JFP **16** (6): 711–750, 2006.
doi:10.1017/S0956796806005909 First published online 9 June 2006 Printed in the United Kingdom

Regular expression filters for XML

HARUO HOSAYA

Department of Computer Science, The University of Tokyo,
Hongo 7-3-1, Bunkyo-ku, Tokyo 113, Japan
email: hahosoya@is.s.u-tokyo.ac.jp

Abstract

XML data are described by types involving regular expressions. This raises the question of what language feature is convenient for manipulating such data. Previously, we have given an answer to this question by proposing regular expression pattern matching. However, since this construct is derived from ML pattern matching, it does not have an iteration functionality in itself, which makes it cumbersome to process data typed by Kleene stars. In this paper, we propose a novel programming feature *regular expression filters*. This construct extends the previous proposal by permitting pattern clauses to be closed under arbitrary regular expression operators. This yields many convenient programming idioms such as non-uniform processing of sequences and almost-copying of trees. We further develop a type inference mechanism that obtains (1) types for pattern variables that are *locally precise* with respect to the type of input values and (2) a type for the result of the whole filter expression that is also locally precise with respect to the types of the body expressions. We discuss how our construct is useful in the practice of XML processing and, in particular, how our type inference is crucial for avoiding changes of programs when types of data to be processed evolve frequently.

1 Introduction

XML (Bray *et al.*, 2000) is a representation of typed data structures for trees. As it becomes popular, a big demand has emerged for special-purpose languages dedicated to processing XML data, in particular, those capable of statically guaranteeing generated data to conform to given types (Cluet & Siméon, 1998; Meijer & Shields, 1999; Fankhauser *et al.*, 2001; Murata, 2001; Milo *et al.*, 2000; Frisch *et al.*, 2002; Tozawa, 2001; Hosoya & Pierce, 2003; Gapeyev *et al.*, 2005; Lu & Sulzmann, 2004).

A unique property of XML data is that there is ordering among sibling nodes and, in order to give structure on these nodes, types for XML data typically involve regular expressions (Bray *et al.*, 2000; Fallside, 2001; Clark & Murata, 2001; Hosoya *et al.*, 2000). This gives rise to the question: what is a convenient programming feature for manipulating such data? Previously, we have given an answer to this question by proposing *regular expression pattern matching* (Hosoya & Pierce, 2002; Hosoya, 2003). In one sense, this design is natural since it combines regular expressions with the ML-style pattern matching paradigm, which has already established its usefulness in tree processing in functional programming languages. However, since pattern matching does not have an iteration functionality in itself,

it is quite cumbersome to process data values typed with regular expressions, in particular, those involving Kleene stars. Typically, an explicit recursive function is required each time the program processes such data.[1] As a construct more suitable for this purpose, many languages for XML, such as XSLT (Clark, 1999), XQuery (Fankhauser *et al*., 2001), and CDuce (Benzaken *et al*., 2003), provide a "for-each" iterator, which performs a given operation on each node of a sequence independently. However, such a construct is *ad hoc* in the sense that it has no connection to regular expressions. More practically, it is difficult for this to manipulate data that repeat groups of several elements. For example, for a value of type `(a,b)*`, we would typically want to process each pair of `a` and `b` rather than each `a` or `b`. (Although such "complex" regular expressions are not very typical, they can occasionally be found in real XML schemas, such as DocBook (OASIS, 2002), MusicXML (LLC., 2004), and RecipeML (FormatData, 2000). We give concrete instances in Appendix B.)

This paper proposes *regular expression filters* as an XML manipulation construct that is elegant, i.e., strongly connected to regular expressions, and has both functionalities of pattern matching and iteration. The idea behind this is to extend regular expression pattern matching, which can be seen as an alternation of pattern clauses, by allowing arbitrary regular expressions over pattern clauses. This yields a significant expressiveness, allowing various convenient programming idioms, such as "non-uniform processing of sequences" (processing nodes with the same label differently depending on their positions) and "almost-copying of trees" (copying the whole tree structure with slight changes), that go beyond the capability of pattern matching or a naive for-each style iterator.

