair which allow the programmer to write more filters without composition, and thus without any annotations.

We also discuss the behaviour of the typing algorithm in practice for fairly complex types (such as the DTDs of XHTML or DocBook) and transformations. In particular, the specific ℂDuce constructs we described in Section 1.4.2 such as `map`, `transform` and `xtransform` can be typed without annotations hence simplifying the type-checking part of the ℂDuce compiler by removing their specific typing rules. Finally, we describe our compilation scheme and some minor run-time optimizations.

1.5.5 XPath encoding (Chapter 7)

XPath expressions are a most wanted feature for the adoption of a language by XML programmers. We provide an encoding of a forward fragment of XPath with predicates into filters. We show that filters can be used to statically type-check XPath-based queries. Indeed, even if the encoding of an XPath expression into a filter requires an annotation, we show that we can derive this annotation from the XPath expression and the input type. The algorithm, unsurprisingly performs an abstract evaluation of the XPath expression over the input type and compute an approximation of the result. Using this technique to infer the annotation for the filter, we provide a *completely automatic* way to type forward XPath expressions. We also show that, using a clever rewriting, many conditions can be statically checked by pushing them into ℂDuce patterns.

1.5.6 Static pruning and typing of XQuery (Chapter 8)

Encoding XPath expressions into filters results in an interesting byproduct: the annotation-inference algorithm. We show that by applying the strategy of "computing the output type by executing a transformation on an input type" we can gather information that optimises the loading of XML documents. We present a a more complex version of the annotation inference algorithm and use it to prune XML documents according to XQuery queries: given a DTD and a query, we can statically apply the latter on the former (that is make a symbolic computation of the query on the type, as we do for filters) and produce a *projector*. Projectors are operators that are used at run-time to determine which part of a document to load in memory. The interest of this technique is to leverage the complexity (in memory) of XML query engines that rely on the DOM data-model. Indeed in many current implementations, even for small documents (for instance 50 MB), the in-memory representation of the document may amount to several hundreds of megabytes sometimes and this makes it impossible to evaluate even a simple query. We use type projectors to help a query engine to retain only the part of the document which is necessary to compute the query and nothing else. We focus on the projector inference algorithm and its implementation as well as experimental results. This chapter is based on the work published in [BCCN06], initiated by Dario Colazzo. Our contributions in this work consists of the implementation and experimental results, the proofs of the main theorems as well as the design of the type inference algorithm.

Publications

Chapter 3 to 6 are an extended version of [CN08]. The content of Chapter 8 was presented in [BCCN06].

Chapter 2

Notations

> This chapter resumes the notations necessary to further developments. We summarize here common definitions and properties and point to the relevant items of the bibliography for further precisions.

Contents

2.1 Basic notations

These first definitions are very common and widely used in language theory. We refer the reader to the litterature for their precise definitions.

Definition 2.1 (Notations)
We use the following symbols, with their canonical meaning unless otherwise specified:

$\equiv$ *: syntactical equivalence. We denote its negation by* $\not\equiv$.

$=$ *: equality, both terms are equal modulo some theory that is clear from the context. We denote its negation by* $\neq$.

$\varnothing$ *: the empty set*

$\subseteq$ *: subset inclusion*

$\subset$ *: strict subset inclusion*

$\cup, \cap, \setminus$ *: union, intersection and difference of sets*

$\mathcal{D}om(f)$ *: the domain of a function* f

$\mathbb{N}$ *: the set of positive integers*

$\mathbb{N}^+$ *: the set of strictly positive integers*

$i..j$ *: the set of integers:* $\{i, i+1, \ldots, j\}$

$\leq_{\mathbb{N}}$ *: natural ordering on positive integers*

$(o_1, \ldots, o_n)_{lex}$ *: lexicographic order based on orders* $o_1,\ldots,o_n$.

Definition 2.2 (Substitution)
A substitution is a mapping *(a partial function) from variables to terms. We use the notation:* $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$ *when the domain is finite.*

We use the notation $t\{x \leftarrow t'\}$ to denote the term t into which every occurence of the variable x has been replaced by the term t'.

Definition 2.3 (Typing environment)
A typing environment is a substitution from variables to types. We use the capital greek letter Γ *to range over type environments.*

Definition 2.4 (Evaluation environment)
An evaluation environment is a substitution from variables to values. We use the lower case greek letter γ *to range over evaluation environments.*

Note It is often useful, especially when presenting an algorithm (that is, a well-defined deterministic process) to take into account *the order* in which elements are inserted in environments. It is quite common to consider environments as stacks, and implicitly have an ordering over their elements corresponding to their order of addition in the environment. This is why we introduce the following definition of the union of two environments:

Definition 2.5 (Union of environments)
Given two environments Γ_1 and Γ_2 we define their union $\Gamma_1 \uplus \Gamma_2$ as:

- $(\Gamma_1 \uplus \Gamma_2)(x) = \Gamma_1(x)$ *if* $x \in \mathcal{D}om(\Gamma_1)$.
- $(\Gamma_1 \uplus \Gamma_2)(x) = \Gamma_2(x)$ *otherwise.*

2.2 Regular trees

All the definition and properties found in this section are taken from the survey by Bruno Courcelle [Cou83].

2.2.1 Symbols

Definition 2.6 (Ranked alphabet)
A ranked alphabet *is a pair $(\Sigma, |\_|)$, were Σ is a countable set of symbols and $|\_| : \Sigma \mapsto \mathbb{N}$ a mapping from symbols (of Σ) to positive integers. $|f|$ is called the arity of the symbol f.*

Definition 2.7 (Path)
A path *is a sequence of strictly positive integers such that:*

- *the empty sequence ϵ is a path.*
- *if π is a path, $i \in \mathbb{N}^+$, π, i is a path.*
- *the concatenation of two paths π and π' is a path, and is noted: $\pi \cdot \pi'$*

We note Π the set of all paths.

2.2.2 Trees

In the following chapters, we will manipulate both finite and infinite (albeit regular) trees, to represent types or terms. We use the following general definitions which regroup both types of trees.

Definition 2.8 (Tree)
Given a ranked alphabet $(\Sigma, |\_|)$, a tree is a partial function $t : P \mapsto S$ where $P \subseteq \Pi$ and $S \subseteq \Sigma$, with the following properties:

- *$t(\epsilon)$ is defined.*
- *if $t(\pi \cdot \pi')$ is defined so is $t(\pi)$ (i.e. $\mathcal{D}om(t)$ is prefix-closed).*
- *$t(\pi, i)$ is defined if and only if $|t(\pi)| \geq i$.*

From the definition of tree derives the definition of sub-tree:

Definition 2.9 (Subtree)
Let t be a tree. Let $\pi \in \mathcal{D}om(t)$ be a path. We call the sub-tree of t rooted at π, and we note $t_{|\pi}$ the tree t' defined by:

- *$\mathcal{D}om(t') = \{\pi' \mid \pi \cdot \pi' \in \mathcal{D}om(t)\}$*
- *$\forall \pi' \in \mathcal{D}om(t'),\ t'(\pi') = t(\pi \cdot \pi')$*

We note: $\mathcal{S}ubtree(t) = \{t_{|\pi} \mid \pi \in \mathcal{D}om(t)\}$

We say that a tree is *finite* if its domain is finite and *infinite* if its domain is infinite. A tree t is *regular* if $\mathcal{S}ubtree(t)$ is finite. Moreover, as trees will be used later on to represent iterators, we will say —by an abuse of terminology— that a tree is *recursive* if and only if it is *regular* and *infinite*.

Definition 2.10 ((strict) subtree relation, equivalence)
Let $\sqsubseteq$ and $\sqsubset$ be the relations defined by:

- *$t' \sqsubseteq t$ if and only if $\mathcal{S}ubtree(t') \subseteq \mathcal{S}ubtree(t)$*
- *$t' \sqsubset t$ if and only if $\mathcal{S}ubtree(t') \subset \mathcal{S}ubtree(t)$*

If $t' \sqsubset t$ we say that t' is a strict sub-tree of t. We denote by $t \overset{\square}{=} t'$ the fact that $t \sqsubseteq t'$ and $t' \sqsubseteq t$.

This sub-tree relation is well suited for our theoretical developments, but is rather peculiar. Indeed, it should be noted that, in general, with regular trees, we cannot rely on a structural ordering. For instance, let us consider the tree t_a, represented in Figure 2.1. The ellipsis and dashed arrow denote the regularity of this infinite tree.

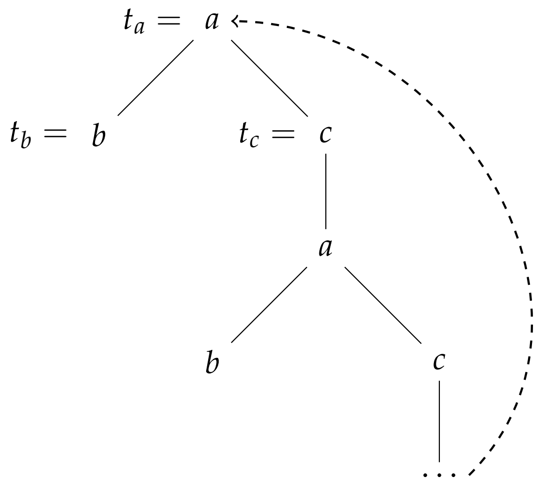


Figure 2.1: A regular tree

This regular tree has three *distinct* sub-trees: t_a itself, which is $t_a|_{(21)^n}, \forall n \geq 0$, t_b which is $t_a|_{(21)^n 1}, \forall n \geq 0$ and t_c, which is $t_a|_{(21)^n 2}, \forall n \geq 0$. Within this tree, the following properties hold:

- $t_b \sqsubset t_a$
- $t_b \sqsubset t_c$
- $t_a \overset{\square}{=} t_c$

The most important property is the third one. Surprisingly, while t_a and t_c are distinct sub-trees, they are equivalent with respect to the sub-tree relation, since they are sub-trees of one another. While this seems odd for trees seen as syntactical objects, it becomes relevant in the context of trees seen as iterators (or terms which are *evaluated*). Indeed, this property states that whenever a "step c" is evaluated, then a "step a" is evaluated afterwards. Hence, if the program (our regular tree) loops and repeats infinitely many c, then it will repeat infinitely many a and conversely. On the contrary, $t_b \sqsubset t_a$ would informally mean that no "a" can occur after a "b" step. In other words, the sub-tree relation is a *well-founded order* for all sub-trees of a given regular tree.

Lemma 2.11 (Well-founded order on regular trees) *Let t be a regular tree. Then $\sqsubseteq$ is a* well-founded order *on $Subtree(t)$.*

Proof Let us show that $\sqsubseteq$ is a well-founded order, that is, $(i.)$ that $\sqsubseteq$ is a partial order and that $(ii.)$ $\sqsubset$ is a well-founded relation.

i. A partial order is a reflexive, antisymmetric and transitive relation. We can remark that: $\forall t_a, t_b, t_c \in \mathcal{S}ubtree(t)$, we have:

reflexivity: $t_a \sqsubseteq t_a$, because $t_a \equiv t_a|_\epsilon$.

antisymmetry: if $t_a \sqsubseteq t_b$ and $t_b \sqsubseteq t_a$ then $t_a \overset{\square}{=} t_b$, by definition.

transitivity: If $t_a \sqsubseteq t_b$, then there exists a path π such that $t_b|_\pi \equiv t_a$. Likewise, if $t_b \sqsubseteq t_c$, then there exists a path π' such that $t_c|_{\pi'} \equiv t_b$. Thus, we have that $t_c|_{\pi'\cdot\pi} \equiv t_a$ and consequently that $t_a \sqsubseteq t_c$. Note that transitivity also holds for $\sqsubset$ (by taking π and π' distinct from ϵ).

ii. To show that $\sqsubset$ is well founded, we must show that there is no infinite chain $t_0 \sqsubset \ldots \sqsubset t_n \ldots$ in $\mathcal{S}ubtree(t)$. First of all, since t is supposed regular, then by definition $\mathcal{S}ubtree(t)$ is finite. Since $\mathcal{S}ubtree(t)$ is finite, in any infinite chain of elements of $\mathcal{S}ubtree(t)$, then there is at least one t' which occurs infinitely many time. Thus an infinite chain has the form: $\ldots \sqsubset t' \sqsubset \ldots \sqsubset t' \sqsubset \ldots$. Hence by transitivity, we have $t' \sqsubset t'$ which is a contradiction since $(t', t') \notin \sqsubset$ (by Definition 2.10). Hence there is no such infinite chain and consequently, $\sqsubset$ is a well-founded relation.

Finally, let us remark that $\sqsubseteq$ is not a *well-order* since it is not *total*. For instance all the leaves in a tree (symbols of arity 0) are not comparable.

2.2.3 Explicit recursion

While infinite trees are a convenient way to reason about recursive terms, another notation is often needed to finitely represent such terms. Indeed, we want to have a machine-representable (hence finite) representation for such terms (particularly if we aim to write algorithms on those terms). We use the well-known μ notation.

Definition 2.12 (Explicit binder)
Let $(\Sigma, |\_|)$ be a ranked alphabet. Let $\mathcal{V}$ be a countable set of symbols called variables such that $\mathcal{V} \cap \Sigma = \varnothing$ and $\forall X \in \mathcal{V}, |X| = 0$. Let μ be a symbol such that $\mu \notin \Sigma$ and $|\mu|$ = 2. A term τ is a term with explicit recursive binder *if τ is a* finite tree *of $(\Sigma \cup \mathcal{V} \cup \{\mu\}, |\_|)$ such that:*

(i) $\forall \pi \in \mathcal{D}om(\tau)$, *if* $\tau(\pi) = \mu$, *then* $\tau(\pi, 1) \in \mathcal{V}$;

(ii) $\forall \tau' \in \mathcal{S}ubtree(\tau)$, *if* $\tau' = \mu(X, \mu(X_1, \ldots, \mu(X_n, \sigma)))$ *then* $\sigma \neq X$.

$$\textbf{(var.)}\ \frac{(X,\bot)\in\Gamma}{\Gamma\Vdash X}\qquad\qquad \textbf{(cst.)}\ \frac{}{\Gamma\Vdash f}\ \text{if } |f|=0$$

$$\textbf{(}\boldsymbol{\mu}\textbf{.)}\ \frac{\{(X,\bot)\}\uplus\Gamma\Vdash\tau}{\Gamma\Vdash\mu X.\tau}\qquad\qquad \textbf{(func.)}\ \frac{\Gamma\Vdash\tau_1\quad\ldots\quad\Gamma\Vdash\tau_n}{\Gamma\Vdash f(\tau_1,\ldots,\tau_n)}\ \text{if } |f|>0$$

Figure 2.2: Well formed μ-terms

Notation We denote the tree $\mu(X,\tau)$ by $\mu X.\tau$. μ is called a *binder* and X is a *bound variable.*

The condition (ii) in Definition 2.12 is known as *contractivity* and rules out meaningless terms such as $\mu X.X$ which cannot be associated to any tree. We define a notion of well-formed closed μ-term which characterizes the scope of a binding in type:

Definition 2.13 (Closed μ-term)
A μ-term is closed if the judgement $\varnothing\Vdash\tau$ is derivable with the rules of Figure 2.2.

Note the use of the $\uplus$ operators which ensures the well-formedness of types with identical nested variables, such as “$\mu X.(X,\mu X.(t,X))$”.

2.2.4 Properties

It is clear that for every μ-term, there is a regular tree (its infinite unfolding) and conversely that to every regular tree we can associate a μ-term. More precisely, as stated by Roberto Amadio and Luca Cardelli in [AC93], the notation $\mu X.\tau$ is used to denote a *canonical* solution of the regular equation $X=\tau$ (see again [Cou83] for a proof that such (systems of) equations have a unique solution, which is regular, and also the tutorial by Pierce *et al.* [GLP03]).

Consider a fixed ranked alphabet $(\Sigma,|\_|)$. We use the following functions to convert a regular tree to and from μ-term.

Definition 2.14 (Infinite expansion)
Let τ be a μ-term. The infinite expansion *(or unfolding) of τ, noted $[\tau]_\infty$ is defined by the coinductive process:*

- $[f(\tau_1,\ldots,\tau_n)]_\infty = f([\tau_1]_\infty,\ldots,[\tau_n]_\infty)$*, for* $f\in\Sigma$
- $[\mu X.\tau]_\infty = [\tau\{X\leftarrow\mu X.\tau\}]_\infty$

Conversely, there exists a notion of recursive folding:

Definition 2.15 (Recursive folding)
Given a regular tree t we note $[t]_\mu$ its canonical μ notation.

2.3 Proofs and trees

A common way to represent type-systems, evaluation rules or algorithms is by giving a set of *inference* rules which proves a *judgement*. Various works present inference systems and proofs techniques such as induction and coinduction (e.g. see [Pie02], [GLP03], [Acz77]). Surprisingly, to the best of our knowledge, it is only recently that a proof-theoretic approach to coinduction was published ([Gra03], [Ler06], [LG07]). These works bridge the gap between inference systems and coinductive terms, allowing one to reason and prove easily properties of infinite regular trees. Most of the definitions and theorems of this section are taken from [Gra03] and [LG07].

2.3.1 Inference systems, derivations

Definition 2.16 (Inference system)
Let $\mathcal{U}$ be a set and let us call its elements judgments. *An inference system ϕ is a set of rules over judgements. A rule is a pair $(\mathcal{A}, z)$ where $\mathcal{A} \subseteq \mathcal{U}$ and $z \in \mathcal{U}$. Elements of A are called the premises of the rule and z is called the goal of the rule.*

The classical notation for an inference rule $(\mathcal{A}, z)$ is $\dfrac{\mathcal{A}}{z}$.

Definition 2.17 (Derivation)
A derivation *(or* proof tree*) of a judgement z within an inference system ϕ is a tree d such that:*

- $d(\epsilon) = z$
- $\forall \pi \in \mathcal{D}om(d), \exists(\{a_1, \ldots, a_k\}, j) \in \phi$ *such that* : $d(\pi) = j$ *and* $\forall i \in 1..k, d(\pi, i) = a_i$

A derivation is conveniently written bottom-up: $\dfrac{\dfrac{a}{b} \quad c \quad \dfrac{d}{e}}{z}$. z is called the conclusion of the derivation.

A derivation is *well-founded* if it is finite. Otherwise it is called *ill-founded*.

2.3.2 Induction and coinduction

An inference system ϕ is associated to an inference operator, $F_\phi : \mathcal{P}(\mathcal{U}) \to \mathcal{P}(\mathcal{U})$ defined as: $F_\phi(S) = \{z \in \mathcal{U} | \exists \mathcal{A} \subseteq S, (\mathcal{A}, z) \in \phi\}$. The *inductive* and *coinductive in-*

terpretations of an inference system ϕ are defined as the least fixed point and greatest fixed point of its inference operator F_ϕ:

- Least fixed point: $\bigcap\{S|F_\phi(S) \subseteq S\}$
- Greatest fixed point: $\bigcup\{S|S \subseteq F_\phi(S)\}$

The proof-theoretic approach to induction and coinduction, define *equivalently* the inductive and coinductive interpretation of an inference system ϕ as follows:

Definition 2.18 (Inductive interpretation)
The inductive interpretation of an inference system ϕ is the set $\mathcal{I}nd(\phi)$ of conclusions of well-founded derivations.

The associated induction principle states that to prove that every judgement in the inductive interpretation is in some set S, it is sufficient to show that for every judgement z, if z is the conclusion of a derivation d and if for all conclusion j of a strict sub-derivation of d, j is in S, then z is in S (structural induction on the derivation of z).

This technique can be used to show that all inductively defined terms have a certain property (of which S is the characteristic set).

As we consider regular (infinite) tree for types and terms, we would like to use regular derivations of such terms. This is exactly the definition of the coinductive interpretation:

Definition 2.19 (Coinductive interpretation)
The coinductive interpretation of an inference system ϕ is the set $\mathcal{C}o(\phi)$ of conclusions of arbitrary derivations.

In particular, the coinductive interpretation contains conclusions of *ill-founded* derivations. The associated coinduction principle is stated as follows: To prove that all judgments in a set S are in the coinductive interpretation, build a system of recursive equations between derivations, with unknowns (x_j) for $j \in S$. Each equation is of the form:

$$x = \frac{x_{j_1} \qquad \ldots \qquad x_{j_n}}{j}$$

and must correspond to an inference rule: $(\{j_1, j_2, \ldots\}, j) \in \phi$. These equations are guarded: there are no equation of the form $x_j = x'_j$. It follows that the system has a unique solution ([Cou83]), and this solution σ is such that for all $j \in S$, $\sigma(x_j)$ is a valid derivation that proves j. Therefore, all $j \in S$ are also in $\mathcal{C}o(\phi)$. This technique is particularly useful to show that a particular infinite derivation is a valid derivation of some inference system ϕ. More precisely, this principle gives a way to *build* such a derivation.

As a last observation, it should be noted that Courcelle's results are not restricted to finite ranked alphabet but also hold for infinite countable sets of symbols. Hence the previous results on induction and coinduction are also valid in the presence of *deduction schemas* instead of deduction rules (an example of deduction schema is given in Figure 2.2 where the schemas **(func.)** and **(cst.)** denote sets of rules, one for each symbol in the alphabet).

2.4 ℂDuce

We succinctly describe the ℂDuce language, its type-system and pattern algebra. The interested reader can refer to [CF05, FCB02, BCF03] for more detailed definitions of the various concepts presented hereafter.

2.4.1 Values

We consider here the maximal subset of values which are not lambda-abstractions. Indeed, in what follows, values will essentially represent documents, hence the restricted definition:

Definition 2.20 (Values)
Values are the terms inductively defined by the following grammar:

$$\begin{array}{rcll} v & ::= & c & c \in \mathbb{C},\ \textit{constants} \\ & \mid & (v,v) & \textit{pairs} \end{array}$$

Note The set of constants consists of 0-ary symbols such as integers —`0`, `1`, …— characters — `'a'`,`'b'`,`'ぼ'`— and atoms (also known as *variants*) which are backquoted symbols — `‘A`, `‘B`, `‘Foo33`.

These values are enough to encode XML documents. For example in ℂDuce, lists are encoded, *à la* Lisp, as nested pairs, the empty list being represented by the atom `‘nil`. An XML document is the pair of its tag, represented by an atom and the list of its children (its content).

Notation We use the square brackets notation as syntactic sugar for lists:

$$\texttt{[v}_1\ \ldots\ \texttt{v}_\texttt{n}\texttt{]} = (v_1, (\ldots, (v_n, \texttt{‘nil})))$$

For XML[1] documents we use the notation:

$$\texttt{<tag>[v}_1\ \ldots\ \texttt{v}_\texttt{n}\texttt{]} = (\texttt{‘tag}, (v_1, (\ldots, (v_n, \texttt{‘nil}))))$$

which represents the document:

[1] We exclude attributes from the formal treatment. They do not pose any difficulty and are presented with the concrete syntax in Chapter 6

$$\begin{array}{l}\texttt{<tag>}\\ \qquad v_1\\ \qquad \vdots\\ \qquad v_n\\ \texttt{</tag>}\end{array}$$

Character strings in ℂDuce are just lists of characters. We use however the standard notation for strings:

$$\texttt{"c}_1 \ldots \texttt{c}_\texttt{n}\texttt{"} = (c_1, (\ldots, (c_n, \texttt{`nil})))$$

2.4.2 Types

We define here ℂDuce types and their subtyping relation.

Definition 2.21 (Types [CF05])
A type *is a possibly infinite term produced by the following grammar:*

$$\begin{array}{rcll} t & ::= & t \wedge t \mid t \vee t \mid \neg t \mid c & \textit{(Boolean connective)} \\ c & ::= & (c \times c) \mid a & \textit{(Type constructor)} \\ a & ::= & b \mid \texttt{Empty} \mid \texttt{Any} \mid t & \textit{(Atomic types)} \end{array}$$

with two additional requirements:

1. *(regularity) the term must be a regular tree;*
2. *(contractivity) every infinite branch must contain an infinite number of product nodes* $(\_ \times \_)$.

Notation We use the notation: $t_1 \setminus t_2$ as syntactic sugar for $t_1 \wedge \neg t_2$. We denote by $\mathbb{T}$ the set of all types.

We use b to range over basic types, while `Empty` and `Any` respectively denote the empty type and the type of all values. Besides, there are product $(t_1 \times t_2)$, union $(t_1 \vee t_2)$, intersection $(t_1 \wedge t_2)$, and negation $(\neg t)$ types. We tighten here the contractivity condition introduced in Definition 2.12 to rule out meaningless regular trees, such as$\neg(\neg(\neg \ldots))$. Both conditions are standard when dealing with recursive types (e.g. see [AC93]). Amongst basic types, one finds the type `Int` of integers, the type `Char` of the characters but also, for each value v in the language a *singleton* type v inhabited only by this constant. We use the notation $\bigvee_{i \in 1..n} t_i$ to denote the finite union $t_1 \vee \ldots \vee t_n$. Finally it is possible to represent intervals of integers using the notation $i..j$ which is syntactic sugar for the finite union of singleton types: $i \vee i+1 \vee \ldots \vee j$.

Notation As for values, these types are enough to denote XML types. The ℂDuce syntax for XML types and documents we sketched in the introduction is related to the type algebra as follows. Sequences are denoted by nested pairs and Boolean connectives and recursive types are used to encode regular expression types. For instance:

$$\texttt{type Book} = \texttt{<book>[Title (Author+}|\texttt{Editor+) Price?]}$$

defines a type `Book` that types elements tagged by `<book>` and that contain a title followed by either a non-empty list of authors or a non-empty list of editors and possibly ended by an optional price (with sensible definitions for `Title`, `Author`, `Editor` and `Price`). The declaration of `Book` above can be considered as syntactic sugar for the following equations:

$$\begin{array}{rcl}
\texttt{Book} & = & (\texttt{`book} \times (\texttt{Title} \times \texttt{X} \vee \texttt{Y})) \\
\texttt{X} & = & (\texttt{Author} \times \texttt{X}) \vee (\texttt{Author} \times \texttt{Z}) \\
\texttt{Y} & = & (\texttt{Editor} \times \texttt{Y}) \vee (\texttt{Editor} \times \texttt{Z}) \\
\texttt{Z} & = & (\texttt{Price} \times \texttt{`nil}) \vee \texttt{`nil}
\end{array}$$

where `‘nil` and `‘book` are singleton types. Note that inside regular expressions, we use “|” instead of “$\vee$” to denote the union.

The semantics of types is expressed in terms of *values*. In the framework of XML processing languages, values are XML documents and, following Hosoya *et al.* [HVP00], an XML type is (interpreted as) the set of XML documents that have that type, which leads to the following definitions:

Definition 2.22 (Type of a value)
We define the judgment $v : t$, where v is a (non-functional) value and t a type. We say that v has type t if the judgement $v : t$ can be derived by the rules given in Figure 2.3.

This definition of typing is sufficient for the restricted subset of values we consider: atoms and products. To type lambda-abstractions one needs to define a type system for all the expressions of the language and thus needs to define the subtyping relation beforehand (which itself relies on the typing relation). While defining such a type-system —despite the circular definitions of typing/subtyping/semantics of the language is possible— it is out of the scope of this thesis, and extensively discussed in [Fri04b] and [FCB02].

Definition 2.23 (Set of values)
We define the set-theoretic interpretation of a type t as: $[\![t]\!] = \{v | v : t\}$.

Definition 2.24 (Subtyping)
A type t is a subtype of a type s, and we note $t \leq s$ if and only if: $[\![t]\!] \subseteq [\![s]\!]$. We denote by $t \lneq s$ the fact that: $[\![t]\!] \subset [\![s]\!]$.

Positive rules

$$\textbf{(i-basic)}\ \frac{v \in t,\ \text{for } t \in b}{v : t} \qquad \textbf{(i-prod)}\ \frac{v_1 : t_1 \qquad v_2 : t_2}{(v_1, v_2) : (t_1 \times t_2)}$$

$$\textbf{(i-ul)}\ \frac{v : t}{v : t \vee s} \qquad \textbf{(i-ur)}\ \frac{v : s}{v : t \vee s} \qquad \textbf{(i-i)}\ \frac{v : t \qquad v : s}{v : t \wedge s} \qquad \textbf{(i-any)}\ \frac{}{v : \texttt{Any}}$$

Negative rules

$$\textbf{(n-basic)}\ \frac{v \notin t,\ \text{for } t \in b}{v : \neg t} \qquad \textbf{(n-n)}\ \frac{v : t}{v : \neg\neg t} \qquad \textbf{(n-empty)}\ \frac{}{v : \neg\texttt{Empty}}$$

$$\textbf{(n-p)}\ \frac{v \not\equiv (v_1, v_2)}{v : \neg(t_1 \times t_2)} \qquad \textbf{(n-p1)}\ \frac{v_1 : \neg t_1}{(v_1, v_2) : \neg(t_1 \times t_2)} \qquad \textbf{(n-p2)}\ \frac{v_2 : \neg t_2}{(v_1, v_2) : \neg(t_1 \times t_2)}$$

$$\textbf{(n-u)}\ \frac{v : \neg t \qquad v : \neg s}{v : \neg(t \vee s)} \qquad \textbf{(n-il)}\ \frac{v : \neg s}{v : \neg(t \wedge s)} \qquad \textbf{(n-ir)}\ \frac{v : \neg t}{v : \neg(t \wedge s)}$$

Figure 2.3: Typing rules for values in ℂDuce.

Since the use of subsumption makes two equivalent types (that is, two types denoting the same set of values) operationally indistinguishable, then we will always work up to type equivalence and consider, e.g. $t \vee t$, $\texttt{Any} \wedge t$ and t as the same type.

When working with types, it is sometimes required to have an order relation much finer than $\sqsubseteq$, which is only a syntactic ordering and does not take into account Boolean equivalence for example. For this purpose, we reuse the notions presented in [Fri04b] (where the reader will find the relevant proofs):

Definition 2.25 (Plinth)
A plinth $\beth \subset \mathbb{T}$ is a set of types with the following properties:

- *$\beth$ is finite*
- *$\beth$ contains* `Any`, `Empty` *and is closed under Boolean connectives* $(\wedge, \vee, \neg)$
- *for all types $t = (t_1 \times t_2)$ in $\beth$, $t_1 \in \beth$ and $t_2 \in \beth$*

One of the reasons why it is not always easy to use $\sqsubseteq$ on types is that a type $t \leq (\texttt{Any} \times \texttt{Any})$ does not necessarily have the form $(t_1 \times t_2)$ for some types t_1 and t_2 but is rather a finite union of products: $\bigvee\limits_{i \in 1..n} (t_1^i \times t_2^i)$ for some n. There are many ways to decompose such a type t in a finite union of products. We therefore use the following:

Definition 2.26 (Product decomposition)
Given a type $t \leq (\texttt{Any} \times \texttt{Any})$, we define a set of pair of types $\pi(t)$ as:

- $t = \bigvee_{(t_1 \times t_2) \in \pi(t)} (t_1 \times t_2)$
- $\forall (t_1 \times t_2) \in \pi(t), t_1 \neq \texttt{Empty}, t_2 \neq \texttt{Empty}$
- *if $\beth$ is a plinth such that $t \in \beth$, then:*$\forall (t_1 \times t_2) \in \pi(t), t_1 \in \beth, t_2 \in \beth$

To make proper use of the Definition 2.25 and 2.26 one need the following theorem:

Theorem 2.27 ([Frio4b]) *Every finite set of types is included in a plinth.*

As stated in [Frio4b], the existence of a plinth and a product decomposition for a type t are crucial. Indeed, we already know that for a regular type t the set $\mathcal{S}ubtree(t)$ is finite. The definitions of the plinth and of the product decomposition ensure that the closure of $\mathcal{S}ubtree(t)$ under Boolean connective is also finite. This is used for instance to show the termination of algorithms working on types.

2.4.3 Patterns

As many functional languages (such as Ocaml, SML or Haskell), CDuce relies on powerful *pattern* operators. Informally, patterns are just types in which capture variables may occur in a controlled way.

Definition 2.28 (Patterns)
Patterns are possibly infinite term produced by the following grammar:

$$\begin{array}{rrcr} p & ::= & t & \textit{(type)} \\ & | & (p,p) & \textit{(product)} \\ & | & p|p & \textit{(union)} \\ & | & p\&p & \textit{(intersection)} \\ & | & x & \textit{(variable)} \end{array}$$

with the following restrictions:

1. *(regularity) the term must be a regular tree;*
2. *(contractivity) every infinite branch must contain an infinite number of product nodes* $(\_,\_)$.

3. *(variables) for every pattern $p_1\&p_2$ (resp. $p_1|p_2$) the set of variables occuring in p_1 and p_2 must be disjoint (resp. equal).*

The set of well formed patterns is noted $\mathbb{P}$.

Definition 2.29 (Capture variables)
The set $\mathcal{V}ar(p)$ of capture variables of a pattern is defined as:

$$\begin{array}{rcc}
\mathcal{V}ar(x) & = & \{x\} \\
\mathcal{V}ar(t) & = & \varnothing \\
\mathcal{V}ar((p_1,p_2)) & = & \mathcal{V}ar(p_1) \cup \mathcal{V}ar(p_2) \\
\mathcal{V}ar(p_1\&p_2) & = & \mathcal{V}ar(p_1) \cup \mathcal{V}ar(p_2) \\
\mathcal{V}ar(p_1|p_2) & = & \mathcal{V}ar(p_1) \; = \; \mathcal{V}ar(p_2)
\end{array}$$

The semantics of patterns is given in terms of matching.

Definition 2.30 (pattern-matching)
The matching of a value v by pattern p, noted v/p is either a substitution from capture variables to values or the special value Ω, denoting an error; and is defined as follows:

$$\begin{array}{lcl}
v/x & = & \{x \mapsto v\} \\
v/t & = & \begin{cases} \varnothing & \textit{if } v \in [\![t]\!] \\ \Omega & \textit{else} \end{cases} \\
v/(p_1,p_2) & = & \begin{cases} v_1/p_1 \oplus v_2/p_2 & \textit{if } v = (v_1, v_2) \\ \Omega & \textit{else} \end{cases} \\
v/p_1\&p_2 & = & v/p_1 \oplus v/p_2 \\
v/p_1|p_2 & = & v/p_1|_1 v/p_2
\end{array}$$

where:

$$\begin{array}{lcl}
\gamma|_1 r & = & \gamma \\
\Omega|_1 \gamma & = & \gamma \\
r \oplus \Omega & = & \Omega \\
\Omega \oplus r & = & \Omega \\
\gamma_1 \oplus \gamma_2 & = & \{x \mapsto \gamma_1(x) | x \in \mathcal{D}om(\gamma_1) \setminus \mathcal{D}om(\gamma_2)\} \\
 & \cup & \{x \mapsto \gamma_2(x) | x \in \mathcal{D}om(\gamma_2) \setminus \mathcal{D}om(\gamma_1)\} \\
 & \cup & \{x \mapsto (\gamma_1(x), \gamma_2(x)) | x \in \mathcal{D}om(\gamma_2) \cap \mathcal{D}om(\gamma_1)\}
\end{array}$$

For the two basic cases —variable binding and type-checking— the definition of matching is straightforward. For a binding pattern, the substitution from the variable to the captured value is returned. For a type check, the pattern succeeds if and only if the matched value inhabits the considered type, and in that case, the empty substitution is returned. The rules for the union, intersection and products are subtler. For the union, the substitution returned is the “first-match union” (denoted by “$|_1$”) of the result of both patterns, thus reflecting the *first-match policy* of ℂDuce’s pattern matching. The rule for intersection and product use a special kind of union, the *cumulative union* denoted by $\oplus$. The simple case is the one for the intersection where, by definition, the set of variables in p_1 and p_2 are disjoint. In this case, the $\oplus$ operator simply acts as a disjoint union. For the product pattern however, the same variable may appear in both components of the pattern. In this situation, the semantics is to bind the common variable to the *pair* resulting of the values matched on each component of the product. This allows to *accumulate* subsequences along a recursive pattern. Let us illustrate the practical interest of this feature by some examples:

Example 2.31

$$\begin{array}{rcl}
\texttt{(0,"foo")}/x\&(\texttt{Int}\&y,z) & = & \{x \mapsto \texttt{(0,"foo")},y \mapsto \texttt{0},z \mapsto \texttt{"foo"}\}\\
\texttt{"bar"}/x\&\texttt{Char} & = & \Omega\\
\texttt{[1 2 3 4 5]}/(x,(\texttt{Any},(x,(\texttt{Any},(x,x))))) & = & \{x \mapsto \texttt{[1 3 5]}\}\\
\texttt{[1 ‘true ‘false ‘true 5 6]}/ & & \\
p \text{ where } (x\&\texttt{‘nil})|((x\&\texttt{Int})|\texttt{Bool},p) & = & \{x \mapsto \texttt{[1 5 6]}\}
\end{array}$$

The first pattern is an intersection where the whole value is captured in x and where the first and second projections are captured in y and z with the added constraint that the first projection must be an integer.

The second pattern fails because it must match something that is a character, while the argument here is a string.

The third pattern illustrates cumulative unions. We recall that lists are encoded as nested pairs, the special atom `‘nil` denoting the empty list. In this pattern, the first element of the list, as well as the third, the fifth and the trailing `‘nil` are bound to the variable x. Following the semantics, the pattern returns a substitution binding x to the nested pairs: $(1,(3,(5,\texttt{‘nil})))$.

The last example illustrates cumulative union and recursive patterns. The special construct “p where ...” creates a binding for a recursive pattern p. This pattern matches either the empty sequence (and binds it to x) or a pair, in which case, the first component can be an integer and is captured by x or a Boolean (and is just checked). The second component is the recursive pattern p itself, thus allowing it to match any sequence of integers or Boolean. Again, the cumulative semantics of the $\oplus$ operator allows the pattern to capture the subsequence $(1,(5,(6,\texttt{‘nil})))$. ℂDuce actually provides some syntactic sugar, with regular expression patterns for such cases:

Example 2.32

```
match [ 1 ‘true ‘false ‘true 5 6 ] with
[ ((x::Int)|Bool)* ] ->x
```

Actually, ℂDuce exposes both the exact intersection (denoted by **&**) and the cumulative one (denoted by `::`). In the above piece of code, all elements of type `Int` are accumulated in x.

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

Theorem 2.33 (Accepted type [FCB02]) *For all $p \in \mathbb{P}$, the set of all values v such that $v/p \neq \Omega$ is a type. We call this set the* accepted type *of p and note it by $\wr p \int$.*

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

Theorem 2.34 (Type environment [FCB02]) *There exists an algorithm that for all $p \in \mathbb{P}$, and $t \leq \wr p \int$ returns a type environment $t/p \in \mathcal{V}ar(p) \to \mathbb{T}$ such that $[\![(t/p)(x)]\!] = \{(v/p)(x) \mid v : t\}$.*

Part II

Filter calculus

Chapter 3

Filters

The goal of this chapter is to formally present the *filters*, give their dynamic semantics as well as termination properties. The use of filters is illustrated through various simple example. A more comprehensive (and realistic) list of filter-based programs is given in Chapter 6.

Contents

3.1 Rationale

As already said in the introduction, our goal is to define a way to precisely type iterators over XML documents and more generally tree-like data structures. Such iterators are meant to be added to a more generic host language and inherit its semantics in order to evaluate *expressions* of the host language on the iterated data.

This language of iterators must be carefully designed, so as to comply with the following requirements:

expressivity : we need to encode complex XML transformations

normalization : the evaluation of an iterator must always terminate

integration : the iterator language must blend seamlessly with the host language.

These three aspects guide the design of the language. While we would like our iterators to be completely independent from any particular host language, their design is actually tightly related to both Hosoya's regular expression filters and ℂDuce's patterns. The first trait we want for our language is expressivity. We want to express complex operations over XML documents. The connection with ℂDuce's patterns is, in this respect, quite tempting. Indeed, as previously described, ℂDuce's patterns allow one to express two things:

- Complex conditions on an XML document
- Capture arbitrary subparts of an XML document.

We recall that ℂDuce's patterns are nothing else but types with capture variable. This is however only a syntactical similarity: patterns (even without capture variables) have a semantics: to *iterate* over a value, checking along conditions on subtrees and optionally capturing some of them. From this point of view, the pattern algebra seems more than well suited to design the filter algebra. Let us recall the basic patterns:

type : t, checks that the matched value inhabits t

variable : x, binds the matched value to x

product : (p_1,p_2), recursively apply p_1 and p_2 to the first and second projection of the input value

union : $p_1|p_2$, return the matching of p_1 if it succeeds and the matching of p_2 otherwise

recursion : patterns are regular, i.e. they can be iterated on an input value

intersection : $p_1\&p_2$, both p_1 and p_2 must match.

This small algebra allows a pattern to iterate over *any XML document*. Starting from this, we design the filter algebra as follows. As the basic case for a pattern is a type check, the basic case for a filter is the transformation which consumes its input and produces a value. However, this value must be *a value of the host language*. Indeed, filters are meant to be part of an host language and are applied to input values from the host language, they should therefore return values from this host language. The most straightforward way to do so is by *executing an expression of the host language*. Our basic filter is then the expression filter, e where e is an expression of the host language.

Of course, an expression filter is only useful if it can output part of the input value. As patterns provide capture variables, filters provide a *pattern filter*, $p \to f$ where p is a pattern provided by the host language and f a filter. When applied to an input value v, a $p \to f$ filter captures subparts of v thanks to p and then evaluates f.

The key point is that now, capture variables can occur in an expression filter e, thus allowing it to copy part of the input in the output. For instance, a filter $x \longrightarrow x+1$ is a pattern filter, with pattern x, for which the sub-filter is the expression $x+1$. This filter simply increments its input.

Similarly to patterns, filters can descend into a product constructor. This is the aim of the *product filter*, (f_1,f_2). As we explained in Chapter 2, the semantics of a pattern of the form (x,x) applied to a value (v_1,v_2) is to return the substitution $\{x \mapsto (v_1,v_2)\}$. This trait allows patterns to capture sub-sequences of an input sequence (encoded as nested pairs), or differently said, to iterate over an input sequence and return matching sub-elements. The product filter uses the same technique to iterate over a product. Given an input value (v_1,v_2), if we apply it to a filter (f_1,f_2) then, the filters are applied component-wise and *the result is returned as the pair of the two partial results*. This aspect coupled with recursion makes it possible to encode list mapping and more generally Hosoya's regular expression filters.

Filters also feature an *alternation operator*, $f_1|f_2$, which conditionally continues the evaluation of a filter. Its semantics is the well known first-match policy, specifying that if the evaluation of f_1 fails on the input value, then f_2 is evaluated on the input. A typical example of failure is when one applies a product filter to a value which is not a pair.

As filters are used to iterate through unbounded trees and sequences, it is natural to consider *regular* filters. This will be reflected in the concrete syntax (presented in Chapter 6) by the possibility to write recursive filters. Of course we need well founded recursion and the usual properties of well-foundness of regular terms are present in the formal definition for filters.

If we only considered the previously mentioned constructs, we would roughly end up with Hosoya's *regular expression filters* translated to ℂDuce instead of XDuce. A feature that as not been explored is *pattern intersection*, which makes little sense in the case of a filter which returns a value. Instead, we introduce a *composition operator* for filters, $f_1;f_2$. Informally, the result of $f_1;f_2$ applied to v is nothing else but $f_2(f_1(v))$. However unguarded composition is way too powerful as it opens the door to Turing-completeness (or at the very least to non-terminating filters). We will see how we restrict composition so as to enforce termination of filters on any input value while retaining enough expressive power to encode the desired iterators. We will also see in Chapter 5 that this composition seriously complicates the type inference process.

All along this work we made, some implicit (and, we think, reasonable) assumptions on the properties of the host and filter languages: typically but not only, that the two languages share the same XML types, share the same variables, and that pattern matching is or can be defined on the values of the host (i.e. it has a constructor for pairs). When dealing with the static semantics we will also assume that given a typing environment Γ, we can deduce the type t of the expression e (of the host language). ℂDuce is an example of such a suitable language. Indeed, the operation defined in Chapter 2 are the only one needed here. However it is clearly not the only possible host language as all of its feature can be found in other generic language with of course different behaviours.

3.2 Filter calculus

Definition 3.1 (Filters)
A filter f is a regular tree co-inductively generated by the following production rules (where e ranges over expressions of the host language)

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

and that satisfies the following conditions:

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

The condition on contractivity is the usual one which rules out meaningless terms. The condition on composition is however rather new and involved. In a nutshell, it states that a recursive filter cannot cross a "**;**". This ensures the termination of the evaluation of a filter on a finite value (as we will illustrate after giving the semantics) as well as the termination of the type inference algorithm (as we explain in Chapter 5). Henceforward we use $\mathbb{F}$ to denote the set of (well-formed) filters.

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

Definition 3.2 (Capture variables)
We define the set of capture variables *of a filter f, $\mathcal{V}ar(f)$ as:*

$$\begin{array}{rcl} \mathcal{V}ar(e) & = & \varnothing \\ \mathcal{V}ar(f_1;f_2) & = & \mathcal{V}ar(f_1) \cup \mathcal{V}ar(f_2) \\ \mathcal{V}ar(f_1|f_2) & = & \mathcal{V}ar(f_1) \cup \mathcal{V}ar(f_2) \\ \mathcal{V}ar((f_1,f_2)) & = & \mathcal{V}ar(f_1) \cup \mathcal{V}ar(f_2) \\ \mathcal{V}ar(p \to f) & = & \mathcal{V}ar(p) \cup \mathcal{V}ar(f) \end{array}$$

Definition 3.3 (Free variables)
We define the set of free variables *of a filter f, $\mathcal{FV}(f)$ as:*

$$\begin{array}{rcl}
\mathcal{FV}(f_1;f_2) & = & \mathcal{FV}(f_1) \cup \mathcal{FV}(f_2) \\
\mathcal{FV}(f_1|f_2) & = & \mathcal{FV}(f_1) \cup \mathcal{FV}(f_2) \\
\mathcal{FV}((f_1,f_2)) & = & \mathcal{FV}(f_1) \cup \mathcal{FV}(f_2) \\
\mathcal{FV}(p \to f) & = & \mathcal{FV}(f) \setminus \mathcal{V}ar(p)
\end{array}$$

We assume that $\mathcal{FV}(e)$ and $\mathcal{V}ar(p)$ are defined for the host language.

3.3 Operational semantics

We define a big step operational semantics for filters and show the termination of the evaluation of filters f on every finite value v. The dynamic semantics is given by the inference rules for the judgement $\gamma \vdash_e f(v) \rightsquigarrow r$ in Figure 3.1 and describes how the evaluation of the application of filter f on a value v in an environment γ yields an object r where r is either a value or Ω. The latter is a special value which represents a runtime error: it is raised either because a filter did not match the form of its argument (e.g. **(e-prod-err2)**) or because some pattern matching failed (e.g. **(e-patt-err)**).

The semantics of filters is quite straightforward and inspired of the semantics of patterns. The *expression* filter discards its input and evaluates (rather, asks the host language to evaluate) the expression e in the current environment (**e-expr**). It can be thought of as the right-hand side of a branch in a `match with` construct.

The *product* filter expects a pair as input, applies its sub-filters component-wise and returns the pair of the results (**e-prod-ok**). This filter is used in particular to express sequence mapping, as the first component f_1 transform the element of the list and f_2 is applied to the tail. In particular, it is often the case where f_2 is a recursive filter, allowing to iterate on arbitrary lists and stopping when the input is `‘nil`. If the input is not a pair, the filter fails (**e-prod-err{1,2}**).

The *pattern* filter first matches its pattern p against the input value v; if it fails it raises an error (**e-patt-err**), otherwise it evaluates its sub-filter in the environment augmented by the substitution v/p (**e-patt-ok**).

The *alternative* filter follows a standard first match policy: If the filter f_1 succeeds, then its result is returned (**(e-union-1)**). If f_1 fails, then f_2 is evaluated against the input value. This filter is particularly useful two write the alternative of two (or more) *patterns* filters, allowing to conditionally continue a computation based on the shape of the input.

Finally, the *composition* allows us to pass the result of f_1 as input to f_2. The composition filter is of paramount importance. Indeed, without it, our only way to iterate (deconstruct) an input value is to use a *product* filter, which always rebuild a pair as result. As already said, this limits us to *map-like* filters (like in Hosoya's regular expression filters). Our composition allows to combine an intermediate result and thus perform more complex operations such as flattening. It should be noted that

(e-expr)
$$\frac{}{\gamma \vdash_e e(v) \rightsquigarrow r} \qquad r = \texttt{eval}(\gamma, e)$$

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

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

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

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

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

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

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

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

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

Figure 3.1: Operational semantics of filters

composition must result in a well-founded filter, in the sense of the second condition of Definition 3.1. The condition prevent a filter from building an infinitely large value (or equally of looping infinitely on an input), as illustrated in the Examples Section hereafter.

3.4 Examples

We illustrate here the semantics of *untyped* filters. The typing of such filters will be discussed in the next chapter.

3.4.1 Simple filters

We start by defining some simple filters. First of all, the identity filter:

Example 3.4

$$id = x \rightarrow x$$

It is a simple *pattern* filter for which the pattern is the capture variable x and the sub-filter is the expression x of the host language. In the same vain, we can define two filters π_1 and π_2 which extract the first and second component of their arguments:

Example 3.5

$$\pi_1 = (x, \_) \rightarrow x \qquad \pi_2 = (\_, y) \rightarrow y$$

Now a `succ` filter which increments its argument can be written:

Example 3.6

$$x \rightarrow x + 1$$

where again $x + 1$ is an expression of the host language.

3.4.2 Alternative, first match policy

The following *neg* filter emulates the behaviour of an overloaded function:

Example 3.7

$$neg = (x\&\texttt{Bool} \rightarrow \texttt{not}\ x) \mid (x\&\texttt{Int} \rightarrow -x)$$

From its definition, it should not be difficult to understand what this filter does: when applied to a Boolean, it returns its negation (*via* the `not` function of the host language) while when applied to an integer, it returns its opposite. We can generalize this concept and see that the well-known pattern-matching operators of the ML language family (such as "`match with`" of OCaml or "`case of`" of Haskell) can be encoded by a filter:

Example 3.8

$$\begin{array}{l}\texttt{match}\ v\ \texttt{with}\\ p_1\ \rightarrow\ e_1\\ |\qquad \vdots\\ |\ p_n\ \rightarrow\ e_n\end{array}$$

becomes:

$$(p_1 \rightarrow e_1 | \ldots | p_n \rightarrow e_n)(v)$$

3.4.3 Recursive filters

Infinite (regular) filters encode recursive transformations. For example, one can write a simple iterator over lists:

Example 3.9

$$\mathit{succList} = \texttt{`nil} \rightarrow \texttt{`nil} | (x \rightarrow x + 1, \mathit{succList})$$

In this filter, if the argument is the constant `` `nil ``, denoting the empty list, the result is the empty list. If the argument is a pair *(head,tail)*, then *head* is incremented and the filter is recursively applied to the *tail*. Afterwards, the pair of the two results is returned. The recursive call being on the second component is only due to the fact that lists are encoded as nested pairs. There is of course no problem putting a recursive call on the first component, allowing to iterate on arborescent data structures.

3.4.4 Composition

Recursive filters, when coupled with the reconstructing *product* filters are enough to encode any *exact* mapping (in the sense of mapping exactly one element of the input to one element of this output, both having thus the same length). It is however impossible to encode two important features such as *almost copying* of a list (or tree), that is deleting some of its elements based on a choice and more generally *fold-like* iterators which operate on an accumulated result. To express such operations by filters, one can use the composition operator ";". The first example we give is the one of list concatenation. In XML languages, concatenation is always given as an hard-coded operator[1] so as to type it precisely. List concatenation can easily be defined as a filter:

[1] Some XML transformation languages provide *flat* sequences, hence the "`,`" is both the `cons` and `concat` operation of functional languages: if $l_1 = \texttt{[}v_1\ v_2\ v_3\texttt{]}$ and $l_2 = \texttt{[}v_4\ v_5\ v_6\texttt{]}$ then $l_1, l_2 =$

Example 3.10

$$\texttt{@} = (x{,}y) \rightarrow (x{;}f) \ \text{ where } \ f = \text{‘}\texttt{nil} \rightarrow y|(z \rightarrow z{,}f)$$

The filter @ takes as input the pair of the two lists to concatenate. The first argument is captured in x, and the second in y. Then, the filter f is applied, through composition to the expression x, *under the scope of the capture variable* y (we will see in the concrete syntax presented in Chapter 6 more a convenient way to write such filters). The filter f simply iterates on the sequence x until it reaches the end, at which point it replaces it by the second list y and rebuilds the whole list.

Another example where the composition is more involved is the list reversal, `rev`. While this is a very common operation on lists, neither ℂDuce nor XDuce provide an hard-coded list reversal operator, which we can define with filters as follows:

Example 3.11

$$\begin{array}{rcl} \mathit{rev} & = & x \rightarrow f{;}(y,\_) \rightarrow y \\ f & = & \text{‘}\texttt{nil} \rightarrow (\text{‘}\texttt{nil}, x)|((z \rightarrow z{,}f){;}\mathit{recomp}) \\ \mathit{recomp} & = & (\_, (\mathit{acc}, (\mathit{head}, \mathit{tail}))) \rightarrow ((\mathit{head}, \mathit{acc}), \mathit{tail}) \end{array}$$

This more complicated filter first binds the whole list to x, then iterates through the list with f. Note that x is in scope in the definition of f. The latter can be seen as a local recursive function. As soon as f reaches the end of the list, it builds the reversed list bottom up while returning from the recursive calls. The last step $((y,\_) \rightarrow y)$ discards the auxiliary argument that was used to build the reversed list. Figure 3.2 gives an example of evaluation of *rev* on the finite list `[1 2 3 4]`. In this figure, evaluation steps are ordered (circled label below the filter expression). First, the list is traversed, (Steps 1 to 5). When the end of the list is reached, an empty accumulator and the whole list are returned (Step 6). Then the accumulator is filled up at each tail of a recursive call (Steps 7 to 11). Finally, only the accumulator is returned (Step 12). Due to the restriction on the composition, if one wants to perform more than list mapping (reversal in our example), one has to resort to using the TABA programming paradigm, as presented by Olivier Danvy and Mayer Goldberg (see “There and back again” [DG05]). In a nutshell, computations occur after returning from a recursive call. While the programming scheme seems unnatural, it has many benefits as Danvy and Goldberg pointed out. In particular, such functions can be efficiently implemented in a continuation passing style. We will further detail this aspects when introducing the compilation scheme for filters in Chapter 6.

$\texttt{[v}_1\ \texttt{v}_2\ \texttt{v}_3\texttt{]}, \texttt{[v}_4\ \texttt{v}_5\ \texttt{v}_6\texttt{]} = v_1, v_2, v_3, v_4, v_5, v_6 = \texttt{[v}_1\ \texttt{v}_2\ \texttt{v}_3\ \texttt{v}_4\ \texttt{v}_5\ \texttt{v}_6\texttt{]}$. This is the case of XDuce but also of XSLT, XQuery and other W3C standards. ℂDuce on the contrary uses a more classical view of lists as nested pairs and provides both `cons` and `@` (also hard-coded).

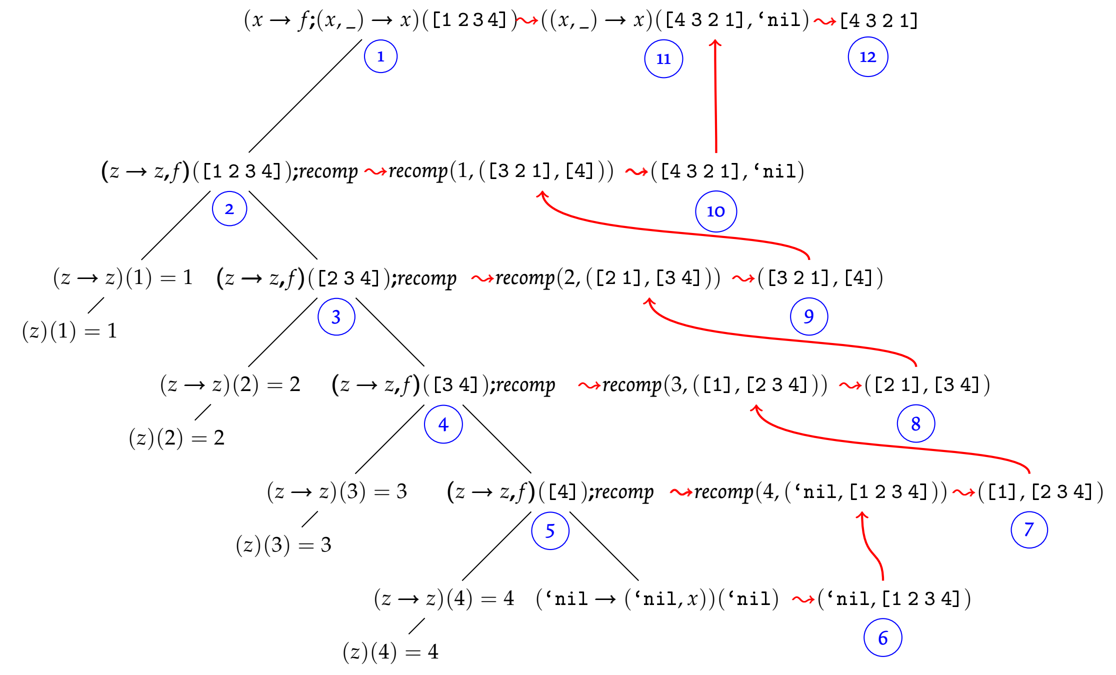

(x → f;(x, _) → x)([1 2 3 4]) ⇝ ((x, _) → x)([4 3 2 1], ‘nil) ⇝ [4 3 2 1]
1
11
12
(z → z,f)([1 2 3 4]);recomp ⇝ recomp(1, ([3 2 1], [4])) ⇝ ([4 3 2 1], ‘nil)
2
10
(z → z)(1) = 1
(z)(1) = 1
(z → z,f)([2 3 4]);recomp ⇝ recomp(2, ([2 1], [3 4])) ⇝ ([3 2 1], [4])
3
9
(z → z)(2) = 2
(z)(2) = 2
(z → z,f)([3 4]);recomp ⇝ recomp(3, ([1], [2 3 4])) ⇝ ([2 1], [3 4])
4
8
(z → z)(3) = 3
(z)(3) = 3
(z → z,f)([4]);recomp ⇝ recomp(4, (‘nil, [1 2 3 4])) ⇝ ([1], [2 3 4])
5
7
(z → z)(4) = 4
(z)(4) = 4
(‘nil → (‘nil, x))(‘nil) ⇝ (‘nil, [1 2 3 4])
6

Figure 3.2: Evaluation of *rev*([1 2 3 4])

Now that we have given some insight of how composition works, we can illustrate the restriction on composition we gave in Definition 3.1. Consider for example the following filter which is not well-formed:

Example 3.12

$$f = (x \to (x, x),x \to (x, x));f$$

On every infinite branch of the filter there is an infinite number of (,) constructors (contractivity), but the second part of the composition, is a sub-term of the whole filter f (as it is f itself). The evaluation of such a filter diverges for every input:

$$\begin{array}{rcc} f((0,0)) & \rightsquigarrow & f((0,0),(0,0)) \\ f((0,0),(0,0)) & \rightsquigarrow & f(((0,0),(0,0)),((0,0),(0,0))) \\ f(((0,0),(0,0)),((0,0),(0,0))) & \rightsquigarrow & \dots \end{array}$$

The problem with this filter is that even if we ensured that the input is always destructured (contractivity), a bigger input is recreated and passed back as argument to the filter. The filter, then loops and creates bigger and bigger values. Let us keep in mind that, for the type inference phase to terminate, filters must always terminate too as they are (abstractly) evaluated on types.

3.5 Termination

As stated before, thanks to the restriction on composition, the evaluation of a filter on a finite value always terminates. To prove termination, we first introduce a particular measure of the filter:

Definition 3.13
We define the function $c : \mathbb{F} \rightarrow \mathbb{N}$ as:

$$\begin{array}{lcll} c(f) & = & 0 & \textit{if } (f_1,f_2) \notin \textit{Subtree}(f) \textit{ for some } f_1 \textit{ and } f_2. \\ c(p \rightarrow f_1) & = & 1 + c(f_1) & \\ c(f_1|f_2) & = & 1 + \textit{max}(c(f_1), c(f_2)) & \\ c(f_1;f_2) & = & c(f_1) & \\ c((f_1,f_2)) & = & 0 & \end{array}$$

To understand why this measure c is useful, it should first be noted that our set of rules does not enjoy the subformula property. First, in the rule **(e-comp-ok)**, the intermediate result r_1 in the first premise is not a sub-term of the goal of the rule. The condition on the "**;**" operator ensures the termination for this case. As for the other rules, the filter(s) in the premise(s) might not be a sub-tree of the filter in the goal as the latter can be recursive (*i.e.* infinite and regular). However there is a well founded order based on both the syntactic tree of the filter and the input value. Intuitively $c(f) = 0$ if either the filter f is finite or if f is a product filter. Let us consider the following example:

Example 3.14
Let f be the recursive filter defined as:

$$\begin{array}{rcl} f & = & \texttt{‘nil} \rightarrow \texttt{‘nil} \\ & | & \texttt{[Int+]} \rightarrow (x \rightarrow x, f) \end{array}$$

And let us consider its application to a value $v =$ `[1]` (which is syntactic sugar for (`1,‘nil`). We have the following derivation steps:

filter		after reduction step	rule	c
f(`[1]`)	$\leadsto$	`[Int+]` $\rightarrow (x \rightarrow x,f)$(`[1]`)	**(e-union2)**	2
`[Int+]` $\rightarrow (x \rightarrow x,f)$(`[1]`)	$\leadsto$	$(x \rightarrow x,f)$(`[1]`)	**(e-patt-ok)**	1
$(x \rightarrow x,f)$(`[1]`)	$\leadsto$	$x \rightarrow x(1)$	**(e-prod-ok)**	0
$x \rightarrow x(1)$	$\leadsto$	$(x)(1)$	**(e-patt-ok)**	0
$(x)(1)$	$\leadsto$	1	**(e-expr)**	0
$(x \rightarrow x,f)$(`[1]`)	$\leadsto$	f(`[]`)	**(e-prod-ok)**	0
	$\vdots$			

To show the normalization theorem, we want to proceed by induction. There are three cases to consider. First, if the filter is finite (such as $x \rightarrow x$ or `'nil` $\rightarrow$ `'nil` in the previous example), the evaluation trivially terminates: each filter in the premises of a rule is a strict-subtree of the filter in the goal of the rule.

Secondly, if a filter (f_1,f_2) is applied to a value $v \equiv (v_1, v_2)$, then the evaluation continues with $f_1(v_1)$ and $f_2(v_2)$. As values are finite, v_1 and v_2 are strict subterms of v. We can use this and show that the size of the input decreases strictly. However, this is only true for product filters. For example, starting from `[Int+]` $\rightarrow (x \rightarrow x,f)((1, \text{`nil}))$ and applying one step of reduction, resulting in $(x \rightarrow x,f)((1, \text{`nil}))$ the input is still $(1, \text{`nil})$. Moreover, as the filter is recursive (in the example, f is a subtree of itself), neither the size of the filter nor the size of the input decrease during this step. This is why we use the special measure c, which represents the contractivity condition for filters. Simply put, $c(f)$ is the depth between the current node in the syntactic tree of f and the next product node (which forces the destructuration of the input). This number is finite even for an infinite filter (thanks to the contractivity condition), allowing us to prove the finiteness of the evaluation. We can now state the normalization theorem:

Theorem 3.15 (Termination of filtering) *Let v be a finite value of the language and f a well-formed filter, in which every expression sub-term terminates for all well-typed substitution. Then the evaluation of $f(v)$ terminates.*

Proof We use an induction on the triple $(f, v, c(f))$ equiped with the well founded order $(\sqsubseteq, \sqsubseteq, \leq_{\mathbb{N}})_{lex}$. The idea is to prove that at each step of the evaluation:

- either we evaluate a strict sub-term of f. The number of distinct subterms of a regular tree being finite, the evaluation terminates;
- or we decompose a pair in its two component (and as values are finite, the evaluation terminates);
- or we get closer to a (f_1, f_2) filter strictly which will either destructurate a value or fail if this value is not a pair.

By case analysis on the evaluation rules:

(e-eval): Trivial as we have assumed that all expressions terminate.

(e-prod-ok): Here the measure decreases strictly in v because we deconstruct the pair v_1, v_2 (and of course, $f_1 \sqsubseteq (f_1,f_2)$ and $f_2 \sqsubseteq (f_1,f_2)$, so the first component of our order does not increase). By induction hypothesis, the two premises terminate and so does the goal of this rule.

(e-prod-err1): here the evaluation terminates directly on Ω (no premises).

(e-prod-err2): like the previous case.

(e-patt-ok): Either f contains an (_,_) in which case $c(f) <_{\mathbb{N}} c(p \to f)$, or it does not and is therefore finite (because an infinite filter would contain an infinite number of (_,_) nodes). In the first case, the order decreases in $c(f)$, in the second case, as the filter is finite, we have $f \sqsubset p \to f$ hence the measure decreases on the first component. In both cases we can apply the induction hypothesis.

(e-patt-err): the evaluation terminates with Ω

(e-comp-ok): for the second premise, f_2 is a strict sub-term of $f_1; f_2$ by definition, hence the evaluation of $f_2(r_1)$ terminates, by induction hypothesis. For the first premise:

- either f_1 contains an (_,_) in which case the third component decreases strictly: $c(f_1) <_{\mathbb{N}} c(f_1; f_2)$.
- or f_1 does not contain any product filter and is therefore finite.

in both cases, it terminates.

(e-comp-err) : terminates with Ω

(e-union1) : let us consider a filter $f_1|f_2(v)$ where $f_1(v)$ is applied. Either f_1 does not contain any (_,_) and it is therefore finite. If this is not the case, then again, $c(f_1) <_{\mathbb{N}} f_1|f_2(v)$, and by induction hypothesis, the evaluation terminates.

(e-union2) : symmetric of the previous case.

Of course, as one can clearly see in Theorem 3.15, if one of the expression subfilters does not terminate, then the whole filter may diverge. However this only means that the non termination is decided by the programmer because either (s)he applied it to a cyclic value or used a diverging expression. In both cases it is not inherently due to the iterating part of the filter. ev

Chapter 4

Static semantics

> This chapter is devoted to the presentation of the type-system for filters. We explain the design choices we made while devising this system and prove that it satisfies the subject reduction property.

Contents

4.1 Type-system

4.1.1 General presentation

Given than filters are computational objects, it is natural to want to associate them with a domain and a co-domain. However, since our goal is to type those objects more precisely than functions, given a subset of their domain, we want to compute an approximation of the image which is much more precise than the co-domain.

Our first attempt is to extend the notion of accepted type of a pattern, $\wr p \int$ to the accepted type of a filter which would be its “domain”. Since $\wr p \int$ is co-inductively defined on the (regular) structure of the pattern p (see [Fri04b]), let us try this approach on filters, seen as regular terms:

Definition 4.1 (Accepted type)
For every filter $f \in \mathbb{F}$ we define the type $\wr f \int$ as follows:

$$
\begin{array}{lcl}
\wr e \int & = & \texttt{Any} \\
\wr (f_1,f_2) \int & = & (\wr f_1 \int \times \wr f_2 \int) \\
\wr f_1 | f_2 \int & = & \wr f_1 \int \vee \wr f_2 \int \\
\wr p \to f \int & = & \wr p \int \wedge \wr f \int \\
\wr f_1 ; f_2 \int & = & \wr f_1 \int
\end{array}
$$

Lemma 4.2 (Necessary inclusion) *Let v be a value and f a filter. If $v \notin \wr f \int$ then $f(v) \rightsquigarrow \Omega$.*

> **Proof** This is immediate since the definition of $\wr f \int$ matches the side conditions of the rules in Figure 3.1. If $v \notin \wr f \int$ then the side conditions do not hold and consequently, $f(v) \rightsquigarrow \Omega$.

Being in $\wr f \int$ is a necessary condition for the evaluation to succeed. Unfortunately, it is not sufficient. For instance, the accepted type of $\texttt{Any} \to \texttt{3} \ ; \ (\_ \times \_) \to \texttt{5}$ is $\texttt{Any}$, but every application of this filter fails, since it tries to match $\texttt{3}$ against a pair pattern. The problem lies, as one would expect, in the composition operator $f_1;f_2$. Indeed, a necessary condition is that the *output* type of f_1 is a subtype of the *input* type of f_2. To ensure type safety, we need to infer the output type of the filter f_1, that is, an approximation of the image set. To that end we define the inference rules of Figure 4.1. The system proves judgements of the form $\Gamma \vdash f(t) = s$ meaning that in a type environment Γ a filter f applied to an expression of type t returns a value of type s. We call $\mathscr{F}$ the associated deduction system and only consider (possibly infinite) regular derivations of this system.

The rules in $\mathscr{F}$ are motivated by our need to be as precise as possible. Indeed, one basic case of type inference is when the input type t of a filter f is a singleton type $\{v\}$, where v is a value of the language. In this case, we expect the type system to infer an output type $s \equiv \{u\}$ where $\varnothing \vdash_e f(v) \rightsquigarrow u$. Our intuition is that, to be precise, the output type of the filter must be computed by an *evaluation* of the filter on the input type. A type is however not a value: a type is at the same time *less precise* then a value (it denotes a set of values) and may be *infinite* (values cannot).

The infinite aspect of types is handled in our system by considering *regular derivations*, co-inductively defined by the set of rules instead of a more classical inductive

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

(t-patt)
$$\frac{t/p \uplus \Gamma \vdash f(t) = s \qquad t \leq \lbag p \rbag \wedge \lbag f \rbag}{\Gamma \vdash (p \rightarrow f)(t) = s}$$

(t-prod)
$$\frac{\pi(t) = \{(t_1^1 \times t_2^1), \ldots, (t_1^n \times t_2^n)\} \quad \Gamma \vdash f_1(t_1^i) = s_1^i \quad \Gamma \vdash f_2(t_2^i) = s_2^i}{\Gamma \vdash (f_1{,}f_2)(t) = \bigvee_{i \in 1..n} (s_1^i \times s_2^i)}$$

(t-union)
$$\frac{\begin{array}{l} t \leq \lbag f_1 \rbag \vee \lbag f_2 \rbag \\ t_1 = t \wedge \lbag f_1 \rbag \\ t_2 = t \setminus \lbag f_1 \rbag \qquad \Gamma \vdash f_1(t_1) = s_1 \qquad \Gamma \vdash f_2(t_2) = s_2 \end{array}}{\Gamma \vdash (f_1 | f_2)(t) = \bigvee_{\{i | t_i \neq \mathsf{Empty}\}} s_i}$$

(t-comp)
$$\frac{\begin{array}{l} t \leq \lbag f_1 \rbag \\ s_1 \leq \lbag f_2 \rbag \qquad \Gamma \vdash f_1(t) = s_1 \qquad \Gamma \vdash f_2(s_1) = s_2 \end{array}}{\Gamma \vdash (f_1;f_2)(t) = s_2}$$

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

Figure 4.1: Deduction system associated with $\mathscr{F}$

definition. This allows us to express only by six rules the complexity we sketched in the introduction, in particular to take *any* valid regular approximation of a non-regular type into account. We will show, in Chapter 5, that forcing an inductive presentation, *i.e.* an algorithmic system, does not provide such flexibility (and that indeed, a particular regular derivation is to be chosen in that case). We reflect this co-inductive nature in our syntax by using a double bar to separate the premises and the goal of a rule.

To achieve a *precise computation* in the presence of less precise objects —types instead of values— we model each rule on the corresponding evaluation rule, presented in Chapter 3, and add specific side-conditions in order to maintain precision.

Regularity both for filters and for deductions prevents Γ from growing indefinitely (regularity of filters guarantees that the number of distinct variables on an infinite branch is finite and regularity of deductions ensures that these variables can be assigned only to finitely many types).

Most of the rules require that the input type is compatible with the accepted type of the considered filter. To type an (host language) expression,the rule (**t-expr**) relies on the type system of the host language with the current environment.

To type a product (rule **(t-prod)**) we use the decomposition π introduced in Definition 2.26. For each pair in the decomposition, we apply the sub-filters f_1 and f_2 respectively to the first and second projection of the product and rebuild a product of the result. Finally, the union of every sub-product is returned. Typing products hence depends on the very nature of the decomposition. We will only assume the properties described in Definition 2.26 to prove subject reduction. However, as we will see in Section 4.2.3 the way products are decomposed has a direct impact on type precision. We will present a particular case of interest, the *maximal product decomposition* (see Definition 4.8), together with its properties.

Typing a pattern filter $p \to f$ merely consists in typing f under an environment enriched with t/p; the latter —defined by Theorem 2.34— is the type environment that assigns to each capture variable in p the most precise type that can be deduced for it when the pattern is matched against a value of type t.

The three remaining rules are subtler: the composition rule, **(t-comp)**, while simple in appearance, shows its power when coupled with the subsumption rule **(t-subs)**. As for the typing of the union, the rule **(t-union)** exhibits some of the details one needs to be aware of to achieve precise type inference. We devote the next two subsection to describe these rules.

4.1.2 Typing the composition

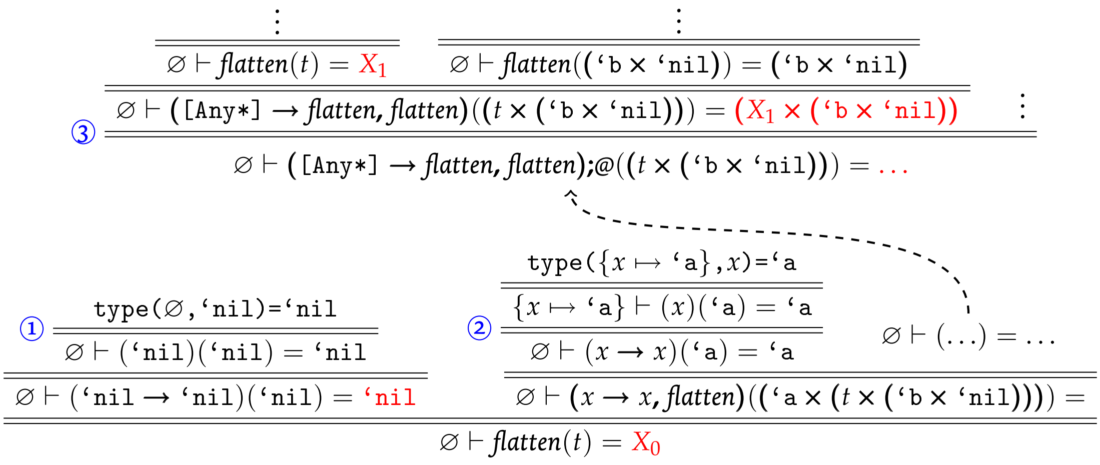
As explained in the previous section, our aim is to *evaluate* a filter on a type (rather than on a value) in order to deduce an output type. In this respect, the rule **(t-comp)** which allows us to type the composition seems quite straightforward. Given an input type t, we apply the first filter f_1 on it, deduce an intermediary type s_1 onto which we can apply the second filter f_2 to deduce the output type s_2. While simple in principle, the effects of this rule on the type-system are quite deep.
To see why this is the case, let us consider the *flatten* filter, defined as follows:

Example 4.3

$$\begin{array}{rcl} \mathit{flatten} & = & \texttt{`nil} \to \texttt{`nil} \\ & | & (\texttt{[Any*]} \to \mathit{flatten}, \mathit{flatten});@ \\ & | & (x \to x, \mathit{flatten}) \end{array}$$

In this definition, @ denotes the concatenation filter defined in Example 3.10. The semantics of this filter is natural: if its argument is the empty list, it returns the empty list. If its argument is a list the first element of which is also a list, then it recursively flattens this first elements,then the tail and concatenates them both. If the first argument is not a list, it recursively flattens the tail of the list. As already said, if this filter is applied to the type $t = \texttt{`nil} \lor (\texttt{`a} \times (t \times (\texttt{`b} \times \texttt{`nil})))$ (which in concrete syntax writes: $t = \texttt{`nil} \lor \texttt{[`a t `b]}$), then the most precise output type is the set of values: $\{\texttt{[`a}^{\texttt{n}}\ \texttt{`b}^{\texttt{n}}\texttt{]} | n \geq 0\}$ which is not a regular set[1]. To see why the "non-regularity" can really arise in the presence of composition, let us try to derive $\varnothing \vdash \mathit{flatten}(t) = ?$, without using the subsumption rule. We see in Figure 4.2 that the

[1] We recall that ‘a denotes the singleton type whose only inhabitant is the atom ‘a.



$\varnothing \vdash \mathit{flatten}(t) = X_1$
$\varnothing \vdash \mathit{flatten}((\texttt{`b} \times \texttt{`nil})) = (\texttt{`b} \times \texttt{`nil})$
$\varnothing \vdash (\texttt{[Any*]} \to \mathit{flatten}, \mathit{flatten})((t \times (\texttt{`b} \times \texttt{`nil}))) = (X_1 \times (\texttt{`b} \times \texttt{`nil}))$
③
$\varnothing \vdash (\texttt{[Any*]} \to \mathit{flatten}, \mathit{flatten});@((t \times (\texttt{`b} \times \texttt{`nil}))) = \ldots$
①
$\texttt{type}(\varnothing, \texttt{`nil}) = \texttt{`nil}$
$\varnothing \vdash (\texttt{`nil})(\texttt{`nil}) = \texttt{`nil}$
$\varnothing \vdash (\texttt{`nil} \to \texttt{`nil})(\texttt{`nil}) = \texttt{`nil}$
②
$\texttt{type}(\{x \mapsto \texttt{`a}\}, x) = \texttt{`a}$
$\{x \mapsto \texttt{`a}\} \vdash (x)(\texttt{`a}) = \texttt{`a}$
$\varnothing \vdash (x \to x)(\texttt{`a}) = \texttt{`a}$
$\varnothing \vdash (\ldots) = \ldots$
$\varnothing \vdash (x \to x, \mathit{flatten})((\texttt{`a} \times (t \times (\texttt{`b} \times \texttt{`nil})))) =$
$\varnothing \vdash \mathit{flatten}(t) = X_0$

Figure 4.2: Typing derivation for the *flatten* filter

output type of the *flatten* filter applied to t is (in informal notation):

$$\mathit{flatten}(t) = \texttt{`nil} \vee (\texttt{`a} \times (\mathit{flatten}(t)@(\texttt{`b} \times \texttt{`nil})))$$

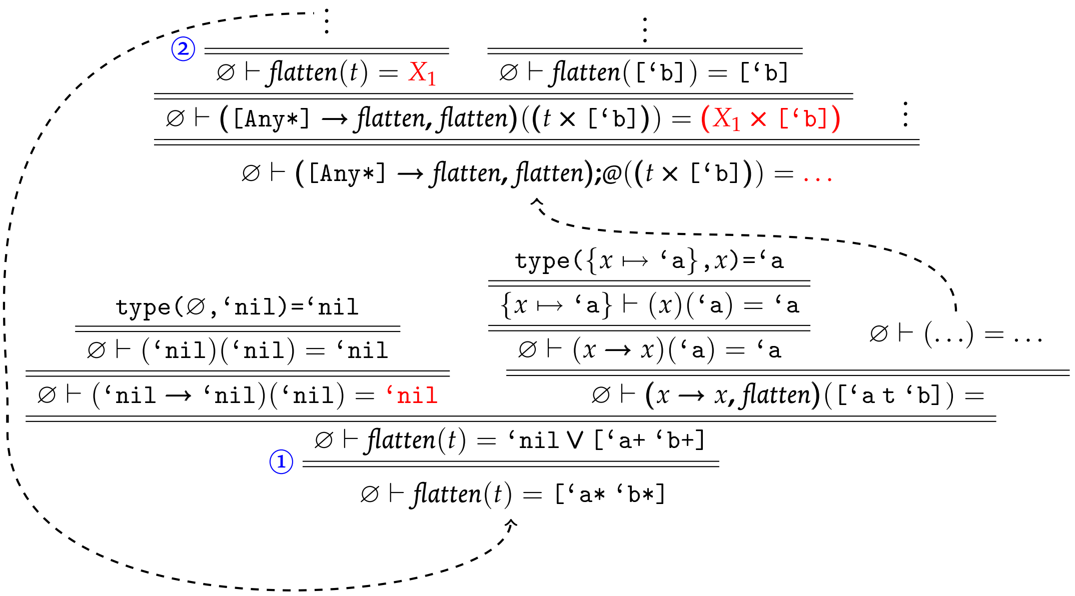
The $\texttt{`nil}$ and $(\texttt{`a} \times \ldots)$ come from Steps 1 and 2 of the derivation (circled). However the latter part is much more tricky. Indeed, it involves the rule **(t-comp)**, (Step 3) which behaves as a logical cut: it introduces an intermediary type. In this case, the worse situation happens as, to be able to type this filter, the only acceptable output for $\mathit{flatten}(t)$ is $\{\texttt{[`a}^{\texttt{n}}\ \texttt{`b}^{\texttt{n}}\texttt{]} | n \geq 0\}$. Indeed if we take $X_1 = \{\texttt{[`a}^{\texttt{n}}\ \texttt{`b}^{\texttt{n}}\texttt{]} | n \geq 0\}$, then at Step 3, the output type is the concatenation of X_1 and $(\texttt{`b} \times \texttt{`nil})$, which is $\{\texttt{[`a}^{\texttt{n}}\ \texttt{`b}^{\texttt{n+1}}\texttt{]} | n \geq 0\}$. Then, at the end of Step 2, we prepend $\texttt{`a}$ to this type to obtain: $\{\texttt{[`a}^{\texttt{n+1}}\ \texttt{`b}^{\texttt{n+1}}\texttt{]} | n \geq 0\}$. Finally, the use of the **(t-union)** rule together with the result $\texttt{`nil}$ of Step 1 gives as final result: $\texttt{`nil} \vee \{\texttt{[`a}^{\texttt{n+1}}\ \texttt{`b}^{\texttt{n+1}}\texttt{]} | n \geq 0\}$ which is nothing else but $\{\texttt{[`a}^{\texttt{n}}\ \texttt{`b}^{\texttt{n}}\texttt{]} | n \geq 0\}$. What happens here is that this derivation yields *to precise* a type, the output type for $\mathit{flatten}(t)$ must verify:

$$\mathit{flatten}(t) = \texttt{`nil} \vee (\texttt{`a} \times (\mathit{flatten}(t)@(\texttt{`b} \times \texttt{`nil})))$$

which only has a non-regular exact solution.

With a derivation such as the one in Figure 4.2, it is not possible to derive an approximation of $\{\texttt{[`a}^{\texttt{n}}\ \texttt{`b}^{\texttt{n}}\texttt{]} | n \geq 0\}$. Indeed, the system, as it is, is too precise to account for regular approximation. For instance, let us try to type this filter with the (reasonable) approximation $\texttt{[`a* `b*]}$ as output type, that is, let us make the assumption that $\forall i, X_i = \texttt{[`a* `b*]}$. Then the result of Step 3 is to append $(\texttt{`b} \times \texttt{`nil})$ to X_1 thus yielding the intermediary type $\texttt{[`a* `b+]}$. At Step 2, $\texttt{`a}$ is prepended to the intermediary result, giving an output type of $\texttt{[`a+ `b+]}$. Finally, the **(t-union)** rule is applied and give as final output type $X_0 = \texttt{`nil} \vee \texttt{[`a+ `b+]}$ which is strictly more precise that $\texttt{[`a* `b*]}$, which was the assumption we made. Indeed, the system returned a *more precise* type, that is a subtype of our approximation. The problem with the derivation of Figure 4.2, is that whatever regular type we use as X_1, we always end-up with $X_0 \lneq X_1$ (X_0 is a strict subtype of X_1).

Fortunately the subsumption rule allows us to loosen this precision a bit, providing a regular derivation for a regular approximation of $\mathit{flatten}(t)$. This is illustrated in Figure 4.3 where the output type is approximated by the regular type `[‘a* ‘b*]`. The important part in this figure are Step 1 and Step 2, which are the same node in the infinite regular proof tree. If, at Step 2, we take X_1 = `[‘a* ‘b*]`, then we end-up at Step 1 with an output type of `‘nil` ∨ `[‘a+ ‘b+]`, as previously. However, now that we have inserted a subsumption rule as “correcting lens”, we can check that `‘nil` ∨ `[‘a+ ‘b+]` ≤ `[‘a* ‘b*]` and then type the filter with the output type `[‘a* ‘b*]`. Thanks to this rule, we have $X_1 = X_0$ = `[‘a* ‘b*]` or said differently, the filter is typed by a regular derivation.



2
$\varnothing \vdash \mathit{flatten}(t) = X_1$
$\varnothing \vdash \mathit{flatten}(\texttt{[‘b]}) = \texttt{[‘b]}$
$\varnothing \vdash (\texttt{[Any*]} \to \mathit{flatten}, \mathit{flatten})((t \times \texttt{[‘b]})) = (X_1 \times \texttt{[‘b]})$
$\varnothing \vdash (\texttt{[Any*]} \to \mathit{flatten}, \mathit{flatten});@((t \times \texttt{[‘b]})) = \ldots$
$\texttt{type}(\varnothing, \texttt{‘nil}) = \texttt{‘nil}$
$\varnothing \vdash (\texttt{‘nil})(\texttt{‘nil}) = \texttt{‘nil}$
$\varnothing \vdash (\texttt{‘nil} \to \texttt{‘nil})(\texttt{‘nil}) = \texttt{‘nil}$
$\texttt{type}(\{x \mapsto \texttt{‘a}\}, x) = \texttt{‘a}$
$\{x \mapsto \texttt{‘a}\} \vdash (x)(\texttt{‘a}) = \texttt{‘a}$
$\varnothing \vdash (x \to x)(\texttt{‘a}) = \texttt{‘a}$
$\varnothing \vdash (\ldots) = \ldots$
$\varnothing \vdash (x \to x, \mathit{flatten})(\texttt{[‘a t ‘b]}) =$
$\varnothing \vdash \mathit{flatten}(t) = \texttt{‘nil} \vee \texttt{[‘a+ ‘b+]}$
1
$\varnothing \vdash \mathit{flatten}(t) = \texttt{[‘a* ‘b*]}$

Figure 4.3: Typing derivation for the *flatten* filter with an approximation

4.1.3 Typing the union

Typing the union filter should be, at first glance quite easy. For example consider the simplified rule:

$$\textbf{(t-union-simple)}\ \frac{\begin{array}{l} t \leq \wr f_1 \wr \vee \wr f_2 \wr \\ t_1 = t \wedge \wr f_1 \wr \\ t_2 = t \setminus \wr f_1 \wr \qquad \Gamma \vdash f_1(t_1) = s_1 \qquad \Gamma \vdash f_2(t_2) = s_2 \end{array}}{\Gamma \vdash (f_1 | f_2)(t) = s_1 \vee s_2}$$

Is not this rule enough? It does not seem unsafe (and it is not) but it may yield to rather imprecise typing. Consider the filter *succList*, presented in Section 3.4.3. A type derivation using the rule **(t-union-simple)** for this filter applied to the type `[ Int+ ]` (which is $t = (\texttt{Int} \times t) \vee (\texttt{Int} \times \texttt{`nil})$) is given in Figure 4.4. As we can

$$\dfrac{\dfrac{\dfrac{\texttt{type}(\varnothing,\texttt{`nil}) = \texttt{`nil}}{\varnothing \vdash (\texttt{`nil})(\texttt{Empty}) = \texttt{`nil}}}{\varnothing \vdash (\texttt{`nil} \to \texttt{`nil})(\texttt{Empty}) = \texttt{`nil}} \qquad \dfrac{\dfrac{\vdots}{\varnothing \vdash (x \to x+1)(\texttt{Int}) = \texttt{Int}} \quad \dfrac{\vdots}{\varnothing \vdash \mathit{succList}(t) = s}}{\varnothing \vdash (x \to x+1,\mathit{succList})(t) = (\texttt{Int} \times s)}}{\varnothing \vdash (\texttt{`nil} \to \texttt{`nil}|(x \to x+1,\mathit{succList}))(t) = \texttt{`nil} \vee (\texttt{Int} \times s)}$$

where $s = \texttt{`nil} \vee (\texttt{Int} \times s)$.

Figure 4.4: Typing derivation for *succList* applied to `[ Int+ ]`

see, the output type is less precise that one would expect: the system built this way infers: `[ Int* ]` instead of `[ Int+ ]`. There was a loss of precision due to the fact that we took into account unnecessary branches of a union filter. Indeed, if the input type for one branch is empty (t_1 or t_2 in rule **(t-union)**), we know *statically* that this branch will not be taken. This explain the check for emptiness of the input type in the rule **(t-union)**. In our example, if at run-time a value is in the type `[Int+]`, then it can never be `` `nil ``, and therefore the first branch is never taken. The type-system must reflect this behaviour and does so by not including the output type of the considered branch to the whole output type. This illustrates an interesting aspect of this type-system: during a typing derivation the same filter (e.g. *succList* in Figure 4.4) can be applied to many different input types (it is nevertheless done finitely many times. This property is crucial for the termination of a type inference algorithm for example and will be proven in Chapter 5). Retyping a filter during a typing derivation is a fundamental difference between our approach and Hosoya's regular expression filters. We will compare in Chapter 6 both approaches on concrete examples and show that ours returns a much more precise output type. Finally this refinement can be used to warn the user of redundant alternatives in a filter. For example in the filter:

Example 4.4

$$\begin{array}{rcl} \mathit{red} & = & \texttt{`nil} \to \texttt{`nil} \\ & | & (x \to x+2,\mathit{red}) \\ & | & (x \to x+2,\mathit{red}) \end{array}$$

the third branch is unnecessary and will never be taken, whatever the input type is. This technique is quite similar to the one used to type *overloaded functions* in the presence of union types (e.g. in the case of ℂDuce as defined in [Fri04b]).

4.2 Properties

As explained in Section 4.1.2, the use of the subsumption rule is necessary to take regular approximations into account. More surprisingly, it is also the *only* case where subsumption is really needed. Indeed, like for the simply typed lambda calculus with subtyping, we can remove all applications of the subsumption rules but those used to prove the first premise of a composition rule. By doing so, we can show that we always obtain a valid derivation which proves a more precise judgement: if $\Gamma \vdash f(t) = s$ can be proven in the type-system by a derivation D, then a derivation D' obtained by removing the "unnecessary" subsumptions from D proves a judgement $\Gamma \vdash f(t) = s'$ where $s' \leq s$. We devote the first part of this section to the proof of this property.

Once this property is proven, we can prove more easily that subject-reduction holds and, consequently, that adding typed filters to an host language does not compromise its safety. We also consider a particular decomposition of products, called maximal product decomposition, and show that by using it, we obtain a property of monotonicity with respect to subtyping for filters. Informally, it states that given a filter f and two types t and s such that $s \leq t$, we have "$f(s) \leq f(t)$". The interest of such a property in terms of programming is that if a data-type t (onto which we apply the transformation f) is refined later on into a type s (making it more restrictive) then the whole program will still type-check and in particular that there will be no need to modify the definition of f.

4.2.1 Use of the subsumption

Before proving the subject-reduction property, we need to prove an important property of the typing derivations in our system, which will also be later used in Chapter 5, for the conception of the typing algorithm. This property (again very similar to what is found for the simply typed lambda calculus with subtyping), that we call *subsumption erasure* states that the only places where the subsumption rule plays a crucial rule (*i.e.* is mandatory to type a filter) are:

- if it is at the bottom of the derivation.
- if it is used to prove the first premise of a composition rule.

Such cases are, for instance, those in Figure 4.3. Before proving this property, we need the following lemma:

Lemma 4.5 *Let f be a filter, Γ a typing environment and t and s two types such that $\Gamma \vdash f(t) = s$ is derivable with a regular derivation D. Then there exists a regular derivation D' for which any application of the rule* **(t-subs)** *is preceded by an application of a rule other than* **(t-subs)** *and followed by an application of a rule other than* **(t-subs)** *or no other rule.*

Proof Let us call P the property we want to prove. We supposed that the derivation D is regular, hence it has a finite number of distinct subderivations. This has two consequences:

i. Every infinite sequence of successive applications of **(t-subs)** has the form (note the reflexivity):

$$\begin{array}{c}
\textbf{(t-subs)}\,\dfrac{\dfrac{\vdots}{\Gamma' \vdash f'(t') = s' \quad s' \leq s'}}{\dfrac{\dfrac{\vdots}{\Gamma' \vdash f'(t') = s' \quad s' \leq s'}}{\vdots}\,\textbf{(t-subs)}}\,\textbf{(t-subs)}
\end{array}$$

ii. There is a finite number $n(D)$ of *distinct* sequences of successive applications of **(t-subs)** in D which do not verify P.

We prove the lemma by induction on $n(D)$.

Basic case: The derivation has no such sequence of consecutive applications of **(t-subs)**. The lemma is trivially true with $D' \equiv D$.

Inductive case: Let us consider a sequence d of successive applications of the rule **(t-subs)** in the derivation D. d can be an infinite sequence of applications of **(t-subs)** as in point (*i*) and we can suppress it altogether (since it does not contribute to the computation of the output type).

- d has the form:

$$d = \begin{array}{c}
\textbf{(t-subs)}\,\dfrac{\dfrac{d_2}{\Gamma' \vdash f'(t') = s' \quad s' \leq s'}}{\dfrac{\dfrac{\vdots}{\Gamma' \vdash f'(t') = s' \quad s' \leq s'}}{d_1}\,\textbf{(t-subs)}}\,\textbf{(t-subs)}
\end{array}$$

  where d_1 and d_2 are not applications of the rule **(t-subs)**. Let us call $d' = \dfrac{d_2}{d_1}$. Let us consider the derivation $D' = D[d \leftarrow d']$, *i.e.* D where we collapsed the unneeded sequences of applications of **(t-subs)**. We have now that d' verifies P, hence that $n(D') < n(D)$, since we removed exactly one sequence of consecutive applications of **(t-subs)**, thus the induction hypothesis holds.

- d has the form:

$$d = \begin{array}{rc}
 & d_1 \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s' \quad s' \leq s'}} \\
\textbf{(t-subs)} & \overline{\overline{\vdots}} \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s' \quad s' \leq s'}}
\end{array}$$

and we take $d' = d_1$. We can apply the induction hypothesis on $D' = D[d \leftarrow d']$ as in the previous case.

Another case is the one where d is an interleaving of infinite sequences and finite numbers of uses of the rule **(t-subs)** where the output type in the premise is a strict subtype of the output type in the conclusion of the rule:

- d has the form: $$d = \begin{array}{rc}
 & d_2 \\
\textbf{(t-subs)} & \overline{\overline{\vdots}} \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s'_k \quad s'_k \leq s'_k}} \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s'_k \quad s'_k \leq s'_{k-1}}} \\
\textbf{(t-subs)} & \overline{\overline{\vdots}} \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s'_1 \quad s'_1 \leq s'_0}} \\
\textbf{(t-subs)} & \overline{\overline{\vdots}} \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s'_0 \quad s'_0 \leq s'}} \\
\textbf{(t-subs)} & \overline{\overline{d_1}}
\end{array}$$

By using the transitivity of the subtyping relation, we have that: $s'_k \leq s'_{k-1} \leq \ldots \leq s'_0 \leq s'$. We can form an equivalent derivation by collapsing all the applications of **(t-subs)** into one. Let

$$d' = \begin{array}{c}
d_2 \\
\overline{\overline{\Gamma' \vdash f'(t') = s'_k \quad s'_k \leq s'}} \\
\overline{\overline{\Gamma' \vdash f'(t') = s'}} \\
\overline{\overline{d_1}}
\end{array}$$

. The property P holds for d'.

If we consider $D' = D[d \leftarrow d']$ then $n(D') < n(D)$, so the induction hypothesis holds for D'.

- The last case is the one for which:

$$d = \begin{array}{rc}
 & d_2 \\
\textbf{(t-subs)} & \overline{\overline{\vdots}} \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s'_k \quad s'_k \leq s'_k}} \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s'_k \quad s'_k \leq s'_{k-1}}} \\
\textbf{(t-subs)} & \overline{\overline{\vdots}} \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s'_1 \quad s'_1 \leq s'_0}} \\
\textbf{(t-subs)} & \overline{\overline{\vdots}} \\
\textbf{(t-subs)} & \overline{\overline{\Gamma' \vdash f'(t') = s'_0 \quad s'_0 \leq s'}}
\end{array}$$

We can use $d' = \dfrac{\dfrac{d_2}{\Gamma' \vdash f'(t') = s'_k \quad s'_k \leq s'}}{\Gamma' \vdash f'(t') = s'}$
and apply the induction hypothesis on the derivation $D' = D[d \leftarrow d']$.

Our next goal is to prove that the only places in a derivation where **(t-sub)** rules are needed are:

- to prove the first premise of a composition
- at the very bottom of the proof

Indeed, using this property, we can then assume without loss of generality that any derivation in the system can be rewritten so as to meet the above conditions. Doing so effectively reduces the number of cases we have to prove for subject reduction to hold.

Lemma 4.6 *Let f be a filter, Γ a typing environment and t and s two types such that $\Gamma \vdash f(t) = s$ is derivable with a regular derivation D. There exists D' such that, D' is a valid regular derivation of $\Gamma \vdash f(t) = s$ and:*

i. $D' = \dfrac{\dfrac{D''}{\Gamma \vdash f(t) = s' \quad s' \leq s}}{\Gamma \vdash f(t) = s}$

ii. an application of the rule **(t-subs)** *in D'' is immediately followed by an application of the rule* **(t-comp)**

Proving such a lemma would not be difficult if we were in the standard situation of *finite typing derivations*. However we are in the context of *co-inductive* derivations, hence using an induction on some well founded order on the derivation would be cumbersome and inelegant. In this case, using the co-induction principle is much cleaner. We use the "proof theoretic" approach to the co-induction principle, recalled in Chapter 2 and developed in [LG07].

Proof First of all, by Lemma 4.5, it is sufficient to consider a derivation D such that an application of **(t-subs)** is immediately followed by a rule other than **(t-subs)**. To apply the co-induction principle, we need to build a *finite* system of guarded recursive equations (see Chapter 2, Section 2.3.2). This system has a unique solution which is a regular derivation that proves $\Gamma \vdash f(t) = s$. To that end, we define a generating function E from typing derivations to systems of equations over derivations. If d is a regular typing derivation, then $E(d)$ is a set of equations of unknowns x_j, where js are judgments occuring in the derivation d. There are three cases:

i. The goal of the derivation is a composition whose first premise is a subsumption:

$$E\left(\ \textbf{(t-comp)}\,\dfrac{\textbf{(t-subs)}\,\dfrac{\dfrac{d_1}{j_1'}}{j_1} \qquad \dfrac{d_2}{j_2}}{j}\ \right) = \{x_j = \textbf{(t-comp)}\,\dfrac{x_{j_1} \qquad x_{j_2}}{j}\} \cup$$

$$\{x_{j_1} = \textbf{(t-subs)}\,\dfrac{x_{j_1'}}{j_1}\} \cup E\left(\dfrac{d_1}{j_1'}\right) \cup$$

$$E\left(\dfrac{d_2}{j_2}\right)$$

ii. The goal of the derivation is a subsumption:

$$E\left(\ \textbf{(t-subs)}\,\dfrac{\dfrac{d}{\Gamma' \vdash f'(t') = s's' \leq s''}}{\Gamma' \vdash f'(t') = s''}\ \right) = E\left(\ \dfrac{d}{\Gamma' \vdash f'(t') = s'}\ \right)$$

iii. The goal of the derivation is neither a subsumption nor a composition:

$$E\left(\ \textbf{(rule)}\,\dfrac{\dfrac{d_1}{j_1} \quad \dots \quad \dfrac{d_n}{j_n}}{j}\ \right) = \{x_j = \textbf{(rule)}\,\dfrac{x_{j_1} \quad \dots \quad x_{j_n}}{j}\} \cup$$

$$E\left(\dfrac{d_1}{j_1}\right) \cup \dots \cup E\left(\dfrac{d_n}{j_n}\right)$$

$$\textbf{rule} \neq \textbf{(t-subs)} \text{ or } \textbf{rule} \neq \textbf{(t-comp)}$$

First of all, note that since the argument of E is a regular derivation d, $E(d)$ is a finite set of equations (E generate at most one equation for each distinct sub-derivation of d). We know that D is a valid regular derivation of $\Gamma \vdash f(t) = s$. Let us consider the system of equations $E(D)$. By construction, we know that each equation in $E(D)$ is guarded, that is there is no trivial equation $x = x'$. By [Cou83] we know that this system has a unique solution, Δ. Furthermore, for every judgement j in the derivation D such that x_j is defined, $\Delta(x_j)$ is a valid typing derivation that proves j. Let us consider D. There are three cases:

- D has the following shape:

  $$D = \textbf{(t-comp)}\,\dfrac{\textbf{(t-subs)}\,\dfrac{d_1}{j_1} \qquad d_2}{j}$$ where $j \equiv \Gamma \vdash f(t) = s$. By construction, $\Delta(x_j)$ is a valid derivation for $\Gamma \vdash f(t) = s$ without subsumptions rules others than on the first premise of a composition rule. The derivation we seek is then:

  $$D' = \textbf{(t-subs)}\,\dfrac{\dfrac{\Delta(x_j)}{\Gamma \vdash f(t) = s\ s \leq s}}{\Gamma \vdash f(t) = s}$$

- D ends by a subsumption rule:

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

where $j \equiv \Gamma \vdash f(t) = s'$. By construction, (case *ii.*), $\Delta(x_j)$ is a valid typing derivation for j. We can then take:

$$D' = \textbf{(t-subs)}\,\frac{\frac{\Delta(x_j)}{\Gamma \vdash f(t) = s' \quad s' \leq s}}{\Gamma \vdash f(t) = s}$$ as the desired derivation.

- The last case is similar to the first one:

$D = \dfrac{d_1 \quad \dots \quad d_n}{j}$ where $j \equiv \Gamma \vdash f(t) = s$. By construction, $\Delta(x_j)$ is a valid derivation for $\Gamma \vdash f(t) = s$ without subsumptions rules others than on the first premise of a composition rule. The derivation we seek is then:

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

4.2.2 Subject reduction

Traditionally (at least for simply-typed lambda calculi), type safety is shown in two steps. The first one is to prove type preservation (or subject reduction), that is if a term t has type s then after one step of reduction, the resulting term t' has type s. The second step is to prove the *progress* property, which states that a well typed term t is either a value or can be reduced to different term t'. In our framework however, we favored a big step semantics to formalise the evaluation of filters. Furthermore, we showed that the evaluation of any filter terminates. Consequently, we only need to prove a variant of type preservation, adapted to our big step semantics.

Theorem 4.7 (Subject reduction) *Let f, Γ, t and s be such that $\Gamma \vdash f(t) = s$. For every $\gamma : \Gamma$ and $v : t$, we have that $\gamma \vdash_e f(v) \leadsto v'$ implies $v' : s$*

Proof To simplify the proof, we can use Lemma 4.6 and consider a typing derivation of $\Gamma \vdash f(t) = s$ where the subsumption can only occur on the first premise of a composition rule. Without loss of generality, we can also consider that the derivation of $\Gamma \vdash f(t) = s$ does not end by a subsumption rule (if it does, then we can consider the judgment $\Gamma \vdash f(t) = s'$ with $s' \leq s$ and the value $v' : s'$). This allows us to avoid distributing the subsumption rule to every case (as the subsumption rule can occur anywhere in a well formed typing derivation), thus simplifying the proof.

We use an induction on the derivation of $\gamma \vdash_e f(v) \rightsquigarrow v'$ which is finite.

- Base of the induction:

  (e-expr) : we know that the filter is well-typed, hence the type of e is s. The evaluation of e gives us a value v' of type s (we rely on subject reduction for the target expression language).

  (e-*-err) : None of the error cases can occur since the filter is supposed to be well typed, and the side conditions in the typing rules ensures that erroneous cases cannot occur.

- Induction:

  (e-prod-ok) : $f \equiv (f_1,f_2)$ and $v \equiv (v_1, v_2)$. By hypothesis, (v_1, v_2) has type $t \simeq \bigvee_{(t_1^i,t_2^i,)\in\pi(t)} (t_1^i, t_2^i)$. Therefore, there exists i, such that $v_1 : t_1^i$ and $v_2 : t_2^i$. We know that the filter is well-typed, hence the premises of rule **(t-prod)** hold:

  $$\Gamma \vdash f_1(t_1^i) = s_1^i$$

  and by induction,

  $$\gamma \vdash_e f_1(v_1) \rightsquigarrow v_1' \text{ with } v_1' : s_1^i.$$

  The same applies for $\gamma \vdash_e f_2(v_2) \rightsquigarrow v_2'$ with $v_2' : s_2^i$. Consequently, $v' \equiv (v_1', v_2') : s_1^i \times s_2^i$. We can finally remark that:

  $$s_1^i \times s_2^i \leq \bigvee_{i\in 1..\mathtt{rank}(t)} (s_1^i \times s_2^i)$$

  and so:

  $$v' \equiv (v_1', v_2') : \bigvee_{i\in 1..\mathtt{rank}(t)} (s_1^i \times s_2^i)$$

  (e-patt-ok) : thanks to Theorem 2.34, we know that if $v : t$ then $v/p : t/p$. Hence, by hypothesis, the premise of this rule holds: $t/p \uplus \Gamma \vdash f(t) = s$; and, by induction hypothesis, $v/p \uplus \gamma \vdash_e f(v) \rightsquigarrow v'$ and $v' : s$ so $\gamma \vdash_e (p \rightarrow f)(v) \rightsquigarrow v'$ and $v' : s$.

  (e-comp-ok) : We know that the filter is well typed. There are two cases for the first premise. Either it is proven by a rule other than **(t-subs)**, in which case the derivation for $\Gamma \vdash f_1(t) = s_1$ does not end by **(t-subs)**, and we can apply the induction hypothesis: $\gamma \vdash_e f_1(v) \rightsquigarrow v'$ and $v' : s_1$. If $\Gamma \vdash f_1(t) = s_1$ is proved using **(t-subs)**, then there is a derivation for $\Gamma \vdash f_1(t) = s_1'$ with $s_1' \leq s_1$ which does not start by **(t-subs)**. We can apply the induction hypothesis and find that $\gamma \vdash_e f_1(v) \rightsquigarrow v'$ and $v' : s_1'$. Finally, by definition of the subtyping relation, if $v' : s_1'$ and

$s'_1 \leq s_1$, then $v' : s_1$. For the second premise, $\Gamma \vdash f_2(s_1) = s_2$, and by induction hypothesis $\gamma \vdash_e f_2(v') \rightsquigarrow v''$ and $v'' : s_2$. Hence, $\gamma \vdash_e (f_1;f_2)(v) \rightsquigarrow v''$ and $v'' : s_2$.

(e-union1) and **(e-union2)**: These two rules must be proved together. Indeed, we know that $v : t_1 \vee t_2$. But, since v is a value, we can refine by saying that either $v : t_1$, or $v : t_2 \setminus t_1$. If $v : t_1$ then, we apply **(e-union1)** and $\gamma \vdash_e f_1(v) \rightsquigarrow v'$ with $v' : s_1$ (in particular, $v' \neq \Omega$). Otherwise, **(e-union2)** applies. If $v : t_2 \setminus t_1$, we have indeed that $v \notin \wr t_1 \int$ (by definition), so $\gamma \vdash_e f_1(v) \rightsquigarrow \Omega$, and, $v : t_2$ so $\gamma \vdash_e f_2(v) \rightsquigarrow v'$ with $v' : s_2$ (by induction hypothesis). Hence, either $v' : s_1$ or $v' : s_2$. Since $s_1 \leq s_1 \vee s_2$ and $s_2 \leq s_1 \vee s_2$,then we get the expected result: $v' : s_1 \vee s_2 \equiv s$

Let us emphasize the importance of the elemination of the subsumption rule here. Without the elimination of the subsumption, a filter of the form (f_1,f_2) is not necessarily proved by the rule **(t-prod)** but can also be proven by a rule **(t-subs)**. This would have made the proof much more tedious, duplicating each case in two, as it is done for the composition rule.

4.2.3 Monotonicity

Typing of Cartesian products (*i.e.* rule **(t-prod)**) can be further refined by carefully choosing the decomposition function π. Indeed, while every subtype of (`Any` × `Any`) can be decomposed into an union of products, the decomposition is not unique. However, there exists a decomposition (that we dub *maximal product decomposition*) given by the operator ピ that has better properties (with respect to subtyping)[2]. Typing Cartesian products can be tricky since not every decomposition of a product in a finite union of Cartesian products is stable with respect to the subtyping relation. Stability with respect to subtyping would give as the monotonicity property presented hereafter. Let us illustrate this on an example involving interval types (the notation of which was given in Section 2.4.2). Consider the following filters:

$$f_1{=}\texttt{0..3} \rightarrow \texttt{A}|\texttt{4..7} \rightarrow \texttt{B} \qquad f_2{=}\texttt{0..4} \rightarrow \texttt{C}|\texttt{0..6} \rightarrow \texttt{D} \qquad f{=}(f_1,f_2)$$

and the types t and s:

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

It is clear that $s \leq t$: by drawing all intervals on a plane, as in Figure 4.5, it is easy to check that the rectangle s is contained in the "▟"-shaped t. However, s overlaps the two rectangles which form t without being completely contained in any of them. If we decompose naively (i.e., syntactically) both types and compute the result type of f by separately applying the filter on each component of the obtained decomposition, then we have:

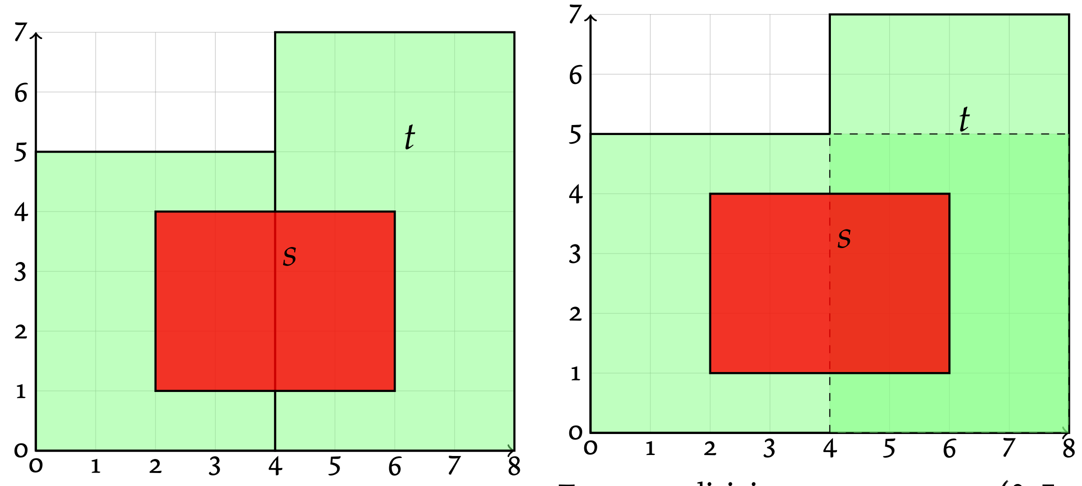
$$\varnothing \vdash f(t) = (\texttt{A} \times \texttt{C})|(\texttt{B} \times \texttt{D})$$

[2] This Japanese character is pronounced *[pi:]* (as for "pea"), which is the French spelling for the Greek letter π. Globalization has an impact, even on function names.

but also:

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

the latter being a super-type of the former. Indeed, in f_1, a value in $4..6$ can match either $0..4$ or $5..8$ (and likewise for f_2), hence the necessity of returning the union of the output type of the two branches, reflecting in the type the fact that at run-time either branch can be taken. Therefore, we would have $s \leq t$ but $f(s) \not\leq f(t)$, which jeopardises subject reduction. The problem is solved by choosing a decomposition that is stable with respect to the subtyping relation, that is which ensures that if $s \leq t$ then $f(s) \leq f(t)$. One such decomposition is the *maximal product decomposition*, noted $\mathcal{U}^{\times}$, which we define as follows:



Two disjoint components: $(0..3 \times 0..4)$ and $(4..7 \times 0..6)$. s overlaps both.