We will give plenty of examples in section 2, but let us see here small ones. Suppose that we have a value `items` of type `Item*` where `Item` is defined as follows

```
type Item = item[Content]
```

(where the `Content` type is defined somewhere else), that is, the value `items` is a sequence of `item` labels each of which contains a value of type `Content`. Let us consider converting such sequence of `items` into some display format. The simplest way would be to format each `item` independently and concatenate all the results. Using our filters, we can express it in a concise manner without involving any recursive definition.

```
filter items {
  (item[Any as c]  { format(c) })*
}
```

A more complex formatting would be to process each item in this value, inserting separators between the resulting values. Make sure not to put a separator in the end, and return the empty sequence if the input is the empty sequence. To do this

[1] This might not be the case if the language supported both parametric polymorphism and higher-order functions as in ML. However, though each of these has been treated separately in Hosoya *et al*. (2005) and in Frisch *et al*. (2002), handling both at the same time is still an open problem.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

with filters, we first regard the input type `Item*` as the following equivalent one

```
() | (Item*,Item)
```

and write the filter expression below, which processes the input value according to the structure indicated by the new regular expression.

```
filter items {
   ()               { () }
 | (item[Any as c]  { format(c), sep[] })*,
   (item[Any as c]  { format(c) })
}
```

That is, when the input is the empty sequence, we use the first clause to emit the empty sequence. In the other case, we use the second clause to format each node and append a separator except for the last node, for which we use the third clause to format the node without adding a separator.

In order for our construct to comfortably be used in a statically typed language, we have designed a type inference mechanism. Similarly to our previous inference technique for regular expression pattern matching (Hosoya & Pierce, 2002), the inference here is *local* and *locally precise*. By local, we mean that we calculate types relevant to a filter by using only type information obtained from its adjacent expressions. By locally precise, we mean that the types to be computed precisely represent the values of the corresponding expressions, with the conservative assumption that the type information from the adjacent expressions is also precise.

More specifically, the inference computes types for two parts of a given filter: the bound variables and the result. For the first part, we have already shown (Hosoya & Pierce, 2002), in the setting of regular expression patterns, how to calculate types for bound variables from a given type for the input. However, the previous technique has the limitation that it can infer types only for "tail" variables (binding a sub-sequence up to the end of the input sequence), which is not acceptable in the present setting since most filters actually use variables to capture intermediate sequences (as in the above example). Therefore we have developed a novel inference technique based on the combination of a tree automata encoding involving "sequence-capturing variables" (Section 4) and product construction of automata (Section 5). (Frisch, Castagna, and Benzaken have solved the same problem independently using a different algorithm (Frisch *et al.*, 2002).) The second part of the inference is to compute a type for the result of the filter from both the type for the input and a given type for each body expression. We have achieved this by using automata augmented with "action annotations" (Section 4). For the practical implication, the inference is, of course, quite convenient for eliding obvious type annotations, thus increasing the writability and readability of programs. However, in the setting of XML, we believe that such type inference is more important than this usual benefit, in particular, it is actually mandatory in reducing the burden of changes of programs caused by evolution of the types the programs work with, as we will argue in Section 2.5.

An important piece of related work is CDuce's `map` and `transform` constructs (Benzaken *et al.*, 2003), which are similar to our regular expression filters except that

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

theirs are restricted to uniform processing of sequences. They also have a somewhat similar but different type inference technique for both bound variables and the result. We will make a more detailed comparison in Section 6.

Although we intend this work to be a feature proposal independent of a specific language, we have incorporated regular expression filters and the type inference in our design and implementation of the typed language XDuce for XML processing. The reader is encouraged to try out our prototype system available through:

```
http://xduce.sourceforge.net
```

The rest of the paper is organized as follows. Section 2 shows a series of examples to illustrate regular expression filters and the type inference. We then formalize these in Section 3. In Section 4, we introduce an automata model for regular expression filters that is suitable for performing the type inference. The inference algorithm itself is described in Section 5. Section 6 compares our work with other work and Section 7 closes this paper. Appendix A shows an algorithm from filters in the surface language to our automata model. Appendix B collects examples of "complex" regular expressions found in real-world DTDs.

2 Examples

This section gives an informal presentation of our language constructs and illustrates their practical uses. The formal definitions can be found in Section 3.

2.1 Values and types

Values in our type system are sequences of labeled values (or base values) and thus representing fragments of XML structures. For example, a sequence of several labeled values like

```
name["Hosoya"],email["hahosoya"],tel["123-456"]
```

and a single label containing some other sequence like

```
person[name["Pierce"],email["bcpierce"]]
```

are values.