Two non-disjoint components: $(0..7 \times 0..4)$ and $(4..7 \times 0..6)$. s is included in $(0..7 \times 0..4)$.

Figure 4.5: Syntactic and maximal product decompositions

Definition 4.8 (Maximal product decomposition)
Let t be a type such that $t \leq (\texttt{Any} \times \texttt{Any})$. Then, there exists $n \in \mathbb{N}$ such that:

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

and that:

i. $\forall s_1, s_2, (s_1 \times s_2) \leq t \implies \exists i \in \{1,..,n\}, (s_1 \times s_2) \leq (t_1^i \times t_2^i)$

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

iii. *For all plinth* $\beth$, $t \in \beth \implies \forall i \in \{1,..,n\}, t_1^i \in \beth$ *and* $t_2^i \in \beth$

While Definition 4.8 states the property of a maximal product decomposition, it is not straightforward that such a decomposition is unique. Furthermore, it does not give a way to effectively compute it. We first prove the unicity of the decomposition:

Although the definition above is not immediate, the intuition it formalises is quite simple: the maximal decomposition of a type is the one formed *only* by (possibly overlapping) rectangles that are as large as possible. The right-hand side of Figure 4.5 shows the maximal decomposition of t, the one of s being s itself. Formally, the maximality of the components is specified by condition (*i.*): every rectangle contained in t is contained in a rectangle of its maximal decomposition (in our example, s is a subtype of the $(0..7 \times 0..4)$ component of t). Condition (*ii.*) instead ensures that *only* maximal components are used, by ruling out redundant ones (in our example $\{(0..7 \times 0..4), (4..7 \times 0..6), (5..7 \times 0..4)\}$, which satisfies (*i.*), would not be a maximal decomposition because of the extra $(5..7 \times 0..4)$). Finally, condition (*iii.*) is the same as in Definition 2.26, and is there to ensure the termination of the algorithms. The key property of our maximal decomposition is that if one product type is smaller than another, then every component of the maximal decomposition of the former is contained in at least one component of the maximal decomposition of the latter.

Lemma 4.9 (Uniqueness of maximal product decomposition) *Let t be a type such that $t \leq (\texttt{Any} \times \texttt{Any})$. The maximal product decomposition of t is unique.*

Proof We use *reductio ad absurdum* to prove this lemma. Let us suppose that there exist T and S, two maximal product decompositions of t:

- $T = \{(t_1^1 \times t_2^1), \ldots, (t_1^k \times t_2^k)\}$
- $S = \{(s_1^1 \times s_2^1), \ldots, (s_1^l \times s_2^l)\}$

We call the elements of T and S *maximal rectangles*. We have three cases:

$k =_{\mathbb{N}} l$**:** Since we suppose that T and S are differents and that they have the same number of elements, then at least one element of T is not an element of S:

$$\exists i \in 1..k \text{ such that } \forall j \in 1..l, (t_1^i \times t_2^i) \neq (s_1^j \times s_2^j)$$

Let us consider such a rectangle $(t_1^i \times t_2^i)$. We have that $(t_1^i \times t_2^i) \leq t$. But since S is a maximal product decomposition, condition $i.$ of Definition 4.8 gives:

$$\exists j \in 1..l \text{ such that } (t_1^i \times t_2^i) \lneqq (s_1^j \times s_2^j)$$

But by applying condition $i.$ of Definition 4.8 to T which is also a maximal product decomposition, we have that:

$$\exists i' \in 1..k, i' \neq i, \text{ such that } (s_1^j \times s_2^j) \leq (t_1^{i'} \times t_2^{i'})$$

Hence, by transitivity of subtyping, we have: $(t_1^i \times t_2^i) \lneq (t_1^{i'} \times t_2^{i'})$, which means that $(t_1^i \times t_2^i)$ is not a maximal rectangle which contradicts the supposition that T is a maximal product decomposition.

$k <_{\mathbb{N}} l$**:** Since there are strictly more rectangles in S than in T, we have that at least two rectangles of S are contained in the same rectangle of T:

$$\exists j, j' \in 1..l, j \neq j', \text{ such that } \exists i \in 1..k, \text{ such that: } \begin{cases} (s_1^j \times s_2^j) & \leq (t_1^i \times t_2^i) \\ (s_1^{j'} \times s_2^{j'}) & \leq (t_1^i \times t_2^i) \end{cases}$$

Consequently, we have $(s_1^j \times s_2^j) \lneq (t_1^i \times t_2^i)$ (because the elements of $(s_1^{j'} \times s_2^{j'}) \smallsetminus (s_1^j \times s_2^j)$ are in $(t_1^i \times t_2^i)$ but not in $(s_1^j \times s_2^j)$). We can now apply condition $i.$ to S:

$$\exists j'' \in 1..l, j'' \neq j \text{ such that } (t_1^i \times t_2^i) \leq (s_1^{j''} \times s_2^{j''})$$

and by transitivity of subtyping: $(s_1^j \times s_2^j) \lneq (s_1^{j''} \times s_2^{j''})$ which means that $(s_1^j \times s_2^j)$ is not a maximal rectangle, which contradicts the assumption that S is a maximal product decomposition.

$k >_{\mathbb{N}} l$**:** Symmetric argument of the previous case.

Definition 4.10 (Maximal product decomposition operator)
Given $t \leq (\texttt{Any} \times \texttt{Any})$ *and its maximal product decomposition:*

$$\{(t_1^1 \times t_2^1), \ldots, (t_1^n \times t_2^n)\}$$

where the t_j^i*s are defined according to Definition 4.8, we write:*

- $\mathcal{U}^{\circ}(t) = \{(t_1^1 \times t_2^1), \ldots, (t_1^n \times t_2^n)\}$
- $\mathcal{U}^{\circ\, i}_{j}(t) = t_j^i$
- $\texttt{rank}(t) = n.$

and call $\mathcal{U}^{\circ}$ *the maximal product decomposition operator.*

We can now prove the existence of such a decomposition by giving an algorithm which computes it. The algorithm `maxprod` which is given in Figure 4.6 uses well-defined operations on Cartesian products which where documented in [Fri04b] and implemented in [ℂDuce]. The algorithm in Figure 4.6 is quite naive. First of all, the algorithm accepts as input *any* decomposition (in the sense of Definition 2.26 of a type $t \leq (\texttt{Any} \times \texttt{Any})$. This can be for instance, the syntactic decomposition, that

Input: any product decomposition $P = \{(t_1^1 \times t_2^1), \ldots, (t_1^n \times t_2^n)\}$

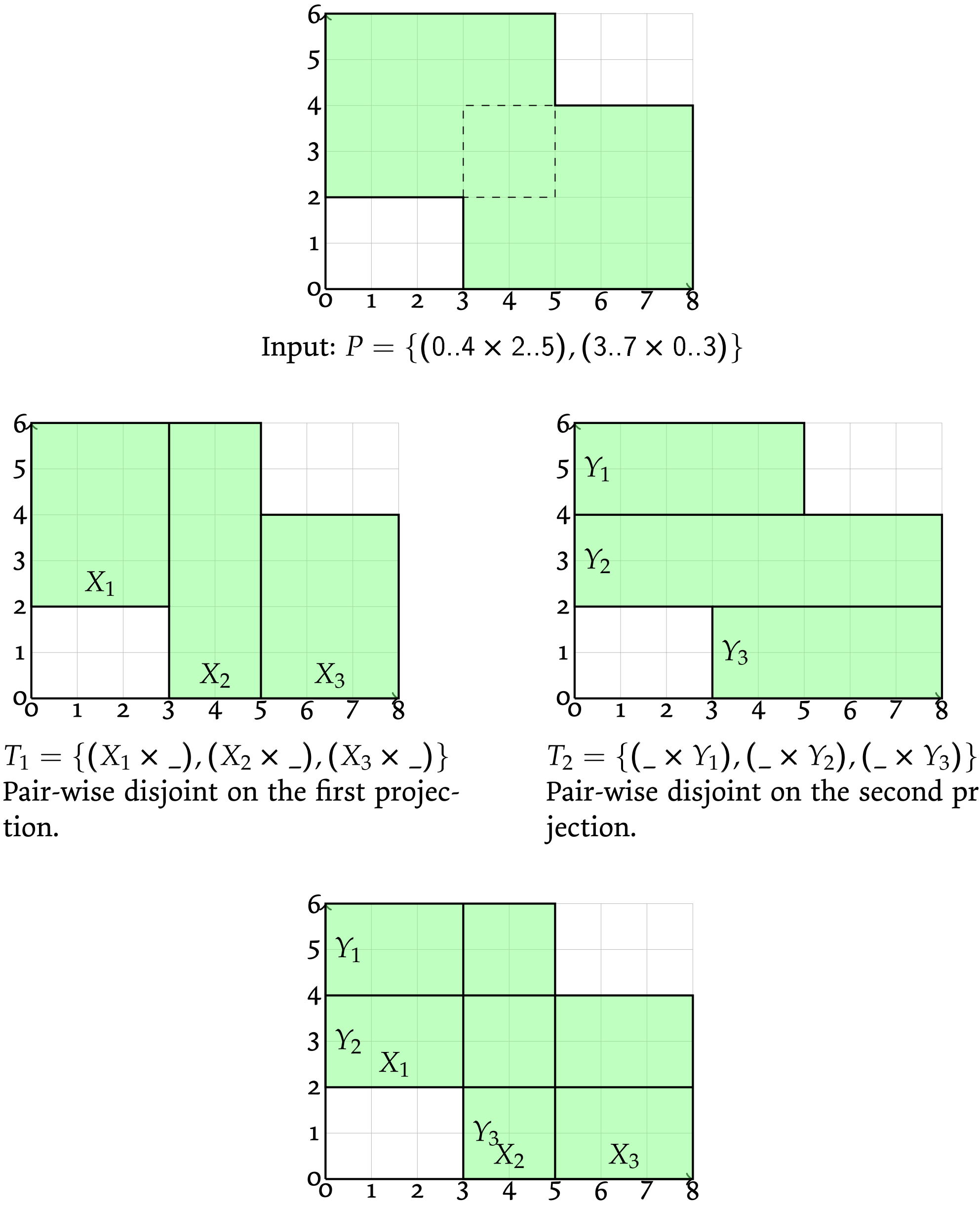
Output: maximal product decomposition $M = \{(s_1^1 \times s_2^1), \ldots, (s_1^m \times s_2^m)\}$

```
1   let $max\_prod$ $P$ =
2   let $t$ = $\bigvee_{(t_1^i \times t_2^i) \in P} (t_1^i \times t_2^i)$ in
3   let $T_1$ = $disjoint\_first$ $P$ in
4   let $T_2$ = $disjoint\_second$ $P$ in
5   let $G = \{(t_x \times t_y) \mid (t_x \times \_) \in T_1, (\_ \times t_y) \in T_2, (t_x \times t_y) \leq t\}$ in
6   do
7   $old\_G := G$;
8   foreach $t_x$ such that $\exists (t_x \times \_) \in old\_G$
9       foreach $t_y$ such that $\exists (t_x \times t_y) \in old\_G$
10          if $(t_x \times t_y) \notin G$
11          then $G := G \cup \{(t_x \times t_y)$

13  foreach $t_y$ such that $\exists (\_ \times t_y) \in old\_G$
14      foreach $t_x$ such that $\exists (t_x \times t_y) \in old\_G$
15          if $(t_x \times t_y) \notin G$
16          then $G := G \cup \{(t_x \times t_y)$

18  until $G = old\_G$
19  return $\{s \mid s \in G, \forall s' \in G, s' \neq s \Rightarrow s' \not\leq s\}$
```

Figure 4.6: Maximal product decomposition algorithm

is the type t as given by the programmer. The function *max_prod*,defined at Line 1 computes the maximal product decomposition of t. This function takes as input a set of products, P which is a decomposition of the type t (Line 2). Then, starting from P, we use the function *disjoint_first* to compute a decomposition of t where the first component of the products are pair-wise disjoint. This decomposition is documented in [Frio4b] and is used for the efficient compilation of pattern-matching. Then, it computes T_2, the symmetrical decomposition where all second projections of the products are pair-wise disjoint. Based on T_1 and T_2, the algorithm builds the set G, of pair-wise disjoint rectangles, the union of which is exactly t. Let us call each element of G a "unit" rectangle (see Figure 4.6). The following steps (Lines 6 to 18) are repeated until the set G is saturated. For each rectangle in G, merge it with another rectangle of G which has the same first projection (Lines 8 to 11) or the same second projection (Lines 13 to 16). Lastly, (Line 19) the set G is sieved to keep only products which are not included in any other of the computed products.

Finaly, we can study the monotonicity for filters, namely that if one refines the input type of a filter (with respect to a previously used input type), then the output



Input: $P = \{(0..4 \times 2..5), (3..7 \times 0..3)\}$

$T_1 = \{(X_1 \times \_), (X_2 \times \_), (X_3 \times \_)\}$
Pair-wise disjoint on the first projection.

$T_2 = \{(\_ \times Y_1), (\_ \times Y_2), (\_ \times Y_3)\}$
Pair-wise disjoint on the second projection.

$G = \{(X_1 \times Y_1), (X_1 \times Y_2), (X_2 \times Y_1), (X_2 \times Y_2), (X_2 \times Y_3), (X_3 \times Y_2), (X_3 \times Y_3)\}$
Partition of t in rectangles.

Figure 4.7: Initialisation of the maximum product decomposition algorithm

type will be more precise:

Lemma 4.11 (Stability of filtering) *For every filter f, types s and t, and type environment Γ, if $s \leq t$ and $\Gamma \vdash f(t) = t'$, then $\Gamma \vdash f(s) = s'$and $s' \leq t'$.*

The proof uses the following observation, which is a direct consequence of the set-theoretic interpretation of types:

Fact 4.12 *Let s, t and u be types such that: $s \leq t$. We have:*

- $s \wedge u \leq t \wedge u$
- $s \vee u \leq t \vee u$

Proof (Stability of filtering) Let $s \leq t$, we want to show that $s' \leq t'$, or, equivalently, that $\forall \gamma : \Gamma$ and $\forall v : s,\ \gamma \vdash_e f(v) \rightsquigarrow v'$, we have $v' \in [\![f(t)]\!]$. With this presentation, we exhibit the value v. We can now reuse the tuple $(f, v, c(f))$ equiped with the well founded order $(\sqsubset, \sqsubset, <_{\mathbb{N}})_{lex}$, and prove the lemma by induction:

- e: we know that `type`$(\Gamma, e) = t'$ and also that `type`$(\Gamma, e) = s'$ so, here, $s' = t'$ hence $s' \leq t'$.
- (f_1, f_2): Let us remark that we have $s \leq t$, and

  $$t \simeq \bigvee_{(t_1^i, t_2^i) \in \mathcal{U}^{\circ}(t)} (t_1^i, t_2^i)$$

  So:

  $$s \simeq \bigvee_{(s_1^j, s_2^j) \in \mathcal{U}^{\circ}(s)} (s_1^j, s_2^j)$$

  By definition of the maximal product decomposition, we know that for all j, there is i such that: $(s_1^j, s_2^j) \leq (t_1^i, t_2^i)$, with $s_1^j \leq t_1^i$ and $s_2^j \leq t_2^i$. We can apply the induction hypothesis on f_1, s_1^j and t_1^i (resp. on f_2, s_2^j and t_2^i) and we get: $f_1(s_1^j) \leq f_1(t_1^i)$ (resp. $f_2(s_2^j) \leq f_2(t_2^i)$), and therefore $f(s) \leq f(t)$
- $f_1 | f_2$: using Fact 4.12, and the typing rule **(t-union)**:

  $$\begin{array}{rcl} s \wedge \wr f_1 \wr & \leq & t \wedge \wr f_1 \wr \\ s \wedge \neg \wr f_1 \wr & \leq & t \wedge \neg \wr f_1 \wr \end{array}$$

  So, by induction hypothesis:

  $$\begin{array}{rcl} f_1(s \wedge \wr f_1 \wr) & \leq & f_1(t \wedge \wr f_1 \wr) \\ f_2(s \wedge \neg \wr f_1 \wr) & \leq & f_2(t \wedge \neg \wr f_1 \wr) \end{array}$$

  hence:

  $$f_1(s \wedge \wr f_1 \wr) \vee f_2(s \wedge \neg \wr f_1 \wr) \leq f_1(t \wedge \wr f_1 \wr) \vee f_2(t \wedge \neg \wr f_1 \wr)$$

- $p \to f'$: direct application of the induction hypothesis: $f'(s) \leq f'(t)$, and so $f(s) \leq f(t)$.
- $f_1; f_2$: Let $f_1(s) = s_1$ and $f_1(t) = t_1$. By induction, $s_1 \leq t_1$. Therefore , $f_2(s_1) \leq f_2(t_1)$, and so $f(s) \leq f(t)$.

Chapter 5

Type inference

> In this chapter, we present an algorithm for the previously defined type inference system. We extend the notion of filters to annotated filters. Based on this formalism, we define a type inference *algorithm*. We show the termination of the algorithm, its soundness with respect to the non-algorithmic system devised in Chapter 4. We also define and prove a property of completeness *up to* annotations, which gives some insight on the use of this algorithm in practice and illustrate the interactions between the type-system and the user-defined annotations.

Contents

5.1 Presentation

In the previous chapter we presented a type-system for the filter algebra which enjoys the desired properties of type-safety and precision. However, in its present state, the system does not translate directly into a typing algorithm. There are two reasons for that:

i. to one filter and input type t there might correspond an infinite number of valid regular derivations. There is hence a need to pick one.

ii. we made some basic assumptions on the *host language*, such as the existence of a type algebra and the associated operators for subtyping or type decomposition (e.g. π to decompose product types).

We have already lengthily discussed point *(i.)*, but let us emphasize that there are not only an infinite number of possible output types, but also that there is not a most precise – *i.e. principal*– one. In the absence of a principal type, even if the algorithm was to choose a particular approximation, there would always be contexts for which the chosen type would not be precise enough. For such cases, the programmer needs a mean to guide the algorithm (which of course still performs a type-checking operation, thus validating or refuting the annotation).

Point *(ii.)* reflects the fact that we devised filters so that their algebra and type-system is *parametrized* by the host language. While it is true that we made rather strong assumptions on the host language, such as the "shape" of its type algebra and the related high-level operations (such as subtyping, type-inference for expressions, and so on), we believe that these assumptions are very reasonable. Indeed, not only ℂDuce, our host language of choice for the implementation, satisfies these requirements, but also XDuce or even various dialects of ML would qualify as relevant host languages for filters. However the extent of the operations (particularly on types) that we can expect from the type algebra is limited. As we will see, typing a filter without annotations, even for some cases where there exists a regular output type may lead to applying a type operation (say subtyping) on ill-formed types, during the algorithmic typing derivations. As the semantics of these operations is obviously not defined for ill-formed types, we need to ensure that "foreign" operations (*i.e.* those provided by the host language) are always applied on well-formed types and will also use annotations in those cases. Before detailing this point some more, let us present the algorithm.

5.2 Type-inference Algorithm

Since the algorithm needs to work on finite representations of (possibly infinite) regular types, we use the "μ" notation introduced in Definition 2.12 to represent types. We use Greek letters τ, σ to range over μ-types and to distinguish them from regular tree types (which are ranged over by t, s,...); recursion variables are ranged over by α, β,....

We extend the definition of filters with annotated filters:

Definition 5.1 (Annotated filters)

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

An annotation is a set of (μ-)types $\mathbb{E}$ among which the algorithm will pick an output type for the annotated filter. The algorithm is described in Figure 5.1 as a set of deduction rules for the judgement $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$, where Γ denotes a type environment for pattern variables and Δ is a memoization environment (which ensures the termination of the algorithm), that is, a set of triples (f, τ, σ) where f is a filter and τ and σ types (intuitively, they respectively are an input and an output type).

There are two sets of rules. The structural rules are adapted from the system by adding an extra environment Δ which they do not use. Memoization rules, on the other hand reflect the co-inductive nature of the type inference system, and handle regularity with the memoization environment Δ in a classical way. The basic case is the **(a-base-rec)** rule, which returns the output type σ for the corresponding pair f, τ if it is already in Δ. As for putting a type in Δ there are three cases. If the filter is not annotated, and a suitable output type does not exist in Δ yet, then we introduce a *fresh* type variable to represent this output type (rules **(a-unfold-rec)** and **(a-unfold-non-rec)**); we then type f against the input type, possibly after unfolding its definition if this type is recursive. The returned output type σ is then closed by the binder $\mu\alpha$ where α is the newly introduced variable. The third case is the one of an annotated filter (rule **(a-annot)**). In that case, we non-deterministically pick an annotation in the set $\mathbb{E}$, and then type the filter f under the assumption that its output type, for the input type τ is the annotation σ that was chosen. In this presentation, we assume that the *choose()* function in rule **(a-annot)** always chooses the right type in the annotation set if it exists. In practice, this is implemented by backtracking, the algorithm trying all the annotations one after the other until a valid one is found (or a type error is raised). We chose to hide this aspect of the algorithm in order not to clutter it with tedious backtracking rules and environments.

Given as it is, the algorithm would not work properly. Indeed, while the structural rules are syntax directed, the memoization ones are not. If we are not careful, the algorithm could then, for any input type and any filter perform the ill-founded derivation: $\dfrac{\dfrac{(f,\tau,\alpha) \in \Delta}{\varnothing, \{f,\tau,\alpha\} \vdash_{\mathscr{A}} f(\tau) = \alpha}}{\varnothing, \varnothing \vdash_{\mathscr{A}} f(\tau) = \mu\alpha.\alpha}$, giving the ill-founded type definition $\mu\alpha.\alpha$ as a result. To avoid such a behaviour, we must enforce a contractivity property on the system, by forcing an alternation between the two kind of rules: a memoization rule must be followed by a structural rule and a structural rule must be followed by a memoization rule. To further formalize the order of application of the rules, we give the algorithm in pseudo code in Figure 5.2.

This specification leads us closer to a practical algorithm but a technical issue of importance has yet to be addressed. Since we use the standard notations $\leq$, $\boldsymbol{\pi}$, τ/p,... that we introduced in Chapter 2 — hence the function given par the host language — we can only assume that these functions are defined on well-formed types, that, is closed "μ"-expressions. Unfortunately, due to the composition rule **(a-comp)**, such an operation can be performed on *open* types (type expressions with free variables). There is then the need to extend these operations on open types. A simple extension, which we will use for now is the following:

Definition 5.2 (Operation on open types)
The operations $\leq$, $\boldsymbol{\pi}$ and τ/p fail *when applied to an open type.*

$\boxed{\text{Structural rules}}$

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

(a-prod)
$$\frac{\begin{array}{l}\boldsymbol{\pi}(\tau) \equiv \{(\tau_1^1 \times \tau_2^1), \ldots, (\tau_1^n \times \tau_2^n)\} \\ j \in 1..2\end{array} \quad \Gamma, \Delta \vdash_{\mathscr{A}} f_j(\tau_j^i) = \sigma_j^i}{\Gamma, \Delta \vdash_{\mathscr{A}} (f_1, f_2)(\tau) = \bigvee_i (\sigma_1^i \times \sigma_2^i)}$$

(a-patt)
$$\frac{\tau / p \uplus \Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma \quad \tau \leq \llparenthesis p \rrparenthesis \wedge \llparenthesis f \rrparenthesis}{\Gamma, \Delta \vdash_{\mathscr{A}} (p \rightarrow f)(\tau) = \sigma}$$

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

(a-comp)
$$\frac{\begin{array}{c}\tau \leq \llparenthesis f_1 \rrparenthesis \\ \sigma_1 \leq \llparenthesis f_2 \rrparenthesis\end{array} \quad \Gamma, \Delta \vdash_{\mathscr{A}} f_1(\tau) = \sigma_1 \quad \Gamma, \Delta \vdash_{\mathscr{A}} f_2(\sigma_1) = \sigma_2}{\Gamma, \Delta \vdash_{\mathscr{A}} (f_1 ; f_2)(\tau) = \sigma_2}$$

$\boxed{\text{Memoization rules}}$

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

(a-unfold-rec)
$$\frac{\begin{array}{c}\nexists \sigma \text{ s.t. } (f, \mu\alpha.\tau, \sigma) \in \Delta \text{ and } \beta \text{ fresh.} \\ \Gamma, \{(f, \mu\alpha.\tau, \beta)\} \uplus \Delta \vdash_{\mathscr{A}} f(\tau[\alpha \leftarrow \mu\alpha.\tau]) = \sigma\end{array}}{\Gamma, \Delta \vdash_{\mathscr{A}} f(\mu\alpha.\tau) = \mu\beta.\sigma}$$

(a-unfold-non-rec)
$$\frac{\begin{array}{c}\nexists \sigma \text{ s.t. } (f, \tau, \sigma) \in \Delta \text{ and } \alpha \text{ fresh.} \\ \Gamma, \{(f, \tau, \alpha)\} \uplus \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma\end{array}}{\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \mu\alpha.\sigma}$$

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

Figure 5.1: Deduction system associated with $\mathscr{F}_{\mathscr{A}}$

What we mean with this informal definition is that the algorithm will fail and consider the term as ill-typed, in the same way as it fails when one of the conditions

```
let rec typeinf $f\ \tau\ \Gamma\ \Delta$ =
  if $\tau \not\leq \lbag f \rbag$
  then failure
  else
  (* we take care of memoization *)

  if $\exists\sigma$ s.t. $(f,\tau,\sigma) \in \Delta$
  then $\sigma$
  else
    match $f$ with
    | $f'_{\mathbb{E}}$ ->
        let $\sigma$ = choose ($\mathbb{E}$)
        (* non deterministic choice *)
        in
        let $\sigma'$ = typeaux $f'\ \tau\ \Gamma$
        $(\{(f',\tau,\sigma)\} \uplus \Delta)$
        in if $\sigma' \leq \sigma$ then $\sigma$ else failure

    | _ ->
        let $\alpha$ = fresh_variable ()
        in
        let $\sigma$ =
typeaux $f\ \tau\ \Gamma\ (\{(f',\tau,\alpha)\} \uplus \Delta)$
        in $\mu\alpha.\sigma$

 (* And now by case on the
 structure of the filters *)

and typeaux $f\ \tau\ \Gamma\ \Delta$ =

  match $f$ with
  | $e$ -> typecheck ($\Gamma$,e)

| $(f_1,f_2)$ ->$\forall(\tau_1,\tau_2) \in \boldsymbol{\pi}(\tau)$
              let $\sigma_1^i$ = typeinf $f_i\ (\tau_1)\ \Gamma\ \Delta$
              and $\sigma_2^i$ = typeinf $f_i\ (\tau_2)\ \Gamma\ \Delta$
              in $\bigvee_i (\sigma_1^i \times \sigma_2^i)$
| $p \to f'$ ->if $\tau \leq \lbag p \rbag \wedge \lbag f \rbag$
             then
             let $\sigma$ = typeinf $f'\ \tau\ (\tau/p \uplus \Gamma)\ \Delta$
             in $\sigma$
             else failure
| $(f_1|f_2)$ ->if $\tau \leq \lbag f_1 \rbag \vee \lbag f_2 \rbag$
               then let $\tau_1 = \tau \wedge \lbag f_1 \rbag$
               and $\tau_2 = \tau \wedge \neg \lbag f_1 \rbag$
               in
               let $\sigma_1$ = typeinf $f_1\ \tau_1\ \Gamma\ \Delta$
               and $\sigma_2$ = typeinf $f_2\ \tau_2\ \Gamma\ \Delta$
               in $\bigvee_{i|\tau_i \neq \texttt{Empty}} \sigma_i$
               else failure
| $(f_1;f_2)$ ->let $\sigma_1$ = typeinf $f_1\ \tau\ \Gamma\ \Delta$
              in
              if $\sigma_1 \leq \lbag f_2 \rbag$
              then
              let $\sigma_2$ = typeinf $f_2\ \sigma_1\ \Gamma\ \Delta$
              in $\sigma_2$
              else failure
```

Figure 5.2: Type inference algorithm given in pseudo-ML

in the premises of a rule does not hold. This allows us not to make any assumption on the internal representations of types for the host language and to provide a truly parametric filter language.

Before discussing the theoretical properties, let us have a look at the behaviour of the algorithm, first on an easy case, where annotations are not required. Let us consider again the filter *succList* (that we introduced in Section 3.4.3) and the input type $\tau = \mu\alpha.(\texttt{`nil} \vee (\texttt{Int} \times \alpha))$ (i.e., lists of integers), and try to find a type σ such that $\varnothing, \varnothing \vdash_{\mathscr{A}} succList(\tau) = \sigma$. We can see the complete typing derivation in Figure 5.3. At the first step, rule **(a-unfold-rec)** is applied. It creates a fresh type variable α_0 which it associates to *succList* applied to the type τ. It then performs a structural rule (Step 2) **(a-union)**. This will type both branches of the union filter, (Steps 3-6 and 7-12) and return the union of the result. Step 3-6 mimic the rules $\mathscr{F}$ interleaving them with memoization rules. It should be noted that in this derivation the exact result should be $\mu\alpha_1.\texttt{`nil}$, $\mu\alpha_1.\mu\alpha_2.\texttt{`nil}$,... but we directly simplified the results. For

$$\textcircled{12}\ \textbf{(a-base-rec)}\ \dfrac{(succList, \mu\alpha.(\texttt{`nil} \vee (\texttt{Int} \times \alpha)), \alpha_0) \in \Delta_1}{\varnothing, \Delta_1 \vdash_{\mathscr{A}} succList(\mu\alpha.(\texttt{`nil} \vee (\texttt{Int} \times \alpha))) = \alpha_0}$$

$$\textcircled{7}\ \textbf{(a-prod)}\ \dfrac{\textcircled{8}\ \textbf{(a-unfold-non-rec)}\ \dfrac{\textcircled{9}\ \textbf{(a-patt)}\ \dfrac{\textcircled{10}\ \textbf{(a-unfold-non-rec)}\ \dfrac{\textcircled{11}\ \textbf{(a-expr)}\ \dfrac{\texttt{type}(\{x \mapsto \texttt{Int}\}, x+1)}{\{x \mapsto \texttt{Int}\}, \Delta_3 \vdash_{\mathscr{A}} (x+1)(\texttt{Int}) = \texttt{Int}}}{\{x \mapsto \texttt{Int}\}, \{(x+1, \texttt{Int}, \alpha_2)\} \cup \Delta_2 \vdash_{\mathscr{A}} (x+1)(\texttt{Int}) = \texttt{Int}}}{\{x \mapsto \texttt{Int}\}, \Delta_2 \vdash_{\mathscr{A}} (x \to x+1)(\texttt{Int}) = \texttt{Int}}}{\varnothing, \{(x \to x+1, \texttt{Int}, \alpha_2)\} \cup \Delta_1 \vdash_{\mathscr{A}} (x \to x+1)(\texttt{Int}) = \texttt{Int}} \qquad \textcircled{12}}{\varnothing, \{((x \to x+1, succList), (\texttt{Int} \times \mu\alpha \ldots), \alpha_1)\} \cup \Delta_0 \vdash_{\mathscr{A}} (x \to x+1, succList)((\texttt{Int} \times \mu\alpha \ldots)) = (\texttt{Int} \times \alpha_0)}$$

$$\textcircled{1}\ \textbf{(a-unfold-rec)}\ \dfrac{\textcircled{2}\ \textbf{(a-union)}\ \dfrac{\textcircled{3}\ \textbf{(a-unfold-non-rec)}\ \dfrac{\textcircled{4}\ \textbf{(a-patt)}\ \dfrac{\textcircled{5}\ \textbf{(a-unfold-non-rec)}\ \dfrac{\textcircled{6}\ \textbf{(a-expr)}\ \dfrac{\texttt{type}(\texttt{`nil},\varnothing)}{\varnothing, \{(\texttt{`nil}, \texttt{`nil}, \alpha_3)\} \cup \Delta_2 \vdash_{\mathscr{A}} \texttt{`nil}(\texttt{`nil}) = \texttt{`nil}}}{\varnothing, \{(\texttt{`nil}, \texttt{`nil}, \alpha_2)\} \cup \Delta_1 \vdash_{\mathscr{A}} \texttt{`nil}(\texttt{`nil}) = \texttt{`nil}}}{\varnothing, \Delta_1 \vdash_{\mathscr{A}} \texttt{`nil} \to \texttt{`nil}(\texttt{`nil}) = \texttt{`nil}}}{\varnothing, \{(\texttt{`nil} \to \texttt{`nil}, \texttt{`nil}, \alpha_1)\} \cup \Delta_0 \vdash_{\mathscr{A}} \texttt{`nil} \to \texttt{`nil}(\texttt{`nil}) = \texttt{`nil}} \qquad \textcircled{7}}{\varnothing, \{(succList, \mu\alpha.(\texttt{`nil} \vee (\texttt{Int} \times \alpha)), \alpha_0)\} \vdash_{\mathscr{A}} succList(\texttt{`nil} \vee (\texttt{Int} \times \ldots)) = \texttt{`nil} \vee (\texttt{Int} \times \alpha_0)}}{\varnothing, \varnothing \vdash_{\mathscr{A}} succList(\mu\alpha.\texttt{`nil} \vee (\texttt{Int} \times \alpha)) = \mu\alpha_0.\texttt{`nil} \vee (\texttt{Int} \times \alpha_0)}$$

Figure 5.3: Derivation of the algorithm on the filter *succList*

space reasons we also did not write extensively the Δ sets but rather aliased them with fresh variables Δ_i and only wrote the new addition to these sets. Step 7 types the product part of the filter against the product part of the type. The type having only one product component, the decomposition π returns the product itself. The two sub-filters are applied component-wise (Steps 8, 11 and 12). The left filter, $x \to x+1$ follows the same pace as Steps 3-6. The important step is Step 12 where a *recursive call* is made, applying *succList* again on τ. The result α_0 which was memoized at Step 1 is returned. The output type is then reconstructed to finally obtain: $\sigma = \mu\alpha_0.(\texttt{`nil} \vee (\texttt{Int} \times \alpha_0))$, which is the expected type.

The next example illustrates what can go wrong in the presence of a composition filter. Imagine the following filter:

Example 5.3

$$\begin{array}{rcl} g & = & \texttt{`nil} \to \texttt{`nil} | (x \to x{+}1, g); h \\ h & = & \texttt{`nil} \to \texttt{`nil} | (x \to x{+}1, h) \end{array}$$

Like $succList$, this filter increments each element of a list of integers but, here, h

is also called on every trailing list, that is:

$$\begin{array}{rcl} g(\texttt{[0 0 0]}) & \rightsquigarrow & h(\texttt{0}+\texttt{1}, h(\texttt{0}+\texttt{1}, h(\texttt{0}+\texttt{1}, \texttt{`nil}))) \\ & \rightsquigarrow & \texttt{[2 3 4]} \end{array}$$

If we try to type this filter, we get at some point of the derivation:

$$\textbf{(a-comp)}\ \frac{\dfrac{\vdots}{\varnothing,\Delta \vdash_{\mathscr{A}} (\mathbf{(}x \to x{+}1\mathbf{,}g\mathbf{)})((\texttt{Int} \times \mu\alpha.(\ldots))) = (\texttt{Int} \times \beta)} \qquad \dfrac{}{\varnothing,\Delta \vdash_{\mathscr{A}} h((\texttt{Int} \times \beta)) = \ldots}}{\varnothing,\Delta \vdash_{\mathscr{A}} (\mathbf{(}x \to x{+}1\mathbf{,}g\mathbf{)};h)((\texttt{Int} \times \mu\alpha.(\ldots))) = \ldots}$$

Here, we must test that $(\texttt{Int} \times \beta) \leq \wr h \int$, where not only β is not bound, but its definition is at this moment incomplete since it is the type we are computing. The test fails and so does the algorithm. However we want to type such filters, that is why we impose that the filter $(x \to x{+}1,g)$, preceding the composition is annotated. This example gives the intuition on why composition filters must be annotated. We will formalize this intuition now, before discussing how this restriction has an impact on the usage of filters.

5.3 Properties

We prove in this section three properties. *(1.)* Termination of the algorithm, which is quite straightforward to prove, thanks to the careful specification of memoization rules and restrictions on filter composition. *(2.)* Soundness of the algorithm, that is correctness with respect to the type-system defined in Chapter 4 which states that if the algorithm infers an output type for a given filter and input type, then there exists a derivation in the system for this filter, input, and output type. This is what makes the algorithm inherits the property of type safety proven for the system. *(3.)* Completeness — which roughly states that if a filter can be typed (in the system), then the algorithm finds the expected type —, requires more work to be precisely defined. Indeed, since the algorithm works on *annotated* filters, the completeness theorem need to take these annotations into account, and more specifically, granted that a typing derivation exists in the system, the completeness theorem must give a way to annotate the filter so that the algorithm succeed.

5.3.1 Termination

Proving the termination of the algorithm is quite straightforward:

Theorem 5.4 (Termination of the typing algorithm) *For all filters f and types τ, the typing algorithm for $f(\tau)$ terminates.*

Proof (Termination of the typing algorithm) Given the type τ, by Theorem 2.27, there exists a plinth $\beth$ such that $\tau \in \beth$. Let us fix the set: $\mathcal{D}(\tau) = \{(f,\tau')|f \in \mathcal{S}\mathit{ubtree}(f)$ and $\tau' \in \beth\}$. $\mathcal{D}(\tau)$ is finite since it is the Cartesian product of two finite sets. We define the function $\Psi(\Delta,\tau) = \{(f,\tau')|(f,\tau') \in \mathcal{D}(\tau)$ and $\nexists \sigma'$ s.t. $(f,\tau',\sigma') \in \Delta\}$. Here, Δ is the set as constructed by the algorithm during a typing derivation. Informally, $\mathcal{D}(\tau)$ describes all the possible applications of a sub-filter of f to a sub-tree of the input type τ.[1] The set $\Psi(\Delta,\tau)$ describes then all the such combinations (f',τ') that have not been encountered at a given step of the derivation. Finally, given a derivation D of the judgment $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$, we associate a measure $m(D)$ such that:

$$\begin{array}{rcl} m(D) & = & 0 \ \ \text{if the last rule is a memoization rule} \\ m(D) & = & 1 \qquad \text{if the last rule is a structural rule} \end{array}$$

We can now prove the theorem. Given a filter f and a type τ, we show that either the algorithm fails, or the derivation D for the judgment $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$ is finite. We prove this by induction on $(f, \Psi(\Delta,\tau), m(D))$, equiped with the order $(\sqsubseteq, \subseteq, \leq_{\mathbb{N}})_{lex}$. If we consider a strict sub-tree of the filter, then the first component decreases strictly. If a filter is applied to a new type during the derivation (that is a type to which it has never been applied earlier in the derivation), then $\Psi(\Delta,\tau)$ decreases strictly. The third measure, $m(D)$ resembles the one introduced to prove the termination of the *evaluation* of filters. Indeed, when going from a memoization rule to a structural rule, either we visit a new input type, and put it in Δ, or we visit an already memoized type and can stop the derivation. In both cases, the second component of our induction order decreases. However, when going from a structural rule to a memoization rule, neither the first component decreases (since the sub-filter might not be strict sub-tree of the input, as in the rule **(a-prod)**), nor $\Psi(\Delta,\tau)$ decreases, since Δ is left untouched by these structural rules. However, we know by definition of the algorithm that at the next step, a memoization rule be applied, thus enforcing a strict decrease of the second component of the order. The measure $m(D)$ merely reflects this intuition. Finally, the first component of the order, $\sqsubseteq$ handles the second premise of the rule **(a-comp)** where the input type may increase (and be completely new) but the filter decreases strictly (identical to the proof of Theorem 3.15).

Basic case :

Failure : One of the pre-condition of a rule does not hold (e.g. $\tau \not\leq (\texttt{Any} \times \texttt{Any})$ for a product filter or the operation τ/p is performed on a type τ with free variables). The algorithm terminates with a failure.

[1] This is an abuse of terminology. In fact it represents all the application of a sub-filter of f to a type τ' that can be generated from τ by Boolean connective and product decomposition, thus a super-set of all the input types which occur in the derivation of $f(\tau)$.

$f \equiv e$: Basic case of the first component. We rely on the type inference of the host language and suppose it terminates.

(a-base-rec) : If the last application is **(a-base-rec)** then the algorithm trivialy terminates (the rule has no premise).

Inductive case :

Structural rules : Here the last applied rule of the derivation D cannot be a **(a-expr)** (as it was dealt with in the base case). If it is either **(a-prod)**, **(a-pat)** or **(a-union)**, then the first component does not decrease, nor does the second (as none of these rules modify Δ, $\Psi(\Delta, \tau)$ is unchanged). For any of these rules, the premises are the last step of a derivation D' which ends by the application of a memoization rule (due to the alternation memoization/structural rule). Hence, $m(D') <_{\mathbb{N}} m(D)$, and we can apply the induction hypothesis. If the last applied rule is **(a-comp)**, the first premise is treated as the other structural rules. The second premise is of the form: $\Gamma, \Delta \vdash_{\mathscr{A}} f_2(\sigma_1) = \sigma_2$ where $f \equiv f_1;f_2$. Here, the second component might increase, has $\Psi(\Delta, \sigma_1)$ might not be related to $\Psi(\Delta, \tau)$. However, the first component decreases strictly, as, by definition of the filters, $f_2 \sqsubset f$. We can apply the induction hypothesis on this premise.

Memoization rules : The rule **(a-base-rec)** does not apply here. It is then one of the other three rules. For any of these rules, the filter in the premises is the same filter f as in the goal (hence, the first component does not change). Let us call Δ' the memoization environment of the premise. We have $\Delta \subset \Delta'$ as these rules add a triplet which is not in Δ. Hence, $\Psi(\Delta', \tau) \subset \Psi(\Delta, \tau)$ (informally, if we add something in Δ it means that we have less types to visit). And so the second component decreases strictly. We can apply the induction hypothesis on the premise.

5.3.2 Soundness

Soundness (and completeness) relates the typing algorithm and the non-algorithmic type-system presented in Chapter 4. However the algorithm is defined for possibly annotated filters. We need to define an operation which removes the annotations of a filter:

Definition 5.5 (Stripping)
Let f be an annotated filter (seen as a regular tree). We define the stripping of f and we note

$\lceil f \rceil$ *the function defined as:*

$$\begin{array}{rclcrcl} \lceil f_{\mathbb{E}} \rceil & = & \lceil f \rceil & \qquad & \lceil f_1;f_2 \rceil & = & \lceil f_1 \rceil;\lceil f_2 \rceil \\ \lceil e \rceil & = & e & & \lceil f_1|f_2 \rceil & = & \lceil f_1 \rceil|\lceil f_2 \rceil \\ \lceil p \to f \rceil & = & p \to \lceil f \rceil & & \lceil (f_1,f_2) \rceil & = & (\lceil f_1 \rceil,\lceil f_2 \rceil) \end{array}$$

Now we need to formalize the fact that "the algorithm behaves like the system" (crude words to express soundness). The idea is pretty straightforward. Given a derivation for a (possibly annotated) filter f, we would like to show that there is a corresponding derivation for $\lceil f \rceil$ in the system. Despite that the algorithmic derivation is finite, it is however not possible to prove the theorem by induction. Indeed, the rule **(a-base-rec)** is not well-founded. While this rule has no premise, it corresponds to an *infinite* derivation in the type-system (this rule was purposely introduced to deal with regular derivations). We see that while the algorithmic derivations are finite, we need a *co-inductive* technique to build the corresponding regular typing derivation. To that end, we reuse the proof technique seen in Chapter 4, (used for Lemma 4.6). We use the finite derivation to build a *finite system of guarded equations* between typing derivations. It follows that the unique solution of this system is exactly the corresponding typing derivation.

Theorem 5.6 (Soundness of the typing algorithm) *For all* $\Gamma, \Delta, f, \tau,$ *and* σ, *if* $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$, *then* $[\Gamma]_\infty \vdash \lceil f \rceil([\tau]_\infty) = [\sigma]_\infty$.

Proof We recall that the $[\,]_\infty$ notation denotes the infinite expansion of a well-formed μ-term (see Chapter 2, Definition 2.14).
First, let us remark that, since $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$ is a successful algorithmic derivation, then to every type variable occuring in this derivation, we can associate a *closed* μ-type. Indeed, since the derivation is successful, every fresh type variable introduced by an application of the rule **(a-unfold-rec)** or **(a-unfold-non-rec)** is associated to a corresponding μ definition, in the goal of the rule. We can use this fact to extend the definition of $[\,]_\infty$ to an *open type* (a type with free recursion variables). Let τ' be an open type occuring in the algorithmic derivation. We define $[\tau']_\infty$ as $[\tau'']_\infty$, where $\tau'' = \tau\{\alpha \leftarrow \mu\alpha.\sigma'\}$, for all free variables $\alpha \in \tau'$ and such that $\mu\alpha.\sigma'$ is the definition corresponding to the introduction of the variable α in the derivation.
We can now inductively build our system of equations (the solution of which is a *co-inductive* derivation). As a commodity, we label every step of the algorithmic derivation by a unique integer. We inductively define the function E from algorithmic derivations to system of equations. The two important points are:

- The rule **(a-base-rec)** which introduces a circular definition, and thus generate a regular solution to the system of equations.

- The rule **(a-annot)** which corresponds to a subsumption rule.

As for the structural rules, they are just "translated" while the **(a-unfold-*)** rules are simply ignored.

(a-base-rec) :$E\left(i\,\dfrac{(f,\tau,\sigma) \in \Delta}{\Gamma,\Delta \vdash_{\mathscr{A}} f(\tau) = \sigma} \right) = \left\{ x_i = \dfrac{x_j}{\Gamma \vdash \lceil f \rceil([\tau]_\infty) = [\sigma]_\infty} \right\}$

where j is the label of the rule adding (f,τ,σ) to Δ. Indeed, the application of the rule **(a-base-rec)** is always preceded by an introduction rule, **(a-unfold-rec)**, **(a-unfold-non-rec)** or **(a-annot)** earlier in the derivation.

(a-annot) : $E\Bigg(\begin{array}{l} j \\ i \end{array} \dfrac{\dfrac{k\,\dfrac{D'}{d}}{\Gamma,\Delta \cup \{(f_{\mathbb{E}},\tau,\sigma)\} \vdash_{\mathscr{A}} f_{\mathbb{E}}(\tau) = \sigma' \quad \sigma' \leq \sigma}}{\Gamma,\Delta \vdash_{\mathscr{A}} f_{\mathbb{E}}(\tau) = \sigma} \Bigg) =$

$\left\{ x_i = \dfrac{x_j}{\Gamma \vdash \lceil f_{\mathbb{E}} \rceil([\tau]_\infty) = [\sigma]_\infty} \right\} \cup$

$\left\{ x_j = \dfrac{\dfrac{x_k}{\Gamma \vdash \lceil f_{\mathbb{E}} \rceil([\tau]_\infty) = [\sigma']_\infty \quad [\sigma']_\infty \leq [\sigma]_\infty}}{\Gamma \vdash \lceil f_{\mathbb{E}} \rceil([\tau]_\infty) = [\sigma]_\infty} \right\} \cup E\left(k\,\dfrac{D'}{d} \right)$

This rule is split into two equations, which form a corresponding subsumption rule. Indeed, when the programmer annotates a filter with a valid type (*i.e.* an annotation which makes the algorithm succeed), the corresponding behaviour in the type-system is to let the system "guess" the annotation. Note that at this step, the annotations are removed from the current filter.

Other rules : Structural rules must be handled carefully. The basic idea is to rewrite a structural rule into its corresponding rule in the type-system, and call E on every sub-derivation of its premises. However we want to erase memoization rules (**(a-annot-rec)** and **(a-annot-non-rec)**) from the derivation as they are not needed in the system. If we perform the operation naively in two steps, we can generate an ill-founded system of equations. Indeed, consider the case of a structural rule with label i:

$$i\,\dfrac{j_1\,\dfrac{k_1\,\dfrac{D_1}{d'_1}}{d_1} \quad \cdots \quad j_n\,\dfrac{k_n\,\dfrac{D_n}{d'_n}}{d_n}}{\Gamma,\Delta \vdash_{\mathscr{A}} f(\tau) = \sigma}$$

Here, since i is a structural rule, then all the rules labeled by js are memoization rules, and conversely all the rules labeled by ks are structural rules. If we were to proceed in two steps, then we would first introduce an equation for x_i depending on x_{j_1},...,x_{j_n} then a call to E which would introduce equations for x_js depending on the x_ks.

This is e.g. what happens if the j rule is an annotation rule (remark in the previous **(a-annot)** case the definitions of x_i and x_j). In the case where the j rule is an **(a-unfold-*)** rule, and since we want to "erase" such rules, we need to take some care so as to ensure the well-formedness of our system of equations. Indeed, naively skipping such rules (by calling E on their premises), can be done in two ways, either by not introducing the x_j variable or by introducing a trivial equality $x_j = x_k$. Both solutions fail: the first one leaves "dangling" variables x_j in the definition of x_i and the second one result in an ill-founded system (we recall that to apply the principle of co-induction related to Definition 2.19, there must be no such trivial equation in the system). We could, in a first pass generate such an ill-founded system and in a second step "clean" the system, by removing any aliasing equation (such as $x_j = x_k$). The solution we propose is then simply to treat both steps at once: if the j rule is an annotation rule, then simply call E on it and if j is an unfolding rule, then call E on the k rule, thus skipping the unfolding rule. Let J be the set : $\left\{ j_1\,\dfrac{k_1\,\dfrac{D_1}{d'_1}}{d_1}, \dots, j_n\,\dfrac{k_n\,\dfrac{D_n}{d'_n}}{d_n} \right\}$. We define the following notations:

$$\begin{array}{rcll} l_J(\iota) & = & j_\iota & \text{if } j_\iota \text{ is an annotation rule} \\ l_J(\iota) & = & k_\iota & \text{if } j_\iota \text{ is an unfolding rule} \end{array}$$

$$\begin{array}{rcll} \delta_J(\iota) & = & j_\iota\,\dfrac{k_\iota\,\dfrac{D_\iota}{d'_\iota}}{d_\iota} & \text{if } j_\iota \text{ is an annotation rule} \\[2ex] \delta_J(\iota) & = & k_\iota\,\dfrac{D_\iota}{d'_\iota} & \text{if } j_\iota \text{ is an unfolding rule} \end{array}$$

We can now define the function E for this case:

$$E\left(\; i\,\dfrac{j_1\,\dfrac{k_1\,\dfrac{D_1}{d'_1}}{d_1} \quad \dots \quad j_n\,\dfrac{k_n\,\dfrac{D_n}{d'_n}}{d_n}}{\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma} \;\right) =$$

$$\left\{ x_j = \dfrac{x_{l_J(j_1)} \quad \cdots \quad x_{l_J(j_n)}}{[\Gamma]_\infty \vdash f([\tau]_\infty) = [\sigma]_\infty} \right\} \cup \bigcup_{\iota \in \{1..n\}} E(\delta_J(\iota))$$

First step: As previously explained, each unfolding rule is erased while rewriting the a structural rule which is its goal. Since the goal of any (complete) typing derivation is mandatorily proven by a memoization rule, we also need to take care of this first step (as it is not handled by the previous case, since there is no structural rule below the first step). This gives the definition: $E\left(i\,\dfrac{j\,\dfrac{D'}{d}}{\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma} \right) = E(\delta_J(i))$

To conclude the proof, we need to remark the following points:

i. The application of E always terminates on a well-formed algorithmic derivation.

ii. The result of E is a system of guarded equations between a *finite* number of variables and rules of the type-system.

Point *i.* is straightforward since E is always applied to a strict subderivation of a finite derivation. Knowing that the number of applications of E is finite, point *ii.* is easily proven by induction on this number. In particular, we can see that since we erase the annotation in the case **(a-annot)**, we provide a derivation for $\lceil f \rceil$. Consequently, the *unique* solution of this system is a valid regular derivation of the non algorithmic type-system, which proves the theorem.

5.3.3 Completeness

Hitherto we showed the following properties:

(Termination) : The algorithm always terminates, either by inferring an output type or by failing.

(Soundness) : When the algorithm finds an output type, this output type is acceptable for the type-system (*i.e.* there exists a regular derivation corresponding to this output type).

However an algorithm that would fail too often (or which would always fail) classifies as sound. Normally, to ensure that the algorithm is useful, one would state the notion of completeness as the exact converse of the soundness:

(Soundness) : "Every judgment derivable by the algorithmic system $\mathscr{F}_{\mathscr{A}}$ corresponds to a judgment derivable in the type system, $\mathscr{F}$"

(Completeness) : "Every judgment derivable by the type system $\mathscr{F}$ corresponds to a judgment derivable in the algorithmic system, $\mathscr{F}_{\mathscr{A}}$"

While we have proven exactly this soundness theorem, we cannot prove its converse as stated. Indeed, we have seen that for a given input type and filter, there are possibly many different derivations of the system. An algorithm, which is deterministic, has to pick one. As previously stated, there is no notion of principal type in our setting.

Indeed, in our framework, the notion of "being more general than" translates to "being a subtype of", that is the one, among all the possible output types, which best approximates the exact set of output values. As already explained, such a set is in general not regular and does not even have a better regular approximation, which forbids any sensible notion of type principality.

A second problem is "how can the completeness theorem take annotations into account?". Indeed, while, starting from a derivation of the algorithm, it was easy to erase the annotations and arrive to a derivation in the system. But if we start from a derivation of the system (and therefore dealing with *unannotated filters*), how can we decorate the filter with annotations — which, in some cases are mandatory for the algorithm to succeed? The notion of completeness we propose allows us to kill these two birds with one stone[2].

The first observation is that we want to relate a derivation of the algorithm to a given derivation of the type-system. So why do not we just take the annotations from this very type-system derivation? Indeed, it would be easy to show that, given a derivation in $\mathscr{F}$ for $\Gamma \vdash f(t) = s$, we can build a derivation in $\mathscr{F}_{\mathscr{A}}$ for $[\Gamma]_\mu, \varnothing \vdash_{\mathscr{A}} g([t]_\mu) = [s]_\mu$ where g is the same filter as f where *every* sub-filter of g is annotated with the corresponding output type in the typing derivation. Unfortunately, such property is not very useful as it states that, for the algorithm to succeed, the programmer has to annotate every sub-filter with the corresponding partial result found in the typing derivation. The algorithm would then not be a "type inference algorithm" (in the sense of: given an input type, infer an output type), but a type-checking algorithm, and a bad one, which would not only require the input and output types but also all the intermediary types. However, as illustrated by Example 5.3, for every structural rule but the composition, the algorithm behaves like the type system. This gives a much lighter constraint on the number of annotations needed to type a filter and, we think, a very reasonable notion of completeness. Indeed we can characterize minimal sets of annotations for a filter to be typable by the algorithm. These annotations would serve as guides for the algorithm and we will show that, in practice (see Chapter 6), these annotations are very light and that the programmer does not have to "guess too much". We now formalise all these notions and state the completeness theorem. We proceed in three steps. First, we highlight the cases where the algorithm fails, and more precisely, fails due to a lack of annotations (or to incorrect annotations). Then, we give a sufficient condition on annotations such that a filter annotated in this way can either be typed or be detected as ill-typed. Finally, we state the completeness theorem, ensuring that if a filter is well typed in the type-system, with respect to a certain input type t, then the algorithm succeeds in typing the filter, provided that the latter is sufficiently annotated. Let us start by pinpointing the cases where the algorithm fails:

Lemma 5.7 (Failure cases) *The algorithm fails if and only if at least one of the following three conditions holds:*

i. One of the side conditions for the current rule is not true (e.g. the input type of a product filter is not a product).

ii. One of the four meta operators τ / p, or $\boldsymbol{\pi}(\tau)$, or testing for equality, or subtyping is applied to a type τ such that $FV(\tau) \neq \varnothing$.

iii. The choice operator cannot find a suitable type amongst the annotations given for a certain filter $f_{\mathbb{E}}$.

[2] No animal was harmed during the writing of this thesis.

Proof (Failure cases) We have to prove two implications:

1. If the algorithm fails then one of the condition holds
2. If one condition hold then the algorithm fails, or equivalently, if the algorithm does not fail, then none of the condition holds.

Case 1 is trivial. (Definition of the algorithm).
Case 2 is proven by induction on the derivation (which we assumed exists, as we assumed the algorithm does not fail). It is easy to show by a case analysis that none of the conditions are true.

A failure for the algorithm may have different meanings. Case *(i.)* means that the term is ill-typed and the algorithm fails with a type error. In case *(ii.)*, the algorithm is deconstructing a type which contains free recursion variables, that is, a type which it is currently computing; it therefore fails due to a lack of information and *more annotations* are required. Of course, we would like to avoid cases *(ii.)* and *(iii.)* while keeping the annotations as minimal as possible. The only problematic case is, as stated before, the composition of two filters. More formally:

Definition 5.8 (Deconstructing subterms)
A filter f deconstructs *its input if and only if f is not an* expression *filter. A* recursive *filter f is a filter such that the associated regular tree is not finite. Let f be a filter. We define the set of all deconstructing sub-terms of f, noted $\mathbb{A}_f$, as the set of all sub-terms g of f such that $g \equiv f_1;f_2$ where f_1 is recursive and f_2 deconstructs its input.*

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

Lemma 5.9 (Mandatory annotations) *Let τ be an input type and f a filter such that for all $f_1;f_2 \in \mathbb{A}_f$, $f_1 \equiv f'_{\mathbb{E}}$ for some $\mathbb{E}$. For all type τ' occurring in the derivation of $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$, if $\mathcal{FV}(\tau') \neq \varnothing$, then τ' is never deconstructed.*

Proof (Mandatory annotations) We assume that the derivation of $\Gamma, \Delta \vdash_{\mathscr{A}} f(\tau) = \sigma$ does not fail for reason $(i.)$, that is, τ is a valid input type for filter. We know that the derivation of the algorithm is finite. As such there is a finite number n of applications of the rule **(a-comp)**. We prove the lemma by induction on this number.

$n = 0$: there is no application of rule **(a-comp)** which means that there is no sub-filter of f of the form $f_1;f_2$. For all the other rules, the premises are applied only on the input type, which is closed (as it is a decomposition of the global input type of the filter). The property holds.

$n >_{\mathbb{N}} 0$: that means that somewhere in the derivation, we have:

$$D \equiv \dfrac{\dfrac{D_1}{\Gamma', \Delta' \vdash_{\mathscr{A}} f_1(\tau') = \sigma'_1} \qquad \dfrac{D_2}{\Gamma', \Delta' \vdash_{\mathscr{A}} f_2(\sigma'_1) = \sigma'_2}}{\Gamma', \Delta' \vdash_{\mathscr{A}} f_1;f_2(\tau') = \sigma'_2}$$

By hypothesis, f_1 is of the form ${f'}_{\mathbb{E}}$ for some $\mathbb{E}$ and so, the derivation D_1 starts by an application of the rule **(a-annot)** which returns a closed type $\sigma'_1 \in \mathbb{E}$. We can also remark that there are strictly less applications of the rule **(a-comp)** in D_2, therefore the induction hypothesis does apply here, which proves the inductive case.

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

Definition 5.10 (Output set)
Let $D = \dfrac{D'}{\Gamma \vdash f(t) = s}$ *be a regular derivation of the type-system. Let* f' *be a sub-filter of* f*, that is* $f' \sqsubseteq f$*. We define the* output type set *of* f' *in* D *as:*

$$\mathscr{O}(D, f') \equiv \{[s]_\mu | \exists \Gamma', t' \textit{ such that } \dfrac{\vdots}{\Gamma' \vdash f'(t') = s} \sqsubseteq D\}$$

We recall that $\sqsubseteq$ is the sub-tree relation and that it also applies to derivations, seen as regular trees. The set $\mathscr{O}(D, f')$ is the collection of all possible output types for a filter f' in a given derivation. It is important to note that, since D is regular, the output type set is always finite. We can now define the t-labelling of a filter:

Definition 5.11 (t-labelling)
Let f *be a filter and* t *a type such that a regular derivation* D *for* $\Gamma \vdash f(t) = s$ *exists, for some type* s*. Let* $\mathbb{A}_f = \{f^1_1;f^1_2, \dots, f^n_1;f^n_2\}$*, The* t-labelling *of* f*, noted* $[\![f]\!]_{t,D}$ *is defined as:*

$$[\![f]\!]_{t,D} = f\{f^1_1 \leftarrow {f^1_1}_{\mathscr{O}(D,f^1_1)}; \dots; f^n_1 \leftarrow {f^n_1}_{\mathscr{O}(D,f^n_1)}\}$$

We can now use Definition 5.11 to state the completeness of the algorithm with respect to t-labellings.

Theorem 5.12 (Completeness) *The algorithm given by the set of rules $\mathscr{F}_{\mathscr{A}}$ is complete with respect to the type-system $\mathscr{F}$, that is:*
if $\Gamma \vdash f(t) = s$ is proved by a derivation D, then $\Gamma, \varnothing \vdash_{\mathscr{A}} \llbracket f \rrbracket_{t,D}([t]_\mu) = [s]_\mu$.

Proof (Completeness) First, let us consider a judgment $\Gamma \vdash f(t) = s$ and its derivation D_0. By Lemma 4.6 (Chapter 4), we know that $\Gamma \vdash f(t) = s$ can be proven by a derivation D_1 for which every instance of the subsumption rule (but the first rule, which proves the goal of the entire derivation) is in the premise of the application of a composition rule, **(t-comp)**. It is also true that there exists a derivation D which proves the judgment and for which the first premise of rule **(t-comp)** is proven by a subsumption rule. It is sufficient to consider D_1: if for one of the instances of the composition rule in D_1 the first premise is not proven by **(t-subs)**, then we can insert a step:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma \vdash f_1(t) = s_1} \qquad s_1 \leq s_1}{\Gamma \vdash f_1(t) = s_1} \qquad \dfrac{\vdots}{\Gamma \vdash f_2(s_1) = s_2}}{\Gamma \vdash f_1;f_2(t) = s_2}$$

Using this remark and Lemma 4.6 we can then assume, without loss of generality, that for any judgement $\Gamma \vdash f(t) = s$ derivable in the system, there exists a derivation D, such that:

i. the goal of the derivation is proven by an application of the subsumption rule

ii. the first premise of an application of the rule **(t-comp)** is proven by a subsumption rule

The idea of the proof is now to build a (finite) algorithmic derivation for $\Gamma, \varnothing \vdash_{\mathscr{A}} \llbracket f \rrbracket_{t,D}([t]_\mu) = [s]_\mu$. Intuitively, an application of **(t-subs)** will be rewritten into an application of the annotation rule **(a-annot)**, while the other rule will straightforwardly be rewritten into structural rules of the algorithmic system. This procedure is formalized by the recursive function $[\_]_{\mathscr{M}}$ defined in Figure 5.4. This function takes three arguments, the memoization environment Δ (initially empty), the non algorithmic derivation D, and an input filter, annotated according to D. Given the judgment $\Gamma \vdash f(t) = s$, $[\varnothing, D, \llbracket f \rrbracket_{t,D}]_{\mathscr{M}}$ computes a finite, well-formed derivation for the judgement $\Gamma, \varnothing \vdash_{\mathscr{A}} \llbracket f \rrbracket_{t,D}([t]_\mu) = [s]_\mu$. Informally, $[\_]_{\mathscr{M}}$ is applied to Δ, D and $\llbracket f \rrbracket_{t,D}$. Depending on the conditions, it produces the appropriate memoization rule. For instance, if the input type has already been visited (*i.e.* if it is in Δ), then it returns an instance of **(a-base-rec)**. If the

input derivation starts by a subsumption rule, it returns a corresponding **(a-annot)** rule, after which it calls $[\_]_{\mathscr{S}}$ on the input derivation. The role of this auxiliary function is to insert, in the output derivation, the corresponding structural rule, as well as to unfold the third argument, the annotated filter. Indeed, during the whole process, the filter f in the typing derivation D, and the t-labelled filter $[\![f]\!]_{t,D}$ are unfolded simultaneously, and the annotated filters are used in place of their corresponding stripped version in the algorithmic derivation.
To prove that the result of $[\_]_{\mathscr{M}}$ is indeed a well-formed algorithmic derivation, we must first prove that it terminates for any input. This can easily be done, in a way very similar to the proof of Theorem 5.4. It is sufficient to consider the triple $(f, \Psi(\Delta, [t]_\mu), m'(D))$, where m' evaluates to 1 for a recursive call to $[\_]_{\mathscr{M}}$ and to 0 for a recursive call to $[\_]_{\mathscr{S}}$ (it serves the same purpose as the measure m in the proof of Theorem 5.4). An induction on this triple, ordered by $(\sqsubseteq, \subseteq, \leq_{\mathbb{N}})$, proves that the function terminates. As for Theorem 5.4, the termination of this functions means either that it outputs a derivation (success), or that it fails. In the present case, a failure can only occur during a call to $[\_]_{\mathscr{S}}$, if the third argument has not the same form as the filter in the goal of the second argument. Let us consider Δ, $D = \dfrac{D'}{\Gamma \vdash f(t) = s}$ and let g be a filter such that $f \equiv \lceil g \rceil$.
Since we know that a call to $[\_]_{\mathscr{M}}$ always terminates, we can show by induction that exactly one of the rule can be applied:

basic case: there are no recursive calls, and $[\_]_{\mathscr{M}}$ is directly applied to the typing derivation of an expression filter. Since by hypothesis, the call is made on $[\varnothing, D, [\![e]\!]_{t,D}]_{\mathscr{M}}$, the first rule of $[\_]_{\mathscr{M}}$ applies and the function does not fail

inductive case: By case analysis on the different forms of f we can verify, for every call to $[\Delta, D, [\![g]\!]_{t,D}]_{\mathscr{M}}$, that D proves exactly the filter $\lceil g \rceil$. Since by definition g and $\lceil g \rceil$ have the same form (the same top level constructor), always one of the rules of $[\_]_{\mathscr{M}}$ applies, and consequently, the function does not fail.

Now that we know that the function always returns a derivation, we can prove the well-formedness of this output algorithmic derivation by induction on the number of call to $[\_]_{\mathscr{M}}$. It is easy to verify, by a case analysis that:

i. for every instance of a composition rule, the first premise is proved by an annotation rule

ii. the filters occuring in the algorithmic derivation are well-annotated

iii. the instance of the annotation rules are well-formed, that is, $[s]_\mu \in \mathbb{E}$ (see the second case in Figure 5.4).

iv. any memoization rule is followed by a structural rule

Case *(i.)* is a direct consequence of particular shape of D: the first premise of every composition rule is proved by a subsumption rule which is translated into an annotation rule. Case *(ii.)* is also straightforward since the filters in the output derivations come from the t-labelling of the input filter. By Lemma 5.7 and 5.9, such a filter makes the algorithm succeed. Case *(iii.)* is also a consequence of the shape of D. By definition, the t-labelling of f annotates every occurrence of filters a filter f_1 occuring as the left-hand side of a composition: $f_1;f_2$. However, in D any output type of such an f_1 is proved by a subsumption rule. Case *(iv.)* is immediate by construction: the alternation between memoization rules and structural rules results from the alternate calls to $[\_]_{\mathscr{M}}$ and $[\_]_{\mathscr{S}}$ in their definition.

$$\left[\Delta, \frac{D}{\Gamma \vdash f(t) = s}, g\right]_{\mathscr{M}} = \frac{}{[\Gamma]_\mu, \Delta \vdash_{\mathscr{A}} g([t]_\mu) = \alpha} \text{ if } (g, [t]_\mu, \alpha) \in \Delta$$

$$\left[\Delta, \frac{\dfrac{D}{\Gamma \vdash f(t) = s' \quad s' \leq s}}{\Gamma \vdash f(t) = s}, g_{\mathbb{E}}\right]_{\mathscr{M}} = \frac{\dfrac{\left[\{(g_{\mathbb{E}}, [t]_\mu, [s]_\mu)\} \cup \Delta, D, g_{\mathbb{E}}\right]_{\mathscr{S}}}{[\Gamma]_\mu, \{(g_{\mathbb{E}}, [t]_\mu, [s]_\mu)\} \cup \Delta \vdash_{\mathscr{A}} g_{\mathbb{E}}([t]_\mu) = [s']_\mu}}{[\Gamma]_\mu, \Delta \vdash_{\mathscr{A}} g_{\mathbb{E}}([t]_\mu) = [s]_\mu}$$

$$\begin{array}{l}\left[\Delta, \dfrac{D}{\Gamma \vdash f(t) = s}, g\right]_{\mathscr{M}} \\ t \text{ is recursive, } [t]_\mu = \mu\alpha.\tau \\ \beta \text{ is fresh,} [s]_\mu \equiv \mu\beta.\sigma \end{array} = \frac{\dfrac{\left[\{(g, [t]_\mu, \beta)\} \cup \Delta, D, g\right]_{\mathscr{S}}}{[\Gamma]_\mu, \{(g, [t]_\mu, \beta)\} \cup \Delta \vdash_{\mathscr{A}} g(\tau[\alpha \leftarrow [t]_\mu]) = \sigma}}{[\Gamma]_\mu, \Delta \vdash_{\mathscr{A}} g([t]_\mu) = \mu\beta.\sigma}$$

$$\begin{array}{l}\left[\Delta, \dfrac{D}{\Gamma \vdash f(t) = s}, g\right]_{\mathscr{M}} \\ t \text{ is not recursive} \\ \alpha \text{ is fresh,} [s]_\mu \equiv \mu\alpha.\sigma \end{array} = \frac{\dfrac{\left[\{(g, [t]_\mu, \alpha)\} \cup \Delta, D, g\right]_{\mathscr{S}}}{[\Gamma]_\mu, \{(g, [t]_\mu, \alpha)\} \cup \Delta \vdash_{\mathscr{A}} g([t]_\mu) = \sigma}}{[\Gamma]_\mu, \Delta \vdash_{\mathscr{A}} g([t]_\mu) = \mu\alpha.\sigma}$$

$$\left[\Delta, \frac{\texttt{type}(\Gamma, e) = s}{\Gamma \vdash e(t) = s}, e'\right]_{\mathscr{S}} = \frac{\texttt{type}_{\mathscr{A}}(\Gamma, e') = [s]_\mu}{[\Gamma]_\mu, \Delta \vdash_{\mathscr{A}} e'([t]_\mu) = [s]_\mu}$$

$$\left[\Delta, \frac{D_1 \qquad D_2}{\Gamma \vdash (f_1, f_2)(t) = s}, (g_1, g_2)\right]_{\mathscr{S}} = \frac{[\Delta, D_1, g_1]_{\mathscr{M}} \qquad [\Delta, D_2, g_2]_{\mathscr{M}}}{[\Gamma]_\mu, \Delta \vdash_{\mathscr{A}} (g_1, g_2)([t]_\mu) = [s]_\mu}$$

$$\left[\Delta, \frac{D_1 \qquad D_2}{\Gamma \vdash f_1; f_2(t) = s}, g_1; g_2\right]_{\mathscr{S}} = \frac{[\Delta, D_1, g_1]_{\mathscr{M}} \qquad [\Delta, D_2, g_2]_{\mathscr{M}}}{[\Gamma]_\mu, \Delta \vdash_{\mathscr{A}} g_1; g_2([t]_\mu) = [s]_\mu}$$

$$\left[\Delta, \frac{D_1 \qquad D_2}{\Gamma \vdash f_1 | f_2(t) = s}, g_1 | g_2\right]_{\mathscr{S}} = \frac{[\Delta, D_1, g_1]_{\mathscr{M}} \qquad [\Delta, D_2, g_2]_{\mathscr{M}}}{[\Gamma]_\mu, \Delta \vdash_{\mathscr{A}} g_1 | g_2([t]_\mu) = [s]_\mu}$$

$$\left[\Delta, \frac{D_1}{\Gamma \vdash p \rightarrow f_1(t) = s}, p \rightarrow g_1\right]_{\mathscr{S}} = \frac{[\Delta, D_1, g_1]_{\mathscr{M}}}{[\Gamma]_\mu, \Delta \vdash_{\mathscr{A}} p \rightarrow g_1([t]_\mu) = [s]_\mu}$$

Figure 5.4: Translation functions $[\ ,\ ]_{\mathscr{M}}$ and $[\ ,\ ]_{\mathscr{S}}$

Part III

Implementation

Chapter 6

Concrete language

> Throughout this chapter, we describe the process of designing a practical language based on the filter algebra presented in Part II. This implementation uses ℂDuce as host language, and benefits from many extensions derived from the core algebra.

Contents

6.1 Introduction

We have presented a theoretical framework for manipulating XML data-structures, in the form of the filter algebra and the associated type-system and inference algorithm. While for the theoretical framework our aim has been to abstract from any particular host language, this chapter will present a tight integration between filters and ℂDuce. This integration is present at many levels. First we show how we extend the syntax of ℂDuce to provide filter declarations and applications. We also show that many (if not all) of the hard-coded operators of ℂDuce can be encoded with filters. A second point of interest is how we deal with annotations, particularly how we achieve our goal of keeping the language filter *modular* even in the presence of annotations. Lastly we present our compilation scheme for filters, and show how to apply optimisations present in the core ℂDuce compiler to filters.

6.1.1 Basic syntax

Filters are a natural extension of ℂDuce's syntax:

Definition 6.1 (Concrete syntax)

$$\begin{array}{lcll}
e & ::= & \dots & \textit{(}\mathbb{C}\textit{Duce expressions)} \\
 & | & \texttt{apply } f \texttt{ to } e & \textit{(filter application)} \\
 & | & \texttt{apply } f \texttt{ to } e \texttt{ where } a & \textit{(filter with annotations)} \\
 & & & \\
f & ::= & \texttt{\textasciitilde\{}e\texttt{\}} & \textit{(}\mathbb{C}\textit{Duce expressions)} \\
 & | & \texttt{(}f\texttt{,}f\texttt{)} & \textit{(pair filter)} \\
 & | & \texttt{\$}p\texttt{->}f & \textit{(pattern filter)} \\
 & | & f\texttt{|}f & \textit{(alternative filter)} \\
 & | & f\texttt{;}f & \textit{(composition filter)} \\
 & | & \texttt{<}f\ f\texttt{>}f & \textit{(XML filter)} \\
 & | & X & \textit{(filter variable)} \\
 & | & \texttt{let filter } X \texttt{ = } f\ [\texttt{and } Y \texttt{=} f \dots]\ \texttt{ in } f & \textit{(recursive filter)} \\
 & & & \\
a & ::= & X_1 \texttt{ = \{|} t_{11} \texttt{;} \dots \texttt{;} t_{1n} \texttt{|\}}\ [\texttt{and} \dots] & \textit{(annotations)} \\
 & & & \\
d & ::= & \dots & \textit{(}\mathbb{C}\textit{Duce toplevel declarations)} \\
 & | & \texttt{let filter } X \texttt{ = } f\ [\texttt{and } Y \texttt{=} f \dots] & \textit{(global filter definition)}
\end{array}$$

In the above definition, we used the standard square bracket notation to denote optional parts of an expression; e is the entry point of ℂDuce's grammar for expressions, t_i the one for types and d the one for top-level declarations.

	Operator
High precedence	~{_} $_->_,(_,_),<_ _>_
	_;_
Low precedence	_|_

Table 6.1: Precedences of filter operators

The reader will easily recognize in the f grammar the basic filter algebra. The additional symbols for expression and pattern filters were added so as to easily disambiguate[1] filter expressions from patterns and ℂDuce expressions. For instance, without the delimiters, the phrase:

Example 6.2

```
(x,x)
```

could denote either the pair expression (x, x) (where x is a variable), or the filter ~{(x, x)} (where x is a filter identifier) or the filter (~{x},~{x}) (where x is again a standard variable). In order to make the examples more readable, we will omit, when possible, the use of parenthesis. The precedence of the filter operators are given in Table 6.1.1 from higher to lower.

6.1.2 XML filters

The XML filter follows the same encoding as the one used for ℂDuce's XML values. We recall that an XML value is encoded as:

$$\begin{gathered}\texttt{<tag a}_1 = \texttt{va}_1\ \ \dots\ \ \texttt{a}_\texttt{k} = \texttt{va}_\texttt{k}\texttt{>[v}_1\dots\texttt{v}_\texttt{n}\texttt{]}\\ \Downarrow\\ (\texttt{`tag},(\{a_1 = va_1\dots a_k = va_k\},(v_1,(\dots,(v_n,\texttt{`nil})))))\end{gathered}$$

where $\{a_1 = va_1\dots a_k = va_k\}$ is an extensible record used to encode the list of attributes of the XML element `tag. The XML filter is then nothing but:

$$\texttt{<}f_1\ f_2\texttt{>}f_3 \equiv \texttt{(}f_1\texttt{,(}f_2\texttt{,}f_3\texttt{))}$$

The first filter f_1 is applied to the tag (which turns out to be very useful in practice for tag conversion), the second filter f_2 to the attributes and the last one, f_3 on the content of the element.

[1] Both for the programmer and the parser

6.1.3 Recursive filters

In the algebra, filters are regular tree. In the programming language, this translates naturally into mutually recursive filter expressions. Recursive declarations are inspired by recursive functions. In this presentation, we underline filter names so as to distinguish them more easily from function names and capture variables. For example the filter *succList* of Section 3.4.3 is written:

Example 6.3

```
let filter succList =  $  [] -> ~{ [] }
| (  $  x -> ~{ x+1 }, succList)
```

This filter can then be used in an `apply to` construct:

Example 6.4

```
let filter succList =  $  [] -> ~{ [] }
                     | (  $  x -> x+1, succList);;

let res = apply succList to [ 1 2 3 4 5 6 ];;
```

The previous code, evaluated in the toplevel interpreter of ℂDuce returns:

```
val res : [ 2 3 4 5 6 7 ] =[ 2 3 4 5 6 7 ]
```

In this code snippet, *res* is a regular ℂDuce variable, defined by the classical `let` binding. The answer of the interpreter means that the (global) variable *res* has type `[2 3 4 5 6 7]` and is equal to `[2 3 4 5 6 7]`. Here the type is the most precise one: it is the singleton type containing *only* the value `[2 3 4 5 6 7]`.

6.1.4 Filter annotations

As we saw in Definition 6.1, the way annotations are specified is quite different from the formal presentations. Indeed, filter annotations are given at the place of the application (that is, in the `apply to` construct) rather than at the declaration of the filter. This feature is fundamental for code reuse and modularity. Indeed, it is a problem every developer faces when (s)he relies on type annotations: an annotated code is marked and "specialized" for the specific annotation, thus forbidding its usage in a different typing context. For example, let us consider the reverse filter:

Example 6.5

```
let filter rev =  $  x ->
  (let filter aux =
    $  [] -> ~{ ([],x) }
     | (  $  z ->  ~{ z }, aux) ;
                (  $ (_,(_,[])) ->~{ ([],[]) }
                  |   $  (_,(y,(h,t))) ->~{ ((h,y),t) }
                )
  in aux); $  (x,_) -> ~{ x };;
```

This filter is the same as the one given in Section 3.4.4. Because of the compositions at Line 4 and 8, some annotations might be needed, but not always. For example, the application of this filter to a constant does not need any annotation:

Example 6.6

```
apply rev to [ 1 2 3 4 ];;
```

returns:

```
— : [ 4 3 2 1 ] =[ 4 3 2 1 ]
```

Indeed, when applied to a singleton types or, more generally, to *non-recursive* types, the typing of filters behaves exactly like the evaluation, hence providing a great precision. For example:

Example 6.7

```
apply rev to ([ 1 "Foo" 3 ‘true ]:[Int String Int Bool]);;
```

returns:

```
— : [ Bool Int String Int ] =[ ‘true 3 "Foo" 1 ]
```

In this example, the “value:type” notation is a coercion or up-cast: the value `[ 1 "Foo" 3 ‘true ]` is not seen as being of type `[ 1 "Foo" 3 ‘true ]` (the corresponding singleton type) but of type `[Int String Int Bool]`. As the latter is finite (non recursive) the typing of the reversal works properly without annotations. If we now up-cast the same value to a recursive type such as `[Any*]`, we have to annotate the filter:

Example 6.8

```
apply rev to ([ 1 "Foo" 3 ‘true ]: [Any*]);;
```

fails with:

```
Characters 0−67:
 Insufficient  annotations
```

Had we implemented filters as their formal counterpart, we would have been faced here with the sole choice of putting a generic annotation (such as `[Any*]`) in the code of the filter. This would have meant rewriting the filter with a new annotation each time list reversal is applied. This is exactly the problem we wanted to avoid while defining filters because this is the exact behaviour of *hand-written functions*. Worse, if we look at the filter then we see that according to the formal algebra, this filter needs *two* annotations. One at line 4 for the filter “`($z->~{z},aux);...`” and one at line 8 for the composition of aux. To offer the user a more flexible way to provide annotations, we designed the language so that annotations are provided at “application time”. The code for rev stays therefore as it is (unannotated) while the application becomes:

Example 6.9

```
let r1 = apply rev to ([ 1 "Foo" 3 ‘true ]:[Any*])
  where aux = {| ([Any*],[Any*]) |};;

let r2 = apply rev to ([ 1 "Foo" 3 ‘true ]:[ (Int|String|Bool)* ])
  where aux = {| ([ (Int|String|Bool)* ],[ (Int|String|Bool)* ]) |};;
```

Here we only need to provide the output type of aux, which is a pair formed by the type of the accumulator and the type of the list we reverse. We see that many instantiations of rev can have different annotations:

```
r1 : [ Any* ] = [ ‘true 3 "Foo" 1 ]
r2 : [ (String | Int | Bool)* ] = [ ‘true 3 "Foo" 1 ]
```

Naturally, this “double annotation” (here, since aux is a product filter, the type `[(Int | String | Bool)*]` needs to be “duplicated” in the annotation) is still quite cumbersome. Further extensions to the filter algebra allow us to alleviate this constraint as we will see hereafter.

6.2 Examples

6.2.1 Pattern-matching

As stated in Chapter 3, ℂDuce's pattern-matching construct, `match with` can be encoded with filters:

Example 6.10

`match e with`
| p_1 `->`e_1
⋮
| p_n `->`e_n

is only syntactic sugar for:

`apply ( $` p_1 `->~{`e_1 `}|...| $` p_n `->~{`e_n `})to e`

Another example of ℂDuce operator that can easily be encoded with filters is the so called *upward coercion*. As already explained, ℂDuce proposes a coercion operator, "`:`" (read *forget*), which up-casts the type of an expression. For instance the expression "`2:Int`" has type `Int` while the expression "2" would have the more precise singleton type "2". Of course, the compiler checks that the upward coercion is possible, that is, that the type of the expression is a subtype of the desired output type. For instance, the expression "`2:Bool`" naturally raises a type error. A variant of this construct is the dynamic cast, "`:?`" which checks at run-time that a value has a particular type, and raises an exception if the tests fails. For instance, the expression "`(2:Int) :?` 2", succeeds. The expression "2" is coerced to `Int` and then, *at run time*, checked against the singleton type "2". The difference with the static up-cast is that, for example, the expression "`(2:Int) :?` 3" type-checks correctly but raises an exception at run-time. Both constructs can be encoded as filters (in the following example, *t* is the type we want to constraint the input to):

Example 6.11

```
let filter stc_cast =  $  x&t -> ~{ x }
;;
let filter dyn_cast =  $  x&t -> ~{ x }
                    |  $  _ -> ~{ raise  "Cast␣error" }
;;
let x = apply stc_cast to 2;;
let y = apply dyn_cast to x;;
```

6.2.2 Map-like filters

Filters are enough to express "map-like" functions and type them as precisely as e.g. the `map` and `xtransform` operators of ℂDuce. The `map` construct iterates expressions guarded by patterns on every element of a sequence:

Example 6.12

```
        map [ 1 2 "0" ‘false ] with
           x&Int -> x+1
          | x&String ->x @"+1"
          | x&Bool -> not x
```

returns:

```
— : [ 2 3 "0+1" ‘true ] =[ 2 3 "0+1" ‘true ]
```

This construction can be simulated by the the following expression, which is as precisely typed:

Example 6.13

```
apply let filter mymap =$  [] -> ~{ [] }
                        | ( (  $  x&Int -> ~{ x+1 }
                             | $  x&String ->~{ x@"+1"}
                             | $  x&Bool -> ~{ not x }),
                            mymap) in mymap
to [ 1 2 "0" ‘false ]

— : [ 2 3 "0+1" ‘true ] =[ 2 3 "0+1" ‘true ]
```

The same holds for the `xtransform` operator which is a generalization of `map` to XML values. It iterates a list of expressions (guarded by patterns) over every (XML) elements of a list (given as input). If a pattern matches then the corresponding expression is evaluated. If no pattern matches, the element is left untouched and the transformation is recursively applied to its content. For instance:

Example 6.14

```
xtransform [ <a>[ <b>[] <b>[ <c>[] <b>[] ] ] ] with
| <b>x -> [ <B>x ]
```

returns:

```
— : [ <a>[ <B>[ ] <B>[ <c>[ ] <b>[ ] ] ] ] =
                           [ <a>[ <B>[ ] <B>[<c>[ ] <b>[ ] ] ] ]
```

We see that while the first and second <b> are capitalized, the third one is not. This is because the second <b> is under a <b> itself. This behaviour can be easily reproduced by a filter:

Example 6.15

```
apply let filter myxtrans = $  [] -> ~{ [] }
                         | ( $  <b>x -> ~{ <B>x }
                             | <( $  x -> ~{x}) ( $  x -> ~{x})>myxtrans
                             |  $  x -> ~{ x }
                             , myxtrans)
      in myxtrans
to
[ <a>[ <b>[] <b>[ <c>[] <b>[] ] ] ]
```

returns:

```
— : [ <a>[ <B>[ ] <B>[ <c>[ ] <b>[ ] ] ] ] =
                           [ <a>[ <B>[ ] <B>[<c>[ ] <b>[ ] ] ] ]
```

However, this limitation of `xtransform` is clearly annoying for real-life types, such XHTML. Indeed, it is common in such types to find mixed and recursive content (such as the tags <b>, <i> and <a> of XHTML) which can occur without restriction below one another. Despite the pervasiveness of such types, hard-coded operators such as `xtransform` fall short and do not allow one, for instance, to rewrite every <b> into a <i> in an XHTML document. This was the primary reason to add filters to ℂDuce: allow the programmer to perform a controlled recursive transformation over an input document and at the same time precisely type this transformation:

Example 6.16

```
apply let filter capitalize  =  $  [] -> ~{ [] }
                         | ( <( $  ‘a -> ~{ ‘A }
                              | $  ‘b -> ~{ ‘B }
                              | $  ‘c -> ~{ ‘C }) ( $  x -> x)> capitalize
                             |  $  x -> ~{ x }
                             , capitalize )
      in  capitalize
to
```

```
[ <a>[ <b>[] <b>[ <c>[] <b>[] ] ] ]
```

returns:

```
- : [ <A>[ <B>[ ] <B>[ <C>[ ] <B>[ ] ] ] ] =
                              [ <A>[ <B>[ ] <B>[<C>[ ] <B>[ ] ] ] ]
```

6.2.3 Non local transformations

Thanks to composition, filters can perform non-local transformations. For example, it is possible to write a filter which takes an XML document as input and replaces the rightmost node with the same values as the leftmost node. Let us show it in detail. We simplify a little bit this definition so that the filter remains readable[2]. To be more precise, we define a filter which takes the leftmost element which must be an *empty element*, that is of the form `<tag>[]` for any tag. This leftmost used to replace the rightmost element of the tree, which must itself also be empty:

Example 6.17

```
let filter id =  $  x -> ~{ x } ;;
let filter leftmost =  $  x & <_ ..>[] -> ~{ x }
                     | <(id) (id)> (leftmost,id); $  <_ ..>(x,_) -> ~{ x };;
let filter replace =  $  x -> ~{ x };
                             leftmost;  $  y ->
                             (~{x};
                                let filter rightmost =  $  <_ ..>[] -> ~{y}
                                                        | <(id) (id)>iter
                                and iter  = ( rightmost,  $ [] -> ~{[]})
                                          | (id, iter)
                                in rightmost);;
```

First of all the leftmost filter returns its argument if it is an empty XML element. If not, it is recursively applied to the first child of this argument, after what, the reconstructed element is composed with a pattern that selects only the interesting part of its input. The second part is the replace filter. It takes the tree we want to transform as argument and saves it in x. Then it applies the leftmost filter on it, whose result is stored in y. After that, x is recalled and the recursive filter rightmost is applied to it. The latter iterates until it finds the desired sub-tree and then replaces it by y. What is interesting with this filter is of course the way the type inference algorithm behaves. And as we can see, it performs quite well:

[2] And we also acknowledge the rather poor choices in syntax and delimiters, but making such ambiguous and overlapping parsers as the one for filters and the one for expressions left us with little choice...

```
let r0 = apply replace to <a>[ <b>[] 1 ]
;;

This expression should have type:
X1 where X1 = <_ (Record)>[ ] | <(Any) (Any)>[ Any+ X1 | X1 ]
but its inferred type is:
<a>[ <b>[ ] 1 ]
which is not a subtype, as shown by the sample:
<a>[ <b>[ ] 1 ]
```

In this case, the algorithm detects that the type of the argument is incompatible, because the rightmost element is not an XML element. If we now apply the filter on a valid input type, everything works as expected:

```
let r1 = apply replace to <a>[ <b>[ <c>[] <d>[] ] ] <z>[] ]
;;

val r1 : <a>[ <b>[<c>[] <d>[]] <c>[] ] =<a>[ <b>[<c>[] <d>[]] <c>[] ]
```

Here the leftmost element is <c>[] and the rightmost is <z>[]. As the filter is applied to a singleton type the typing can be very precise: the output type is the singleton type containing only the result of the application.

Since the replace and leftmost filters use the composition operation in many places, one may thus wonder what happens if the input type is recursive.

```
let r3 = apply replace to (<a>[ <b>[] <c>[] ]:<a>[<b>[]+ <c>[]+]);;

val r3 : <a>[ X1 X1+ | X1 X1+ X2+ X3 | X1 X2+ X3 ] where
         X1 = <b>[ ] and
         X2 = <c>[ ] and
         X3 = <b>[ ] = <a>[ <b>[ ] <b>[ ] ]
```

Since compositions are lightly used (in the sense of "do not deconstruct their input too much") the type inference algorithm can infer a precise output type without any annotations. The input type being a document <a>[...] with a non-empty list of <b>[] and a non-empty list of <c>[] as content, the computed output type is of course an <a>[...] whose content falls under one of the following three categories. First it can be [<b>[] <b>[]+]. It would be the case if the input value was in <a>[<b>[]+ <c>[]]. The <c>[] is replaced by a <b>[] hence the result. The input could also be in <a>[<b>[]+ <c>[] <c>[]+] (as the case for only one <c>[] was in the previous category there are at least two <c>[]). The last <c>[] is replaced by a <b>[] giving <a>[<b>[] <b>[]+ <c>[]+ <b>[]]. Finally the input type could be in <a>[<b>[] <c>[]+], which naturally gives an output type of <a>[<b>[] <c>[]+ <b>[]].

6.2.4 Annotations

Let us now present some interesting filters that require annotations. We start with the paradigmatic example: flattening.

Example 6.18

```
let filter concat =
 $  (x,y)  -> ( ~{x} ; (
                  let filter aux =
                    $  []  -> ~{ y }
                  | (   $  z  -> ~{ z }, aux )
                   in aux ))

let filter flatten  =
    $  []  -> ~{ [] }
     | (( $  [ Any* ] -> flatten , flatten  ) ; concat)
     | (  $  x -> ~{ x } ,  flatten  )
```

This filter flattens nested lists. For example the problematic input type `T` defined hereafter requires an annotation:

```
type T = [ ‘a T ‘b ] | []
apply flatten to ([‘a [] ‘b]:T) where flatten = {| [ (‘a|‘b)* ] |}

— : [ (‘a |‘b)* ] =[ ‘a ‘b ]
```

If we omit the annotation, then the typechecker fails as expected:

```
apply flatten to ([‘a [] ‘b]: T);;
Insufficient Annotations.
```

6.3 Syntactic extensions

We already saw that the core filter algebra is very powerful and allows one to write many interesting transformations and type them precisely. However, we made several extensions that we present next.

6.3.1 Deletion

The actual algebra makes the writing of *almost* copying filters cumbersome. Indeed, if one wants to write a filter which removes all the integers of a given list, (s)he may do so with the following filter:

Example 6.19

```
let filter remInt =  $  [] -> ~{ [] }
                  | (  $  x&Int -> ~{x},remInt);$ (_,y) -> ~{y}
                  | (  $  x -> ~{x}, remInt)
```

As one can see throughout the different examples we gave, a composition such as the one in Line 2 of the above example is quite common. Indeed, it is often the case that we only want to apply a filter on one projection of a pair or only on the content of an XML value (and discard its tag or attributes). To that end, we define the following filters:

Definition 6.20 (Projection filer)
The filter grammar is extended with the following productions:

$$
\begin{array}{rcll}
f & ::= & \dots & \textit{(previous definitions)} \\
 & | & \underline{\text{left}}\ f & \textit{(left projection)} \\
 & | & \underline{\text{right}}\ f & \textit{(right projection)} \\
 & | & \underline{\text{content}}\ f & \textit{(XML content)} \\
 & | & \underline{\text{tagcontent}}\ f_1\ f_2 & \textit{(discard attributes)}
\end{array}
$$

All these filters can be encoded in the core algebra. For instance:

$$\underline{\text{left}}\ f$$

is equivalent to:

```
(f,id);$(x,_)->~{x}
```

The interest of *inlining* the composition is that:

1. In practice, we remarked that those cases are very common
2. Getting rid of a composition gets rid of the associated annotations.

The previous filter can then be rewritten:

Example 6.21

```
let filter remInt =  $  [] -> ~{ [] }
                  |  $  (Int,_) -> right  remInt
                  | (  $  x -> ~{x}, remInt)
```

Similarily, discarding the tag (thus returning the content of an XML element) or discarding the attributes of an element will be the building blocks of the XPath encoding, presented in Chapter 7. As a teaser, the knowledgeable reader could consider the following example:

Example 6.22

```
let filter slash = $  [] -> ~{ [] }
                 | ( $  <a>x  -> ~{ x } , slash);concat
                 | right  slash
```

which is nothing but the XPath expression “`child::a`”. The semantics of XPath as well as our full encoding will be thoroughly presented in Chapter 7.

6.3.2 Filter parameters

As we saw earlier, filters can be used to encode ℂDuce iterators such as `xtransform` or `map`. However one may still be reluctant to use filters to express such constructs because of the burden of always *rewriting the same iterating part*. Indeed, there are two parts in such filters. The “branches” part which constitutes the code to execute on every traversed node and the actual code which iterates over an XML or list structure. As we aim for a better code modularity than with explicitly typed function, we also need a way to factorise the code of such filters. Indeed, a generic computation should only be written once. A first step has been taken in the previous section with the hardcoded projection filters. For instance, special typing aside, the left filter is a filter which apply its filter argument to the first component of its input and discards the second component. We generalize this approach with *macro filters*, which are toplevel filters with filters as parameters.

Definition 6.23 (Macro filters)
The toplevel definitions are extended with the productions:

$$
\begin{array}{rcll}
d & ::= & \ldots & \textit{(previous definitions)} \\
 & | & \texttt{let filter}\, X\, X_0 \ldots X_n = f\ [\ \texttt{and}\ \ldots] & \textit{(macro filters)}
\end{array}
$$

The term *macro filter* was inspired by Philip Wadlers *higher order macros*, introduced in his well-known work on deforestation [Wad90]. With an argument to Wadler’s, we claim that while macro filters are not “higher order” filters (since they do not have a type and are not even first class values), they suffice in practice to write concise code and obtain a behaviour one would expect from higher order functions:

Example 6.24

```
let filter map f =  $  [] -> ~{ [] }
                 | ( f, map);;

let filter trans1 =  $  x -> ~{ x+1 };;
let filter trans2 =  $  x&Int -> ~{ x+1 }
                  |  $  x&Bool -> ~{ not x};;

apply map (trans1) to [ 1 2 3 4 ];; (* returns [ 2 3 4 5 ] *)
apply map (trans2) to [ 1 ‘true 3 4 ];; (* returns [ 2 ‘false 4 5 ] *)
```

We can also define a generic xmap filter which takes 3 filters as parameter and iterate through an XML value. The arguments are applied to the tags, attributes and non-XML elements respectively:

Example 6.25

```
    let filter xmap ftag fatt  felem =
     <(ftag)  (fatt)>(let filter xmap_rec = $  [] -> ~{ []}
                                          | ( $  <_ ..>_ -> xmap, xmap_rec)
                                          | (felem, xmap_rec) in xmap_rec)
```

It should be noted that arguments are only allowed for toplevel filter definitions, and are *inlined* within the body of the main filter, e.g. with the filters:

```
let filter atoA =  $  ‘a -> ~{ ‘A };;
let filter id =  $  x -> ~{ x };;
let filter incr =  $  Int&x -> ~{ x+1 } |  $  x -> ~{ x };;
```

the expression:

```
apply xmap (atoA) (id) (incr) to <a>[ <a> [ 3 ‘Foo ]];;
```

is compiled as:

```
apply
 let filter xmap =
    <( $  ‘a -> ~{ ‘A } ) ( $  x -> ~{ x })>(
          let filter xmap_rec = $  [] -> ~{ []}
                              | ( $  <_ ..>_ -> xmap, xmap_rec)
                              | ( $  Int&x -> ~{ x+1 }
                                  |  $  x -> ~{ x },
                                  xmap_rec)
                                  in xmap_rec)
 in xmap
to <a>[ <a> [ 3 ‘Foo ]];;
```

This imposes that argument filters must be well-defined and therefore cannot be the filter which is being defined. The syntactic restriction that, in their definitions, arguments of filters are implicit (for example in the definition of xmap, one writes: “xmap” and not “ xmap (ftag) (fatt) (felem)” to make a recursive call), prevents from writing ill-founded recursive filters such as:

Example 6.26

```
let filter f (g) = ( g (f), f(g) )
```

Despite these restrictions, the previous examples show that macro filters, even if of purely syntactical nature, are quite useful in practice to write reusable code.

6.3.3 Regular expression filters

While sequences and XML documents are encoded via pairs in the formal algebra, providing syntactic support for such data structures clearly eases the writing of programs. This affects all the aspects of ℂDuce: values use the square brackets and triangle brackets notations, types and patterns are extended with regular expression types (and patterns) to express XML types. The same goes for filters, as it is possible to add syntactic sugar for regular expression filters:

Definition 6.27 (Regular expression filters)
The filter grammar is extended by the following productions:

$$\begin{array}{rcll} f & ::= & \ldots & \textit{(previous definitions)} \\ & | & [\ \mathtt{r}\ ] & \textit{(Regular expression filter)} \end{array}$$

$$
\begin{array}{lcll}
r & ::= & \epsilon & \textit{(Empty expression)} \\
 & | & t & \textit{(}\mathbb{C}\textit{Duce type)} \\
 & | & r\ o & \textit{(Regexp operator)} \\
 & | & r\ r & \textit{(Concatenation)} \\
 & | & r|r & \textit{(Alternation)} \\
 & & & \\
o & ::= & * \mid ? \mid *? \mid ?? & \textit{(Regexp operators)}
\end{array}
$$

The regular expression operators are, quite classically the Kleene star $*$, and the ? meaning at most one repetition. Their weak counterpart is denoted by $*$? and ?? respectively. The semantics of regular expression filters is given by mean of a rewriting function from regular expressions to filters, given in Figure 6.1. This function is inspired from the one used to rewrite ℂDuce patterns (which can be found in [Fri04b]). Its first argument is the regular expression we rewrite and is deconstructed at each call. The second and third arguments are respectively continuations, that is, filters which are called after the current regular expression filter has been applied. We need to distinguish two cases: the one where the regular expression accepts the empty sequence, in which case we use the continuation f_2 and the case where the regular expression matches at least one element, in which case f_1 is used as a continuation.

$$
\begin{array}{lcl}
\mathit{Reg}(f, f_1, f_2) & = & (f, f_1) \\
\mathit{Reg}(r_1 r_2, f_1, f_2) & = & \mathit{Reg}(r1, \mathit{Reg}(r_2, f_1, f_1), \mathit{Reg}(r_2, f_1, f_2)) \\
\mathit{Reg}(r_1|r_2, f_1, f_2) & = & \mathit{Reg}(r1, f_1, f_2) \mid \mathit{Reg}(r_2, f_1, f_2) \\
\mathit{Reg}(r*, f_1, f_2) & = & \texttt{let filter } X = \mathit{Reg}(r, X \mid f_1, \epsilon) \mid f_2 \texttt{ in } X \\
\mathit{Reg}(r*?, f_1, f_2) & = & \texttt{let filter } X = f_2 \mid \mathit{Reg}(r, f_1 \mid X, \epsilon) \texttt{ in } X \\
\mathit{Reg}(r?, f_1, f_2) & = & \mathit{Reg}(r, f_1, f_2) \mid f_2 \\
\mathit{Reg}(r??, f_1, f_2) & = & f_2 \mid \mathit{Reg}(r, f_1, f_2)
\end{array}
$$

X is a fresh filter variable.

Figure 6.1: Rewriting function *Reg* from regular expression filters to plain filters

Initially, for a given regular expression r the rewriting function is called with $\mathit{Reg}(r, \underline{\text{nilf}}, \underline{\text{nilf}})$, where nilf is the filter matching and returning the empty sequence:

```
let filter nilf =  $  [] -> ~{ [] }
```

The ϵ filter found in the encoding of the Kleene star and its weak counterpart is a "dummy" filter which is defined by:

```
let filter ϵ = $  x & Empty -> ~{ x };;
```

This filter is never taken —since its input type is `Empty`— and does not interfere with the typing of the whole filter, since its output type is `Empty`. The ϵ filter is only a "dummy" element which is used to build well-founded recursive filters and is, in practice, removed during the compilation process. Let us consider the following example to illustrate how this function works:

```
let filter f = [ (( $  x -> ~{x}) ?) * ]
```

f is a problematic regular expression since the internal regular expression: `$x->~{x}?` matches the empty sequence. The result of *Reg*(*f*, *nilf*, *nilf*) is detailed in Figure 6.2.

First step:

```
let filter f =
let filter X =
  Reg($x->~{x}?, X | nilf, ϵ)
  | nilf
in X
```

Second step:

```
let filter f =
let filter X =
  Reg($x->~{x}, X | nilf, ϵ)
  | ϵ
  | nilf
in X
```

Third step:

```
let filter f =
  let filter X =
    ( $  x -> ~{x} , X|nilf)
    | ϵ
    | nilf
  in X
```

Simplification:

```
let filter f =
    let filter X =
    ( $  x -> ~{x} , X)
    | nilf
    in X
```

Figure 6.2: The *Reg* function in action.

Our syntax for regular expression filters resembles very much Hosoya's filters [Hos04]. The notable differences are that our union operator "|" is first match while Hosoya's is non deterministic and that, thanks to use of *flat* sequences in its algebra, he obtains node deletion for free, by substituting a node with the empty sequence. This differences aside, we can express the same map style filters:

Example 6.28

```
let filter succList = [ ( $  x -> ~{x+1})* ];;
let filter xmap ftag fatt  felem = <(ftag) (fatt)>[ ( $  AnyXML ->xmap
                                                 |felem)* ];;
```

The difference between greedy and weak regular expression is quite straightforward when one looks at the encoding of Figure 6.1. A greedy regular expression ($*$ or ?) will match as much as possible while a weak operator will match as little as possible. This is reflected e.g. in the encoding of $*$, where there is an alternation between (f, X) (first choice) and $Reg(r)$ (second choice). Thanks to the first-match policy, the filter will try to apply f as much as possible on a sequence. When the application of f fails, it falls back on the second choice and applies the tail of the regular expression on the remaining value. For the weak version $*?$, the encoding is reversed. One first tries to match the tail. If it does match, then the evaluation continues from it since it would be considered as a "matching of length zero". Again, we can see that the core filter algebra allows to express higher level constructs, which are more suitable for a real language and can nevertheless be typed in a very precise way.

6.4 Type inference algorithm

The implementation of the type inference algorithm is quite straightforward. It can be written in a very natural way by transposing the rules of $\mathscr{F}_{\mathscr{A}}$. Even a naive algorithm allows us to quickly type large terms *and* input types. Indeed, in the XML framework, types are often quite large (one can for example think of the XHTML DTD or the DocBook DTD): there may be several hundreds of different sub-trees, all mutually recursive and guarded by regular expression operators.

There is however quite a difference between filters seen as regular trees and concrete filter expressions. This has a direct impact on the precision of type inference in presence of annotations. Let us once again consider the flatten filter. We can compare the regular tree (given as a recursive equations) and its concrete syntax equivalent:

Example 6.29

$$\begin{array}{rcl} \mathit{flatten} & = & \texttt{`nil} \longrightarrow \texttt{`nil} \\ & | & (\texttt{[Any*]} \longrightarrow \mathit{flatten}, \mathit{flatten});@ \\ & | & (x \longrightarrow x, \mathit{flatten}) \end{array}$$

```
let filter flatten =
   $  []  -> ~{ [] }
      | ( $  [ Any* ] -> flatten , flatten  ) ;@
      | (  $  x -> ~{ x } ,  flatten  )
```

While those filters seems syntactically equivalent, they are most certainly not, as far as annotations are concerned. Indeed if we consider the following type:

type t = [`a t `b] | [`a `b]

then we know that the most precise output type for "flatten(t)" is $\{\,\texttt{[`a}^{\texttt{n}}\texttt{`b}^{\texttt{n}}\texttt{]} \mid n \geq 1\}$. We already dealt with a similar case: the algorithm could infer (with the appropriate annotations) the output type [(`a|`b)∗] for the input type s = `nil|[`a s `b].

In the present case we would like the algorithm to infer `[(‘a|‘b)+]`. The formal algorithm shows this to be possible. It is indeed sufficient to consider the filter *flatten*, annotated this way:

$$\begin{array}{rcl} \mathit{flatten} & = & \texttt{‘nil} \longrightarrow \texttt{‘nil} \\ & | & (\texttt{[Any*]} \longrightarrow \mathit{flatten}, \mathit{flatten})_{\{(\texttt{[(‘a | ‘b)+]} \times \texttt{[(‘a | ‘b)+]})\}};@ \\ & | & (x \longrightarrow x, \mathit{flatten}) \end{array}$$

However when “converting” this filter to the concrete syntax, we are faced with the problem of adequately placing the annotation. There are two options. Either we proceed as in Example 6.18 and write:

```
apply flatten to ... where flatten={| [ (‘a|‘b)+ ] |};;
```

which *does not* type check. Indeed, the whole filter (as we annotated the whole filter for simplicity and not some sub part) must have type `[(‘a | ‘b)+]` yet it contains a branch (the first one) which returns `‘nil`. Another way of doing is to *alias* the relevant part of the filter by rewriting it like:

```
let filter flatten =
  $ [] -> ~{ [] }
    | let filter alias = ( $ [ Any * ] -> flatten , flatten )
      in ( alias ;@)
    | ( $ x -> ~{ x } , flatten )
```

and annotate the application like this:

```
apply flatten to ... where alias={| ([ (‘a|‘b)+ ],[ (‘a|‘b)+ ]) |};;
```

While this solution works, it is quite cumbersome. In the worst case, the programmer did not think that this annotation would be necessary and did not write the filter with the extra alias, or in the best case (s)he did and can annotate the filter without touching any other part of the code (especially the filter definition) but ends up with a less readable code (and therefore less maintainable code). The solution we used in the implementation is inspired by the refinement for the typing rule of the union, presented in Section 4.1.3. We associate to each filter and input type a notion of *context* which tells us whether the result of a filter is part of the whole output type or not. Its effect on annotations (which are constraints over the output type) is that if the result of an annotated filter is “intermediary” and is not part of the final output type, then it is not necessary to check whether its output type is not a subtype of the annotation or not. Said differently, the constraint of the rule **(a-annot)** has to be enforced only for filters the output type of which contributes to the global output type. With our current example, we can write:

```
let filter flatten =
  $  []  -> ~{ [] }
     | ~( $  [ Any* ] -> flatten , flatten  );@
     | ~(  $  x -> ~{ x } ,  flatten  )
;;
apply flatten  to ... where flatten={| [ (‘a|‘b)+ ] |};;
```

The output type of the first branch is not considered since the input type is:

```
type t = [‘a t ‘b] | [‘a ‘b]
```

which means that the filter can *never* return the empty sequence.

6.5 Compilation

Although the filter algebra does not provide an abstract execution model (automata, virtual machine,...), we have experimented many compilation techniques which take advantage of the fact that the core filter algebra is small and well defined and that filters are precisely typed to provide some optimizations to a rather naive implementation.

6.5.1 Compilation target

After many experiments, we chose to compile filters into OCaml closures instead of writing an interpreter for a low-level algebra of filter terms. These closures take as argument two environments (local and global represented as stacks of values), and the input value and return the output value. The big advantage with this compilation scheme is that it allows us to extend the values of ℂDuce with a new internal representation: *lazy filter application*. Indeed internally, this value which is the result of an `apply to` expression is represented by an OCaml variant `Filter`(f,v) where f is a closure and v a value. Unfreezing the value merely consists in evaluating the application $(f\ g\ l\ v)$, where g and l are the current global and local environments. This was motivated by the fact that in ℂDuce the concatenation operator `@` is lazily applied and evaluated, to achieve an amortized linear complexity while concatenating sequences, instead of the quadratic behaviour that one would have with standard concatenation. This gave us the idea of lazily evaluating filters. However, freezing/unfreezing each step of the evaluation of a filter adds a big overhead to the whole process which kills the benefits of lazy evaluation. We experimented with some heuristics to decide when to freeze a filter evaluation and the one which gave the best results was to freeze recursive calls. Indeed, consider the concat filter and its application:

```
let filter concat =
 $  (x,y)  -> ( ~{x} ; (
                 let filter aux =
                   $  []  -> ~{ y }
                |  (   $  z  -> ~{ z }, aux )
                 in aux ));;

apply concat to ([ 1 2 3 4 ],[5 6 7]);;
```

Here, recursive calls to aux will be frozen, so that the application will return a value v which is internally encoded as (OCaml code):

```
Pair(1, Filter (f,[ 2 3 4]))
```

where `Pair(  ,  )` is the internal representation for ℂDuce pairs and f the closure encoding the application of aux. It should be noted that this closure retains in its environment the variable y, bound to the value `[5 6 7]`.

6.5.2 Tail-recursive list traversal

A well known drawback of languages such as OCaml or ℂDuce is that list mapping is *not* tail-recursive. List mapping is however one of the building blocks of XML processing and an efficient evaluation of list traversal is required. An existing approach for OCaml is the ExtLib library [Ext]. This library provides tail recursive versions of standard list functions of OCaml by making use of the unsafe module `Obj`. This module allows one to temper with the low level representation of OCaml values e.g. to perform in-place modifications of non mutable values. Such a technique was already used in ℂDuce to compile the iterators `map`, `transform` and `xtransform` in a tail-recursive fashion when possible.

We reused this technique to compile filters more efficiently. Indeed, a naive evaluator for pair filters would be (given in ML pseudo-code):

```
let rec eval f g l v = match f with
 | ...
 |  FilterPair (f1,f2) -> Pair(eval f1 g l (fst v), eval f2 g l (snd v))
 | ...
```

It is clear that the recursive calls to *eval* are not in tail position, since the pair value must be reconstructed afterwards. If we assume that it is possible to make in-place modifications of pairs then we can write the previous code more efficiently as:

```
let rec eval f g l v output  set_res  = match f with
 | ...
 |  FilterPair (f1,f2) ->
   let cell  = new_pair()
   in
    set_res (output, cell);
    eval f1 g l (fst v) cell  set_fst ;
    eval f2 g l (snd v) cell  set_snd;
 | ...
```

In this code snippet, the *output* argument is an accumulator which represents the current output value of the filter and *set_res* is a function which given the accumulator and a value, sets the value at the right place within the accumulator, and does so by an in-place modification. We see now that evaluating a pair filter consists in:

1. allocating a new pair *cell*

2. setting it as part of the output

3. evaluating $f1$ on the first projection of the input, the result of which will be put as the first component of *cell*

4. evaluating $f2$ on the second projection of the input, the result of which will be put as the second component of *cell*

In this way, the second call to *eval* is tail-recursive (and then efficiently compiled by the OCaml compiler). This technique combined with the previously explained lazy evaluation gives good performances in practice.

6.5.3 Filter specialization

As we saw in Chapter 4, the precise typing of the union allows us to detect unused code in a filter. But, for some filters, some branches may be unused, depending of the input type:

Example 6.30

```
let filter convert =  $  [] -> ~{ [] }
                    | ~( $  x&Int -> ~{ "Integer:" @ ( string_of x)}
                       | $  x&Bool -> ~{ "Boolean:" @ ( string_of x)}
                       | $  x&AnyXML ->~{"XML:" @ (string_of x)},
                       convert)
```

The accepted type for this filter (which is exact as there is no composition) is `[ (Int | Bool | AnyXml)∗ ]`. However, if the type of its argument does not contain any XML element, then the third branch of the first component of the product filter will never be used. We cannot completely issue a “warning” to the user at the time of the definition because this branch might be used for some input types. However we can keep the information computed during the typing phase to discard this branch at compile time and effectively obtain a more efficient filter. Indeed, thanks to the efficient compilation of pattern matching in ℂDuce (described in [Fri04a]), the previous filter , when applied to a value of type `[ (Int | Bool)∗ ]`, is compiled as follows:

```
let filter convert =  $   [] -> ~{ [] }
                     | (  $  x&Int -> ~{ "Integer:" @ ( string_of  x)}
                          |  $  x -> ~{ "Boolean:" @ ( string_of  x)},
                     convert)
```

Not only is the third branch discarded, but the second test is simplified as (i). we know that the filter is well-typed and (ii). the current element is not of type `Int` (as the first branch failed) so it is of type `Bool` and the test is superfluous.

6.5.4 Evaluation without backtracking

The use of ℂDuce’s pattern matching framework is of great interest for filters. Indeed, as shown by Alain Frisch [Fri04a, Fri04b], ℂDuce performs pattern matching *without backtrack*. In a nutshell, the expression:

```
   match v with
 | p1 ->e1
 ⋮
 | pn ->en
```

is compiled as:

```
run_matching [ (p1,e1);...;(pn,en)] v
```

Essentially, the `run_matching` function matches the value `v` against the patterns $p_1,\ldots,p_n$ *in parallel*, without backtracking and returns a pair $(v/p_i, e_i)$, where p_i is the successful pattern.

Since a union filter, “$f_1|f_2$” behaves like the branches of a `match with` construct, we would like to evaluate it without backtracking, that is, evaluate f_2 directly if f_1 fails at some point. To achieve this, we reuse the information obtained during the typing of the filter. Indeed, after a filter has been typed with respect to some input type, every node in its internal representation is *decorated* with its accepted type. It is then possible to rewrite a filter "$f_1|f_2$" as : "$\wr f_1 \int \rightarrow f_1 | \wr f_2 \int \rightarrow f_2$" (we recall that $\wr f \int$ is the accepted type of f). The union of two filters is then compiled as:

$$\mathit{run\_matching}\ \texttt{[}\ \texttt{(}\wr f_1 \int \texttt{,} f_1\texttt{)}\texttt{;}\texttt{(}\wr f_2 \int \texttt{,} f_2\texttt{)]}\ v$$

This ensures that the evaluation of the filter f_1 does not fail, since its application is guarded by the pattern $\wr f_1 \int$.

Chapter 7

XPath encoding, approximations

We present in this chapter a language of path expressions, based on a forward fragment of XPath and ℂDuce types. We show that such paths can be encoded as a set of mutually recursive filters. Since these filters make use of the composition operator, they must be annotated. We exhibit an algorithm which computes an approximation of the output type and uses it to automatically infer the annotations. The algorithm is a variation of the algorithm presented in [BCCN06], which is detailed in Chapter 8. Finally, we show how we encode XPath predicates and tests into ℂDuce types to achieve a typed implementation of XPath into filters.

Contents

7.1 XPath-like expressions

XPath expressions are one of the standard way to select parts of an XML document (the other one being *patterns*). As they are a w3c standard, their use is widespread and well known by the average XML programmer. On the contrary, patterns, while efficient and allowing great typing precision are less known and the average programmer is often clueless when having to use a language designed around pattern-matching. What is true for patterns is even more true for filters. Indeed as we have seen in the previous examples, while filters allow one to write complex XML transformations, their code can be quite intricate and complex. Our goal is to show an encoding from XPath like expressions into filters thus reusing their typing discipline and compilation model.

7.1.1 XPath^t expression, automata

We first present a particular subset of paths based on ℂDuce types that we dub XPath^t. We show later on in Section7.3.2 that a non-trivial subset of XPath can be encoded in this formalism.

Definition 7.1 (XPath^t)
A path is a finite production from the following grammar, with entry point p:

$$\begin{array}{rcll}
p & ::= & \epsilon \mid s \mid q & \textit{(path)} \\
q & ::= & s/q \mid s & \\
s & ::= & a :: \tau & \textit{(step)} \\
a & ::= & \texttt{self} \mid \texttt{child} \mid \texttt{desc} \mid \texttt{d-o-s} & \textit{(axis)}
\end{array}$$

Here, τ ranges over ℂDuce types. The empty path —which is denoted by ϵ and is the base case for the definitions— is only a commodity. It is not part of the standard and is therefore not exposed to the programmer in the implementation. An example of such XPath^t expression is:

Example 7.2

```
child::<b>Any/d-o-s::<c>Any
```

This path, when applied to an XML document, returns the list of all elements with tags `<c>` which are below an element with tag `<b>` which must be itself the root of the document (here, the root of the document is to be understood as the root of the ℂDuce value corresponding to the document. The w3c has a different specification for the root element which we will discuss in the next paragraphs). XPath^t axes mimic those of XPath and are used to *navigate* through a value. The `self` axis "stays" on the current input value, the `child` axis applies its test to every child of the input value, the `desc` axis applies its test to every strict sub-tree of the input value and finally the `d-o-s` axis applies its test to every sub-tree of the input value including itself.

While it is simple to illustrate, on an example, what such a path does, it is not easy to directly translate the XPath specification ([XPa]) into the ℂDuce framework. Indeed, this specification relies on a different data model than the one used in ℂDuce. In particular, the specification requires that every node (or sub-tree) of an XML document has a *unique* identifier. Unfortunately, the ℂDuce data model is rather simple and two structurally equal sub-trees are indistinguishable. This results in a poor and very liberal interpretation of the XPath standard, where duplicate elements can occur in a result (while they are forbidden in the standard specification, and easily filtered out thanks to the unique node identifier). For instance, if we consider the following expression[1]:

Example 7.3

```
[ <a>[<a>[]] ]/d-o-s::<a>Any/d-o-s::<a>Any
```

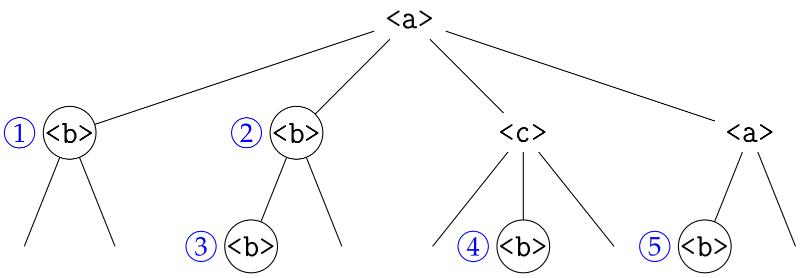
then the result is not what one would expect:

```
— : [ <a>Any* ] =[ <a>[ <a>[ ] ] <a>[ ] <a>[ ] ]
```

The problem here is that the inner element `<a>[]` has been duplicated in the result. The reason is that, in the current implementation, XPath steps are interpreted in a compositional fashion, one after another. In our example, the expression `[ <a>[<a>[]]]/desc::<a>Any` is evaluated first and returns `[ <a>[<a>[]] <a>[] ]`, which is the expected result for this step. To this intermediary result is applied the second step, `/desc::<a>Any`, which naturally returns `[ <a>[<a>[]] <a>[] ]` for the first element of the intermediary input and `[ <a>[] ]` for the second element.

We see then that, to give a proper semantics of XPath^t for the ℂDuce language, we cannot rely directly on the XPath specification. Fortunately, other semantics exists for XPath, such as the one described in [GGM$^+$04, DAF$^+$03, IHW02]. In these works, forward fragments of XPath (such as the one presented in Definition 7.1), are defined by translation into a deterministic finite *word* automaton (DFA). The language recognized by this automaton denotes exactly the set of sub-trees of the input document which match the XPath^t expression. An example of such matching is illustrated in Figure 7.1. In this figure, the node matching `desc::<a>Any/desc::<b>Any` are numbered in the order in which they must be returned (the *document order*). The sequence of the tags along the matched paths (in the tree) is clearly the regular language “.*, <a>, .*, <b>” (with informal notations, where “*” denotes the Kleene star and “.” a wild-card symbol). An interesting node is Node 5, which is matched either by “.^{0}, <a>, .^{1}, <b>”, or by “.^{1}, <a>, .^{0}, <b>”. Nevertheless, this node must be returned only *once* in the final result. To achieve both goals (document order as well as unicity of nodes in the result), we simply translate the XPath^t expression

[1] We have slightly adapted ℂDuce’s syntax here in order to ease the reading of the examples. `/d-o-s::` corresponds to `//` in the current implementation of ℂDuce.



The path `desc::<a>Any/desc::<b>Any` matches the following words:

1. `<a>,<b>`
2. `<a>,<b>`
3. `<a>,<b>,<b>`
4. `<a>,<c>,<b>`
5. `<a>,<a>,<b>`

Figure 7.1: Paths matching an XPath^t expression

`desc::<a>Any/desc::<b>Any` — seen as the regular expression “.*, `<a>`, .*, `<b>`”— into a NFA (through standard techniques, e.g. see [Wat94]). Then we determinise the NFA to obtain a DFA (again using standard techniques). Finally, we encode the DFA as a set of mutually recursive filters, which *capture* elements matching the automaton (thanks to patterns filters) and return the captured sub-trees. However, since our automata are used as an intermediary step to produce filters, hence to process ℂDuce values, we label the transitions by ℂDuce *types* instead of using symbols as it is usually the case:

Definition 7.4 (Finite Type Automaton)
A finite automaton is a 5-tuple $\mathcal{A} = (\Sigma, Q, q_0, Q_a, \delta)$ *where:*

Σ**:** *is a finite set of types*

Q**:** *is a finite set of states*

q_0**:** *is the initial state* ($q_0 \in Q$)

Q_a**:** *is the set of accept states* ($Q_a \subseteq Q$)

δ**:** $Q \times \Sigma \mapsto \mathcal{P}(Q)$ *is the transition function.*

Here, $\mathcal{P}(Q)$ denotes the power-set of Q. We also introduce the following notation:

Definition 7.5 (Out-going transitions)
let $\mathcal{A} = (\Sigma, Q, q_0, Q_a, \delta)$ be an automaton. The set of out-going transitions of a state $q \in Q$, denoted by $out(q)$ is:

$$out(q) = \{t \mid \exists Q' s.t. \delta(q, t) = Q'\}$$

Our definition of automaton is not merely syntactical. We choose to label transitions with types but still to match sequences of values. That is, given an input value, a transition is taken if the value is in the *type* labelling the transition. We formalise this new notion of run (or acceptance) of an automaton with the definition hereafter:

Definition 7.6 (Run of an automaton)
Let $s = v_0, \ldots, v_n$ be a finite sequence of values. Let $\mathcal{A} = (\Sigma, Q, q_0, Q_a, \delta)$ be a finite type automaton. The run of $\mathcal{A}$ against s is defined as follows:
We call q the current state and v the current input.

1. *initialize q with q_0 and i to 0.*
2. *set v to v_i*
3. *let $T = \{t \mid t \in out(q) \wedge v \in t\}$*
4. *let $Q' = \{q' \mid \exists t \in T \text{ s.t. } \delta(q, t) = q'\}$*
5. *if Q' is not empty, choose non-deterministically q in Q'. Set i to $i + 1$ and go to Step 2.*
6. *if $i = n$ and $q \in Q_a$, then success, else failure.*

From this definition, we see that an automaton can be non-deterministic if, in the current state, there are two transitions, labelled t_1 and t_2 where $t_1 \wedge t_2$ is not empty. We say that $\mathcal{A}$ is deterministic if for any input sequence there is exactly one possible run ($\mathcal{A}$ is a DFA) and that $\mathcal{A}$ is non-deterministic otherwise ($\mathcal{A}$ is a NFA).

The transformation of an XPath^t expression into a NFA is achieved by the function $[\![\_]\!]_{NFA}$, given in Figure 7.2. In the definition of $[\![\ \_\ ]\!]_{NFA}$, ϵ denotes the empty path. The translation function relies on an auxiliary function, $[\ \_,\ \_\ ]_{NFA}$ which takes an automaton and a step as argument and "chains" together the input automaton with the piece of automaton corresponding to the input step. The case for the `child` axis is the simplest. It is sufficient to start from the accepting state of the input automaton and make a τ transition to the newly created accepting state. Similarly, the automaton for the `desc` axis makes a τ transition to its final state. However, it can also loop on the state q_1, with the transition `AnyXml`, thus ignoring elements until the τ transition can be taken. The other two cases are subtler. For a step `self::`τ we must not add a new accept state (since we do not descend in the document) but rather "correct" the transitions arriving to the current accepting state. If a transition

$$\begin{array}{lcll}\llbracket\, \epsilon \,\rrbracket_{NFA} & = & (\varnothing, \{q_0\}, q_0, \{q_0\}, \varnothing) & q_0 \text{ is a fresh state}\\ \llbracket\, p/s \,\rrbracket_{NFA} & = & [\, \llbracket\, p \,\rrbracket_{NFA},\ s \,]_{NFA} & \end{array}$$

$[\, (\Sigma, Q, q_0, \{q_1\}, \delta),\ \texttt{child::}\tau \,]_{NFA} = (\Sigma \cup \{\tau\}, Q \cup \{q_2\}, q_0, \{q_2\}, \delta')$
where $\delta' = \delta \cup \{(q_1, \tau) \mapsto q_2\}$ and q_2 is a fresh state

$[\, (\Sigma, Q, q_0, \{q_1\}, \delta),\ \texttt{desc::}\tau \,]_{NFA} = (\Sigma \cup \{\tau, \texttt{AnyXml}\}, Q \cup \{q_2\}, q_0, \{q_2\}, \delta')$
where $\delta' = \delta \cup \{(q_1, \tau) \mapsto q_2, (q_1, \texttt{AnyXml}) \mapsto q_1\}$ and q_2 is a fresh state

$[\, (\Sigma, Q, q_0, \{q_1\}, \delta),\ \texttt{self::}\tau \,]_{NFA} = (\Sigma', Q, q_0, \{q_1\}, \delta')$
where:

$$\Sigma' = \Sigma \cup \bigcup_{((q,\sigma)\mapsto q_1)\in\delta} \{\sigma \wedge \tau\}$$

$\delta'(q, \sigma \wedge \tau) = q_1$ if $((q, \sigma) \mapsto q_1) \in \delta$
$\delta'(q, \sigma) = \delta(q, \sigma)$ else

$[\, (\Sigma, Q, q_0, \{q_1\}, \delta),\ \texttt{d-o-s::}\tau \,]_{NFA} = (\Sigma', Q \cup \{q_2\}, q_0, \{q_2\}, \delta')$
where:

$$\Sigma' = \Sigma \cup \{\tau, \texttt{AnyXml}\} \cup \bigcup_{((q,\sigma)\mapsto q_1)\in\delta} \{\sigma \wedge \tau\}$$

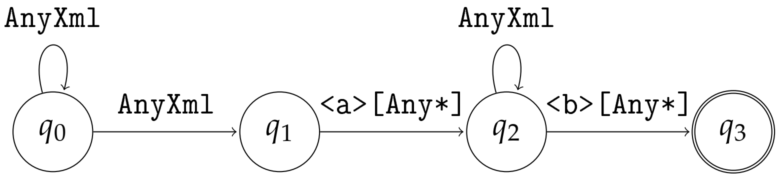
$\delta' = \delta \cup \{(q_1, \tau) \mapsto q_2, (q_1, \texttt{AnyXml}) \mapsto q_1\} \cup \bigcup_{((q,\sigma)\mapsto q_1)\in\delta} \{(q, \sigma \wedge \tau) \mapsto q_2\}$
and $q_2$is a fresh state

`AnyXml` ≡ `<AnyTag>[Any*]`
`AnyTag` is the type of all tags (*i.e.* the type of all atoms).

Figure 7.2: Translation from XPath^t to a NFA

matched a type σ while arriving on the current node, then since the current node must also match `self::`τ, the σ transition is replaced by a $\sigma \wedge \tau$ transition. Lastly, a `d-o-s::`τ is similarly to `desc::`τ. However, we must take into account the current node in the result. We proceed as for the `self` axis, adding new "intersection" transitions to the accepting state.

A close inspection of the $\llbracket\ \_\ \rrbracket_{NFA}$ function shows some peculiar features. For instance, any path starting by a `self::`τ step results in an ill-formed automaton, where the initial state q_0 is disconnected from the other states. The problem actually also arises in the XPath standard, where the first "real" element of an XML document, the document node, is a child of a fictive *root element* which does not correspond to any real element in the XML file. An XPath query starting with a `self` step yields then *always* an empty result. This corresponds to our ill-formed automaton which does not accept any input. Likewise, thanks to our use of the intersection transitions, an ill-formed XPath^t expression like `child::<a>Any/self::<b>Any` will return an automaton where the transition is labeled by the `Empty` type (since `<a>Any` `desc::*/child::a/desc::b:`



v_0 = `<c>[...<c>[...<a>[...<c>[...<b>[] ...] ...] ...] ...]`
v_1 = `<c>[...<a>[...<c>[...<b>[] ...] ...] ...]`
v_2 = `<a>[...<c>[...<b>[] ...] ...]`
v_3 = `<c>[...<b>[] ...]`
v_4 = `<b>[]`

$s = v_0, v_1, v_2, v_3, v_4$ is accepted by the NFA.

Figure 7.3: Matching of an input sequence by an NFA

and `<b>Any` are disjoint), thus rejecting also any input.

An example of valid automaton is given in Figure 7.3 together with an accepted input sequence. In this figure, v_1 is a sub-tree of v_0, v_2 is a sub-tree of v_1, and so on. We see that v_0 is accepted by the transition `AnyXml` which stays in q_0, v_1 by the transition from q_0 to q_1, v_2 by the `<a>` transition, v_3 by the loop on q_2 and finally v_4 is accepted by the `<b>` transition which leads to an accepting state. Since v_4 leads to an accepting state it must be in the result of the XPath^t expression applied to v_0.

Since our goal is to encode such an automaton into filters, we need to determinise the NFA to obtain a DFA. Instead of adapting the determinisation procedure to the general case of type NFAs as introduced in Definition 7.4, we specialize the determinisation procedure to automata obtained as output of the $[\![\_]\!]_{NFA}$ function, which simplifies the presentation quite a bit. Indeed, by construction, the only ambiguity in the NFA is when an `AnyXml` transition is introduced. Such transitions are used to treat "wild-card" $*$ or recursion (`descendant` axis). In [GGM$^+$04], the authors use a special transition, labeled `[other]` to denote "any tag that is not explicitly occuring in an outgoing transition". Thus `[other]` is not a fixed symbol of the alphabet (of the automaton) but rather a "default transition" which depends of the other transitions going out of a given state. The use of ℂDuce types and their Boolean connective allows us to explicit the value of this default branch in our algorithm. The determinisation algorithm is given in pseudo-code in Figure 7.4. Like the well-known power-set algorithm, our variation creates new states, indexed by a set of states of the input automaton. This set represent the possible positions in the NFA during one of its run. The only difference is our handling of the `AnyXml` transition.

Lines 1 to 3 consist of initialization of variables. At Line 5, the algorithm chooses from the sets of generated states one for which it has not computed the outgoing transitions yet. Initially the only state is $q_{\{q_0\}}$ indexed by the initial state of the NFA.

Input: XPath^t expression p.

Output: a DFA $(\Sigma',Q',q_{\{q_0\}},Q_a,\delta')$

```
let $q_0$ =new_state()
let $(\Sigma, Q, q_0, Q_a, \delta)$ =$[\![p, q_0]\!]_{NFA}$
let $(\Sigma', Q', Q_a, \delta)$ =$\varnothing, \{q_{\{q_0\}}\}, \varnothing, \varnothing$
do
    let $q_E$ =$choose(Q')$
    let $out\_t$ = $\bigcup_{q' \in E}$ out(q')
    let $out\_t$' = $out\_t \setminus \{$AnyXml$\}$
    $\Sigma' := \Sigma' \cup out\_t$'
    for $t \in out\_t$'
      do
      let $E'$ = $\bigcup_{Q'' \mid \exists q'' \in E \text{ s.t. } \exists t' \geq t \text{ s.t. } \delta(q'',t')=Q''} Q''$
      if $q_{E'} \notin Q'$ then
         begin
         let $q_{E'}$ = new_state();
         $Q' := Q' \cup \{q_{E'}\}$;
         if $\exists q' \in E'$ s.t. $q' \in Q_a$
         then $Q'_f := Q'_f \cup \{q_{E'}\}$;
         end;
      $\delta' := \delta' \cup \{(q_E, t) \mapsto \{q_{E'}\}\}$;
      done;
   if AnyXml $\in out\_t$ then
      begin
      let $t$ = AnyXml $\wedge \neg \bigvee_{t' \in out\_t'} t'$;
      let $E'$ = $\bigcup_{Q'' \mid \exists q'' \in E \text{ s.t. } \exists t' \geq t \text{ s.t. } \delta(q'',t')=Q''} Q''$
      if $q_{E'} \notin Q'$
      then
         begin
         let $q_{E'}$ = new_state();
         $Q' := Q' \cup \{q_{E'}\}$;
         if $\exists q' \in E'$ s.t. $q' \in Q_a$
         then $Q'_f := Q'_f \cup \{q_{E'}\}$;
         $\Sigma' := \Sigma' \cup \{t\}$;
         end;
      $\delta' := \delta' \cup \{(q_E, t) \mapsto \{q_{E'}\}\}$;
      end;
while $\delta'$changes
```

Figure 7.4: Determinisation procedure

Steps 6 to 11 compute the set of accessible states from the current state q_E. This requires a little bit more work than in the standard case. First (Step 6) it is necessary to obtain the set of types which labels the out-going transitions. If one (or more) transitions are labeled by `AnyXml`, then this case must be treated separately (Step 21) since it represents "all the transitions that are not explicitly taken from the current state". Step 8 only adds the possible transition labels to the alphabet of the DFA we are computing. Then, the algorithm differs slightly from the standard power-set algorithm. For every possible type in a transition, we must find the reachable states. This is done at Step 11, where E' regroups every state connected by a transition labeled by a *super type* of the current type. For instance, consider that the current state in the DFA is $q_{\{q_0\}}$. Imagine that, in the NFA, there are four transitions going out of q_0, two labeled by `<a>[Any*]`, going respectively in q_1 and q_2 and two others, labeled by `AnyXml`, going in q_3 and q_4. Then the accessible states in the NFA from q_0, if the input value is in `<a>[Any*]`, are q_1, q_2, q_3 and q_4. Thus, these are grouped, in the DFA in a single state $q_{\{q_1,q_2,q_3,q_4\}}$ reachable from $q_{\{q_0\}}$ by a transition labeled by `<a>[Any*]`. Line 12 to 19 only updates the DFA we are constructing. Step 21 repeats the same operation for the case where `AnyXml` was in the out-going transitions. Going on with our example, if the only transitions from q_0 in the NFA were to q_1 and q_2 with `<a>[Any*]` and to q_3, q_4 with `AnyXml`, then, in the DFA, there are two transitions: one from $q_{\{q_0\}}$ to $q_{\{q_1,q_2,q_3,q_4\}}$ labeled by `<a>[Any*]` and one to $q_{\{q_3,q_4\}}$ labeled by `AnyXml` $\wedge$ $\neg$`<a>[Any*]`. Like the standard algorithm, ours stops when we saturate the set of transitions, δ', of the DFA. Since the number of new states is finite (in the worst case it is the number of partition of Q') the algorithm clearly terminates. The output of the algorithm, applied to the input path `desc::a/child::b` is illustrated in Figure 7.5. Note that, for all the states of the DFA, the types on the outgoing transitions are pair-wise disjoint (thus ensuring determinism), and that the union of the types labelling the outgoing transitions of the DFA is equal to the union of the types labelling the outgoing transitions of the NFA.

7.1.2 Filter encoding

Once the XPath^t expression is converted into a DFA, the hard part of the work is done (typing aside). Indeed, the rewriting procedure to transform the DFA into a set of mutually recursive filters is purely syntactical. To illustrate how the filter works, let us consider again the example of Figure 7.5. There are three states in the DFA, each one marking "how much" of the desired input has been recognized. In state q_o, nothing has been recognised. In state $\{q_0, q_1\}$, we have found an `<a>` somewhere in depth. In state $\{q_0, q_2\}$, we have found a `<b>` directly below an `<a>`, thus the state is an accepting one. If below this `<b>` there is an `<a>`, then we are not done yet and we go back to state $\{q_0, q_1\}$ since there might be again an `<b>` directly below the new `<a>`, and so on.

The semantics we wish to achieve is the following: given an XPath^t p and an input sequence `[v`$_0$`...v`$_n$`]` we want to return the list $p(v_0)@\dots@p(v_n)$, where "@" denotes the concatenation and $p(v_i)$ informally denotes the application of the XPath^t expression p to the tree v_i. What we simply need to do is:

Input: `desc::a/child::b`
“Syntactic” NFA:

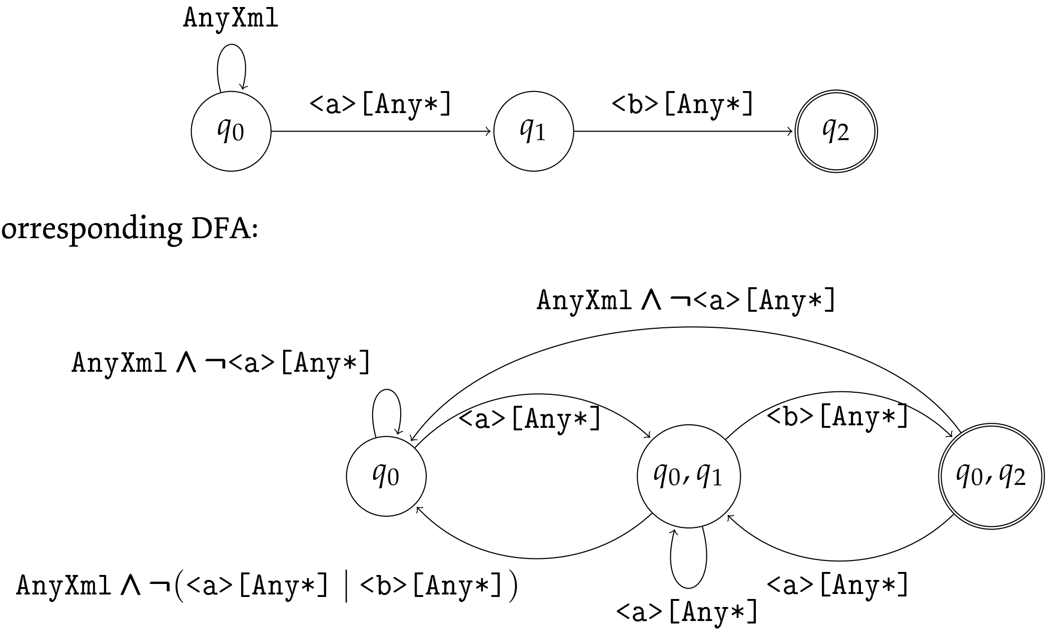


Figure 7.5: Transformation of an XPath^t expression into a NFA and determinisation

1. Iterate the starting state of the DFA corresponding to p on each element of the sequence v_0, v_1,...
2. depending on the tag of the input value, one transition exactly (since we are in a DFA) applies. It returns a new state q_1.
3. Iterate q_1 on every child element of v_0
4. and so on...

Furthermore, if at Step 2 the transition we take leads to an accepting state, then we have to remember the current input as it is part of the result. Before giving the encoding, let us recall the semantics of some useful filters:

```
let filter mapconcat f = $  [] -> ~{ [] }
                       | ( f, mapconcat) ;concat;;
```

The mapconcat filter expects a list as input and apply its filter argument to every element of this list and concatenates all the results. For instance:

```
let filter id =  $  x -> ~{ x };;
apply mapconcat id to [ [ 1 ] [ 2 ] [ [ 3 ] 4 ] ];;

val — : [ 1 2 [ 3 ] 4 ] =[ 1 2 [ 3 ] 4 ]
```

Here, since the input type is not recursive, the filter mapconcat does not require any annotations; even though in the general, annotations are mandatory for this filter.

We also recall the behaviour of the filter content which was presented in Section 6.3.1.

```
let filter id =  $  x -> ~{ x };;
let filter content f = <(id) (id)>f;  $  <_ >x -> ~{ x };;
```

when applied to an XML element, content f applies f to the sequence of children of that element and discards the tag. We also recall that in the extended version of the filter calculus, this filter can be typed without annotations despite the presence of a composition.

The algorithm for translating a DFA is given in Figure 7.6. It is easy to see that the resulting filters are well-formed. Indeed, since the input are the transitions of a DFA, all the generated branches for a given filter are independent (except for the default case, added as the last branch at Line 16). Secondly, every recursive call to an f_i occurs on the left-hand side of a composition operator (we recall that composition operators are hidden in the definition of mapconcat and content). The result of the algorithm run on the DFA corresponding to the XPath^t expression `desc::a/child::b` is:

Example 7.7

```
let filter fo =
          |  $  <a>[Any*] ->content (mapconcat fo1)
          |  $  AnyXml \<a>[Any*] ->content (mapconcat fo)
          |  $  _ -> []

and filter fo1 =
      |  $  <a>[Any*] ->content (mapconcat fo1)
      |  $  x&<b>[Any*] ->content (mapconcat fo1) ;y ->~{ (x,y) }
      |  $  AnyXml \(<a>[Any*] | <b>[Any*]) ->content (mapconcat fo)
      | _ -> []

and filter fo2 =
      |  $  <a>[Any*] ->content (mapconcat fo1)
      |  $  AnyXml \<a>[Any*] ->content (mapconcat fo)
```

```
     | $ _ -> []
;;
let filter fxpath =mapconcat (content fo)
```

Let us check on an example that the filter fxpath performs the desired computation:

```
let v0 =<a>[ <b>[1] ]
let v1 =<c>[<a>[ <c>[ <b>[2] ] ] <c>[] <d>[<a>[<b>[3] <d>[] ] ]
              <a>[<b>[ 4 <a>[<b>[5] ]]]
   ]
let input = [ v0 v1 ]
```

The input is composed of two documents, v_0 and v_1. v_0 is a simple case and should the `<b>[1]` sub-tree should appear in the result. For v_1, `<b>[2]` should *not* appear since it is not directly the child of an `<a>` node. `<b>[3]`, `<b>[4...]` and `<b>[5]` should appear in this order in the result. Let us check:

Input: a DFA$\mathcal{A} = (\Sigma, Q, q_0, Q_a, \delta)$

Output: the filter `fx`

```
for $q_i \in Q'$
 do
     create a new union filter $f_i$
     for every transition $\delta(q_i, t) \mapsto q_j \in \delta$
     do
     if $q_j \in Q_a$
       then
       create a branch
           "|␣ $ ␣x&t␣->␣(content␣(mapconcat␣$f_j$) ;y-> ~{(x,y)})"
       else
        create a branch
          "|␣ $ ␣t␣->␣content␣(mapconcat␣$f_j$)"
     done
 create a branch
     "|␣ $ ␣_␣->␣~{␣[]␣}"
 done
create a new filter "fx␣=␣(content␣(mapconcat␣$f_0$)"
```

Figure 7.6: Translation of a DFA into a filter

```
apply fxpath to input ...;;
val — : ... =[ <b>[ 1 ]
              <b>[ 3 ]
              <b>[ 4 <a>[ <b>[ 5 ] ] ]
              <b>[ 5 ] ]
```

Here we see that the document order is respected. The ellipsis hide (for now) the typing parts. Indeed, since the filter makes an heavy use of composition, annotations are required to type it. Before giving our typing discipline for the XPath fragment we have defined, let us show another example. Here we consider the filter corresponding to the XPath^t expression `d-o-s::<a>Any/d-o-s::<a>Any`:

Example 7.8

```
let filter fo =
         | $  x&<a>[Any*] ->content (mapconcat fo12);y ->~{ (x,y) }
         | $  AnyXml \<a>[Any*] ->content (mapconcat fo)
         | $  _ -> []

and filter fo12 =
     | $  x&<a>[Any*] ->content (mapconcat fo12);y ->~{ (x,y) }
     | $  AnyXml \<a>[Any*] -> content (mapconcat fo1)
     | _ -> []

and filter fo1 =
     | $  x&<a>[Any*] ->content (mapconcat fo12);y ->~{ (x,y) }
     | $  AnyXml \<a>[Any*] ->content (mapconcat fo1)
     | $  _ -> []
;;
let filter fxpath =mapconcat (content fo)
;;
(*  XPath expression in CDuce, second <a> node is duplicated *)
let buggy = [ <a>[ 1 <a>[ 2 ] ] ] //<a>Any //<a>Any ;;

# val buggy : [ <a>Any* ] =[ <a>[ 1 <a>[ 2 ] ] <a>[ 2 ] <a>[ 2 ] ]

(* The filter corresponding  to the  DFA *)
let ok = apply fxpath to [ <a>[ 1 <a>[ 2 ] ] ] ...;;

# val ok : ... =[ <a>[ 1 <a>[ 2 ] ] <a>[ 2 ] ];;
```

Since the traversal of the value is done in one pass for the whole filter, we see that a sub-tree is kept in the result only if it matches the path expression and that, afterwards, the automaton is iterated on its strict-subtree and siblings. This prevent any duplication of an input tree in the output.

7.2 Type annotations

The encoding of XPath^t into filters, while satisfactory from a dynamic semantics perspective seems less appealing from a typing perspective. If one considers the code for the filters (mapconcat f_i), it is quite clear that such a filters needs to be annotated. Indeed, they are very similar to—and are in fact an "inlined" version of—the flatten filter, root of all evil in this world (of typing). The situation seems even more problematic because, since the programmer writes an XPath^t expression which is then compiled into a set of filters, (s)he does not have any knowledge of the number (let alone the names) of the filter that are introduced by the transformation.

The situation is however not desperate. A careful analysis of the generated filters shows that all the generated filters must be annotated by the *same* type: the final output type. As we have seen, the most precise instance of this output type might not be regular. To see why this is true in the case of XPath^t expressions, it is sufficient to consider the following input type and XPath^t expression:

Example 7.9

```
type in_t = <c>[ <a>[] in_t <b>[] ] | <c>[]

d-o-s::(<a>[Any*]|<b>[Any*])
```

For this particular type, the XPath^t expression is nothing but a flatten filter, where the most precise output type is $\{\texttt{[<a>[]}^{n}\texttt{<b>[]}^{n}\texttt{]} \mid n \geq 0\}$.

A possible solution is to reuse the type information occurring in the last step of the XPath^t expression. That is, for a path p`/axis` $::$ τ (where `axis` denotes any of the four axes), annotate the filters with the output type `[` τ`*` `]`. Such solution is however barely satisfactory. Indeed, if one writes a path such as: `d-o-s::<a>Any/self::AnyXml`, we know at least, just by looking at the path, that the result will be both in `[ <a>Any* ]` and in `[AnyXml* ]`, that is in `[ <a>[Any*]*]`. To achieve a greater precision, we can, as for filters, take the input type into account. The idea is again to evaluate the XPath^t expression on the input type. Since we know the precise semantics of the XPath expression, we can tailor an algorithm so that computes an approximation of the output type. Contrary to filters, where the composition can be used to define arbitrary transformations and hence requires annotations, we know the extent of all the operation that must be done on the input type and can thus tailor the typing rule for theses specific rules. In particular we have the following pieces of information:

- the output type is a subtype of `[Any*]`
- the recursive calls are made to iterate through XML elements only

We give in Figure 7.7 a typing algorithm for XPath^t expressions. This algorithm is given as a set of inference rules, deriving judgment of the form $\Vdash_x p(T) = S$. Contrarily to the type inference algorithm of Chapter 5 this algorithm works on sets of types rather than types directly. Here, $\Vdash_x p(T) = S$ means that if a path is applied to a value whose type is in T, then the type of its result is in S. Informally, we abstractly execute the path p on every elements of T and collect the resulting types in S. Suppose now that $\tau \leq$ `[Any*]` is the type of the sequence of elements we wish to apply the path p onto. If $\Vdash_x p(\{\tau\}) = S$, then we choose for output type: $[(\bigvee_{\sigma\in S} \sigma)*]$.

This judgment requires an auxiliary judgment, $\Delta \vdash_x s(T) = S$ where s is a single step, T and S sets of types and where Δ a memoization environment, that is, a set of pairs (s, τ) where s is a step and τ a type. In the presentation, `<_>`τ stands for an XML type with some label, which we do not care about and `axis` represent any of the four XPath^t axes. Additionally, for the **(s-*)** rules, a memoization rule must be applied whenever possible and must be followed by a non-memoization rule (as for the type inference algorithm of Chapter 5). The algorithm for paths is quite simple: it applies step by step the path on the set of input types. The **(s-*)** rules are where the real work takes place. The **(s-*)** rules are divided in four categories. The first three—step, iteration and set—are structural rules. Step rules handle the application of a step on a singleton[2]. The **(s-self)** rule restricts the input by using an intersection with the "test type" of the step. The **(s-child)** checks that the input type is compatible with an XML element, extract the type τ'' corresponding to the sequence of child and apply a `self` step on the types of the children. **(s-d-o-s)** returns the output types for both the current input and **(s-d-o-s)** recursively applied to the types of the children. Finally, **(s-desc)** is equivalent to applying **(s-d-o-s)** on the children directly.

Iteration rules apply the step rules on every elements of a sequence type. Indeed, the encoding of sequences in ℂDuce is made explicit in this algorithm. Since sequences are encoded as nested pairs, we use the product decomposition $\boldsymbol{\pi}$ (defined in Chapter 2, Section 2.26). As an example, consider an input set $\{\tau\}$, where $\tau \equiv$ `[ (A|B) * C ]` and `A`, `B` and `C` are three disjoint types (for the sake of simplicity). More formally, τ is defined by:

$$\tau = ((\mathtt{A} \mid \mathtt{B}) \times \tau) \mid (\mathtt{C} \times \mathtt{`nil})$$

What we want is that the algorithm apply the current step on `A`, `B` and `C`. Like for the typing of product filters, this depends of the semantics of $\boldsymbol{\pi}$. Let us assume here that $\boldsymbol{\pi}(\tau) = \{(\mathtt{A} \times \tau), (\mathtt{B} \times \tau), (\mathtt{C} \times \mathtt{`nil})\}$, then in rule **(s-cons)** the step $s$ is applied to `A`, `B`, `C`, τ and `` `nil ``, which is the expected behaviour. It is also possible that $\boldsymbol{\pi}(\tau) = \{((\mathtt{A} \mid \mathtt{B}) \times \tau), (\mathtt{C} \times \mathtt{`nil})\}$ in which case, s would be applied to $\mathtt{A} \mid \mathtt{B}$. It is worth noticing that in the ℂDuce implementation, $\boldsymbol{\pi}$ returns a set of products whose first components are pairwise disjoint.

[2] Note that here, a singleton is a set containing only one type, an *not* a singleton type.

Step rules

$$\textbf{(s-self)}\ \frac{}{\Delta \vdash_x (\texttt{self::}\tau)(\{\tau'\}) = \{\tau \wedge \tau'\}}$$

$$\textbf{(s-child)}\ \frac{\texttt{<\_>}\tau'' = \tau' \wedge \texttt{AnyXml} \qquad \Delta \vdash_x (\texttt{self::}\tau)(\{\tau''\}) = S}{\Delta \vdash_x (\texttt{child::}\tau)(\{\tau'\}) = S}$$

$$\textbf{(s-desc)}\ \frac{\texttt{<\_>}\tau'' = \tau' \wedge \texttt{AnyXml} \qquad \Delta \vdash_x (\texttt{d-o-s::}\tau)(\{\tau''\}) = S}{\Delta \vdash_x (\texttt{desc::}\tau)(\{\tau'\}) = S}$$

$$\textbf{(s-d-o-s)}\ \frac{\begin{array}{r} \Delta \vdash_x (\texttt{self::}\tau)(\{\tau'\}) = S_1 \\ \texttt{<\_>}\tau'' = \tau' \wedge \texttt{AnyXml} \qquad \Delta \vdash_x (\texttt{d-o-s::}\tau)(\{\tau''\}) = S_2 \end{array}}{\Delta \vdash_x (\texttt{d-o-s::}\tau)(\{\tau'\}) = S_1 \cup S_2}$$

Iteration rules

$$\textbf{(s-nil)}\ \frac{}{\Delta \vdash_x (s)(\texttt{[]}) = \varnothing} \qquad \textbf{(s-cons)}\ \frac{\begin{array}{l} \boldsymbol{\pi}(\tau') \equiv \{(\tau_1^1 \times \tau_2^1), \ldots, (\tau_1^n \times \tau_2^n)\} \\ j \in 1..2 \\ \Delta \vdash_x (s)(\{\tau_j^i\}) = S_j^i \end{array}}{\Delta \vdash_x (s)(\{\tau'\}) = \bigcup_i S_1^i \cup S_2^i}$$

Set rules

$$\textbf{(s-set1)}\ \frac{}{\Delta \vdash_x (s)(\varnothing) = \varnothing} \qquad \textbf{(s-set2)}\ \frac{n > 1, i \in 1..n\ \Delta \vdash_x (s)(\{\tau_i\}) = S_i}{\Delta \vdash_x (s)(\{\tau_1, \ldots, \tau_n\}) = \bigcup_i S_i}$$

Memoization rules

$$\textbf{(s-mem1)}\ \frac{(s, \tau') \in \Delta}{\Delta \vdash_x (s)(\{\tau'\}) = \varnothing}$$

$$\textbf{(s-mem2)}\ \frac{(s, \tau') \notin \Delta \qquad (s, \tau') \cup \Delta \vdash_x (s)(\{\tau'\}) = S}{\Delta \vdash_x (s)(\{\tau'\}) = S}$$

Path rules

$$\textbf{(p-empty)}\ \frac{}{\Vdash_x (\epsilon)(T) = \varnothing} \qquad \textbf{(p-step)}\ \frac{\varnothing \vdash_x (s)(T) = S \qquad \Vdash_x (p)(S) = S'}{\Vdash_x (s/p)(T) = S'}$$

Figure 7.7: Type inference algorithm for XPath^t

Set rules are used when the input set contains more than one type. Lastly, memoization rules are used to record an already encountered type and stop the recursion (occuring for a **d-o-s** step).

What the step algorithm does, is to collect every sub-tree of a type in the input set which matches the step. This is clearly seen in the **(s-d-o-s)** and **(s-cons)** Where the result is the *union* of what is returned in the premises of the rule. The set of **(s-*)** rules defines an algorithm, since the termination is ensured by the memoization environment Δ.

The set of **(p-*)** rules also defines an algorithm (the derivation clearly terminates by induction on the length of the path).

Let us illustrate how the algorithm performs on a few examples.

Example 7.10

```
type in_t = <c>[ <a>[] in_t <b>[] ] | <c>[]

d-o-s::(<a>[Any*]|<b>[Any*])

let v:in_t = <c>[ <a>[] <c>[] <b>[] ]
#val v : int_t = <c>[ <a>[] <c>[] <b>[] ]

apply d-o-s::(<a>[Any*]|<b>[Any*]) to [ v ];;
#val — : [ (<a>[]|<b>[])* ] =[ <a>[] <b>[] ]
```

We see here that the result is more precise than by simply using the path to annotate the filter, which would yield the output type: `[(<a>[Any*]|<b>[Any*])*]`. A more interesting example is the following.

Example 7.11

```
type a_int =  <a>[ Int * ]
type a_bool = <a>[ Bool * ]
type doc = <c>[ <b>[ a_int * ] *  a_bool * ]

let v:doc = <c>[ <b>[ <a>[ 1 2 3 ] ] <a>[ ‘false ] ]
#val v : doc = <c>[ <b>[ <a> [ 1 2 3 ] ] <a>[ ‘false ] ]

apply d-o-s::AnyXml/child::<b>Any/child::<a>Any to [ v ];;
#val — : [ <a>[ Int * ] * ] =[ <a>[ 1 2 3 ] ]
```

Here the algorithm correctly deduced that the only <a> sub-trees that could occur in the result where those below a <b>, hence only nodes of type a_int and not a_bool. Let us details the algorithm step by step[3]:

1. d-o-s::AnyXml is applied to v, of type doc≡ <c>[<b>[a_int*]* a_bool*]. This returns a set of types: $S = \{$doc,<b>[a_int*],a_int,Int,a_bool,Bool$\}$

2. On this intermediary type is applied child::<b>Any is then applied to S. First of all, only the XML part of the type is kept (since the child axis must descend into an input element). This correspond to having the step self::<b>Any applied to $S' = \{doc,$<b>[a_int*],a_int,a_bool$\}$.

3. This returns the intermediary set $S'' = \{$<b>[a_int*]$\}$ onto which is applied child::<a>Any, which returns $\{$a_int$\}$, hence the final result: [a_int*].

7.3 XPath and XPath^t

We present some encoding of standard XPath expression into XPath^t. As we will show, many conditions can be encoded into ℂDuce types and thus checked *statically*.

7.3.1 Basic features

Aside from renaming descendant-or-self and descendant into d-o-s and desc respectively, one can easily encode XPath tests into ℂDuce types:

- axis::a where a is a tag name is encoded as axis::<a>Any
- axis::* where the special character * matches any element is encoded as axis::AnyXml
- axis::text(), which matches raw text nodes is encoded as: axis::Char

The attribute axis can be also added as a basic axis of XPath^t, since its treatment is roughly similar to the one of the child axis.

7.3.2 Predicates

A more complex and more wanted feature are XPath *predicates*. Predicates are Boolean conditions that are used to restrict the set of selected nodes. For instance, the XPath expression child::a[child::text() = "foo" or child::b] selects only the child of the current node, whose tag is <a> and for which, either the content is the character string "foo", or one of the children is a <b>. We define a notion of predicates, which we call structural predicates, and which correspond to a subset of *non-nested* XPath predicates:

[3] Pun intended...

Definition 7.12
A structural predicate is a finite production of the following grammar, with entry point c:

$$
\begin{array}{rcll}
c & ::= & c\ \texttt{and}\ c \mid c\ \texttt{or}\ c \mid \texttt{not}\ c & \textit{(Boolean operations)} \\
& \mid & s & \textit{(XPath}^t\textit{ step)} \\
& \mid & \texttt{child::text()} = \textit{STRING} \mid \texttt{desc::text()} = \textit{STRING} & \textit{(string search)}
\end{array}
$$

The step condition is evaluated to true if the step applied to the current node does not yield an empty result. The "child" string test is evaluated to true if the string – obtained by concatenating all the child of the current node which have type Char – is the same as the constant *STRING* we test against. Finally, the "desc" string test if the *string-value* of the current node is equal to some string. The string value, as defined by [XPa] is the concatenation of all the raw text node occuring inside a given sub-tree.

Our goal here is to show that such predicates can be encoded into ℂDuce *types*. An XPath^t step $\texttt{axis::}\tau\texttt{[}c\texttt{]}$ can then simply be rewritten into $\texttt{axis::}(\tau \wedge \tau_c)$ where the type τ_c is the encoding of the predicate c. Before defining the translation from predicates to types, we define some useful ℂDuce types.

Definition 7.13 (Many occurrences)

$$\mathit{Many}(\tau) \equiv \mu X.(\tau \mid \texttt{<AnyTag>[Any*}\quad X\quad \texttt{Any*]})$$

It is clear that if $v \in \mathit{Many}(\tau)$, then at least one sub-tree of v has type τ (by induction on the value v).

We can also easily define exactly one and exactly zero occurrence for types that are not XML (that is for basic types such as `Int`, `Char`,...which are ranged over by B). We use this feature to encode string matching.

Definition 7.14 (Zero occurrence)
Let τ be a type such that $\tau \wedge \texttt{AnyXml} = \texttt{Empty}$:

$$\mathit{Zero}(\tau) \equiv \texttt{Any} \setminus \mathit{Many}(\tau)$$

Definition 7.15 (One occurrence)
Let τ be a type such that $\tau \wedge \texttt{AnyXml} = \texttt{Empty}$:

$$\mathit{One}(\tau) \equiv \mu X.(\tau \mid \texttt{<AnyTag>[}\mathit{Zero}(\tau)\texttt{*}\ X\ \mathit{Zero}(\tau)\texttt{*]})$$

Again, let v be a value of type $One(\tau)$. Then it is clear that exactly one sub-tree of v has type τ (by induction on v).

The conversion from predicates to types is given by a function $[\_]_T$ from predicates to types given in Figure 7.8. We can now express $XPath$ expressions such as:

$$\begin{array}{lcl}
[c_1 \texttt{ and } c_2]_T & = & [c_1]_T \wedge [c_2]_T \\
[c_1 \texttt{ or } c_2]_T & = & [c_1]_T \mid [c_2]_T \\
[\texttt{not } c]_T & = & \neg[c]_T \\
[\texttt{self::}\tau]_T & = & \tau \\
[\texttt{child::}\tau]_T & = & \texttt{<(AnyTag)>[Any* } \tau \texttt{ Any*]} \\
[\texttt{desc::}\tau]_T & = & \texttt{<(AnyTag)>[Any* } \mathit{Many}(\tau) \texttt{ Any*]} \\
[\texttt{d-o-s::}\tau]_T & = & \mathit{Many}(\tau) \\
[\texttt{child::text() = "}c_0 \ldots c_n\texttt{"}]_T & = & \texttt{<(AnyTag)>[}\mathit{NC}\ '\mathrm{c}'_0\ \mathit{NC}\ \ldots\ \mathit{NC}\ '\mathrm{c}'_n\ \mathit{NC}\texttt{]} \\
[\texttt{desc::text() = "}c_0 \ldots c_n\texttt{"}]_T & = &
\end{array}$$

$$\texttt{<(AnyTag)>[}\mathit{NCN}\ \mathit{One}('\mathrm{c}'_0)\ \ldots \mathit{NCN}\ \mathit{One}('\mathrm{c}'_n)\ \mathit{NCN}\texttt{]}$$

where:
$\mathit{NC} = \texttt{Any} \setminus \texttt{Char}$
$\mathit{NCN} = \mathit{Zero}(\texttt{Char})$

Figure 7.8: Translation from predicates to types

Example 7.16

```
d-o-s::a[child::b]/d-o-s::*[ child::text() = "foo" or (not child::d)]
```

We spare the reader's sanity by *not* giving the corresponding ℂDuce type for each step.

Chapter 8

Type-based XML projection

We introduce in this chapter a practical use of the *abstract evaluation*, that is, the evaluation of a term on a type instead of a value. Our motivation is to optimize query engines based on the DOM data-model which is known to be very expensive in memory. Indeed, for such engines, even if the query does not access the whole document, the latter has to be entirely loaded in memory. Our approach is to perform a static analysis on the query and the *type* of the document to determine which parts of it will be needed at run-time.

The work presented in the present chapter is based on an original idea by Dario Colazzo. For the sake of comprehension, we give here the collaborative results published in [BCCN06], and highlight our personal contribution namely the design of the type inference algorithm, the formal proofs of the theorems in this chapter as well as the implementation and experimental results.

Contents

8.1 Document pruning

As we developed in the introduction, the typing discipline for filters, that is, the evaluation of a filter on a type rather than on a value, was designed so as to be independent from the type algebra or particular data-model used for the host language. However, the language we designed was still somewhat related to ℂDuce/XDuce. Indeed, the use of semantic subtyping and a pattern algebra allowed us to design a very precisely typed language. As for the implementation, it was even more dependent on the ℂDuce compiler and the particular data-model and type algebra it implements. To see whether the technique of abstract evaluation scales to other data-models and type algebra, we propose the following study, inspired by the type inference algorithm for XPath^{t} in Chapter 7. Our aim in this chapter is to present a very practical use of typing, namely its use to optimize the loading of an XML document in memory.

Languages, or more specifically *query engines*, for XML can be divided in three categories with respect to memory usage. The first category is the one of *streaming* engines, for which the transformation is evaluated *on the fly*, with bounded memory, performing the outputs as soon as they are computed. Unfortunately, not every transformation can be performed in streaming. For instance, the reversal of the list of children of the root of a document needs to buffer all these elements up to the last to start the output. The second category, is the one of *main-memory* query engines. In these engines, the whole input document is loaded in memory and organised following a *data-model* such as DOM ([DOM04]). While providing an easy access to any part of the document, such models are known to be expensive in memory due to the presence of various *meta-data*. The situation is such that even for relatively small documents, of hundreds of megabytes, the main-memory representation takes several gigabytes, thus making such engines unusable on standard desktop setups or small servers. A more recent and maybe more realistic use case is the one of the "interactive web". In this setup web sites are not static anymore but the client (the web browser) continuously exchanges data with the server. Such data exchanges are often encoded in XML (for instance by using the AJAX framework). In the current situation, client-side programming is performed with Java-Script, in which the use of the DOM data-model is pervasive. In this context, the bottleneck for XML processing is not the memory but the network bandwidth, which is much more constrained. Finally, the last category is the one relying on a persistent storage of the XML document, pretty much like a DBMS. While bigger documents can be processed, the penalty of disk access with respect to memory access is a huge price to pay. Another remark is that persistent query engines can be much more difficult to set-up and fine-tune. As remarked by Amélie Marian and Jérôme Siméon in [MS03], the whole document is often not needed to perform a transformation. Following this idea, they proposed XML projections, a method where they evaluate the query (given in XQuery, [XQu]) *while* loading the document, hence discarding part of it *before* constructing the memory representation. Of course, as the document is not complete, the exact result cannot be computed, but this "loading time evaluation" provides them enough information to choose which nodes of the input to keep in memory and which to discard. This results in a smaller version of the in-memory document, onto which they can execute the query/transformation to obtain the expected result. While the reduced memory occupation of the projected document is certainly appealing, their method suffers from a major drawback: the evaluation of the query at loading time might require (even if temporary) a significant amount of memory especially in the presence of a `descendant-or-self` axis. Another drawback is that they do not handle *backward* axes, namely the `parent` or `ancestor` axes of the XPath specification [XPa].

While retaining their approach of document pruning, we extend their technique by taking types into account. In the same way as we previously typed *filters* by evaluating them on an input type, we evaluate queries on the *type of the input document*. This application allows us to compute a *type projector*, which can then be used at runtime to perform the pruning process. The advantage is that given a type projector and a document, the pruning can be performed *in streaming*, and be interleaved with the loading/validation process, thus avoiding any memory or time penalty at runtime. Our approach also takes all XPath axes into account, particularly the `ancestor` and `parent` one.

Our algorithm works as follows: given an input XQuery query, we perform a *path extraction*, that is, we collect every XPath expression that is used in the query to navigate in the input document. Given this set of XPath expressions, we apply it on the input type of the document, given by a DTD, thus computing a type projector. The latter can then be used to prune the documents that are fed to the query engine.

We introduce our notion of data-model, XPath expressions and DTD before formally giving the algorithms. The type projector inference algorithm is shown to be sound—a query applied to a projected document always gives the same result as on the original document—and also complete for a specific class of DTDs and queries—if we try to remove more nodes than those specified by the type projector, then we change the semantics of the query.

For the case where the input document is not in this specific class of DTD, we show through our implementation that it remains very precise in practice.

8.2 Notations

8.2.1 Data Model

As for XPath$^{\ell}$ in Chapter 7, we exclude the treatment of attributes, for which our approach can straightforwardly be extended, as we did in our implementation. Indeed, from a theoretical point of view, the treatment of attributes is very similar to the one of the `child` axis.

Definition 8.1 (XQuery data-model)
A value is a finite production of the following grammar, with entry-point f:

$$\begin{array}{rcll} f & ::= & () \mid f,f \mid t & \textit{(forest)} \\ t & ::= & s_i \mid \texttt{<}l_i\texttt{>[}\ f\ \texttt{]} & \textit{(tree)} \end{array}$$

Essentially, a value is an ordered sequence of labelled ordered *trees* (ranged over by t), that is an ordered *forest* (ranged over by f), where each node has a unique *identifier* (ranged over by i) and where `()` denotes the empty forest. Tree nodes are labelled by *element tags* (ranged over by l) while, without loss of generality, we consider only leaves that are text nodes (that is, strings, ranged over by s) or empty trees (that is, elements that label the empty forest). The main difference with the data-model used in the previous chapters is the presence of a unique identifier for strings and trees.

We define a complete partial order $\preceq$ on forests (and thus on trees) by relating a forest with the forests obtained either by adding or by deleting sub-forests:

Definition 8.2 (Projection ($\preceq$))
Given two forests f and f' we say that f' is a projection *of f, noted as $f' \preceq f$, if f' is obtained by replacing some sub-forests of f by the empty forest.*

Definition 8.3 (Good formation)
A forest is well formed *if every identifier i occurs in it at most once. Given a well-formed forest f and an identifier i occurring in it, we denote by $f@i$ the unique sub-tree t of f such that $t = s_i$ or $t = \texttt{<}l_i\texttt{>[}\ f'\ \texttt{]}$. The set of identifiers of a forest f is then defined as* $\mathbf{Ids}(f) = \{i \mid \exists\, t.\ f@i = t\}$

Henceforth we will consider only well-formed forests and confound the notions of a node with that of the identifier of the node.

Definition 8.4 (Root id)
Given a tree t, if $t = s_i$ or $t = \texttt{<}l_i\texttt{>[}\ f\ \texttt{]}$ then we define $\mathbf{RootId}(t) = i$.

8.2.2 DTDs and validation

The approach we present in this chapter is for DTDs, but the treatment for XML Schema is similar.

Following [LMM00] we define a DTD as a *local tree grammar*:

Definition 8.5 (DTD)
A DTD is a pair (X, E) where X is a distinguished name *and E is a set of productions rules (or* edges*) of the form $\{X_1 \to R_1, \ldots, X_n \to R_n\}$, such that:*

1. *the X_i's are pairwise distinct;*
2. *each R_i is of the form `<`a_i`>[`r_i`]` or String, where a_i is an element tag, and each r_i is a regular expression over* names *$\{X_1, \ldots, X_n\}$;*
3. *for each pair $X_i \to$ `<`a_i`>[`r_i`]` and $X_j \to$ `<`a_j`>[ `r_j`]`, $i = j$ if and only if $a_i = a_j$;*
4. *X is in $\{X_1, \ldots, X_n\}$ (it denotes the root element type).*

In the following we write $\mathit{Names}(r)$ for the set of all names used in r and $\mathit{DN}(E)$ for the set of names defined in E (that is, $\{X_1 \ldots X_n\}$). We also say that r is a regular expression over (X, E), if r is a regular expression over names in $\mathit{DN}(E)$. We will use W, X, Y, Z to range over *names*. We use Greek letters to range over sets of names (in particular we use π to stress that the set of names is a *type projector*, as in Definition 8.8, κ and τ to stress that the set is used as a context or as a type, respectively (see Section 8.4.1) and S to range over sets of (node) identifiers. When speaking of DTDs we will often identify them with the set of their edges E, leaving the root X implicit.

We see that DTDs are far more restricted than ℂDuce types. In particular, point *(3.)* in Definition 8.5 states that for a given tag, there is only a unique definition for the associated content. For instance, it is possible to define `<a>[ (c | d)* ]` but not both `<a>[ c* ]` and `<a>[ d* ]` within the same DTD. In other words, the *tag* of an element determines its contents.

We define now the validity of a document with respect to a DTD:

Definition 8.6 (Valid Trees)
A tree t is valid *with respect to a DTD (X, E), if there exists a mapping (interpretation) $\Im$ from $\mathit{Ids}(t)$ to $\mathit{DN}(E)$ such that:*

1. *$\Im(\mathit{RootId}(t)) = X$*
2. *for each i in $\mathit{Ids}(t)$, if $t@i = s_i$ then $\Im(i) = Y$ and $(Y \to \mathit{String}) \in E$*
3. *for each i in $\mathit{Ids}(t)$, if $t@i =$ `<`l_i`>[`$t_1 \ldots t_n$`]`, then $\Im(i) \to$ `<`l`>[ `r` ]` $\in E$ and $\Im(\mathit{RootId}(t_1)), \ldots, \Im(\mathit{RootId}(t_n))$ is generated by r.*

In this case we say that t is $\Im$-valid with respect to (X, E) and write $t \in_{\Im} (X, E)$.

Algorithms to validate XML trees are well known (see [LMM00] for a comparison between various XML schema specifications). Every validation algorithm produces, as a side effect, an interpretation for the validated tree. Note that if t is valid with respect to a DTD, then there is a unique interpretation $\Im$ from t to the DTD. This is a direct consequence of point (3.) in Definition 8.5.

8.2.3 Type projectors

Given a tree t valid with respect to a DTD (X, E), we can use subsets of $DN(E)$ to project that tree. Essentially, only nodes that are associated with names in the projecting subset of $DN(E)$ are kept in the projection. Of course not every subset of names can be used to project a tree, since we want to delete whole sub-trees (not nodes in the middle of a tree), thus if we discard some name, we must also discard all the names it generates. In order to define formally this notion we need to define the reachability relation $\Rightarrow_E$.

Definition 8.7 (Forward Reachability)
Given a DTD (X, E) and $Z \in DN(E)$, we write $Z \Rightarrow_E Y$ if and only if $Z \rightarrow \texttt{<a>[}\ r\ \texttt{]} \in E$ and $Y \in Names(r)$. We use $\Rightarrow_E^+$ and $\Rightarrow_E^*$ to denote respectively the transitive closure and the transitive and reflexive closure of $\Rightarrow_E$.

Strings of names are called *chains* and ranged over by c, c_i, c',... In particular we use $Chains_{(X,E)}(Y)$ to denote the set of all chains rooted at Y, defined as $\{Y X_1 \dots X_n \mid Y \Rightarrow_E X_1 \Rightarrow_E \dots \Rightarrow_E X_n, n \geq 0\}$. We use $Names(c)$ to denote the set of all names occurring in a chain c.

Definition 8.8 (Type-Projectors)
Given a DTD (X, E), a (possibly empty) set of names $\pi \subseteq DN(E)$ is a type projector *for (X, E) if and only if there exists $C \subseteq Chains_{(X,E)}(X)$ such that*

$$\pi = \bigcup_{c \in C} Names(c)$$

A type projector is thus a set of names generated (i.e. reached) by a suite of productions starting from the root of the DTD. A type projector can be used to prune a valid tree as follows:

Definition 8.9 (Type Driven Projections)
Let π be a type projector for (X, E) and t a forest or tree such that $t \in_{\Im} (X, E)$. The π-projection of t, noted as $t \backslash_{\Im} \pi$,is defined as follows:

$$\begin{array}{lcll}
\texttt{<}l_i\texttt{>[}\ f\ \texttt{]} \backslash_{\Im} \pi & = & \texttt{<}l_i\texttt{>[}\ f \backslash_{\Im} \pi \texttt{]} & \Im(i) \in \pi \\
s_i \backslash_{\Im} \pi & = & s_i & \Im(i) \in \pi \\
\texttt{<}l_i\texttt{>[}\ f \backslash_{\Im} \pi \texttt{]} & = & \texttt{()} & \Im(i) \notin \pi \\
s_i \backslash_{\Im} \pi & = & \texttt{()} & \Im(i) \notin \pi \\
(f, f') \backslash_{\Im} \pi & = & (f \backslash_{\Im} \pi), (f' \backslash_{\Im} \pi) &
\end{array}$$

The interesting cases are the third and fourth cases of Definition 8.9. If a tree $<l_i>$`[`f`]` is the production of a name Y in the DTD (that is if $\Im(i) = \{Y\}$), and if Y is not in the projector, then this tree must be erased that is replaced by the empty forest; and the same holds for strings. In other words, pruning erases every node that corresponds to a name not in π.

Lemma 8.10 *Let π be a type projector for (X, E). Then for every tree $t \in_\Im (X, E)$ it holds $(t\backslash_\Im \pi) \preceq t$.*

Proof By straightforward induction the finite tree t equiped with the order $\sqsubseteq$ (sub-tree).

8.3 XPath and XPath$^\ell$

As we have already sketched in Chapter 7, XPath queries are expressed by defining a path of steps separated by /. For instance,

$$\begin{array}{rcl} Q & = & \texttt{/descendant} :: \mathit{author} \\ & & \texttt{/child::text()[self::node="}\mathit{Dante}\texttt{"]} \\ & & \texttt{/parent::*/parent::}\mathit{book}\texttt{/child::}\mathit{title} \end{array}$$

is the query that returns all titles of books whose author is "Dante". First, the navigational part instructs to descend to all text nodes whose parent is an author (`/descendant` :: *author*/`child` :: `text()`), then the predicate selects those nodes that are the string "Dante" (`[self::node="`*Dante*`"]`), and finally the navigation ascends to the book element and descends to the title.

The inference rules we define in Section 8.4.2 do not work directly on queries such as Q. The rules are defined for XPath$^\ell$, a subset of XPath that we introduce in this section. Contrary to XPath^t, introduced in Chapter 7, XPath$^\ell$ includes downward and upward axes and a special kind of predicates. In order to statically analyse Q (or any other XPath query that is not in XPath$^\ell$), we will find a XPath$^\ell$ query that approximates Q soundly with respect to the pruning inferred by the rules (Section 8.3.3), and use it to deduce the pruning for Q.[1] Of course, these approximations, as well as those we introduce later on, will only be used to determine the pruning: the pruned document will be queried by the original query.

For the sake of presentation, we first deal with "simple paths", that is, path expressions with upward and downward axes in which no predicate occurs. Then, in Section 8.3.2 we add XPath$^\ell$ predicates, i.e. disjunctions of simple predicates, and finally in Section 8.3.3 we show how to approximate generic XPath conditions into XPath$^\ell$. The missing axes are dealt with in Section 8.4.3.

[1] For instance, the approximation of our sample query Q is obtained by replacing in Q the predicate `[self::node]` for the current one.

8.3.1 Simple paths

Definition 8.11 (Simple Path)
Simple paths are defined by the following grammar:

$$\begin{array}{rcl} \mathit{SPath} & ::= & \mathit{Step} \mid \mathit{SPath}/\mathit{SPath} \mid /\mathit{SPath} \\ \mathit{Step} & ::= & \mathit{Axis}::\mathit{Test} \\ \mathit{Axis} & ::= & \texttt{self} \mid \texttt{child} \mid \texttt{descendant} \\ & \mid & \texttt{parent} \mid \texttt{ancestor} \mid \texttt{ancestor-or-self} \\ & \mid & \texttt{descendant-or-self} \\ \mathit{Test} & ::= & \mathit{tag} \mid \texttt{node()} \mid \texttt{text()} \end{array}$$

Here, *tag* is a meta-variable ranging over element tags. Henceforward, we omit the treatment of leading / (i.e., absolute paths) and of `descendant-or-self` and `ancestor-or-self` axes: their handling would blur definitions and can be easily deduced from the rest.

The formal semantics of paths is given in three definitions. First, we formalise *Test* filtering, then *Axis* selections, and finally we combine the two notions to define the semantics of a single step *Axis* :: *Test*. The definitions comply with the W3C XPath semantics [XPa].

Definition 8.12 (Filtering)
Given a tree t and a set of nodes $S \subseteq Ids(t)$ we define

$$\begin{array}{lcl} S ::_t l & = & \{i \in S \mid t@i = \texttt{<}l_i\texttt{>[}\ f\ \texttt{]}\} \\ S ::_t \texttt{node()} & = & S \\ S ::_t \texttt{text()} & = & \{i \in S \mid \exists\, s\,.\, t@i = s_i\} \end{array}$$

Definition 8.13 (Axes selection)
Given a tree t and a set of nodes $S \subseteq Ids(t)$ (called context nodes), we define $[\![Step]\!]_t(S)$ as the set of nodes resulting by applying Step to each node in S

$$\begin{array}{lcl} [\![\texttt{self}]\!]_t(S) & = & S \\ [\![\texttt{child}]\!]_t(S) & = & \bigcup_{i\in S}\{i' \mid (i,i') \in \mathcal{E}(t)\} \\ [\![\texttt{parent}]\!]_t(S) & = & \bigcup_{i\in S}\{i' \mid (i',i) \in \mathcal{E}(t)\} \\ [\![\texttt{descendant}]\!]_t(S) & = & \bigcup_{i\in S}\{i' \mid (i,i') \in \mathcal{E}(t)^+\} \\ [\![\texttt{ancestor}]\!]_t(S) & = & \bigcup_{i\in S}\{i' \mid (i',i) \in \mathcal{E}(t)^+\} \end{array}$$

where $\mathcal{E}(t)$ is the edge relation *of t, that is:*

$$\mathcal{E}(t) = \{(i,i') \mid t@i = \texttt{<}l_i\texttt{>[}\ f\ \ t'\ \ f'\ \texttt{]} \wedge RootId(t') = i'\}$$

and $\mathcal{E}(t)^+$ is its transitive closure.

Definition 8.14 (Simple Path Semantics)
Given t, a set $S \subseteq Ids(t)$ and a path SPath, we define the evaluation of path SPath over S nodes as follows:

$$\begin{array}{lcl} [\![Axis :: Test]\!]_t(S) & = & ([\![Axis]\!]_t(S)) ::_t Test \\ [\![SPath_1 \,/\, SPath_2]\!]_t(S) & = & [\![SPath_2]\!]_t([\![SPath_1]\!]_t(S)) \end{array}$$

Contrary to the XPath fragment we defined in Chapter 7, the semantics we present in Definition 8.14 is quite simple and compositional: the semantics of $SPath_2$ in the definition is applied to the result returned by the application of the semantics of $SPath_1$ to the argument S. This is because the result set here is a *set of ids*, which ensures that at most one occurrence of a given sub-tree appears in the result. This simplicity is to be contrasted with the fairly complex automaton encoding we gave in Chapter 7.

8.3.2 Predicates

XPath queries use predicates to express some filtering conditions that cannot be expressed by simple paths. Predicates mix *structural conditions* (directly expressed by means of paths) with *non-structural conditions* (expressed by functions, operators, values, etc...).

We have seen an example of a non-structural condition in the query Q extracting all book titles of books written by Dante, defined at the beginning of the section. The best pruning for the query Q is the one that deletes all books whose authors do not include Dante. To implement such a pruning, one should extract value-based conditions (e.g. being equal to “Dante”) from the query . This would drastically complicate the treatment without bringing a significant gain: previous experiments have shown that navigational specifications are already sufficient to obtain important improvements in memory reduction and query execution time [MS03]. Hence we would rather abstract out non-structural conditions and only retain structural ones. From this derives our definition of $XPath^\ell$:

Definition 8.15 (XPath $^\ell$ query)
XPath $^\ell$ queries are the finite production of the following grammar:

$$\begin{array}{lcll} Path & ::= & Step \mid Step[Cond] \mid Path/Path & (path) \\ Cond & ::= & SPath \mid Cond \texttt{ or } Cond & (condition) \end{array}$$

We will use meta-variables *Path* and P to range over these paths, and reserve *SPath* for simple paths and Q for general XPath queries. Note that the definition of *Cond* uses simple paths, therefore in XPath$^\ell$ conditions are not nested. The definition for the semantics of conditions is straightforward:

Definition 8.16 (Condition Semantics)
The truth value of a static condition is given by:

$$\begin{array}{lcl} Check_t[Path](i) & = & [\![Path]\!]_t(\{i\}) \neq \emptyset \\ Check_t[C_1 \texttt{ or } C_2](i) & = & Check_t[C_1](i) \vee Check_t[C_2](i) \end{array}$$

Semantics of XPath$^\ell$'s paths is defined by extending the one in in Definition 8.14:

Definition 8.17 (XPath$^\ell$ Semantics)
Given a tree t, a set $S \subset Ids(t)$ and a path Path, we define the evaluation of path Path over S nodes as follows:

$$\begin{array}{lcl} [\![Axis :: Test]\!]_t(S) & = & ([\![Axis]\!]_t(S)) ::_t Test \\ [\![Axis :: Test[C]]\!]_t(S) & = & [\![Axis :: Test/\texttt{self} :: \texttt{node}[C]]\!]_t(S) \\ [\![\texttt{self} :: \texttt{node}[C]]\!]_t(S) & = & \{i \in S \mid Check_t[C](i)\} \\ [\![Path_1/Path_2]\!]_t(S) & = & [\![Path_2]\!]_t([\![Path_1]\!]_t(S)) \end{array}$$

This semantics allows us to directly handle *structural conditions*. We present now a way to approximate general XPath conditions.

8.3.3 Handling XPath predicates

The predicates of the previous section cover only a small part of XPath. If we want to apply our analysis to XPath and XQuery we must be able to deal with the more general expressions used in conditions.

In this section we show how to rewrite every predicate *Exp* expressible in XPath to a simple condition *Cond* such that *Cond* is a sound approximation of *Exp* with respect to data needs: the pruning determined for *Cond* preserves the semantics for *Exp*. In other words, if we take a generic XPath query Q and approximate all its predicates to infer a projector π, then the execution of (the original) Q on a given document or on the document pruned by π yield the same result. This rewriting, together with the treatment of missing axes of Section 8.4.3, allows us to deal with a large subset of XQuery and XPath queries, covering all those in XPathMark [Fra05] and XMark [SWK$^+$02] benchmarks. First of all, let us define XPath conditions:

Definition 8.18 (XPath Conditions)

$$\begin{array}{lcll} Exp & ::= & Q & \textit{(XPath query)} \\ & | & Exp\ op\ Exp & \textit{(Operator)} \\ & | & f(Exp_1,\ldots,Exp_n) & \textit{(Function)} \\ & | & AExp & \textit{(Arithmetic expression)} \end{array}$$

$$Q \quad ::= \quad Step \mid Step[Exp] \mid Step/Q \mid Step[Exp]/Q$$

where $op \in \{$`eq`, `ne`, `lt`, `le`, `gt`, `ge`, `=`, `!=`, `<`, `<=`, `>`, `>=`, `is`, `<<`, `>>`, `or`, `and`$\}$ is an operator, *AExp* ranges over arithmetic expressions (see [XPa]) and base values (PCDATA), f ranges over XPath and XQuery functions and operators (see [MMW05] for a complete reference) such as `count`, `contains`, `is-zero`, `not`, `empty`, etc., and Q is a generic XPath query. The rewriting is obtained by a path-extracting function which, when applied to an expression *Exp* returns a set of *simple* paths whose "`or`" constitutes the approximation of *Exp*.[2]

Let us illustrate the rewriting by an example. Consider the predicate following predicate:

[`position()`>1 `and` `parent::node`/*book*/*author*="*Dante*" `and` *year*>1313]

In our system this predicate is approximated by:

[`self::node()` `or` `parent::node()`/*book*/*author* `or` *year*].

Indeed, if at run-time one wants to check that:

1. the position of the current node in the result set is greater than one;

2. and that there is a *book* under the parent node for which the *author* is *Dante*;

3. and that the *year* is 1313

then, the nodes that must be kept are those that:

1. match the current step (`self::node()`);

2. or are the node matching `parent::node()/child::`*book*`/child::`*author*, as well as its content;

3. or match *year*

[2] In order not to clutter this presentation with verbose details about XPath and XQuery specifications, we do not formally define the approximation for each function of the standard, nor do we handle other aspects such as absolute paths that are not relevant to the formal development.

Since these conditions are "dynamic" (we cannot determine them in advance since their truth value depends on a particular instance of the DTD), we must keep all the node matching either of the three conditions above. This ensures that at run-time, the node is present and that the dynamic condition can be tested on it.

Essentially, given a predicate *Exp* we obtain a condition *Cond* that soundly approximates it by retaining the disjunction of all structural conditions (for instance `parent::node`/*book*/*author* and *year* in the previous example), plus either `self::node` or `descendant-or-self::node` if some non-structural condition is present (for instance, `position()>1`). Indeed, for some conditions, keeping only the node is sufficient, while for other it is necessary to also keep the content of all the sub-trees. For instance functions like `position` or `count` require `self::node` since their execution requires only the root nodes; instead a function such as `string`, which concatenates all the strings occuring in the sub-tree in document order, needs the whole tree. The choice between `self::node` and `descendant-or-self::node` depends on the functions and operators used in the condition. We give here a partial definition of this function[3]:

Definition 8.19 (F function)
We define $\mathbf{F} : \mathit{String} \times \mathbb{N} \to \{\texttt{descendant-or-self} :: \texttt{node}, \texttt{self} :: \texttt{node}\}$ *as:*

$$\begin{array}{lcl}
\mathbf{F}(\texttt{fn:string}, 1) & = & \texttt{descendant-or-self} :: \texttt{node} \\
\mathbf{F}(\texttt{fn:count}, 1) & = & \texttt{self} :: \texttt{node} \\
\mathbf{F}(\texttt{fn:deep-equal}, 1) & = & \texttt{descendant-or-self} :: \texttt{node} \\
\mathbf{F}(\texttt{fn:deep-equal}, 2) & = & \texttt{descendant-or-self} :: \texttt{node} \\
\mathbf{F}(\texttt{fn:empty}, 1) & = & \texttt{self} :: \texttt{node} \\
 & \vdots &
\end{array}$$

For the sake of generality we suppose that this function depends on the position of the argument in n-ary functions. For instance, if we consider the function `fn:deep-equal` which takes two arguments and returns true if they are structurally equal, then the approximation needs to keep all the nodes for both arguments of the functions. Hence, $\mathbf{F}(\texttt{fn:deep-equal}, 1)$ returns `descendant-or-self :: node` and so does $\mathbf{F}(\texttt{fn:deep-equal}, 2)$. On the contrary, the `fn:empty` function, which returns true if its argument is the empty sequence only needs to keep the context node, not all its content. We can now define the approximation for general XPath predicates:

[3] A complete definition would be quite tedious since there are almost two hundred operators (as a rough count of those defined in [MMW05] indicates). We only give a few of them to illustrate our approach and refer the reader to our implementation for a more detailed list of what is currently handled.

Definition 8.20 (Predicate approximation)

$$
\begin{array}{lcl}
\mathbf{P}(\mathit{Step}) & = & \{\mathit{Step}\} \\
\mathbf{P}(\mathit{Step}[\mathit{Exp}]) & = & \mathit{Step}/\mathbf{P}(\mathit{Exp}) \\
\mathbf{P}(\mathit{Step}/Q) & = & \mathit{Step}/\mathbf{P}(Q) \\
\mathbf{P}(\mathit{Step}[\mathit{Exp}]/Q) & = & \mathit{Step}/(\mathbf{P}(Q)\cup\mathbf{P}(\mathit{Exp})) \\
\mathbf{P}(\mathit{Exp}\ \mathit{op}\ \mathit{Exp}') & = & \mathbf{P}(\mathit{Exp})\cup\mathbf{P}(\mathit{Exp}') \\
\mathbf{P}(f(\mathit{Exp}_1,\ldots,\mathit{Exp}_n)) & = & \bigcup_{i=1,n}(\mathbf{P}(\mathit{Exp}_i)/\mathbf{F}(f,i))\cup \\
 & & \qquad \cup\{\texttt{self} :: \texttt{node}\}
\end{array}
$$

where we used the notation *Step*/A as a shorthand for $\{\mathit{Step}/\mathit{SPath} \mid \mathit{SPath} \in A\}$ when A is a set of simple paths (similarly for A/*Step*). The first three cases of the definition are straightforward. For Case 4, since we need to check at run-time if a node matches *Exp and* if it matches Q, then we cannot discard nodes which would match either of them. Hence we need to keep the union of the approximations. Case 5 is similar, we need to keep the approximation of both operands. As for Case 6, the presence of {`self` :: `node`} is motivated by the fact that when we have a non structural condition, paths must not be used to restrict the inferred projectors, since this would not yield a sound approximation. More precisely, when *Exp* is purely structural, that is it only involves paths in (possibly nested) conditions, then these paths are extracted to refine the projection. For instance, in `descendant` :: `node`[`child` :: *a*] we can use the condition [`child::`*a*] to refine projection inference : we select only element types having an *a* child. On the other hand, when *Exp* is not purely structural, as in:

$$\texttt{descendant} :: \texttt{node}[\texttt{not}(\texttt{child} :: a)] \tag{8.1}$$

or in :

$$\texttt{descendant} :: \texttt{node}[\texttt{count}(\texttt{child} :: a)<5] \tag{8.2}$$

we can not use the same projector as for `descendant` :: `node`[`child` :: *a*]: if we use [`child` :: *a*] to restrict the projection, we would alter the result of the last two queries, so the projector would be unsound. To guarantee soundness, we extract paths from the arguments `not` and `count` and add the condition {`self` :: `node`} to ensure that we do not prune nodes necessary to the evaluation of the functions. So, for the Queries 8.1 and 8.2, after condition rewriting, we have the approximating query `descendant` :: `node`[`child` :: *a* `or` `self` :: `node`], yielding a sound projector.

To resume, to indicate the fact that, in the presence of not purely structural conditions, paths must not be used to restrict inferred projectors, we add the always true condition {`self` :: `node`}. Of course, we could have adopted more precise (and complex) techniques, but we preferred this solution since we consider it as a good compromise between precision and simplicity.

We want also to stress that here we reach the limits of XQuery and XPath type systems. Indeed, as we saw in Chapter 7, the use of a more advanced type-system allows us to encode dynamic conditions more precisely. For instance, we were able to encode the test against a character string as a ℂDuce type. Some other example of more precise encoding of XPath conditions can be found in [BCM05].

8.4 Static Analysis

In this section we define deduction rules to statically infer from a XPath$^\ell$ path P and a DTD E a type-projector for an input document validating E. We show that the analysis is sound, and that it enjoys completeness for a large class of queries when E is a $*$-guarded and non-recursive DTD (see Definition 8.25 below). Soundness means that executing the query on the original document and on the document pruned by the inferred projector yields the same result. Completeness means that if we take a type projector smaller (i.e., more selective) than the inferred one, then there exists a document validating E for which the result of the two executions is not the same. When the conditions on DTDs or on queries are relaxed the analysis is still sound but it may be not complete. Nevertheless, as we will illustrate, it still is very precise.

In order to define our static type inference we proceed in two steps.

1. Given a path P and a DTD E we type P by the set of all elements that may appear in the result of applying P to a document validating E. This is done in Section 8.4.1 (actually, we will be more precise and type P by the set of all names of E that *generate* the elements in the result).
2. We use the type inference at the previous point to define the inference of type projectors. In particular we will use the cases in which the previous type inference returns the empty set to determine the points in which pruning must be performed. This is done in Section 8.4.2.

8.4.1 Type inference

Given a path *Path* and a DTD E we want to find a set of names of E that generates elements that can be found in the result of P. Formally, we want to infer a set $\tau \subseteq DN(E)$ such that

$$\forall t \in_{\Im} E.\ \Im([\![Path]\!]_t(RootId(t))) \subseteq \tau \tag{8.3}$$

which states the soundness of the analysis. Moreover, we aim at an analysis which is precise enough to guarantee, on a large class of types and for a large class of queries, that whenever the path semantics is empty over all possible instances of the input DTD, then the inferred type τ is empty, as well:

$$\forall t \in_{\Im} E.\ \Im([\![Path]\!]_t(RootId(t))) = \emptyset \;\Rightarrow\; \tau = \emptyset \tag{8.4}$$

(the converse is a consequence of (8.3)). The precision described by (8.4) will then be used during the inference of type-projectors to discard elements that are useless in the evaluation of *Path*.

We start by inferring types for single-step paths.

Definition 8.21 (Single Step Typing)
Let E be a DTD and $\tau \subseteq DN(E)$, then:

$$\begin{array}{rcl}
A_E(\tau, \texttt{ancestor}) & = & \bigcup_{Y\in\tau}\{Z \mid Z \Rightarrow_E^+ Y\} \\
A_E(\tau, \texttt{child}) & = & \bigcup_{Y\in\tau}\{Z \mid Y \Rightarrow_E Z\} \\
A_E(\tau, \texttt{parent}) & = & \bigcup_{Y\in\tau}\{Z \mid Z \Rightarrow_E Y\} \\
A_E(\tau, \texttt{descendant}) & = & \bigcup_{Y\in\tau}\{Z \mid Y \Rightarrow_E^+ Z\} \\
A_E(\tau, \texttt{self}) & = & \tau \\
T_E(\tau, a) & = & \{Y \mid Y \in \tau, E(Y) = \texttt{<}a\texttt{>[}\ r\ \texttt{]}\} \\
T_E(\tau, \texttt{node}) & = & \tau \\
T_E(\tau, \texttt{text}) & = & \{Y \mid Y \in \tau,\ E(Y) = \mathit{String}\}
\end{array}$$

The type of a single step query $Axis :: Test$ for the DTD (X, E) is then given by $T_E(A_E(\{X\}, Axis), Test)$. As we can see, this definition is quite similar in spirit to the structural **(s-*)** rules of the algorithm presented in 7, Section 7.2. The present version is however simpler since the tag of an element determines its content.

Soundness of this definition, i.e. property (8.3), is given by the following lemma.

Lemma 8.22 *Let t be a tree $\Im$-valid with respect to the DTD E. For every $S \subseteq Ids(t)$ and type τ, if $\Im(S) \subseteq \tau$ then:*

1. $\Im([\![Axis]\!]_t(S)) \subseteq A_E(\tau, Axis)$
2. $\Im(S ::_t Test) \subseteq T_E(\tau, Test)$

> **Proof** By structural induction on the finite tree t and case analysis, it is easy to check that the property holds, for each case of Definition 8.13.

While the typing of composed paths was rather simple in the case of forward axes, as in Chapter 7, the presence of upward axes complicates the situation. To ensure precision, i.e. property (8.4), we have to be careful in dealing with DTDs in which an element may occur in the content of different elements. The naive solution consisting of inferring a type for composed paths by composing the functions we just defined for single steps, works only in the absence of upward axes. This can be illustrated by an example. Consider the following DTD rooted at X:

$$\{X \rightarrow \texttt{<c>[}Y\ \ Z\texttt{]}, Y \rightarrow \texttt{<a>[}W\ \mathit{String}\texttt{]},\ Z \rightarrow \texttt{<b>[}\mathit{String}\texttt{]}, W \rightarrow \texttt{<d>[}Y?\texttt{]}\}$$

and observe that Y occurs in two different element content definitions. If we consider the path $\texttt{self} :: c/\texttt{child} :: a/\texttt{parent} :: \texttt{node}$ over documents of the above DTD, then the precise type that this path should have is $\{X\}$. However, by using Definition 8.21 we would end up with $\{X, W\}$. This is because the first step selects $\{Y\}$ and then, according to Definition 8.21, the second step selects $\{X, W\}$, as Y is in the content definition of these two names.

To solve this problem we introduce particular types, called *contexts*, to be updated at each step and containing names already encountered in previous steps. We then use them to refine type inference for upward axes. In the previous example, when typing the first step we build a *context* $\{X, Y\}$ indicating that for the moment the two names are the only ones visited by the traversal. Then, we use Definition 8.21 to type `parent` thus obtaining $\{X, W\}$, as before, but this time we intersect it with the context thus obtaining the precise answer $\{X\}$.

This idea is formalised by the (deterministic) type system of Figure 8.1. We use the meta-variables τ to range over types and κ over contexts, both denoting sets of names defined by the input DTD E. An environment, ranged over by Σ, is a pair (τ, κ); we use Σ_τ and Σ_κ to denote the first and second projection of Σ, respectively. The judgement $(\tau_c, \kappa_c) \vdash_E \mathit{Path} : (\tau_r, \kappa_r)$ means that given a DTD E, starting from the names in τ_c and the current context κ_c, the path *Path* generates the names τ_r in an updated context κ_r.

$\boxed{\text{Primitive Single Step}}$

$$\frac{\mathit{Axis} \in \{\texttt{self}, \texttt{child}, \texttt{descendant}\}}{\Sigma \vdash_E \mathit{Axis} :: \texttt{node} :\ (A_E(\Sigma_\tau, \mathit{Axis})\ ,\ \Sigma_\kappa \cup A_E(\Sigma_\tau, \mathit{Axis}))}$$

$$\frac{\mathit{Axis} \in \{\texttt{parent}, \texttt{ancestor}\}}{\Sigma \vdash_E \mathit{Axis} :: \texttt{node} :\ (A_E(\Sigma_\tau, \mathit{Axis})) \cap \Sigma_\kappa\ ,\ A_E(\Sigma_\kappa, \mathit{Axis}) \cap \Sigma_\kappa}$$

$$\frac{\mathit{Test} \neq \texttt{node}}{\Sigma \vdash_E \texttt{self} :: \mathit{Test} :\ (T_E(\Sigma_\tau, \mathit{Test}), (\Sigma_\kappa \cap A_E(T_E(\Sigma_\tau, \mathit{Test}), \texttt{ancestor})) \cup T_E(\Sigma_\tau, \mathit{Test}))}$$

$$\frac{\forall X_i \in \Sigma_\tau, P_j \in \mathit{Cond}\ , \quad (\{X_i\}, \Sigma_\kappa) \vdash_E P_j :\ \Sigma^{ij}}{\Sigma \vdash_E \texttt{self} :: \texttt{node}[\mathit{Cond}] :\ (\ \tau\ ,\ (\Sigma_\kappa \cap A_E(\tau, \texttt{ancestor})) \cup \tau)}$$

where $\tau = \{X_i \mid \exists j.\Sigma^{ij}_\tau \neq \emptyset\}$

$\boxed{\text{Encoded Single Step}}$

$$\frac{\Sigma \vdash_E \mathit{Axis} :: \texttt{node}/\texttt{self} :: \mathit{Test} :\ \Sigma'}{\Sigma \vdash_E \mathit{Axis} :: \mathit{Test} :\ \Sigma'} \qquad \frac{\Sigma \vdash_E \mathit{Axis} :: \mathit{Test}/\texttt{self} :: \texttt{node}[\mathit{Cond}] :\ \Sigma'}{\Sigma \vdash_E \mathit{Axis} :: \mathit{Test}[\mathit{Cond}] :\ \Sigma'}$$

for $\mathit{Test} \neq \texttt{node}$ and $\mathit{Axis} \neq \texttt{self}$

$\boxed{\text{Composed paths}}$

$$\frac{\Sigma \vdash_E \mathit{Step} :\ \Sigma'' \qquad \Sigma'' \vdash_E \mathit{Path} :\ \Sigma'}{\Sigma \vdash_E \mathit{Step}/\mathit{Path} :\ \Sigma'}$$

Figure 8.1: Inference rules for single step queries

An environment (τ, κ) is well-formed with respect to E, if $\tau \subseteq \mathit{DN}(E)$, and $\kappa \subseteq \tau \cup A_E(\tau, \texttt{ancestor})$, that is, if the context contains only names that occur in chains ending with names in τ. A judgement $\Sigma \vdash_E \mathit{Path} : \Sigma'$ is well formed if both Σ and Σ' are well formed with respect to E. It is easy to see that the type inference rules of Figure 8.1 preserve well-formedness.

The rules are relatively simple to understand. The first two rules implement our main idea: when we follow an axis *Axis*, we compute the type by $A_E(\Sigma_\tau, \mathit{Axis})$; if the axis is a downward one, then we add this type to the current context, otherwise if the axis is an upward one, then we intersect it with the current context (both for the type part and for the context part). The rule for `self` :: *Test* is slightly more difficult since it discards from the current set of nodes those that do not satisfy the test: the type is computed by $T_E(\Sigma_\tau, \mathit{Test})$, while the context is obtained by erasing all the names that were in there just because they generated one of the discarded nodes; to do it it generates (the type of) all ancestors of the nodes satisfying the test, and intersects them with the current context. These first three rules are enough to type all the paths of the form *Axis* :: *Test* since, as stated by the fifth typing rule, all remaining cases are encoded as *Axis* :: `node/self` :: *Test*. The fourth rule is the most difficult one: recall that *Cond* is a disjunction of *simple* paths; the type τ is obtained by discarding from Σ_τ all (names of) nodes for which *Cond* never holds; thus for each X_i in Σ_τ we compute the type of all the paths in *Cond*, and keep in τ only names for which at least one path may yield a non-empty result; the context then is computed as in the third rule, by discarding from the context all names that generated only names discarded from Σ_τ. Once more, all the remaining cases of conditional steps are encoded by this one, as stated by the sixth rule. Finally, step composition is dealt as a logical cut.

First of all, let us prove that the deduction rules in 8.1 define a deterministic total and terminating algorithm:

Theorem 8.23 *Let (X, E) be a DTD and P a path and Σ an environment. There exists a unique finite derivation for the judgment $\Sigma \vdash_E P : \Sigma'$, for some Σ'.*

Proof Uniqueness of the derivation is immediate, since the rules are syntax-directed: at each step, exactly one of the rules applies. We prove finiteness by induction on the pair $(n(P), l(P))$ where:

$n(P)$**:** is the number of steps *Axis* :: *Test* for which $\mathit{Axis} \neq$ `self` and $\mathit{Test} \neq$ `node`:

$$\begin{array}{lcl}
n(\texttt{self} :: \texttt{node}) & = & 1 \\
n(\mathit{Axis} :: \mathit{Test}) & = & 0 \quad \text{if } \mathit{Axis} \neq \texttt{self} \wedge \mathit{Test} \neq \texttt{node} \\
n(\mathit{Axis} :: \mathit{Test}[C]) & = & n(\mathit{Axis} :: \mathit{Test}) + \sum_{P \in C} n(P) \\
n(P/P') & = & n(P) + n(P')
\end{array}$$

$l(P)$**:** is number of steps in P:

$$\begin{array}{lcl}
l(\mathit{Axis} :: \mathit{Test}) & = & 1 \\
l(\mathit{Axis} :: \mathit{Test}[C]) & = & 1 + \sum_{P \in C} l(P) \\
l(P/P') & = & l(P) + l(P')
\end{array}$$

Basic case: An application of any of the rules 1, 2 or 3 terminates the derivation since the three of them are axioms.

Inductive case: Let us show for the other rules that the measure decreases strictly in the premise of the rules:

- For Rule 4, it is clear that $n(P_j) \leq n(P)$ and that $l(P_j) < l(P)$.
- For Rule 5, if $\mathit{Axis} \neq \texttt{self} \wedge \mathit{Test} \neq \texttt{node}$ then $n(\mathit{Axis} :: \mathit{Test}) = 1$, while for the premise of the rule: $n(\mathit{Axis} :: \texttt{node/self} :: \mathit{Test}) = 0$.
- For Rule 6, if $\mathit{Axis} \neq \texttt{self} \wedge \mathit{Test} \neq \texttt{node}$ then $n(\mathit{Axis} :: \mathit{Test}[C]) = 1 + \sum_{P \in C} n(P)$, while for the premise: $n(\mathit{Axis} :: \texttt{node/self} :: \mathit{Test}[C]) = \sum_{P \in C} n(P)$.
- For Rule 7, it is clear that the first component does not increase in the premises, that $l(\mathit{Step}) < l(\mathit{Step/Path})$, and that $l(\mathit{Path}) < l(\mathit{Step/Path})$

We can state the first important property, soundness of the type-system:

Theorem 8.24 (Soundness of type inference) *Let (X, E) be a DTD and P a path. If $(\{X\}, \{X\}) \vdash_E P : (\tau, \kappa)$ then:*

$$\bigcup_{t \in_{\Im} E} \Im([\![P]\!]_t(\mathit{RootId}(t))) \subseteq \tau$$

Proof Let us consider the following, more general judgment:

$$(\tau, \kappa) \vdash_E P : (\tau', \kappa')$$

We show *simultaneously* the following properties:

1. Soundness : For all tree t $\Im$-valid with respect to (X, E) and all set $S \subseteq \mathit{Ids}(t)$, if $\Im(S) \subseteq \tau$ then:

$$\Im([\![P]\!]_t(S)) \subseteq \tau'$$

2. Context well-formedness, if:

$$\kappa = \{Y \mid \forall Z \in \tau, X \Rightarrow_E^* Y \Rightarrow_E^* Z\}$$

then:

$$\kappa' = \{Y \mid \forall Z \in \tau', X \Rightarrow_E^* Y \Rightarrow_E^* Z\}$$

Property 1 is a generalisation of the soundness property we are proving, and states that the interpretation of the semantics of a path P applied to all sub-trees of t produced by a name in τ is a subset of the output type τ'. Property 2 states that the algorithm maintains well-formed contexts. We prove both properties by induction on the depth of the typing derivation, which is finite by Theorem 8.23:

Base case:

Rule 1: Property 1 is true by a direct application of Lemma 8.22. Property 2 holds by definition of $A_E(\_,\_)$

Rule 2: As for the previous case, by Lemma 8.22, $\Im([\![Axis :: \texttt{node}]\!]_t(S)) \subseteq A_E(\tau, Axis)$. Moreover since κ is a well-formed context, $\kappa = \{Y \mid \forall Z \in \tau, X \Rightarrow_E^* Y \Rightarrow_E^* Z\}$. Let us first consider the case $Axis = \texttt{ancestor}$. By Definition 8.13, $[\![\texttt{ancestor} :: \texttt{node}]\!]_t(S) = \{i' \mid i \in S \wedge (i', i) \in \mathcal{E}^+(t)\}$. Thus, $\Im(\{i' \mid i \in S \wedge (i', i) \in \mathcal{E}^+(t)\}) = \{Y | Z \in \Im(S) \wedge Y \Rightarrow_E^+ Z\}$. Since we supposed $\Im(S) \subseteq \tau$, then clearly $\{Y | Z \in \Im(S) \wedge Y \Rightarrow_E^+ Z\} \subseteq \kappa$, thus $\Im([\![Axis :: \texttt{node}]\!]_t(S)) \subseteq A_E(\tau, Axis) \cap \kappa$, which is Property 1. The case for $Axis = \texttt{parent}$ is a particular instance of $Axis = ancestor$. As for Property 2, κ is the set of names visited up to the context node type τ. $A_E(\kappa, Axis)$ is the set of all parents (or ancestors) of those names. Consequently, the intersection is still a well formed context.

Rule 3 : Similarly to the case of Rule 1, Property 1 is a direct application of Lemma 8.22. For Property 2, we can remark that $\kappa' = \kappa \cap A_E(T_E(\tau, Test), \texttt{ancestor})$ contains all the names leading to a node in τ for which $Test$ succeeds (including the name of the selected node), hence it is a well-formed context.

Inductive case:

Rule 4 : The induction hypothesis holds for all the premises of the rule. Let us call $\tau_{ij} = \Sigma_\tau^{ij}$ and $\kappa_{ij} = \Sigma_\kappa^{ij}$. We have that: $\Im([\![P_j]\!]_t(S_i)) \subseteq \tau_{ij}$ where S_i is such that $\Im(S_i) \subseteq \{X_i\}$ (Property 1). By Definition 8.17 :

$$[\![\texttt{self} :: \texttt{node}[Cond]]\!]_t(S) = \bigcup_{\{i | i \in S \wedge \exists P_j \in Cond \text{ s.t. } [\![P_j]\!]_t(\{i\}) \neq \varnothing\}} \{i\}$$

If $[\![P_j]\!]_t(\{i\}) \neq \varnothing$, then $\Im([\![P_j]\!]_t(\{i\})) \neq \varnothing$, and by Property 1, $\tau_{ij} \neq \varnothing$, which implies the $X_i \in \tau'$. Consequently, Property 1 holds for the goal of the rule. Property 2 holds, similarly to Rule 3.

Rule 5 and Rule 6: The induction hypothesis holds for the premise of the rule. Since τ' and κ' are unchanged in the goal of the rule, both properties hold.

Rule 7 : Property 1 is true by induction hypothesis on both premises. Property 2 is true for the first premise, by induction hypothesis. In particular, Σ''_κ is a well-formed context. We can then apply the induction hypothesis on Σ'' and we have that Σ'_κ is a well-formed context too.

□

The type system is also complete for DTDs that are $*$-guarded, non-recursive, and parent-unambiguous. Intuitively, a DTD is $*$-guarded when every union occurring in its productions is guarded by $*$ (or by +); it is non recursive if the depth of all documents validating it is bound; it is parent-unambiguous if no name types both the parent and a strict ancestor of the parent of another name. Formally, we have the following definition

Definition 8.25
Let (X, E) be a DTD.

1. *E is* $*$-guarded *if for each $Y \to$ <l>[r] in E, the regular expression is a product $r = r_1, \ldots, r_n$ and whenever r_i contains a union, then $r_i = (r')*$;*
2. *E is* non-recursive *if it is never the case that $Y \Rightarrow^+_E Y$, for any name $Y \in DN(E)$;*
3. *E is* parent-unambiguous *if for all chains c and names Y, Z such that $cYZ \in Chains_{(X,E)}(X)$ the following implication*

$$cYc'Z \in Chains_{(X,E)}(X) \quad \Longrightarrow \quad c' = \varepsilon$$

holds (ε denotes the empty chain).

Non-recursivity and $*$-guardedness are properties enjoyed by a large number of commonly used DTDs. As an example, the reader can consider the DTDs of the XML Query Use Cases [CFF$^+$03]: among the ten DTDs defined in the Use Cases, seven are both non-recursive and $*$-guarded, one is only $*$-guarded, one is only non-recursive, and just one does not satisfy either properties. Furthermore our personal experience is that most of the DTDs available on the web are $*$-guarded. Concerning the parent-unambiguous property, although DTDs satisfying this property are less frequent (five on the ten DTDs in [CFF$^+$03]), its absence is in practice not very problematic since, as we will see, only the presence of the `parent` axis may hinder completeness.

Theorem 8.26 (Completeness) *Let (X, E) be a DTD and P a path. If (X, E) is $*$-guarded, non-recursive, parent-unambiguous, and if $(\{X\}, \{X\}) \vdash_E P : (\tau, \kappa)$, then we have:*

$$\tau \subseteq \bigcup_{t \in_{\Im} E} \Im(\llbracket P \rrbracket_t(RootId(t)))$$

Proof Like for the proof of Theorem 8.24, we consider the following, more general judgment:

$$(\tau, \kappa) \vdash_E P : (\tau', \kappa')$$

We show that for each tree t, $\Im$-valid with respect to (X, E) and set $S \subseteq Ids(t)$, if $\tau \subseteq \Im(S)$ then:

$$\tau' \subseteq \Im(\llbracket P \rrbracket_t(S))$$

We proceed by induction on the depth of the typing derivation:

Basic case :

Rule 1 :

`self` **axis** : We have $A_E(\tau, \texttt{self}) = \tau$. By definition of $\llbracket\ \rrbracket_t(\_)$, $\llbracket \texttt{self} :: \texttt{node} \rrbracket_t(S) = S$, hence $\Im(\llbracket \texttt{self} :: \texttt{node} \rrbracket_t(S)) = \Im(S)$ which, by hypothesis is such that $\tau \subseteq \Im(S)$.

`descendant` **axis** : Let $Y \in \tau$. Let $\tau' = A_E(\{Y\}, \texttt{descendant})$. By definition:

$$\tau' = \{Z | \exists Z_1, \ldots, Z_n \text{ s.t. } Y \Rightarrow_E Z_1 \Rightarrow_E \ldots \Rightarrow_E Z_n \Rightarrow_E Z\}$$

Here, the chain $YZ_1 \ldots Z_n Z$ is finite and the Z_k, Y and Z are distinct from one another since the DTD is not recursive. Let us suppose that $\exists i_0, \ldots, i_n, l \in Ids(t)$, where $\Im(i_k) = Z_k$ and $\Im(l) = Z$. Now let us consider j such that $\Im(j) = Y$. j, l and the i_k's are distinct from one another (because their $\Im$-interpretations are).

We can conclude that if $Z \in \tau'$, then $Z \in \Im(\llbracket \texttt{descendant} :: \texttt{node} \rrbracket_t(\{j\}))$. Indeed, $\llbracket \texttt{descendant} :: \texttt{node} \rrbracket_t(\{j\}) = \{m | (j, m) \in \mathcal{E}^+(t)\}$. Let us call T this set. Since, $(j, i_1), \ldots, (i_n, l) \in \mathcal{E}^+(t)$, we have that l is in T, and consequently that $Z = \Im(l) \in \Im(\llbracket \texttt{descendant} :: \texttt{node} \rrbracket_t(\{j\}))$. Hence, for all Z, if $Z \in \tau'$ then $Z \in \Im(\llbracket \texttt{descendant} :: \texttt{node} \rrbracket_t(\{j\}))$ which proves the theorem for this case.

`child` **axis** : is a particular instance of the previous case.

Rule 2 : We only treat the case of the `ancestor` axis, of which the `parent` axis is a particular instance. Let $Y \in \tau$. Let $\tau'' = A_E(\{Y\}, \texttt{ancestor})$. By definition:

$$\tau'' = \{Z | \exists Z_1, \ldots, Z_n \text{ s.t. } Z \Rightarrow_E Z_1 \Rightarrow_E \ldots \Rightarrow_E Z_n \Rightarrow_E Y\}$$

Again the chain $ZZ_1 \ldots Z_n Y$ is finite since the DTD is not recursive, and all names are pairwise distinct. We know that κ is a valid context, hence that:

$$\tau' = \tau'' \cap \kappa = \{Z | X \Rightarrow_E^* Z \Rightarrow_E^+ Y\}$$

Furthermore, given a name Y there is a *unique* chain of names $X \ldots Y$, because the DTD is parent-unambiguous. Let us consider Z in τ' with the associated unique chain $XZ_1 \ldots Z_n Z$. Then there exists a document t $\Im$-valid with respect to the DTD, such that i,i_1,...,i_n,l $\in$ $Ids(t)$ and $\Im(i) = \{X\}$, $\Im(i_k) = \{Z_k\}$ and $\Im(l) = \{Z\}$. Let us now consider $[\![\texttt{ancestor} :: \texttt{node}]\!]_t(\{l\})$. By definition, we have that: $[\![\texttt{ancestor} :: \texttt{node}]\!]_t(\{l\}) = \{l, i_n, \ldots, i_1, i\}$, hence that: $\Im(\{l, i_n, \ldots, i_1, i\}) = \{X, Z_1, \ldots, Z_n, Z\}$, and thus that $\tau' \subseteq \bigcup_{t \in_\Im E} \Im([\![\texttt{ancestor} :: \texttt{node}]\!]_t(S))$

Rule 3: is similar to the case `self` of Rule 1.

Inductive case :

Rule 4: Let us consider the condition:

$$Cond \equiv P_1 \texttt{ or } \ldots \texttt{ or } P_n$$

Let us note $X_{i1}, \ldots, X_{in_1}$ the names for which P_i may yields a non-empty result. Without loss of generality, we can restrict ourselves to the case where $Cond \equiv P_1 \texttt{ or } P_2$, and P_1 and P_2 are incompatible that is, $\forall j \in Ids(t)$:

$$[\![P_1]\!]_t(\{j\}) \neq \varnothing \implies [\![P_2]\!]_t(\{j\}) = \varnothing$$
$$[\![P_2]\!]_t(\{j\}) \neq \varnothing \implies [\![P_1]\!]_t(\{j\}) = \varnothing$$

Let us assume that $X_1 \in \tau'$ is such that $\tau_{11} \neq \varnothing$ and $X_2 \in \tau'$ is such that $\tau_{22} \neq \varnothing$ (that is X_1 yields a non empty type for P_1 and X_2 yields a non-empty type for P_2. Furthermore, since there are at least two distinct names X_1 and X_2 in τ' and thus in τ, we can assume that we are not at the root of the document (in which case τ would be $\{X\}$). Then there exists a document t, $\Im$-valid with respect to the DTD, such that, i_1, $i_2 \in Ids(t)$ and $\Im(i_1) = \{X_1\}$ and $\Im(i_2) = \{X_2\}$. This is possible because the DTD is $*$-guarded, which means that X_1 and X_2 occur either as $(X_1|X_2)*$ or X_1, X_2 or X_2, X_1. For each case, it is possible to have two distinct sub-trees, one generated by X_1 the other by X_2 on the same level (if the DTD were not $*$-guarded, for instance with $X_1|X_2$, then our algorithm would have kept X_1 and X_2 in the type while only one tree could be present in a document, thus compromising completeness). We can conclude by remarking that:

$$\Im([\![\texttt{self} :: \texttt{node}[Cond]]\!]_t(S)) = \{X_1, X_2\}$$

thus that $\tau' \subseteq \bigcup_{t \in_\Im E} \Im([\![\texttt{self} :: \texttt{node}[Cond]]\!]_t(S))$

Rule 5 and Rule 6: are a straightforward application of the induction hypothesis

Rule 7 : Again, we can apply the induction hypothesis on the premises of the rule. We only need to note that, as for Rule 4, if *Step* and *Path* are incompatible, then the $*$-guardness of the DTD guarantees the existence of a document with at least two nodes: one for which *Step* is successful and one for which *Path* is successful (see the example hereafter).

□

To see why completeness does not hold in general consider the following DTD rooted at X and which is recursive and not $*$-guarded

$$\{X \rightarrow \texttt{<c>[}Y|Z\texttt{]},\ Y \rightarrow \texttt{<a>[}Y{*}\ \textit{String}\texttt{]},\ Z \rightarrow \texttt{<b>[}\textit{String}\texttt{]}\}$$

and the following two queries $\texttt{self} :: c[\texttt{child} :: a]/\texttt{child} :: b$ and $\texttt{self} :: c/\texttt{child} :: a/\texttt{parent} :: \texttt{node}$. The type inferred for the first query contains both Y and Z. These are useless since the query is always empty. This is due to the non $*$-guarded union $Y|Z$: if we had $(Y|Z)*$ instead, then the query might yield a non-empty result, therefore Y and Z must correctly (and completely) be in the query type. The second query shows the reason why completeness does not hold in presence of recursion and backward axes (recursion with only forward axes does not pose any problem for completeness). The type of the second query should be $\{X\}$, but instead the type $\{X, Y\}$ is inferred. This is due to the recursion $Y \rightarrow \texttt{<a>[}Y{*}\ldots\texttt{]}$: since $Y \Rightarrow_E Y$, once Y is reached it is kept in the inferred type for every backward step.[4]

For queries over parent-ambiguous DTDs, completeness does not hold because the fourth rule in Figure 8.1—the one defined for $\texttt{self} :: \texttt{node}[\textit{Cond}]$—is not precise for the parent axis. For instance, consider the following DTD rooted at X

$$\{X \rightarrow \texttt{<a>[}Y\ Z\texttt{]},\ Y \rightarrow \texttt{<b>[}Z\texttt{]},\ Z \rightarrow \texttt{<c>[]}\}$$

and the query $\texttt{self} :: a/\texttt{child} :: b/\texttt{child} :: c/\texttt{parent} :: \texttt{node}$. The precise type of this query should be $\{Y\}$. However, the inferred type is $\{X, Y\}$. This is because the last step $\texttt{parent} :: \texttt{node}$ is typed with the context $\{X, Y, Z\}$ and this contains $A_E(\{Z\}, \texttt{parent}) = \{X, Y\}$. Here Z is the type for the c node selected by $\texttt{child} :: c$ and the $A_E(,)$ operator assigns it $\{X, Y\}$ as parent type, even if the *real* parent type for Z in this case should be $\{Y\}$. Hence, the intersections operated by the type rule for $\texttt{parent}$ are not powerful enough to guarantee precision for cases like this one. In a nutshell, this happens because in the presence of parent-ambiguous DTDs the type analysis may produce contexts containing false parent types (with respect the current type τ). This suggests that to be extremely precise, instead of sets of names, contexts should rather be sets of *chains* of names, computed and opportunely managed by the type analysis. However (i) managing sets of chains instead of simple sets of names dramatically complicates the treatment, due to recursive axes like

[4] The techniques developed in [CGMS04, Col04] can be adapted to recover completeness for cases like the first query, while a more sophisticated type analysis could solve the problem with the second. In view of the precision of the current approach this is not a priority and we leave this investigation as future work.

descendant, (ii) the problem may arise only for queries that use parent axis and the concomitance of parent-ambiguity make the event rare in practice, and (iii) the loss of precision looks in most cases negligible. Therefore we considered that such a small gain (remember that completeness is just some icing on the cake since while it helps to gauge the precision of the approach its absence does not hinder its application) did not justify the dramatic increase in complexity needed to handle this case.

Note also that the type system, hence the completeness result, is stated for predicates of the form described in Section 8.3.2, therefore it does not account for the approximations introduced in Section 8.3.3. However very few non-structural conditions can be expressed at the level of types, so the impact of these approximations on completeness is very light.

8.4.2 Type-Projection inference

In this section we use the type inference defined previously to infer type-projectors. Once again, naive solutions do not work. For instance, if we consider simple paths $Step_1/\dots/Step_n$, we may consider as type projector with respect to (X, E) the set $\bigcup_{i=1\dots n} \tau_i \cup \{X\}$, where for $i = 1 \dots n$:

$$(\{X\}, \{X\}) \vdash_E Step_1/\dots/Step_i : \ (\tau_i, -)$$

(we use "$-$" as a placeholder for uninteresting parameters). This definition is sound but not precise at all, as can be seen by considering $\texttt{descendant} :: \texttt{node}/Path$: the use of the above union yields a set containing τ_1 defined as

$$(\{X\}, \{X\}) \vdash_E \texttt{descendant} :: \texttt{node} : \ (\tau_1, -)$$

that is, all descendants of the root X (no pruning is performed). Instead, we would like to discard, at least, all names that are descendants of X but that are not ancestors of a node matching *Path*. These are the names $Y \in T_E(A_E(\{X\}, \texttt{descendant}), \texttt{node})$ such that:

$$(\{Y\}, \kappa) \vdash_E \texttt{descendant} :: \texttt{node}/Path : \ (\emptyset, -)$$

for some appropriate context κ. A similar reasoning applies to ancestor.

Such a selection is performed by the inference rules of Figure 8.2. For paths formed by a single step, if the step has no condition (first rule), then the type inference of the previous section is enough; otherwise (second rule) the step is transformed into a complex path (a simple trick to avoid the definition of several rules). Thanks to the third rule the type inference can work on just one node at a time, and thanks to the fourth and fifth rule, it just analyses paths whose components have one of the following three forms: (i) $\texttt{self}::Test$, (ii) $\texttt{self}::\texttt{node}[Cond]$, or (iii) $Axis::\texttt{node}$. These three cases are handled by the "Primitive Rules" of Figure 8.2: The first rule handles the case (i) simply by collecting the current context. The second rule handles the case (ii), by collecting besides the context also all the parts that are necessary to compute the condition (which in the rule is expanded in its more general form); the case (iii) is handled by the last three rules which are nothing but slight variations of the same rule according to the particular axis taken into account: each rule infers the type τ obtained by discarding from the type $\{X_1, ..., X_n\}$ of the step, all

Base and induction

$$\frac{\Sigma \vdash_E \mathit{Step} :\ (\tau,\kappa)}{\Sigma \Vdash_E \mathit{Step} : \tau \cup \kappa} \qquad\qquad \frac{\Sigma \Vdash_E \mathit{Step}[\mathit{Cond}]/\texttt{self} :: \texttt{node} : \tau}{\Sigma \Vdash_E \mathit{Step}[\mathit{Cond}] : \tau}$$

$$\frac{(\{X_1\},\kappa) \Vdash_E P : \tau_1 \qquad \ldots \qquad (\{X_n\},\kappa) \Vdash_E P : \tau_n}{(\{X_1,\ldots,X_n\}\,,\,\kappa) \Vdash_E P : \bigcup_{i=1..n}\tau_i}$$

Encoded Rules

$$\frac{\Sigma \Vdash_E \mathit{Axis} :: \texttt{node}/\texttt{self} :: \mathit{Test}/P : \tau}{\Sigma \Vdash_E \mathit{Axis} :: \mathit{Test}/P : \tau}\ \begin{array}{c}\mathit{Test} \neq \texttt{node}\\ \wedge\\ \mathit{Axis} \neq \texttt{self}\end{array}$$

$$\frac{\Sigma \Vdash_E \mathit{Axis} :: \mathit{Test}/\texttt{self} :: \texttt{node}[\mathit{Cond}]/P : \tau}{\Sigma \Vdash_E \mathit{Axis} :: \mathit{Test}[\mathit{Cond}]/P : \tau}\ \begin{array}{c}\mathit{Test} \neq \texttt{node}\\ \vee\\ \mathit{Axis} \neq \texttt{self}\end{array}$$

Primitive Rules

$$\frac{(\{Y\},\kappa) \vdash_E \texttt{self} :: \mathit{Test} :\ \Sigma \quad \Sigma \Vdash_E P : \tau}{(\{Y\},\kappa) \Vdash_E \texttt{self} :: \mathit{Test}/P : \{Y\} \cup \tau}$$

$$\frac{(\{Y\},\kappa) \vdash_E \texttt{self} :: \texttt{node}[P_1\texttt{or}\ldots\texttt{or}P_n] :\ \Sigma \quad \Sigma \Vdash_E P : \tau \quad \Sigma \Vdash_E P_i : \tau_i}{(\{Y\},\kappa) \Vdash_E \texttt{self} :: \texttt{node}[P_1\texttt{or}\ldots\texttt{or}P_n]/P : \{Y\} \cup \tau \cup \tau_1 \cup \cdots \cup \tau_n}\ n \geq 1$$

$$\frac{(\{Y\},\kappa) \vdash_E \mathit{Axis} :: \texttt{node} :\ (\{X_1,...,X_n\},\kappa') \quad (\{X_i\},\kappa') \vdash_E P :\ \Sigma^i \quad (\tau,\kappa') \Vdash_E P : \tau'}{(\{Y\},\kappa) \Vdash_E \mathit{Axis} :: \texttt{node}/P : \{Y\} \cup \tau \cup \tau'}$$

where $\mathit{Axis} \in \{\texttt{parent},\texttt{child}\}$ and $\tau = \{X_i \mid \Sigma^i_\tau \neq \emptyset\}$

$$\frac{\begin{array}{l}(\{Y\},\kappa) \vdash_E \texttt{desc} :: \texttt{node} :\ (\{X_1,\ldots,X_n\},\kappa')\\ (\{X_i\},\kappa') \vdash_E \texttt{desc} :: \texttt{node}/P :\ \Sigma^i \qquad\qquad (\tau,\kappa') \Vdash_E \texttt{child} :: \texttt{node}/P : \tau'\end{array}}{(\{Y\},\kappa) \Vdash_E \texttt{desc} :: \texttt{node}/P : \tau \cup \tau'}$$

where $\tau = \{X_i \mid \Sigma^i_\tau \neq \emptyset\} \cup \{Y\}$.

$$\frac{\begin{array}{l}(\{Y\},\kappa) \vdash_E \texttt{ancs} :: \texttt{node} :\ (\{X_1,...,X_n\},\kappa')\\ (\{X_i\},\kappa') \vdash_E \texttt{ancs} :: \texttt{node}/P :\ \Sigma^i \qquad\qquad (\tau,\kappa') \Vdash_E \texttt{parent} :: \texttt{node}/P : \tau'\end{array}}{(\{Y\},\kappa) \Vdash_E \texttt{ancs} :: \texttt{node}/P : \tau \cup \tau'}$$

where $\tau = \{X_i \mid \Sigma^i_\tau \neq \emptyset\} \cup \{Y\}$

Figure 8.2: Projectors inference rules (where `ancs` and `desc` are shorthands for `ancestor` and `descendant`)

names that are useless for the rest of the path, and then uses this τ to continue the inference of the projector. It is easy to show that the process terminates:

Theorem 8.27 (Termination of projector inference) *Let (X, E) be a DTD and P a path and Σ an environment. There exists a unique finite derivation for the judgment $\Sigma \Vdash_E P : \Sigma'$, for some Σ'.*

> **Proof** We can use the same measure as for the termination of type inference, since paths are deconstructed the same way.

The projector inference algorithm is sound:

Theorem 8.28 (Soundness of projector inference) *Let (X, E) be a DTD and P a path. If $(\{X\}, \{X\}) \Vdash_E P : \tau$, then τ is a type projector for (X, E) and for every $t \in_{\Im} E$:*

$$[\![P]\!]_{t\backslash_{\Im}\tau}(\mathit{RootId}(t)) = [\![P]\!]_t(\mathit{RootId}(t))$$

> **Proof** This is a straightforward induction on the depth of the derivation. We use 8.24 to show that whenever the type inference algorithm returns $\varnothing$ then the query is always empty hence the node name is safe to remove.

The above theorem states that executing the query P on a tree t returns the same set of nodes as executing it on $t\backslash_{\Im}\tau$ (see Definition 8.9) the tree t pruned by the inferred projector. From a practical perspective it is important to notice that according to standard XPath semantics, the semantics of a query contains *only* the nodes of the result of the query not their sub-trees. The latter may thus be pruned by the inferred projector. Therefore, if we want to *materialise* the result of a query we must not cut these nodes, and rather use the projection $\tau = \tau' \cup A_E(\tau'', \mathtt{descendant})$ where $(\{X\}, \{X\}) \Vdash_E P : \tau'$ and $(\{X\}, \{X\}) \vdash_E P : (\tau''; -)$.

Completeness requires not only completeness of the type system ($*$-guarded, non-recursive, and parent-unambiguous DTDs), but also the following condition on queries:

Definition 8.29
An XPath query Q is strongly-specified *if:*

i. *its predicates do not use backward axes,*

ii. *along Q and along each path in the predicates of Q there are no two consecutive (possibly conditional) steps whose Test part is* `node`

iii. *each predicate in Q contains at most one path and this does not terminate by a step whose Test is* `node`.

For instance, among the following queries, only the first two are strongly-specified:

1. $\texttt{descendant} :: \texttt{node}/\texttt{self} :: a/\texttt{ancestor} :: \texttt{node}$
2. $\texttt{descendant} :: \texttt{node}[\texttt{child} :: b]/\texttt{self} :: a/\texttt{parent} :: \texttt{node}$
3. $\texttt{descendant} :: \texttt{node}/\texttt{ancestor} :: \texttt{node}/\texttt{self} :: a$
4. $\texttt{descendant} :: \texttt{node}[\texttt{child} :: b/\texttt{child} :: \texttt{node}]/\texttt{self} :: a$
5. $\texttt{child} :: a[\texttt{descendant} :: \texttt{node}/\texttt{parent} :: b]/\texttt{child} :: c$

Once more, we are in presence of a very common class of queries: for instance, almost all paths in the XMark and XPathMark benchmarks are strongly specified.

Theorem 8.30 (Completeness of projector inference) *Let (X, E) be a $*$-guarded, non-recursive, and parent-unambiguous DTD, and P a strongly-specified path. If $(\{X\}, \{X\}) \Vdash_E P : \tau$, then there exists $t \in_{\Im} E$ such that for each $Y \in \tau$, if $\pi = \tau \setminus (\{Y\} \cup A_E(\{Y\}, \texttt{descendant}))$, then:*

$$[\![P]\!]_{t \backslash_{\Im} \pi}(\mathit{RootId}(t)) \neq [\![P]\!]_t(\mathit{RootId}(t))$$

Proof By induction on the length of the typing derivation which is finite. We use Theorem 8.26 to show that if we remove a name Y inferred by the type inference algorithm, then we remove nodes from the result of the query applied to the projected document. The fact that P is strongly specified is used for the treatment of predicates. Indeed, it forces any path in a predicate to be matched exactly by one node. If a path in a predicate could be matched by two (or more) nodes, then removing one of the nodes would not change the semantics of the query, since there would still be a node present to make the predicate succeed. We illustrate this in the example hereafter.

The fact that completeness may not hold for not $*$-guarded, non-recursive, or parent-ambiguous DTDs, is a consequence of the analogous property of the type system. To see that also strong-specification is a necessary condition consider documents valid with respect to the following DTD rooted at X:

$$\{X \to a[Y, W],\ W \to c[\,], Y \to b[Z],\ Z \to d[\,]\}.$$

Query them by the following query which is not strongly-specified since it does not satisfy condition (ii) of Definition 8.29

$$\texttt{self} :: a[\texttt{child} :: \texttt{node}].$$

$\{X, Y\}$ is an optimal projector for this query, but the presence of the condition $\texttt{self} :: \texttt{node}$ makes the system to include also W in the inferred projector, thus breaking completeness. Concerning the presence of backward axes in predicates, consider the query $\texttt{self} :: a[\texttt{descendant} :: \texttt{node}/\texttt{ancestor} :: a]$ which does not satisfy condition (i). An optimal projector for this query on the same DTD is $\{X, Y\}$. However, since the `ancestor` condition is true for all descendants of a nodes, $\{W, Z\}$ is included in the projector as well. Finally, it is straightforward to check that the query $\texttt{self} :: a[\texttt{child} :: b \texttt{ or } \texttt{child} :: c]$, which does not satisfy condition (iii), is not complete for the same DTD.

Of course, it is possible to state completeness for other classes of queries but, once more, this seems a satisfactory compromise between simplicity and generality.

8.4.3 Adding sibling, preceding and following axes.

We could deal with the missing XPath axes by adding specific inference rules. Instead we opt to use an approximation of these axes in term of the previous ones, since it appears as the best compromise between simplicity and efficiency.

The approximation is performed by two logical rewriting passes. In the first pass we rewrite preceding and following axes as specified in the W3C specifications [DFF+04]. Namely, we substitute each step:

$$\mathit{Axis} :: \mathit{Test} \text{ where } \mathit{Axis} \in \{\texttt{preceding}, \texttt{following}\}$$

by the following equivalent path:

$$\texttt{ancestor-or-self} :: \texttt{node}/(\mathit{Axis}\texttt{-sibling}) :: \texttt{node}/\texttt{descendant-or-self} :: \mathit{Test}$$

The second pass is the one which introduces the approximation since it replaces all steps of the form $\mathit{Axis}::\mathit{Test}$ with $\mathit{Axis} \in \{\texttt{preceding-sibling}, \texttt{following-sibling}\}$ by the path $\texttt{parent}::\texttt{node}/\texttt{child}::\mathit{Test}$.

Clearly, the static analysis of the approximation yields a less precise projection than the one we could obtain by working directly on the original query. However, we still achieve good precision of pruning in practice as we will show in Section 8.6. For instance, by applying the above rewriting to XPathMark queries Q9 and Q11, we were able to prune a document down to 7.5% of its original size.

8.5 Extension to XQuery

In this section we extend the technique to XQuery. More precisely to the FLWR core of XQuery described by the following grammar:

Definition 8.31 (XQuery core)

$$\begin{array}{rcll}
q & ::= & () \mid q\,,\,q \mid \textit{<tag>}q\textit{</tag>} \mid \mathit{Exp} & \textit{(Simple expression)} \\
& \mid & \texttt{for}\ x\ \texttt{in}\ q\ \texttt{return}\ q & \textit{(for)} \\
& \mid & \texttt{let}\ x = q\ \texttt{return}\ q & \textit{(let)} \\
& \mid & \texttt{if}\ q\ \texttt{then}\ q\ \texttt{else}\ q & \textit{(if)}
\end{array}$$

where the definition of *Exp* (given in Section 8.3.3) is extended with variables, and with generic XPath expressions Q of Section 8.3.3 that can be rooted at a variable or at / :

$$Exp ::= x \mid Q \mid x/Q \mid /Q \mid Exp\ op\ Exp \mid f(Exp, .., Exp) \mid AExp$$

Without loss of generality, we assume that FLWR expressions do not occur in `if`-conditions nor in predicates (every query can be put into this form by adding appropriate `let`-expressions). Also, we do not consider either queries which first construct new elements and then navigate on them (these are rarely used in practice), nor those containing XQuery clauses like `order_by`, `switch_case`, etc.: our approach can be easily extended to both cases.

$$
\begin{array}{lll}
1. & \mathbf{E}((), \Gamma, m) & = \varnothing \\
2. & \mathbf{E}(AExp, \Gamma, 1) & = \{P \mid (x;\ \text{for}\ P) \in \Gamma\} \\
3. & \mathbf{E}(AExp, \Gamma, 0) & = \varnothing \\
4. & \mathbf{E}((q_1, q_2), \Gamma, m) & = \mathbf{E}(q_1, \Gamma, m) \cup \mathbf{E}(q_2, \Gamma, m) \\
5. & \mathbf{E}(\texttt{<}tag\texttt{>}q\texttt{</}tag\texttt{>}, \Gamma, m) & = \{P \mid (x;\ \text{for}\ P) \in \Gamma\} \cup \mathbf{E}(q, \Gamma, 1) \\
6. & \mathbf{E}(x, \Gamma, 1) & = \bigcup_{(x;\ -\ P) \in \Gamma} \{P/\texttt{descendant-or-self} :: \texttt{node}\} \\
7. & \mathbf{E}(x, \Gamma, 0) & = \bigcup_{(x;\ -\ P) \in \Gamma} \{P\} \\
8. & \mathbf{E}(/P, \Gamma, 1) & = \{/P/\texttt{descendant-or-self} :: \texttt{node}\} \\
9. & \mathbf{E}(/P, \Gamma, 0) & = \{/P\} \\
10. & \mathbf{E}(x/P, \Gamma, 1) & = \bigcup_{(x;\ -\ P') \in \Gamma} \{P'/P/\texttt{descendant-or-self} :: \texttt{node}\} \\
11. & \mathbf{E}(Step/q, \Gamma, m) & = Step/\mathbf{E}(q, \Gamma, m) \\
12. & \mathbf{E}(Step[Exp]/q, \Gamma, m) & = Step[\texttt{or}(\mathbf{P}(Exp))]/\mathbf{E}(q, \Gamma, m) \\
13. & \mathbf{E}(Exp_1\ op\ Exp_2, \Gamma, m) & = \mathbf{E}(Exp_1, \Gamma, m) \cup \mathbf{E}(Exp_2, \Gamma, m) \\
14. & \mathbf{E}(f(Exp_1, \ldots, Exp_n), \Gamma, m) & = \bigcup_{i=1,n}(\mathbf{E}(Exp_i, \Gamma, 0)/\mathbf{F}(f, i)) \cup \{\texttt{self} :: \texttt{node}\} \\
15. & \mathbf{E}(\texttt{if}\ q\ \texttt{then}\ q_1\ \texttt{else}\ q_2, \Gamma, m) & = \mathbf{E}(q, \Gamma, 0) \cup \mathbf{E}(q_1, \Gamma, m) \\
 & & \ \cup \mathbf{E}(q_2, \Gamma, m) \cup \{P \mid (x;\ -\ P) \in \Gamma\} \\
16. & \mathbf{E}(\texttt{for}\ x\ \texttt{in}\ q_1\ \texttt{return}\ q_2, \Gamma, m) & = \mathbf{E}(q_1, \Gamma, 0) \cup \mathbf{E}(q_2, \Gamma \cup \Gamma', m) \\
 & & \ \text{where}\ \Gamma' = \{(x;\ \text{for}\ P) \mid P \in \mathbf{E}(q_1, \Gamma, 0)\} \\
17. & \mathbf{E}(\texttt{let}\ x\ \texttt{=}\ q_1\ \texttt{return}\ q_2, \Gamma, m) & = \mathbf{E}(q_1, \Gamma, 0) \cup \mathbf{E}(q_2, \Gamma \cup \Gamma', m) \\
 & & \ \text{where}\ \Gamma' = \{(x;\ \text{let}\ P) \mid P \in \mathbf{E}(q_1, \Gamma, 0)\}
\end{array}
$$

Figure 8.3: XQuery path extraction

In order to apply the previous analysis to infer a projector for q, we first extract a set of XPath$^\ell$ expressions from q, denoting the data needs for q. This set of paths is extracted from the query by the extraction function **E**, whose definition is given in Figure 8.3. The extraction function has the form $\mathbf{E}(q, \Gamma, m)$. The first parameter is the query at issue. The second parameter Γ is an environment that keeps track of bindings of the form $(x;\text{ for } P)$ or $(x;\text{ let } P)$, whose scope q is in (see the definition of Γ' in the last two lines of Figure 8.3, and observe, by a simple induction reasoning, that environments contain paths already in XPath$^\ell$). Finally, m is a flag indicating whether q is a query that serves to materialise a partial or final result ($m = 1$), or that just selects a set of nodes whose descendants are not needed ($m = 0$). Thus, the set of path expressions (possibly containing qualifiers) extracted from a top-level query q is $\mathbf{E}(q, \emptyset, 1)$.

Once the set of paths are extracted from a query q, we use it to infer a projector for q according to rules in Section 8.4.2. Formally, for each P_i extracted from q we deduce a projector π_i, and use for the whole q the union of these projectors (projectors are closed by union). Also, note that the extracted path of a closed query will not contain free variables since possible free variables are persistent roots that must be solved before the analysis.

Most of the rules in Figure 8.3 are not difficult to understand, therefore only few of them deserve further commentary. The flag is needed since each path determining the result ($m = 1$) must be extended with `descendant-or-self`, in order to project on all nodes needed in the query result. This is done by the lines 6, 8, and 10 of the definition. Expressions are dealt in a way similar to the path extractor **P** of Section 8.3.3; the extractor **P** itself is used in line 12 to produce simple paths (where we used the notation $\texttt{or}(\{P_1, ..., P_n\})$ for $P_1\texttt{or}\ldots\texttt{or}P_n$, and omitted the—straightforward—rules for single step paths). Also note that when a result is computed (lines 2 and 5) paths in "for"-environments are added ("let" are added only if their binding variable is used).

These rules subsume and enhance the technique of Marian and Siméon [MS03]. In particular, (i) the technique we use to exclude useless intermediate paths is simpler and more compact, (ii) we do not need to distinguish between two kinds of extracted paths but, more simply, we always manage a unique set of path expressions, and (iii) last but not least, our path extractor can be used even if the user cannot access an XQuery to XQuery-Core compiler, which is necessary for [MS03].

Before applying the extraction function **E** to a query q we apply some heuristics that rewrite q so to improve the pruning capability of the inferred paths. Among these heuristics the most important is the one that rewrites

```
for y in Q/descendant-or-self::node
  return if C(y) then q else ()
```

into

```
for y in
  Q/descendant-or-self::node[C(self :: node)]
return q
```

whenever $C(y)$ is a condition referring only to y and does not use external functions ($C(\texttt{self} :: \texttt{node})$ is obtained by replacing `self :: node` for all occurrences of y free in

C). If we apply **E** to the first query, then a path ending by `descendant-or-self::node` is extracted thus annulling further pruning: the entire forest selected by Q is loaded in main memory. This also happens with the approaches of Bressan *et al.* [BCL$^+$05] and of Marian and Siméon [MS03]. In our and Marian and Siméon's approach the query can be rewritten as above (this is not possible in [BCL$^+$05] since their subset of XQuery does not include predicates). However, Marian and Siméon's path based pruning degenerates (no further pruning is performed) also for the second query, since the `descendant-or-self::node` ends up in the set of pruner paths, thus selecting all nodes. This is because their approach cannot manage predicates. In our approach instead predicates are taken into account and therefore only nodes satisfying $C(y)$ are kept by the projector, thus yielding a very precise pruning.

It is important to stress that despite their specific form the first kind of queries is very common in practice since they are generated from XQuery→XQuery-Core compilation of a non negligible class of queries (for instance Q13 of the XPathMark) or when rewriting upward axes into downward ones. This latter observation shows that the application of rewriting rules of [OMFB02] to extend Marian and Siméon's approach to upward axes is not feasible since the rewriting may completely compromise pruning.

8.6 Experiments

We have implemented a complete version of the algorithm defined for full XPath. The code is written in OCaml, uses the PXP library for parsing XML documents, and its correctness was verified for all tests. After the path extraction of Section 8.5, it performs the rewriting presented in Sections 8.3.3 and 8.4.3, and the static analysis defined in Section 8.4. The latter is extended to deal with attributes, with the wild-card test `element()`, with `{descendant,ancestor}-or-self` as well as `{preceding,following}-siblings` axes, and with absolute paths. It also uses a couple of heuristics. One heuristic rewrites the DTD E so that every name Y defined as $Y \rightarrow$ *String* occurs exactly once in the right hand side of an edge of E; this enhances the precision of pruning by reducing the number of conflicts on the leaves of the tree. The other heuristic keeps track of the depth of elements in the paths in order to improve pruning, especially in presence of recursive DTDs (this latter heuristics could be embedded in the formal treatment, but we preferred to keep it simpler). Pruning is then performed *in streaming* and merely consists of a one-pass traversal of the document. We also added an optional validation option, that makes it possible to prune the document while validating it. Programs that use an external validator can therefore prune their document without any overhead.

We performed our tests on a GNU/Linux desktop, with 3GHz processor, 512 MB of RAM and a single S-ATA hard-drive, using DTDs, document generator, and queries of XMark and XPathMark (the latter is interesting because its queries use all the available axes). Queries were processed by the latest version of Galax. Virtual memory (a.k.a. swap) was disabled to test memory limits.

For what concerns the overhead of the optimisation, tests confirmed that it is always negligible, both in memory and time consumption: the only noticeable overhead is pruning time, which is linear in the size of the pruned document, but can be

	QM03	QM06	QM07	QM14	QM15	QM19	QP01	QP02	QP03	QP04	QP05	QP06
Original Document Size (MB)	930	2048*	1100	202	2048*	964	112	313	258	291	123	190
Pruned Document Size(MB)	25	5,3	42	139	24	24	89	50	46	50	98	133
Main Memory Usage (MB)	374	90	380	512	245	512	391	399	433	434	418	485
Gain in Size (% of original)	2.5	0.3	3.4	69.6	1.15	2.5	80.4	15.7	17.5	16.8	80.4	69.6
Gain in Speed (× faster)	17.8	110.1	28.2	3.9	62.6	7.5	1.5	3.6	3.7	4.3	1.5	2.9

	QP07	QP08	QP09	QP10	QP11	QP12	QP13	QP21	QP23
Original Document Size (MB)	168	123	459	123	369	134	79	224	403
Pruned Document Size(MB)	123	99	35	98	28	107	78	152	42
Main Memory Usage (MB)	467	466	466	483	456	460	504	459	465
Gain in Size (% of original)	73.2	80.4	7.5	80.4	7.5	80.4	98.2	67.9	10.4
Gain in Speed (× faster)	2.6	1.1	4.9	1.6	4.2	1.6	1.0	3.6	3.6

*: biggest file the XMark generator was able to produce.

Table 8.1: Sizes (in MBytes) of the biggest document processed thanks to pruning, size of its pruned version, and memory used to process the latter. Percent of the pruned document and speedup of the execution time for a 56MB document.

embedded in document parsing and/or validation (e.g., for 60MB documents computing the projector took around 0.5s while pruning and saving the pruned document to disk was always below 10s). These results were confirmed by further experiments on large DTDs (e.g. XHTML) and long XPath expressions (twenty steps or so).

In Table 8.1 we report the result of our test (we omit in this presentation queries with similar behaviour).

Projector efficiency. The fourth line of Table 8.1 reports the effect of inferred projectors and it is an indicator of the selectivity of the query. For several XMark queries the size of the pruned document is around 70-80% of the size of the original document. This is due to the fact that XMark documents contain mixed-content `<description>` elements which account for about 70% of the total size. Thus, queries whose execution requires the whole content of `<description>` elements, preserve a large part of the file. On the contrary, for very selective queries like QM06, 99.7% of the document is discarded. Finally, for queries that are very little selective, like QP13, the whole document has to be kept. It should be noted in Table 8.1, fourth line, that for all XMark queries but QM14 we could prune more than 95% of the original document.

Execution time and memory occupation. The comparison of performances of the Galax query engine on an original document and its pruned version is given in Figures 8.4 and 8.5, which respectively report the processing times and main memory occupation for documents of 56MB. They show that time and memory gains are similar.

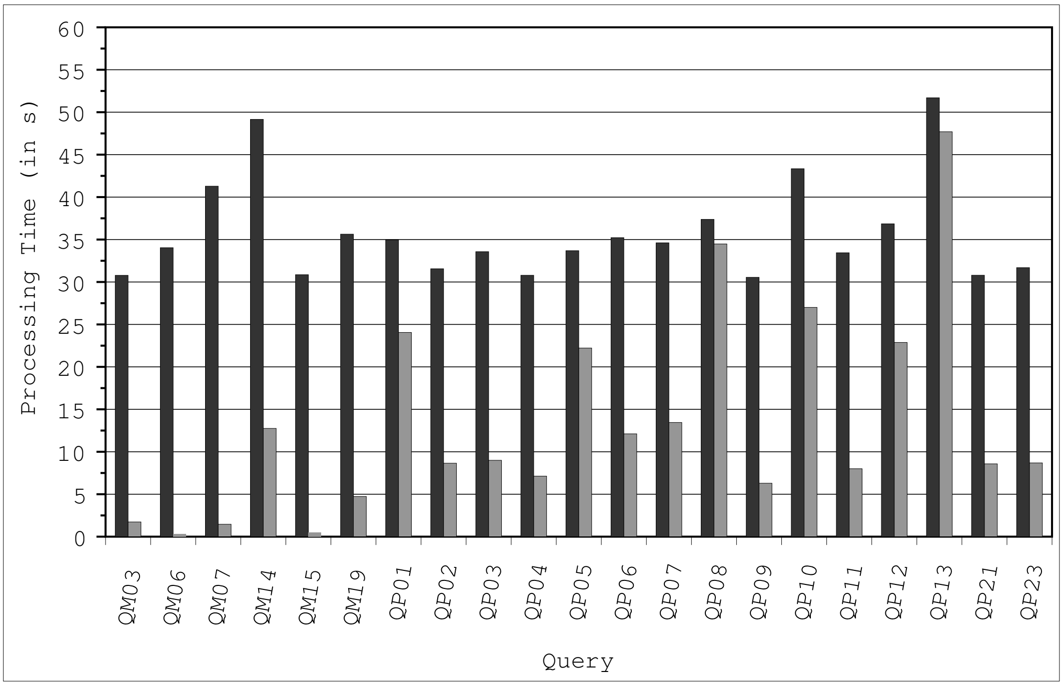


Figure 8.4: Processing time of a query on original (56MB) and pruned documents

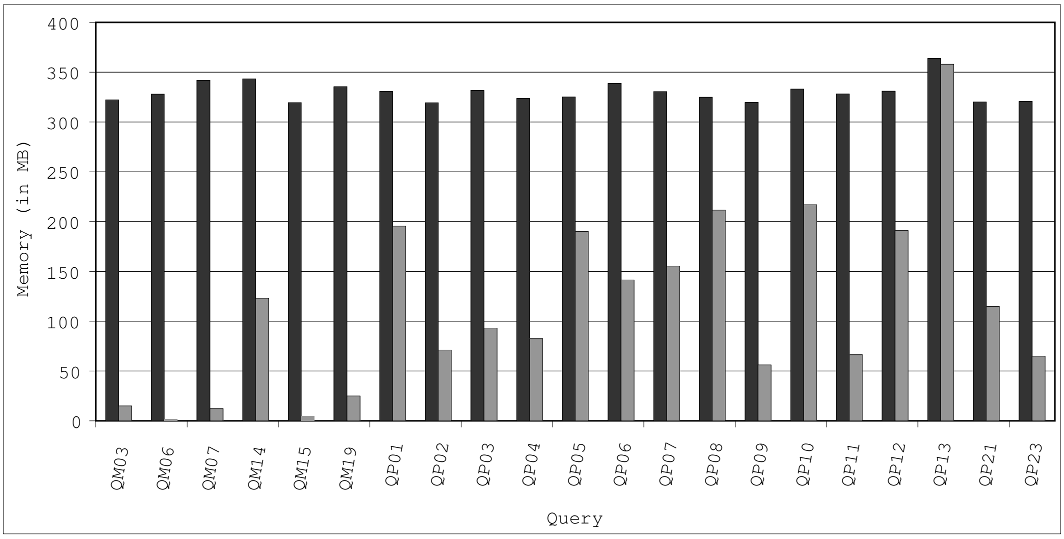


Figure 8.5: Memory used to process a query on original (56MB) and pruned documents

These gains translate in practice into much faster executions and the possibility to process much larger documents. The improvement can be measured by looking at the first and last lines of Table 8.1. The first line reports the size of the largest document it was possible to process thanks to pruning. This must be compared with the fact that, for all queries, the largest document that can be processed without pruning is 68MBytes large. The last line reports how many times the execution on a pruned document is faster than the execution on the original document. It is important to note that, depending on the nature of the query, the gain can be much higher than the proportion given by the percent of the size of the pruning. For instance, for queries such as QM14, QP6, and QP21 the size of the pruned document is two-thirds of the size of the original document, but they can then be processed from three to four times faster and, as Figure 8.5 shows, using three times less memory than when processed on the original. The latter is a huge gain when one knows that memory usage is one of the main bottlenecks for real life query processing (e.g., in DOM-based implementations of XPath or XSLT processors).

Quite informative, as well, is the data in the second line of Table 8.1 which reports, for each query, the size in MB of the maximum pruned document. It is interesting to see that, while the maximum size for an unpruned document is 68MB, we can process documents for which the projection has a size of 152MB (on disk). This is due to the fact that projecting a document not only reduces its size but also its *complexity* by reducing the number of types of nodes. This simplification of the document reduces the amount of extra-information the query engine has to keep for each node and, consequently, its memory usage. More precisely, the benefit of pruning out some (types of) nodes is twofold: first, the fan out of the document is reduced and this may impact memory usage for engines that chase sibling pointers and, second, the number of element names is reduced, which may reduce memory occupation when shredding.

These results are a clear-cut improvement over current technology. While we cannot directly compare processing performances since no implementation of the other pruning approaches is publicly available, we want to stress two points: (*i*) with one exception (QM14) the amount of pruning on common experiments is always equal or better with our approach than the others and (*ii*) performing pruning never is a bottleneck in our case thanks to fact that our solution consists of a single bufferless one pass traversal of the input document (on our 512MB machine we were able to efficiently prune arbitrary large documents, while in case of [MS03] pruning can end up using as much memory as the execution of the query).

Part IV

Conclusion

Chapter 9

Conclusion

9.1 Summary

We have studied in this dissertation a wide range of techniques to define, precisely type and efficiently implement iterators for XML. A fair amount of our work consists in the definition of a language of combinators, named *filters*. This filters achieve balance between the following conflicting requirements:

Expressivity: filters were designed to enrich a given host language with iterators. We made very few assumptions on the host language, namely that it provides a product constructor, atoms, a type system with subtyping, and a pattern algebra. Many languages fulfill these requirements: ℂDuce, XDuce, ML, Haskell,.... Despite their general design, filters allow the programmer to define very precise and specific transformation operators on data-structures, providing a way to arbitrarily iterate on an input while maintaining the strong and desirable property of termination. The scope of transformations one can express with filters goes well beyond hard-coded iterators (*à la* ℂDuce) or even Hosoya's regular expression filters since they can encode, for instance, a non-trivial subset of XPath. We also want to stress out that, since there is no need for the host language to be XML-oriented, filters can be used to extend a generic language with XML transformations, or, more generally with generic and/or precisely typed iterators such as SYB ([LP03, Läm07]) or the work on pattern calculus by C. Barry Jay ([Jay04]).

Precise typing: despite their great expressive power, filters still enjoy a precise typing. Indeed, since it was a requirement that filters could encode flattening or XPath expressions, we had to deal with the fact that the most precise output type would escape regular languages. We introduced a type-system, based on the *abstract evaluation* of a filter on a type rather than on a value. The main property of our type-system is that it accounts for any valid regular approximation when no better regular type exists for a given filter (which can occur in the setting of forward typing) and of course, that it provide type safety to filters through a classical subject-reduction property and progress.

Type inference: we exhibited a practical typing algorithm for filters, relying on typing annotations that are in practice very light. In particular, annotations are very well localised which permits to study their inference separately. We proved that the algorithm was sound and complete with respect to the type-system. A particular case of annotation inference has been studied for the case of filter encoding XPath expressions.

The formal developments have been supported by an implementation of *filters* into the ℂDuce language. Rather than being a consequence of the theoretical formalism, the implementation was developed alongside the theoretical framework, giving many useful insights on the behaviour of the type-system. Indeed, since our iterators were designed *for* XML, the type-system had to keep a foot in the real world, having to deal with types such as the XHTML or DocBook DTDs or Schemas. The size and complexity of such types, compared to what is found for instance, in the ML world were a real obstacle to hand-written examples and whiteboard research. The prototype was a precious help to test various examples and ensuring that the work was going "in the good direction". Lastly, once the prototype was developed, we used it to study many design issues related to filters, and more generally to very precise type-systems. First of all, the problem of code modularity was particularly difficult since it seemed, at first, contradictory with type annotations and the strict semantics of filters (which only expects one argument, the value they transform). We provided two effective solutions to this problem, the first one by designing the language so that annotations are required only at the place of their application, and the second one by defining macro filters, which allow the programmer to write generic iterator filters and parametrize them with local transformations. We also improved the language by adding regular expression filters whose syntax is much more user-friendly than mutually recursive filters and a typed XPath extension, thus adding support for a well-known paradigm.

We believe that our work has been, with this respect, a good answer to one of the problem posed by Alain Frisch in the conclusion of his thesis ([Fri04b]): "Give a way for the ℂDuce programmer to define precisely typed iterators". However our desire to abstract ourselves from ℂDuce made our work reusable in various context: XDuce, ML, or, following the approach of [GLPS05] in an object-oriented language such as C♯.

From this main axis of research emerged an orthogonal, more practical, work on XML standards such as XPath and XQuery. Rather than being antagonist, these two aspects of our work shed some light on one another. The formal work on filters, their use as a *typed* compilation target for XPath hinted on how to perform type-based optimisations to address very practical concern such as memory occupation. On the other hand, working on – and thus formalizing – the semantics of XPath and XQuery gave invaluable insight on what features were necessary to an XML language which desire to stay on edge with the XML standards.

9.2 Future work

There are several possible continuations to our work on filters. We list them here, following the order of the chapters of this dissertation.

9.2.1 Dynamic semantics, expressivity

While we have carefully designed filters so that they always terminate and so that, in practice, they allow us to write the desired iterators (list and tree mapping, almost copying, XPath encoding), little is known on their expressive power. An approach would be to use a restricted set of expressions: constants, variables and constructors (thus forbidding function calls in expression filters) and compare such filters to *top-down* tree transducers with regular look-ahead (as introduced by Joost Engelfriet in [Eng77]). Indeed, filters are evaluated top down, and pattern matching precisely consists in regular tree recognition. However it is unclear how much the ability of patterns to capture variables and the use of composition enhance the expressive power of filters. Following the current XML trend, a comparison with MTTs ([EV85]) would be much wanted but seems rather challenging. The use of accumulators in the latter in particular, makes the writing of functions such as the flattening much simpler than with filters.

9.2.2 Type-system, approximations

We devised in Chapter 7 an algorithm to automatically infer some type for an XPath expression. Unfortunately, how to do so in the general case still eludes us. A first approach could be based on the work of Mark-Jan Nederhof [Nedoo], who defines an algorithm to approximate context-free grammar with regular ones. However, even in that case, the technique seems very dependent on the syntax of the types, rather than their semantics and would need further refinements to handle the large and complex types of XML in a satisfactory way.

9.2.3 Concrete language, compilation

Even if it performed well as a prototype, our implementation is based on a rather naive compilation of filters. Many things can be done to further enhance the performances of filters. First of all, once a filter is typed, it could automatically be rewritten to permit a recursive call to traverse a composition (which actually goes against the restriction on recursive filters and composition). This would allow us to apply the well-known deforestation techniques, introduced by Philip Wadler in [Wad90]. It would also provide an efficient execution model, with minimal creation of intermediary structures while retaining the expressivity of filters. Adapting the static analysis we made in Chapter 8 to filters, hence determining which part of a value is visited by a filter, also seems feasible, but tricky in the presence of ℂDuce types instead of DTDs. A more interesting approach would be to compile filters to a streaming language, such as XStream ([FN07]). XStream is a low level functional language, introduced by Alain Frisch and Keisuke Nakano, compiled with *streaming* in mind. Indeed, it allows a programmer to write an XML transformation which will use as few memory as possible, thus outputing a result as soon as it is ready. This approach is appealing since it allows one to write *any* XML transformation in XStream and do the tedious work of interleaving computations and I/O operations automatically. Since XStream does not have a type system, compiling filters to XStream programs would allow the programmer to benefit from both a safe and precise type system and an efficient (throughput and memory -wise) program.

9.2.4 XPath encoding

The encoding of XPath, based on the determinisation of an automaton may yield some combinatorial explosion. Since our encoding is very similar to the one found in [GGM$^+$04], we are affected by the same worst case scenario: for a given path p the number of states in the final automaton is $O(2^{a(p)})$ where $a(p)$ is the number of occurrences of `//*//x`, for any tag `x`. However, since in this case the code of the filter is not required to type it (as we rely on an ad-hoc typing algorithm), we can use a lazy determinisation process which would, in our setting, consist in a *just-in-time* compilation of the resulting filter.

9.2.5 Type projectors

On the side of type-projectors, several aspects of importance can still be developed. First of all, even if DTDs are widespread and represent a vast majority of typed document available, it seems desirable to extend our approach to XML-Schema. While this does not seem theoretically difficult, special care should be taken so as to handle the so-called local elements. Another aspect is to extend our approach to *untyped* documents. This can be done if a set of documents is known beforehand. Indeed, given a set of documents, it is possible to infer not only a DTD, but as Bex *et al.* have shown in [BNST06], a somewhat general DTD which could then be reused for all the documents in the set. Lastly, the pruning information could also be used in other contexts, e.g. to optimise disk access in the case of an XML DBMS. Indeed, if we know in advance which parts of the document are used and which are not, then this information can be used so as useful parts are clustered in the same area of the disk, thus reducing the disk access to the document.

9.2.6 To infinity… and beyond!

As we have shown, filters seem to be a very valuable language to express transformations of XML documents and type them, and illustrate, in some sense, how far we can go with forward typing[1]. However, if one really wants to achieve exact annotation-free typing, backward type inference seems the way to go.

Indeed, it provides some valuable insight onto what is actually practical with forward typing. The idea would be to reuse these techniques in the context of MTTs and backward type inference. Very informally (and as a wild conjecture) it should be possible to mix forward and backward type inference so as to make backward type inference more practical. Our encoding of XPath and the CDuce integration would also benefit to MTT-based approach. In particular, to the best of our knowledge no

[1] Not that we claim that our approach is the “best” possible forward approach, but it seems more than likely that any equivalently expressive language has to resort to some types annotations, either as we did to provide one possible approximation or to use more complex type-systems (non-regular tree languages, dependant types) for which completely automatic type inference do not exists.

implementation based on MTTs is currently available. If making the typing of MTTs practical is one goal, another one, perhaps more challenging is to actually design a full fledged language for XML based on MTTs. This means taking into account—besides typing which is the main focus of theoretical languages—user friendly syntax and error messages, efficient compilation, and bindings with external languages to allow one to use system libraries, and so on. Last, but not least, such a language would need to make use of a great deal of research led by the database community on *e.g.* query optimization, persistency, streaming, distributed querying, memory efficiency.... Indeed, as we have seen with the study of type-projectors, typing can be used to further enhance already known and efficient database techniques.

Creating such a full fledged language is still, one of the most daring challenges in the field of XML programming.

Appendix

Langage de combinateurs pour XML : conception, typage, implantation

(résumé étendu)

A.1 Contexte

Depuis l'avènement des réseaux, en particulier de l'internet, est apparu un besoin croissant *d'échange* et de *partage* des données. Par exemple, nombreux sont les sites web permettant d'*agréger* le contenu de plusieurs autres sites. Un site web , actuellement, n'est plus une collection de pages statiques (au contenu figé) mais plutôt un ensemble de pages dynamiques, dont le contenu est actualisé en permanence, en réutilisant des données externes. Un site d'actualités, par exemple, pourra agréger les nouvelles venant de différents sites d'agences de presses ou de journaux et les présenter de manière unifiée à un utilisateur. Ce même site pourra, à son tour, devenir une *source* et voir ses données récupérées, sélectionnées et réorganisées par un site tiers.

Ce type de sites et plus largement d'applications interconnectées doit son succès à l'adoption d'un format commun de représentation des données, le format XML (eXtensible Markup Language). XML est une spécification, standardisée par le W3C (World Wide Web Consortium) décrivant comment organiser un ensemble de données sous forme textuelle (voir [XML]). Un exemple, représentant un carnet d'adresses est donné à la figure A.1. Comme on peut le voir, un tel document est composé de :

- texte brut (tel que `0123456789`)
- balises (par exemple `<contact group=''Work'' >` et `</addressbook>`)

Dans une balise telle que `<contact group=''Work''>`, `<contact>` est appelé une *étiquette* (*label*) et `group` un *attribut*. De plus, `<contact>` est une balise *ouvrante* alors que `</addressbook>` est une balise *fermante*. Le standard ne fixe pas le nom des étiquettes ou des attributs : c'est à l'utilisateur de définir une sémantique pour les documents, en définissant les balises, les attributs, leur ordre, ...

La spécification ([XML]) n'impose en effet que peu de contraintes sur un document. Par exemple, elle impose des contraintes de « bas niveau » sur l'encodage des caractères (XML ayant été conçu pour fonctionner avec des encodages complexes tel qu'Unicode). Pour ce qui est des contraintes de structures, la seule imposée par la norme est que le document représente un arbre. Plus précisément :

- Il doit y avoir une balise englobant tout le document (par exemple `<addressbook> ...</addressbook>` dans la figure A.1). Une telle balise est appelée *racine* du do-

```
<addressbook>

  <contact group="Work">
    <name>
     <first>Giuseppe</first>
     <last>Castagna</last>
    </name>
    <phone>0123456789</phone>
    <email>gc@pps.jussieu.fr</email>
  </contact>

  <contact group="Family">
   <name>
    <first>Yohanna</first>
    <last>Nguyễn</last>
   </name>
   <phone>0987654321</phone>
   <email>yoh@yoh.org </email>
   <address>
    <nb>4</nb>
    <street>Rue du Yahourt</street>
    <zip>75000</zip><city>Paris</city>
   </address>
  </contact>

</addressbook>
```

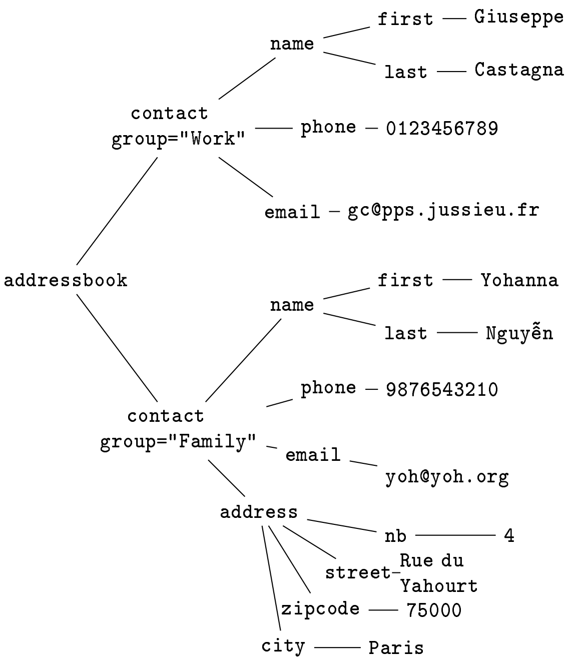


Fig. A.1 – Un exemple de document XML

cument.

- Une balise ouvrante doit être fermée par une balise fermante correspondante (ayant la même étiquette, précédée de /). De manière équivalente, on peut dire que les balises doivent être bien parenthésées.

Cette spécification donne un cadre générique pour structurer des documents, et sépare ainsi leur sémantique de la représentation textuelle. Il est en effet possible d'écrire des bibliothèques génériques permettant de parser et afficher des documents XML sans avoir besoin de connaître la sémantique de ces derniers. Cette particularité à fait d'XML un standard de choix, et les types de fichiers utilisant XML comme format de représentation de données sont légion. On peut par exemple citer Xhtml (une version compatible XML d'Html), le format de fichier Svg utilisé pour représenter des images vectorielles, le format OpenDocument, conçu pour stocker des fichiers d'application bureautique tels que documents mis en forme, feuilles de tableurs, présentations. XML est aussi un format de choix pour les fichiers de configuration. Un autre cas d'utilisation d'XML est celui des services web. Ces services sont des applications réseau qui acceptent des requêtes et renvoie les résultats correspondants encapsulés dans un document XML. Par exemple, un site de commerce en ligne peut fournir un service web permettant d'interroger son catalogue. Cela permet au service web d'utiliser XML pour publier des résultats complexes, organisés selon une structure riche mais qui reste cependant aisée d'utilisation pour un client, qui peut utiliser n'importe quelle bibliothèque XML générique pour lire un tel résultat.

Évidement, il faut pouvoir donner des contraintes *sémantiques* à un document. Par exemple, on veut pouvoir préciser quelles étiquettes peuvent apparaître dans un document, leur ordre, leur imbrication, *etc.* Il existe plusieurs moyens de décrire de telles contraintes. Des informations supplémentaires telles que spécifiées par des DTD (Document Type Definition, [DTD06]), un fichier Relax-NG ([Rel]) ou des XML-Schemas ([XSc]) peuvent être ajoutée à un document XML. On appelle communément de telles informations, *types*, *schémas* ou encore *contraintes*. Il est possible d'utiliser un *validateur* pour vérifier qu'un document respecte bien les diverses contraintes qui lui sont associées. La figure A.2 donne un exemple de tel type XML. Plus précisément, la figure représente une DTD qui spécifie des documents similaires au carnet d'adresse présenté figure A.1.

```
<!ELEMENT addressbook  (contact*)>
<!ELEMENT contact      (name, (phone|email)+,address?)>
<!ATTLIST contact      group CDATA #REQUIRED>
<!ELEMENT name         (first,last)>
<!ELEMENT phone        (#PCDATA)>
<!ELEMENT email        (#PCDATA)>
<!ELEMENT first        (#PCDATA)>
<!ELEMENT last         (#PCDATA)>
<!ELEMENT address      (nb,street,zip,city)>
<!ELEMENT nb           (#PCDATA)>
<!ELEMENT street       (#PCDATA)>
<!ELEMENT zip          (#PCDATA)>
<!ELEMENT city         (#PCDATA)>
```

FIG. A.2 – Un exemple de type XML (une DTD pour un document *addressbook*)

Ce type[2] consiste en un ensemble de définitions, une pour chaque élément pouvant apparaître dans un document de ce type. Les définitions sont représentées par des expressions régulières. Par exemple, un `<addressbook>` contient une séquence possiblement vide de `<contact>` (dénoté par l'étoile de Kleene « * »). Le contenu d'un `<contact>` est d'abord un `<nom>`, suivit par une séquence non vide (dénotée par le « + ») de `<phone>` ou `<email>` (l'alternance étant dénotée par un « | ») et finalement un élément `<address>` optionnel (dénoté par « ? », la mise en séquence étant notée « , »). `CDATA` (dans la définition de l'attribut) et `#PCDATA` sont des fragments de texte brut.

Le lecteur familier avec la théorie des langages formels peut aisément voir qu'une DTD n'est autre qu'une grammaire régulière d'arbres. De plus, bien que plus expressifs que les DTDs, les schémas, (XML-Schema ou Relax-NG) sont exprimables par une grammaire régulière d'arbres. Nous présenterons par la suite un formalisme équivalent, celui des expressions régulières de types (*regular expression types*). Vérifier qu'un arbre (un document) donné est une production d'une grammaire particulière (d'un type) est aisé. Par exemple, il est montré dans [SV02] que pour une classe de DTDs, la validation peut se faire en temps linéaire (en la taille du document) et mémoire constante. Plus généralement, la validation d'un document par rapport à une

[2] nous utilisons par la suite les termes types, schémas et contraintes de manière interchangeable.

DTD peut être faite en temps linéaire et en espace logarithmique (en la taille du document; il suffit pour cela de garder une pile égale à la profondeur max du document, vu comme un arbre).

A.2 Programmer avec XML

S'il est une chose de spécifier des documents XML, c'en est une autre que de les *manipuler*. En effet, comme des documents XML sont une manière d'organiser des données, il est nécessaire de pouvoir *interroger* le document pour extraire cette information, *transformer* l'information pour produire un *résultat* et finalement *publier* (ou présenter) ce résultat.

La manière la plus simple de faire est de considérer un document XML sous ça forme la plus brute : un fichier texte. Cette technique atteint rapidement ses limites. En effet XML étant un standard très complexe et verbeux, écrire à la main un parseur de document se révèle être une tâche ardue. De plus, les contraintes de documents ne sont pas du tout considérées et *a fortiori* vérifiée par une telle méthode. Il est douteux qu'une application sérieuse basée sur XML puisse être écrite de la sorte.

Si l'on se place d'un point de vue plus abstrait, les documents XML ont deux aspects —le contenu et le type— que l'on peut retrouver dans un langage de programmation. En effet, un langage peut fournir un support syntaxique pour XML (aspect valeur) et il peut garantir les contraintes du document (aspect typage).

Les différentes combinaisons de ces deux aspects permettent de classer les langages utilisés pour manipuler des documents XML. Premièrement on trouve les langages sans support syntaxique ni sémantique particulier. Ce sont les langages génériques, tels que Perl, Python, C,.... Ces langages manipulent des documents XML au moyen de bibliothèques spécialisées qui permettent de parser, manipuler et valider des documents XML. L'intérêt de cette approche réside dans le fait que ces langages sont largement supportés, disposent de nombreuses bibliothèques nécessaires à l'écriture de programmes complexes (gestion du réseau, appels systèmes, bibliothèques graphiques,...). Cependant, l'interface entre le « monde XML » et le langage lui-même est limitée au stricte minimum. L'écriture de ces programmes est donc souvent malaisée (pas de support syntaxique) les programmes difficiles à déboguer (pas de support sémantique; si une erreur se produit du côté XML, elle ne peut pas être détectée par le compilateur du langage, et produira le plus souvent une erreur à l'exécution).

Mieux adaptés sont les langages possédant une algèbre de types suffisamment riche pour pouvoir y encoder des contraintes de document XML. Deux tels exemples sont Java et OCaml. Cette technique, connue sous le nom de *data-mapping* consiste à encoder certaines contraintes XML dans le système de type du langage utilisé. Cela peut être fait au moyen d'une hiérarchie de classe (voir [JAX] pour un exemple d'implantation en Java) ou par d'autres moyens (par exemple, [Bal06, OCS] encodent des contraintes XML dans les *variants polymorphes* d'OCaml). Ces techniques permettent d'utiliser la *logique* du compilateur du langage généraliste pour détecter des erreurs XML. Elles se limitent cependant à la portion des contraintes XML exprimables dans le système de type du langage considéré.

À un niveau encore supérieur, on retrouve les standards du W3C (tels que XPath, XQuery ou XSLT). Ceux si sont conçus spécialement pour la manipulation de documents XML. Ils permettent donc d'écrire facilement des transformations XML. De plus, étant « ciblés XML », ils peuvent être optimisés spécifiquement pour cette tâche. Ils ne bénéficient cependant pas toujours des systèmes de types permettant de détecter statiquement la violation de contraintes XML.

La dernière catégorie, dans laquelle se placent les travaux de cette thèse est celle des langages statiquement typés pour XML.

A.2.1 Langages statiquement typés pour XML

La manière la plus sûre (et selon nous la plus élégante) de programmer avec XML et de « considérer les types sérieusement ». En effet, dans un système de type XML, si l'on peut vérifier qu'une transformation (une fonction) a le type $t \rightarrow s$, alors on a la garantie que la transformation renverra *toujours* un document de type s pour une entrée valide de type t. Toute validation du document de sortie devient donc inutile. De plus, une erreur de *typage*, permet de localiser précisément, au sein du code source, la ligne incriminée; celle qui viole une des contraintes du type de sortie.

Cette approche a été initiée par Haruo Hosoya ([Hos00, HP01]) et a notamment consisté en la réalisation du langage XDuce. Dans ces travaux, Hosoya explique que les DTDs et les divers schémas XML peuvent être représentés par un formalisme plus abstrait et générique : les expressions régulières de types (*regular expression types*). Ces types ne sont qu'une manière de représenter les langages réguliers d'arbres et possèdent des propriétés importantes de clôtures par opérateurs booléens. De plus, toutes les propriétés importantes (décision du vide, calcul du complémentaire, appartenance) sont décidables (cf. [CDG$^+$97]).

En XDuce, les documents sont des valeurs de première classe (au même titre que les entiers, les chaînes de caractères,...le sont dans un langage ordinaire). On peut voir à la figure A.3 un exemple de code XDuce[3]

La première contribution importante d'Hosoya avec XDuce est la définition de *filtrage par motifs réguliers* (*regular pattern matching*). Ces derniers sont une généralisations des motifs que l'on peut retrouver dans plusieurs langages comme OCaml ou Haskell. Ces opérateurs permettent de sélectionner très précisément une partie de l'entrée (un document) de manière typée (on connaît le type exact de la sous-partie extraite). La deuxième contribution majeure est celle du *sous-typage sémantique*. Cette relation de sous-typage est particulièrement adaptée à la vérification de contraintes fines, telles que celle exprimable dans le contexte d'XML (ces deux aspects sont présentés formellement au chapitre 2).

XDuce a été étendu dans plusieurs directions. La première est due à Benjamin Pierce *et al.* avec leur travaux sur XTatic (voir [GLPS05]). Ces travaux consistent à ajouter au langage objet C♯ le filtrage par motif de XDuce. Ils ont en particulier donné lieu a plusieurs travaux connexes sur l'implantation efficace de tels motifs. Il est cependant nécessaire de remarquer que la fusion entre la relation de sous-typage

[3] Les syntaxes de XDuce et ℂDuce étant très proches, nous avons quelque peu modifié l'exemple pour n'avoir à présenter qu'une seule syntaxe, celle de ℂDuce qui sera introduite par la suite.

```
type city = <city>[ Char* ]
type zip = <zip>[ Char* ]
type street = <street>[ Char* ]
type number = <nb>[ Char* ]
type address = <address>[ number street zip city]
type email = <email>[ Char* ]
type first = <first>[ Char* ]
type last = <last>[ Char * ]
type name = <name>[ first last ]
type data = <contact group=String>[ name (phone|email)+ address ? ]
type addressbook = <addressbook>[ data* ]

<addressbook>[ <contact group="Work">[
                 <name>[
                        <first>[ 'Giuseppe' ]
                        <last>[ 'Castagna' ]
                       ]
                 <phone>[ '0123456789' ]
                 <email>[ 'gc@pps.jussieu.fr' ]
               ]
               <contact group="Family">[
                 <name>[
                        <first>[ 'Yohanna' ]
                        <last>[ 'Nguyễn' ]
                       ]
                 <email>[ 'yoh@yoh.org' ]
                 <address>[
                        <nb>[ '4' ]
                        <street>[ 'Rue du Yahourt' ]
                        <zip>[ '75000' ]
                        <city>[ 'Paris' ]
                         ]
               ]
]
```

Fig. A.3 – Définitions pour le type `addressbook` ainsi qu'un exemple de document.

sémantique et le sous-typage de C♯ ne pose pas de problème conceptuel; cette dernière étant très pauvre (héritage) et complètement renseignée par le programmeur (la relation est étendue lorsque le programmeur déclare que telle classe hérite de telle autre).

Une autre extension importante —dans le cadre de laquelle nous positionnons nos travaux— est celle du langage ℂDuce, définit par Alain Frisch dans sa thèse ([Fri04b]). Les contributions de cette thèse sont nombreuses : ajout des types flèche et enregistrement extensible à XDuce, compilation efficace du filtrage par motifs (cf. [Fri04a]), mais aussi, de manière plus pragmatique, interface entre ℂDuce et OCaml, représentation efficace des types de donnée XML,....

Au vu de tout ces travaux, il est nécessaire de se poser la question : peut on aller plus loin dans le typage d'XML? Ces langages sont-ils suffisants ?

A.2.2 « Y'a pas que le statique et le précis dans la vie »

À l'exception d'XTatic les langages cités précédemment sont fonctionnels. Il n'est donc pas difficile pour un programmeur OCaml (par exemple) de programmer en ℂDuce. Deux aspects manquent cependant :

- le polymorphisme;
- l'inférence de types.

Le problème vient du fait que les systèmes de types polymorphes traditionnels ne sont pas du tout adaptés à la programmation en XML. Nous illustrons cela à l'aide de la fonction `concat` —qui effectue la concaténation de deux séquences— et de son typage. Dans un système de type à la ML, une telle fonction a généralement le type :

$$\texttt{concat} : \alpha\ \texttt{list} \rightarrow\ \alpha\ \texttt{list} \rightarrow\ \alpha\ \texttt{list}$$

où le type α `list` est définit inductivement[4] par :

$$\texttt{type}\ \alpha\ \texttt{list} = (\alpha \times \alpha\ \texttt{list})|\texttt{[]}$$

Comme il est d'usage, ce type est quantifié universellement selon la variable de type « α »[5]. Ainsi, une telle liste est soit la liste vide, notée « `[]` », soit une paire formée d'un élément de type α et d'une liste. Dans une expression, cette variable de type peut être instanciée pour fournir par exemple le type « liste d'entiers » `int list`. On remarque que le fait de n'utiliser qu'une variable de type *force* tous les éléments de la liste à être du même type, quel qu'il soit. Cela est particulièrement inadapté à XML, où les séquences peuvent être hétérogènes. Par exemple, le contenu du type `contact` défini à la figure A.3 est dénoté par une séquence hétérogène : `[name (phone|email)+ address ?]`. Il apparaît donc clairement que le polymorphisme à la ML n'est pas suffisant pour représenter aisément des types XML. Bien sûr, un polymorphisme plus puissant tel que celui du Système F pourrait suffire, mais il est bien connu que la vérification de types pour ce système est indécidable ([Wel99]).

La situation pour les types ℂDuce n'est guère meilleure. En effet, en l'absence de polymorphisme, la fonction `concat` doit être typée comme :

$$\texttt{concat} :\ \texttt{[}\ \texttt{Any}*\ \texttt{]} \rightarrow \texttt{[}\ \texttt{Any}*\ \texttt{]} \rightarrow \texttt{[}\ \texttt{Any}*\ \texttt{]}$$

Cette fonction devant pouvoir être appliquée à n'importe quel type de séquences, il est nécessaire de la typer avec un type suffisamment générique pour englober toutes les séquences possibles, ou autrement dit, un sur-type de tous les types séquences. En typant la fonction de cette manière, il est possible de l'appliquer à n'importe quelle paire de listes, mais alors, le type du résultat devient très imprécis : on perd l'information des types d'entrée et on sait uniquement que le résultat est une liste quelconque.

Le comportement que l'on attendrait du système de type est illustré table A.1. À la première ligne de cette table, on remarque que la concaténation de deux listes

[4] Nous utilisons la syntaxe OCaml, similaire à celle de nombreux langages fonctionnels.

[5] Nous rappelons qu'un tel type se lit : $\forall\alpha.(\alpha \times \alpha\ \texttt{list})|\texttt{[]}$. Ce type de quantification est connu sous le nom de *forme prénexe*.

type de x	type de y	type de `concat` x y
`[ Any* ]`	`[ Any* ]`	`[ Any* ]`
`[ Int* ]`	`[ Char* ]`	`[ Int * Char* ]`
`[ Int* ]`	`[ Int Bool? ]`	`[ Int+ Bool? ]`
...	...	...

TAB. A.1 – Types attendus pour la concaténation de deux listes.

doit être une liste quelconque, de type `[ Any* ]`. Cependant, la concaténation d'une liste d'entiers et d'une liste de caractères (ligne 2) doit donner une liste contenant d'abord des entiers puis des caractères. Finalement (ligne 3), on souhaite conserver les cardinalités : concaténer une liste d'entier avec une liste comportant au moins un entier donne une liste contenant au moins un entier, ce qui apparaît dans le type du résultat sous la forme d'`Int+`.

Comme nous l'avons expliqué, ce type de polymorphisme est plus puissant que celui disponible dans les langages disponibles actuellement. En effet, on souhaite l'opposé du polymorphisme paramétrique (à la ML). On souhaite pouvoir *spécialiser* le type d'une transformation en fonction du type d'entrée, alors que le polymorphisme paramétrique consiste justement à ne pas spécialiser les variables de types. Il ne peut pas non plus être exprimé par du polymorphisme de sous-typage, car le type le plus générique est complètement non-informatif (`[Any*]` de l'exemple précédant). Enfin, même s'il est similaire au polymorphisme *ad-hoc* des fonctions surchargées, ces dernières ne peuvent traiter qu'un nombre fini de cas, spécifiés dans la signature de la fonction alors que notre typage doit pouvoir mettre en relation un nombre potentiellement infini de types d'entrée avec des types de sortie bien précis.

Une solution partielle, utilisée dans ℂDuce est de fournir un opérateur de concaténation *ad hoc*, noté « `@` ». Cet opérateur n'est *pas* une fonction ni un objet de première classe. Il n'a pas de valeur en soi. Cependant, utilisé dans une expression du style « l_1@l_2 », il est *re-typé*, en fonction du type de l_1 et l_2. Ceci est possible parce que la sémantique de l'opérateur est connue : il effectue la concaténation de ses deux arguments. Le typage consiste donc à concaténer les deux types séquences correspondants, obtenant ainsi un typage exact de l'expression. Ce typage est identique à celui illustré dans la table A.1. Ceci est l'une des idées directrice de nos travaux : l'opérateur est évalué sur les *types de ses arguments* pour calculer un type de sortie *précis*. Dit autrement, nous effectuons une *exécution abstraite* de l'opérateur sur son type d'entrée.

Cette solution est déjà mise en œuvre dans de nombreux langages XML typés : XDuce propose l'opérateur `map` pour itérer une transformation sur des listes, ℂDuce propose les opérateurs `@` (concaténation), `transform` (itération sur une séquence) et `xtransform` (itération sur un document XML) et XTatic propose `foreach` et `iterate` à des fins similaires. Tous ces opérateurs sont typés précisément (au sens ou le type de sortie est calculé à partir du type d'entrée) et permettent d'exprimer certaines opérations de base sur les documents XML. Cependant, pour des transformations plus complexes (par exemple supprimer d'un document XHTML tous les éléments obsolètes) ces opérateurs ne sont plus suffisants. Les choix du programmeur à ce niveau sont alors limités. Il peut demander gentiment aux mainteneurs du langage de rajouter l'opérateur dont il a besoin, ce que les mainteneurs refuseront de faire[6]. Le programmeur peut aussi écrire lui-même les fonctions effectuant les transformations qu'il désire, mais il devra alors fournir lui même le type de sortie. Prenons l'exemple (en ℂDuce) d'une fonction retirant tous les liens hypertextes (balises <a>) d'un document XHTML. Comme nous l'avons dit précédemment, cette fonction ne peut être écrite au moyen d'un simple opérateur. Le programmeur écrit donc une fonction f effectuant la transformation. Comme ℂDuce ne possède pas d'inférence de types, le programmeur doit annoter sa fonction par le type d'entrée et le type de sortie. Le type d'entrée est simple, c'est le type XHTML des documents du même nom. Ce type étant courant, on peut supposer qu'il est prédéfini dans la bibliothèque standard de ℂDuce. À l'inverse, le type de sortie est bien plus fastidieux à écrire. Il s'agit en effet du type XHTML dans lequel toutes les occurrences de la balise <a> ont été supprimées. Sachant que le type XHTML (et les types XML en général) est très verbeux (plusieurs dizaines de tags différents) cette opération est malaisées et propice à l'introduction d'erreurs. Le programmeur peut aussi choisir de donner à sa fonction le type de sortie XHTML qui est un sur-type du type exact. C'est un type valide (un document XHTML sans lien est bien un document XHTML). Cependant ce n'est pas un type assez précis, le type de sortie ne reflétant pas le fait que le document de sortie ne contient pas de lien. Nos travaux de thèse se proposent de répondre à cette problématique : écrire des transformations *complexes* typées précisément et automatiquement par le compilateur.

A.2.3 Une solution

L'une des solutions possible à ce problème, qui est celle que nous développons dans cette thèse, est de définir un langage restreint de combinateurs, suffisamment expressifs pour écrire des transformations complexes de documents XML mais suffisamment simples pour pouvoir être typés précisément. Plus exactement, nous proposons d'ajouter à un langage « hôte » préexistant (tel que ML, ℂDuce, ...) un sous langage de manipulation de document XML. Cette technique a été explorée par Haruo HOSOYA par l'ajout « d'expressions régulières de filtres » (*regular expression filters*, [Hos04]) à XDuce. Dans cette approche, le langage de combinateurs est paramétré par les expressions et les motifs du langage hôte (dans le cas d'HOSOYA, les expressions et les motifs de XDuce). Cela permet au programmeur *d'itérer* une expression XDuce de manière particulièrement précise sur un document d'entrée, de capturer des sous-parties du document à l'aide de motifs et de les réutiliser pour produire un résultat.

Cette solution semble naturelle. En effet, il n'est pas rare d'étendre le noyau d'un

[6] En effet, ajouter un tel opérateur touche toutes les parties du langages : la syntaxe (l'ajout de nouveaux mots-clés qui peut rendre invalide du code existant), le typage (l'ajout d'une règle de typage spécifique à cet opérateur n'est souvent pas triviale), et enfin la compilation (rajouter de nouvelles constructions dans le langage peut casser certains invariants nécessaires à l'optimisation du code). Maintenir un tel code devient rapidement un cauchemar.

langage avec un sous langage, moins expressif mais ayant de meilleures propriétés de typage, d'optimisation, ... On peut citer par exemple le cas d'OCaml et de son système de modules, de ℂDuce et de son extension ℂQL (qui consiste en l'ajout d'une primitive `select from where` à ℂDuce), ou encore de C++ et de son langage de templates. Dans le problème qui nous concerne, la solution n'est cependant pas triviale, et demande de remplir un certain nombre de critères :

1. Le langage de combinateurs doit pouvoir appeler n'importe quelle expression du langage hôte. La conception du langage de combinateurs doit donc être indépendante du langage hôte choisi.
2. Le sous-langage doit être statiquement typé. Cela a deux conséquences importantes sur le système de types. Il doit (i) être capable de donner un type à toute expression de combinateurs —c'est à dire donner un ensemble d'expressions pour lesquelles l'évaluation n'échoue pas— et (ii) être capable de déduire un type de sortie précis pour chaque type d'entrée correspondant, et ce en *évaluant l'itérateur sur le type de l'entrée.*
3. Une conséquence de (ii) du point précédant est que le langage doit permettre de définir uniquement des itérateurs qui *terminent* lorsqu'ils sont appliqués à des types potentiellement infinis (l'infinitude d'un type peut par exemple être liée au fait que ce dernier soit récursif ou qu'il contienne des opérateurs de répétitions, telle que l'étoile de Kleene). En effet, il faut que la phase de *typage* termine, et donc il faut garantir que l'évaluation d'un itérateur sur un type termine toujours.
4. Le pouvoir expressif des combinateurs doit être suffisant pour exprimer des transformations XML courantes : concaténation, *map*, exploration d'arbres, expressions XPath, et ainsi de suite.
5. Le langage introduit ne doit pas compromettre les propriétés de modularité et réutilisation de code déjà présente dans le langage hôte.

Il y a évidement un conflit entre les points 3 et 4 : pouvoir expressif et terminaison sont deux aspects contradictoires, il faut donc trouver un équilibre entre ces deux points. En effet entre le fait de pouvoir simplement itérer un document dans l'ordre et écrire une transformation quelconque (donc avoir un langage de combinateurs Türing-complet), il existe un grand nombre de classes de transformations possibles. En considérant que des itérateurs de type *map* (qui traversent un document dans l'ordre, appliquant une transformation locale à chaque noeud) sont insuffisants pour travailler avec XML, nous posons, comme pré-requis que le langage de combinateurs soit capable d'exprimer l'aplatissement d'un arbre XML, c'est à dire donner, à partir d'un document d'entrée, la liste de tout ses noeuds. Il convient de noter que ce type d'opérations est classique dans le cadre de la programmation XML. Malheureusement, ce pré-requis qui nous semble minimal nous interdit à tout jamais d'espérer avoir une inférence de type *exacte* pour nos transformations. En effet, une propriété bien connue des langages d'arbres réguliers est que leur image par homomorphisme n'est pas régulière (voir à ce sujet [CDG⁺97]). Par exemple. considérons le type récursif suivant :

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

Ce type est l'union (dénotée par `|`) du singleton `[]` ou d'une liste de trois éléments, `<a>[]` en premier, `<b>[]` en dernier et une liste de type `T` en deuxième position. Bien que ce type soit régulier, son image exacte par la transformation d'aplatissement n'est pas régulière car c'est l'ensemble $\texttt{S} = \{\,\texttt{[<a>[]}^{\texttt{n}}\ \texttt{<b>[]}^{\texttt{n}}\texttt{]}\,|n \geq 0\}$. Un autre point important est qu'en général il n'existe pas de *meilleure* approximation régulière pour de tels ensembles. Par exemple l'ensemble de valeurs précédant peut être approximé par la suite de $\texttt{S}_i$ définie ci-après, où $\texttt{S}_{i+i}$ est strictement plus précis (au sens de l'inclusion) que $\texttt{S}_i$:

$$\begin{array}{lcl}
\texttt{type S}_0 & = & \texttt{[<a>[]* <b>[]*]} \\
\texttt{type S}_1 & = & \texttt{[] | [<a>[] <b>[]] | [<a>[] <a>[]+ <b>[] <b>[]+]} \\
 & \vdots & \\
\texttt{type S}_n & = & \texttt{[] | [<a>[] <b>[]]| ...| [<a>[]}^{n}\texttt{<a>[]+ <b>[]}^{n}\texttt{<b>[]+]} \\
 & \vdots &
\end{array}$$

Ces deux points de typage rendent délicate la conception d'un tel langage, typé précisément et automatiquement.

A.3 État de l'art

Nous ne présentons ici que les différentes approches du problème. Un état de l'art détaillé se trouve dans la section 1.4 du présent manuscrit.

Dans cette thèse, nous présentons un langage de combinateurs, nommés *filtres*[7]. Les filtres sont suffisamment expressifs pour écrire des transformations complexes de documents XML tout en gardant une discipline de typage simple. Les filtres sont intégrés à ℂDuce, de la même manière que les filtres d'Hosoya sont intégrés à XDuce. L'intérêt est que le programmeur n'est pas limité à un langage restreint pour toutes les tâches « non-XML » et peut par exemple bénéficier des fonctions système et autres facilités fournies par ℂDuce et plus généralement par un vrai langage de programmation. Le programmeur peut donc définir ses propres itérateurs polymorphes et bénéficier d'un typage précis, tout en restant dans le cadre d'un langage (hôte) complet. Notons que dans toute cette thèse, nous faisons de nombreuses comparaisons avec le travail d'Hosoya, qui est le plus proche de notre formalisme.

Afin de mieux présenter nos travaux, nous donnons un aperçu des différents formalismes XML utilisés pour concevoir des langages de manipulation modulaires, précisément typés et expressifs. Les solutions à ce problème que l'on retrouve dans la littérature peuvent être divisées en quatre catégories.

A.3.1 Système de types polymorphes pour XML

De nombreux travaux entrent dans cette catégorie. Ils consistent en un rapprochement d'un système de type à la XDuce avec un système de type à la Ml. Ce rapprochement peut être profond : ce sont les travaux de Haruo Hosoya, Giuseppe Castagna et Alain Frisch, [HFC05] et de Jérôme Vouillon [Vou06] pour un

[7] appellation empruntée aux «filters» d'Haruo Hosoya.

rapprochement avec Ml et de Martin Sulzmann et Kevin Zhuo Ming Lu pour un rapprochement avec Haskell (cf. [SL06a]). Ces travaux ont en commun une modification importante des deux disciplines de typage pour faire cohabiter le polymorphisme paramétrique avec par exemple le sous-typage sémantique.

Le rapprochement peut aussi être plus superficiel, comme dans le cadre d'OCamlDuce (voir [Fri06]), qui juxtapose les deux systèmes de types (XDuce et Ml) dans un même langage. Une valeur a donc *soit* un type Ml (et peut donc être polymorphe) soit un type XDuce (et donc être précisément typée) mais le mélange est interdit.

Ces approches ont en commun qu'elles facilitent l'écriture de code polymorphe dans un langage XML (ou de manière équivalente qu'elle facilitent l'écriture de transformations XML dans un langage avec polymorphisme) mais ne permettent pas d'écrire des transformations *à la fois* polymorphes *et* précisément typées.

A.3.2 Itérateurs prédéfinis

Comme nous l'avons mentionné précédemment, ℂDuce, XDuce et Xtatic fournissent des itérateurs prédéfinis pour les transformations XML. Les avantages de ces derniers sont la précision de leur typage ainsi que leur polymorphisme (ils peuvent être appliqués à n'importe quel type d'entrée compatible et être re-typés précisément). L'inconvénient majeur est leur expressivité très limitée et le fait qu'ils ne facilitent pas la réutilisation du code ainsi que la modularité. En effet, de tels itérateurs ne peuvent être encapsulés dans une fonction, car il souffriraient alors du typage « manuel » de celle-ci et doivent être copiés/collés tout au long du code.

A.3.3 Langages d'itérateurs

Notre solution d'intégrer un sous-langage avec un langage hôte (ℂDuce dans notre implantation) soulève la question du choix du sous-langage à utiliser. En effet il existe de nombreux langages restreints, spécialisés pour XML. En premier lieu, XPath ([XPa]) est le standard pour la navigation dans un arbre XML. Cependant, XPath n'est qu'un langage de requête et ne permet que d'explorer un document, pas de le transformer. Il peut être vu comme un équivalent des patterns de XDuce, qui sont aussi une primitive de navigation dans un document. Un autre langage plus puissant est XQuery basé sur les primitives FLWR (`For Let Where Return`). XQuery fournit des constructions permettant d'extraire des sous-parties d'un document (au moyen de chemins XPath), d'itérer sur ces sous-parties et de créer de nouvelles valeurs en utilisant des sous-arbres capturés pendant l'itération. Ainsi, il semble être un bon candidat pour un langage de transformation. Cependant, XQuery est aussi très lié au modèle de donnée d'XML qui spécifie par exemple que chaque noeud d'un document doit avoir un identifiant unique. XQuery permet aussi (au travers d'XPath) de naviguer d'un noeud vers les parents de ce dernier, ce qui impose une représentation cyclique du document XML (pointeurs vers le père, la racine du document, ...). Ces aspects particuliers ne facilitent pas l'intégration d'XQuery avec un langage fonctionnel de type XDuce où le modèle de donné est beaucoup plus simple et ne peut être modifié aisément, en raison notamment des opérateurs de filtrage.

A.3.4 Transducteurs d'arbres et inférence de type arrière

Plus éloignée des standards, nous trouvons une littérature très riche sur les combinateurs XML, issue de la théorie des transducteurs d'arbres. Le plus grand atout des formalismes à base de transducteurs d'arbres est leur capacité à être typés *exactement* (*i.e.* sans approximation) au moyen d'une technique dite de typage arrière. Dans cette approche, étant donné une transformation f (exprimée par un transducteur) et un type *de sortie* s, on calcul le plus grand type d'entrée t (au sens de l'inclusion) tel que :

$$f(t) \subseteq s$$

Cela revient, en pratique à calculer l'image inverse de s par f :

$$t = f^{-1}(s)$$

La propriété des langages réguliers, que leur image par homomorphisme inverse est toujours régulière, garantit que le type t ainsi calculé représente exactement l'ensemble des entrées valides pour f. Cette technique a été appliquée à de nombreux types de transducteurs : *k-peeble transducers*, *macro-tree transducers*, ...

L'inconvénient majeur de cette technique est sa complexité en temps (une tour d'exponentielle en la taille de la transformation et du type). Même si cette approche est inutilisable actuellement en pratique, on peut noter les progrès substantiels de Helmut Seidl, Thomas Perst et Sebastian Maneth ([MPS07]) ainsi que d'Haruo Hosoya et Alain Frisch ([FH07]) qui peuvent typer certaine classes de transformations en temps polynomial et diminuent quelque peu la complexité dans le cadre général.

A.4 Contributions

Nous présentons ici les principales contributions de cette thèse, correspondant chacune à un chapitre du présent manuscrit. La première partie est composée d'une introduction ainsi que de rappels syntaxiques et sémantiques nécessaires aux développements formels. La seconde partie donne les définitions formelles du langage de filtres, leur sémantique, ainsi que leur système de type et un algorithme d'inférence de type. La troisième partie présente des résultats pratiques d'implantation, ainsi qu'un encodage d'XPath dans les filtres et une application de la discipline de typage des filtres à XQuery, en vue de l'exécution efficace de requêtes. La quatrième partie du manuscrit conclut notre étude et propose une discussion sur les travaux futurs. Nous présentons ici les deuxième et troisièmes parties, contenant les contributions.

A.4.1 Les filtres et leur sémantique (chapitre 3)

En prenant pour point de départ les motifs de ℂDuce ([FCB02] eux même étant une extension des motifs XDuce, [HP01]), nous les généralisons à des combinateurs et en faisons un langage à part entière, permettant de former des termes calculatoires, pouvant par exemple itérer un document d'entrée et le transformer. Si l'on pense aux motifs en termes d'automates d'arbres, alors les filtres ne sont rien d'autre que des *transducteurs d'arbres* : ils peuvent à la fois reconnaître (itérer) une valeur d'entrée et la transformer.

Notre calcul permet d'exprimer : les opérations usuelles sur les séquences et les arbres (renversement, concaténation, aplatissement, *map*,...) mais aussi d'encoder un fragment non trivial d'XPath, de même que des transformations XSLT . La nouveauté est l'introduction d'un opérateur de *composition*, absent du formalisme d'HOSOYA et rendant nos filtres strictement plus expressifs. Pour donner un aperçu du calcul, nous présentons le filtre effectuant la concaténation de deux listes :

Exemple A.1
Le filtre de concaténation :

$$\begin{array}{ccl} \texttt{concat} & = & (x,y) \to (x;f) \\ f & = & \texttt{`nil} \to y \\ & | & (z \to z,f) \end{array}$$

est équivalent à la fonction OCaml:

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

Bien que l'expression du filtre `concat` semble un peu complexe, il est aisé de la comprendre en regardant le code de la fonction OCaml `concat`, équivalent. Les filtres sont appliqués à une entrée unique. Le premier filtre, $(x,y) \to (x;f)$ est un filtre *motif* dans lequel le côté gauche de la flèche est un motif CDuce et le côté droit la continuation du filtre. Dans le cas présent, le motif (x,y) capture dans x le premier argument, dans y le deuxième et effectue la continuation dans l'environnement ainsi augmenté. De la même manière, dans le code OCaml, la fonction `f` est imbriquée sous la définition de `concat` de sorte que `x` et `y` sont visibles depuis `f`. La partie droite du filtre (la continuation) est une *composition*, symbolisée par « ; », de deux filtres. La composition de deux filtres $f_1;f_2$ appliquée à une valeur v consiste à calculer $f_2(f_1(v))$. Bien que cette opération semble naturelle, nous verrons par la suite que c'est d'elle dont est issue toute le pouvoir expressif du langage et par là-même la complexité du typage. Pour en revenir à notre exemple, les deux filtres composés sont le filtre *expression* x (qui est une expression du langage hôte) et le filtre f défini après `concat`. La composition consiste ici à calculer $f(x)$ et correspond exactement à l'appel `f x` que l'on trouve à la fin du code OCaml.

Le filtre f quand à lui, consiste en l'union (notée « | ») de deux filtres; de manière similaire aux branches de l'opérateur `match with` que l'on voit dans le code OCaml. La première branche de l'union est le cas de base du filtre récursif f. Si la première liste est vide, alors il retourne le deuxième argument, y. Si l'entrée est une liste (représentée par une paire, à la Lisp), alors l'identité $z \to z$ est appliquée au premier élément de la liste et f et récursivement appliqué au reste. Après l'appel à f, la paire des deux sous-résultats est renvoyée.

Cet exemple simple présente tous les filtres possibles : expressions de base, filtrage par motif, déconstruction de paire et reconstruction, union, composition et régularité (par appels récursifs). Nous montrons dans ce chapitre la propriété fondamentale de terminaison :

Théorème A.1 (Terminaison)
Soit f un filtre bien formé dont tous les sous-filtres expression terminent. Alors l'évaluation de $f(v)$ termine, pour toute valeur finie v.

A.4.2 Système de type (chapitre 4)

Comme expliqué précédemment, la discipline de typage pour les filtres n'est pas standard. Plutôt que de typer la définition d'un filtre (comme on le fait avec des fonctions), on les types à l'endroit de leur application, lorsque le type précis de l'entrée est connu. L'idée est donc d'évaluer le filtre directement sur le type d'entrée pour calculer un type de sortie. Dans certains cas, le type exact peut être calculé. Dans d'autre cas, le type de sortie exact n'est pas régulier. Plutôt que de choisir une approximation particulière, nous définissons un système de type qui, étant donné un filtre et un type d'entrée, infère toutes les dérivations régulières possibles. Plus précisément, nous définissons des jugements de la forme :

$$\Gamma \vdash f(t) = s$$

signifiants qu'un filtre f appliqué à une valeur de type t dans un environnement Γ renvoie une valeur de type s. Pour pouvoir exprimer le fait qu'il y a potentiellement plusieurs s valides (et pas de meilleur), nous définissons le système de manière à ce que pour tout s valide, il existe une dérivation de typage. Pour chaque approximation, il existe donc une façon de typer le filtre de manière à ce que le type de sortie soit exactement cette approximation. À l'aide de ce système, nous pouvons énoncer une première propriété fondamentale du système de type, la préservation du typage.

Théorème A.2 (Préservation du typage)
Soient Γ, f, t, s et v tels que :

$$\Gamma \vdash f(t) = s$$

et v et de type t. Alors, $f(v)$ est une valeur v' de type s.

Comme le théorème de terminaison nous donne la progression de l'évaluation (un filtre ne peut pas boucler à l'infini), nous pouvons en déduire la sûreté : un filtre bien typé appliqué à une valeur du bon type n'échoue jamais.

Un autre point important développé dans cette partie est celui de la précision de ce système et de son comportement vis-à-vis de la relation de sous-typage. Nous donnons plusieurs résultats techniques et exemples permettant de cerner la précision du système. Il est en particulier plus précis que celui d'Hosoya pour les filtres que les deux formalismes sont à même d'exprimer.

A.4.3 Algorithme d'inférence de type (chapitre 5)

Le système de type développé, bien que précis n'est pas utilisable dans une implantation. En effet, nous avons vu que pour un type d'entrée et un filtre, il peut admettre une infinité de dérivations valides et qu'il n'y a pas moyen, *a priori*, d'en choisir automatiquement une meilleure. Nous résolvons ce problème en ajoutant aux filtres des annotations de types. L'idée est de décorer le corps du filtre à certains endroits pour *forcer* l'algorithme à choisir l'une des approximations possible. Dit autrement, nous laissons le soin au programmeur d'annoter le filtre, donc de choisir lui même l'approximation qui lui convient. Le système, ramené à des filtres avec annotations devient algorithmique et donc utilisable en pratique. Concevoir un système avec des annotations n'est pas choses aisée. Nous nous attachons notamment à démontrer les points suivants :

- un filtre n'a pas besoin d'être annoté partout. Nous analysons précisément dans quel cas un filtre doit être annoté et quel est l'endroit précis où il doit l'être. Cette analyse fine nous permet d'affirmer qu'en pratique, les annotations ne sont pas pesantes pour le programmeur et que des filtres complexes peuvent êtres écrits avec peu ou pas d'annotation.
- Nous montrons bien sûr un théorème de correction de l'algorithme vis-à-vis du système de type général. Cela nous permet de montrer que l'algorithme est sûr.
- Nous énonçons aussi un théorème de complétude. La présence d'annotations rend ce théorème particulièrement important. Il spécifie que si un filtre est annoté avec des types venant d'une dérivation de typage du système, alors l'algorithme se comportera comme le système. Cela signifie que l'algorithme n'est qu'une instance particulière du système dont le programmeur aura guidé les choix, mais aussi que de mauvaises annotations ne permettrons jamais de typer un filtre, et donc que les annotations ne sont que des « aides » et non pas des opérateurs de cast (transtypage) aveugles.

A.4.4 Langage concret, implantation (chapitre 6)

La motivation de notre calcul formel est de donner un langage concret, utilisable pour l'écriture de transformations XML. Nous étudions donc (et fournissons une implantation de) l'intégration des filtres avec CDuce. Nous dotons les filtres d'une *syntaxe concrète* afin de rendre leur écriture plus aisée. Le filtre `concat` présenté précédemment devient donc :

Exemple A.2

```
let filter concat =
 $  (x,y)  -> ( ~{x} ; (
                         let filter f =
                          $  []  -> ~{ y }
                        | (  $  z  -> ~{ z }, f )
                        in f ))
```

Mis à part les caractères spéciaux `$` et `~` utilisés pour désambiguïser les motifs, les filtres et les expressions, nous introduisons un lieur « `let filter` »permettant d'écrire des filtres récursifs.

Bien évidement, l'un des points clés est la spécification des annotations. Nous ne pouvons, en effet, demander au programmeur de décorer le *corps* du filtre d'annotations de type, comme cela était fait dans le modèle formel. Faire ainsi casserait invariablement la modularité et la réutilisabilité du code, un filtre annoté se trouvant « marqué » par des types et donc inutilisable dans un autre contexte. Nous résolvons se problème en utilisant des annotations *tardives*. L'idée est d'annoter non pas la définition mais l'application du filtre, comme cela est illustré à la figure A.4. Nous y définissons le filtre d'aplatissement, `flatten` dont on sait qu'il nécessite des annotations. Deux annotations différentes sont données à la fin du code, à l'endroit ou `flatten` est appliqué à un argument. De cette manière, aucun type ne vient polluer la définition du filtre qui peut être utilisé dans plusieurs contextes différents.

Nous étendons aussi le langage avec de nouvelles constructions syntaxiques et sémantiques, exprimables dans l'algèbre de base des filtres mais facilitant l'écriture de programmes. De telles extensions sont les filtres paramétrés par d'autres filtres (nommés macro-filtres), les expressions régulières de filtres (similaires à celle d'Hosoya) ainsi que d'autres combinateurs utiles : première projection et seconde projection d'un produit, extraction du tag ou du contenu d'un document XML, ... Finalement, nous montrons que les filtres autorisent un modèle d'exécution relativement efficace.

A.4.5 Encodage d'XPath (chapitre 7)

Les expressions XPath étant fortement prisées dans la communauté XML, leur présence semble indispensable dans un langage pour favoriser l'adoption de ce dernier. Nous montrons comment encoder un fragment non trivial d'XPath, comportant les axes avant ainsi que les prédicats. Bien que le filtre correspondant à un chemin XPath ait besoin d'annotations, nous montrons comment inférer automatiquement une approximation suffisante dans le cadre d'XPath. En effet, le fait de connaître la sémantique du chemin XPath nous permet de déduire l'annotation du type de sortie et donc d'annoter le filtre en conséquence. On obtient alors des expressions XPath automatiquement typées. Nous attachons un grand soin à la préservation exacte de la sémantique d'XPath pour le fragment considéré (ordre de parcours, absence de doublons,. . .). Nous montrons en particulier que plusieurs prédicats peuvent être testés

```
  type T = [] | [ <a>[] T <b>[] ]
  type S = [<c>[] <d>[] ] | [ <c>[] S <d>[] ]


let filter flatten =
   $  []  -> ~{ [] }
      |  $  ([ Any* ] ,_) -> (( flatten , flatten ) ; concat)
      | (  $  x & (Any ∖ [Any*]) ->~{ x } , flatten )


let v1 = (* une valeur de type T *)
let v2 = (* une valeur de type S *)
(*
 ... autres definitions
*)

let r1 = apply flatten to v1 where {| flatten = [ <a>[]* <b>[]* ] |}
let r2 = apply flatten to v2 where {| flatten = [ <c>[]+ <d>[]+ ] |}
```

Fig. A.4 – Un exemple de code ℂDuce avec des filtres

statiquement, en les encodant sous forme de motifs ℂDuce.

A.4.6 Élagage statique et typage d'XQuery (chapitre 8)

Cette dernière contribution s'éloigne un peu des filtres et montre comment appliquer leur discipline de typage à un langage déjà existant, et quels bénéfices en retirer. L'idée est toujours la même : « appliquer la transformation sur le type d'entrée pour en calculer le type de sortie ». Nous appliquons cette technique, à XQuery et XPath, en utilisant une version plus complexe de l'algorithme d'inférence d'annotations utilisé au chapitre 7. L'idée est de calculer non seulement le type de sortie d'une requête mais surtout un opérateur de projection. Ce dernier est une information statique (au même titre que le type) qui permet de déterminer quels noeuds d'un document vont être effectivement utilisés par la requête (et quels autres seront simplement ignorés). Cela nous permet d'équiper le moteur d'exécution de requêtes pour qu'il puisse effectuer un chargement partiel du document en mémoire, ne chargeant que les parties nécessaires à l'exécution. Ce type d'optimisations est particulièrement intéressant dans le cadre de moteurs DOM, ces derniers utilisant une représentation des document XML en mémoire peu optimale. En effet, pour assurer une exécution efficace, de nombreuses méta-données sont gardées en mémoire avec le document. Un document dont la taille sur le disque n'est que de 50 Mo pourra donc avoir une taille de 250 Mo une fois chargé en mémoire et complètement saturer la mémoire système lors de l'exécution d'une requête. Nous fournissons les outils formels nécessaires à l'analyse des projecteurs ainsi que nos résultats d'implantation qui montrent des performances nettement supérieures aux travaux existants.

A.5 Conclusion et travaux futurs

Comme nous l'avons décrit dans ce résumé, cette thèse s'attache à l'étude d'un langage restreint et typé pour XML. Nous définissons un langage appelé langage de filtres dont nous montrons formellement qu'il a les propriétés souhaitées de précision, modularité et expressivité. Les résultats formels sont augmentés par une étude pratique de l'implantation des filtres dans ℂDuce, ainsi que de l'application des techniques développées aux formalismes standards d'XML (XPath et XQuery).

Le point majeur pour la poursuite des travaux est bien évidement l'étude de l'inférence automatique d'annotations dans le cas général. Les travaux sur l'approximation de langages algébriques de Mark Jan NEDERHOF ([Nedoo]) sont un point de départ. Utiliser les nombreux résultats obtenus pour la compilation efficace des transducteurs d'arbres permettrait aussi d'améliorer le modèle d'exécution des filtres. Une piste intéressante est aussi l'application des techniques de typage arrière. Bien qu'étant inutilisables en l'état, elles pourraient devenir performantes après un prétraitement du filtre par une passe de typage avant, approximative peu coûteuse en temps. Un autre point demandant une attention particulière est celui de la syntaxe concrète et en particulier des annotations. En effet, bien que conservant la modularité, notre politique d'annotation reste fragile au regard de plusieurs aspects. Dans un premier lieu, il faut se préocuper des problèmes de nommages et de portée (nous avons considéré ici que tous les filtres portaient des noms différents). Un autre problème est celui de pouvoir annoter un filtre « profondément enfoui » dans le corps d'un autre (ce qui est actuellement possible mais demande l'introduction de noms artificiels, poluant le code).

En ce qui concerne les travaux de projection et typage d'XQuery, une extension naturelle semble être de passer des DTDs à un formalisme plus général, regroupant DTDs, XML-Schemas, Dataguides,.... Un travail d'intégration avec un moteur existant et éprouvé est aussi un point important.

Bibliography

[ℂDuce] The ℂDuce language implementation. `http://www.cduce.org`.

[AC93] R. M. Amadio and L. Cardelli. Subtyping recursive types. *ACM Trans. on Programming Languages*, 15(4):575–631, 1993.

[Acz77] P. Aczel. Introduction to inductive definitions. *In: Barwise, J. (Ed.), Handbook of Mathematical Logic, North-Holland Publishing Company, Amsterdam. pp. 739-782*, 1977.

[Bal06] V. Balat. The Ocsigen: Typing Web Interaction with Objective Caml. ACM Sigplan Workshop on ML, 2006.

[BCCN06] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyễn. Type-based XML projection. In *VLDB 2006*, pages 271–282, 2006.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *ICFP '03*, pages 51–63, 2003.

[BCL$^+$05] S. Bressan, B. Catania, Z. Lacroix, Y-G Li, and A. Maddalena. Accelerating queries by pruning XML documents. *Data Knowl. Eng.*, 54(2):211–240, 2005.

[BCM05] V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL 05*, number 3350 in LNCS, pages 235–252, 2005.

[BNST06] G. J. Bex, F. Neven, T. Schwentick, and K. Tuyls. Inference of concise dtds from xml data. In *VLDB*, pages 115–126, 2006.

[CAS] The Castor project. `http://www.castor.org/`.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CDG$^+$97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at: `http://www.grappa.univ-lille3.fr/tata`, 1997. Release October, the 1st 2002.

[CF05] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proc. of *PPDP '05* (full version) and *ICALP '05*, LNCS n. 3580, (summary), 2005. Joint ICALP-PPDP keynote talk.

[CFF$^+$03] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. XML Query Use Cases. Technical Report 20030822, World Wide Web Consortium, 2003.

[CGMS04] D. Colazzo, G. Ghelli, P. Manghi, and C. Sartiani. Types for Path Correctness for XML Queries. In *ICFP '04, 9th ACM Int. Conf. on Functional Programming*, 2004.

[CN08] G. Castagna and K. Nguyễn. Typed iterators for xml. In *PLAN-X*, page to appear, 2008.

[Col04] D. Colazzo. *Path Correctness for XML Queries: Characterization and Static Type Checking*. PhD thesis, Dip. di Informatica, Università di Pisa, 2004.

[Cou83] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983.

[DAF$^+$03] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. M. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[DFF$^+$04] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, February 2004. W3C Working Draft.

[DG05] O. Danvy and M. Goldberg. There and back again. *Fundam. Inform.*, 66(4):397–413, 2005.

[DOM04] W3C: DOM specifications. `http://www.w3.org/TR/2004/REC-DOM-Level-3-Core-20040407/`, 2004.

[DTD06] W3C: DTD specifications. `http://www.w3.org/TR/REC-xml/#dt-doctype`, 2006.

[Eng77] J. Engelfriet. Top-down tree transducers with regular look-ahead. *Mathematical Systems Theory*, 10:289–303, 1977.

[EV85] J. Engelfriet and H. Vogler. Macro tree transducers. *J. Comput. Syst. Sci.*, 31(1):71–146, 1985.

[Ext] The extlib library. `http://ocaml-lib.sourceforge.net`.

[FCB02] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02*, pages 137–146. IEEE Computer Society Press, 2002.

[FH07] A. Frisch and H. Hosoya. Towards practical typechecking for macro tree transducers. In *DBPL*, pages 246–260, 2007.

[FN07] A. Frisch and K. Nakano. Streaming xml transformation using term rewriting. In *PLAN-X*, pages 2–13, 2007.

[Fra05] M. Franceschet. XPathMark - An XPath benchmark for XMark generated data. In *XSym 2005, 3rd Int. XML Database Symposium*, LNCS n. 3671, 2005.

[Fri04a] A. Frisch. Regular tree language recognition with static information. In *Proc. IFIP Conf. on Theor. Comput. Sci. (TCS)*. Kleuwer, 2004.

[Fri04b] A. Frisch. *Théorie, conception et réalisation d'un langage adapté à XML.* PhD thesis, Université Paris 7 Denis Diderot, 2004.

[Fri06] A. Frisch. OCaml + XDuce. *SIGPLAN Not.*, 41(9):192–200, 2006.

[GGM$^+$04] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[GLP03] V. Gapeyev, M. Levin, and B. Pierce. Recursive subtyping revealed. *Journal of Functional Programming*, 12(6):511–548, 2003.

[GLPS05] V. Gapeyev, M. Y. Levin, B. C. Pierce, and A. Schmitt. The Xtatic experience. In *PLAN-X*, 2005.

[GP04] V. Gapeyev and B. C. Pierce. Paths into patterns. Technical Report MS-CIS-04-25, University of Pennsylvania, October 2004.

[Gra03] H. Grall. *Deux critères de sécurité pour l'execution de code mobile.* PhD thesis, Ecole Nationale des Ponts et Chaussées, 2003.

[HFC05] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *POPL '05*, pages 50–62, 2005.

[Hos00] H. Hosoya. *Regular Expression Types for XML.* PhD thesis, University of Tokyo, 2000.

[Hos04] H. Hosoya. Regular expression filters for XML. In *Programming Languages Technologies for XML (PLAN-X)*, pages 13–27, 2004.

[HP01] H. Hosoya and B.C. Pierce. Regular expression pattern matching for XML. In *POPL '01*, 2001.

[HVP00] H. Hosoya, J. Vouillon, and B. Pierce. Regular expression types for XML. In *ICFP '00*, volume 35(9) of *SIGPLAN Notices*, 2000.

[IHW02] Z. G. Ives, A. Y. Halevy, and D. S. Weld. An xml query engine for network-bound data. *VLDB J.*, 11(4):380–402, 2002.

[JAX] The jaxb api. `http://java.sun.com/developer/technicalArticles/WebServices/jaxb/`.

[Jay04] C. Barry Jay. The pattern calculus. *ACM Trans. Program. Lang. Syst.*, 26(6):911–937, 2004.

[Läm07] R. Lämmel. Scrap your boilerplate with XPath-like combinators. In *POPL'07, Proceedings*. ACM Press, January 2007.

[Ler06] X. Leroy. Coinductive big-step operational semantics. In *European Symposium on Programming (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer, 2006.

[LG07] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 2007. Accepted for publication in the special issue on Structured Operational Semantics, to appear.

[LMM00] D. Lee, M. Mani, and M. Murata. Reasoning about XML Schema Languages using Formal Language Theory. Technical report, IBM Almaden Research, 2000.

[LP03] R. Lämmel and S. Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. *ACM SIGPLAN Notices*, 38(3):26–37, March 2003. Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

[MBPS05] S. Maneth, A. Berlea, T. Perst, and H. Seidl. XML Type checking with macro tree transducers. In *ACM PODS*, pages 283–294, 2005.

[Mia06] C. Miachon. *Langages de requêtes pour XML à base de patterns : conception, optimisation et implantation.* PhD thesis, Université Paris-Sud 11, 2006.

[MMW05] A. Malhotra, J. Melton, and N. Walsh. XQuery 1.0 and XPath 2.0 Functions and Operators, 2005. `http://www.w3.org/TR/xpath-functions/`.

[MPS07] S. Maneth, T. Perst, and H. Seidl. Exact XML type checking in polynomial time. In *ICDT*, pages 254–268, 2007.

[MS03] A. Marian and J. Siméon. Projecting XML documents. In *VLDB '03*, pages 213–224, 2003.

[MSV03] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *J. Comput. Syst. Sci.*, 66(1), 2003.

[Ned00] M.-J. Nederhof. Practical experiments with regular approximation of context-free languages. *Computat. Linguistics*, 26(1):17–44, 2000.

[OCS] The Ocsigen Project. `http://www.ocsigen.org/`.

[OMFB02] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proc. EDBT Workshop (XMLDM)*, volume 2490 of *LNCS*, pages 109–127. Springer, 2002.

[Pie02] B. C. Pierce. *Types and programming languages.* MIT Press, 2002.

[Rel] OASIS Committee Specification: Relax-NG. `http://relaxng.org/spec-20011203.html`.

[SL06a] M. Sulzmann and K. Zhuo Ming Lu. A type-safe embedding of XDuce into ML. *El. Notes Theor. Comp. Sci.*, 148(2):239–264, 2006.

[SL06b] M. Sulzmann and K. Zhuo Ming Lu. XHaskell. In *PLAN-X*, 2006.

[SV02] L. Segoufin and V. Vianu. Validating streaming xml documents. In *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 53–64, New York, NY, USA, 2002. ACM.

[SWK$^+$02] A. Schmidt, F. Waas, M. L. Kersten, M. J. Carey, I. Manolescu, and R. Busse. XMark: A benchmark for XML data management. In *VLDB '02*, pages 974–985, 2002.

[Toz01] A. Tozawa. Towards static type checking for xslt. In *DocEng '01: Proceedings of the 2001 ACM Symposium on Document engineering*, pages 18–27, New York, NY, USA, 2001. ACM.

[Vou06] J. Vouillon. Polymorphic regular tree types and patterns. In *POPL*, pages 103–114, 2006.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990.

[Wat94] B. W. Watson. A taxonomy of finite automata construction algorithms. Technical Report Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, May 1994.

[Wel99] J. B. Wells. Typability and type checking in system f are equivalent and undecidable. *Ann. Pure Appl. Logic*, 98(1-3):111–156, 1999.

[XML] W3C: XML Version 1.0 (Fourth Edition). `http://www.w3.org/TR/REC-xml/`.

[XPa] W3C: XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`.

[XQu] W3C: XML Query (XQuery). `http://www.w3.org/TR/xquery`.

[XSc] W3C: XML Schema. `http://www.w3.org/XML/Schema`.