Types are regular expressions over labeled types (or base types such as `String`). For example, the following type

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

allows a sequence of a `name` label followed by zero or more `email` labels and an optional `tel` label where each label contains a string. We can also nest types as in the following.

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

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

We also allow recursive types (and type abbreviations) through type definitions. For example, the following defines a type for trees where each node can have an arbitrary number of subtrees.

```
type Tree = node[Tree*] | leaf[String]
```

We do not impose any restriction on regular expressions, such as determinism or unambiguity (e.g., we allow unions of the same labels like `a[b[]]|a[c[]]`). This makes types to correspond to *nondeterministic finite tree automata*, which form the basis of our framework. More discussions can be found in Hosoya *et al.* (2000).

A labeled type can actually have a *label set* instead of a single label. For example, we can use union label sets as in `(name|email)[String]`, the universal label set as in `~[String]`, and negation label sets as in `^(tel|email)[String]`. In particular, we use the types `Any` (matching any value) and `AnyOne` (matching any singleton sequence) defined as follows.

```
type Any = AnyOne*
type AnyOne = ~[Any] | String
```

(We assume here that the only base type is `String`.)

The subtype relation between two types is simply inclusion between the sets of values that they denote. For example, `(Name*,Tel*)` is a subtype of `(Name|Tel)*` since the first one is more restrictive than the second. That is, `Name`s must appear before any `Tel` in the first type, while `Name`s and `Tel`s can appear in any order in the second type.

2.2 Regular expression filters

The basic blocks of regular expression filters are *clauses*. A clause has the form *pattern* {*expression*} and means "if the input value is matched by the pattern, execute the expression," just like ML pattern matching. Patterns are syntactically identical to types except that patterns may contain variables to which the corresponding substructures of the input value are bound. (Therefore a type itself can be used as a pattern by putting no variable.) We do not give a specific definition of expressions in this paper; however, examples shown in the sequel use the following kinds of expression: labeling constructors `l[e]`, concatenations `e1,e2`, the empty sequence `()`, variables `x`, base values such as strings, function calls `f(e)`, and filter expressions themselves.

We can connect or enclose clauses by regular expression operators, forming *filters*. For example, the following filter expression gives a value `v` to the filter connecting several clauses by the union operator `|`.[2]

[2] The precedence rule for operators usable in filters is as follows (from stronger to weaker):

```
* + ? as {}   (suffix operators)
,
|
```

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

```
filter v {
   person[Any as i]    {  li[...]  }
 | company[Any as j]   {  li[...]  }
 | comment[Any as s]   {  p[]     }
}
```

Here, the form *pattern as variable* binds the variable to the value matched by the pattern. Thus, the above filter matches any singleton sequence where the label of the only element is either `person`, `company`, or `comment`. In the first case, it executes the first body expression creating a label `li` (containing some sequence not shown here), and similarly for the other cases. As one can see, this use of filters is similar to ML pattern matching and, in fact, is exactly the same as our previous proposal, regular expression pattern matching (Hosoya & Pierce, 2002) (except that the union operator in filter expressions does not have the first-match semantics, i.e., top-to-bottom evaluation of clauses, but instead it chooses an arbitrary match if there are multiple possibilities; we will come back to this point in Section 3.)

A filter can be enclosed by Kleene closure `*`, enabling iteration on sequences. For example, the following wraps the above filter by `*`.

```
filter v {
 ( person[Any as b]    {  li[...]  }
 | company[Any as f]   {  li[...]  }
 | comment[Any as s]   {  p[s]     } )*
}
```

The filter matches any sequence of labels `person`, `company`, or `comment`, converts each label to either `li` or `p` in the same way as above, and concatenates all the results in the left-to-right order.

The concatenation of two filters splits the given sequence so that the first sub-sequence matches the first filter and similarly for the second one, then evaluates each filter with the corresponding sub-sequence, and finally concatenates the two results. (Again, when there are multiple ways of matching, the system chooses an arbitrary one.) For example, the following filter expression

```
filter v {
  (email[Any as e]  { emailAddress[e]  }),
  (tel[Any as t]    { telNumber[t]     })
}
```

matches any value of type `email[Any],tel[Any]` and replaces the label `email` with `emailAddress` and `tel` with `telNumber`.

When a filter is enclosed by a label, it matches a value with this label, processes the content with the enclosed filter, and puts the label back to the result. For example, the following filter

```
filter v {
  person[ Any as pc  { proc_person_content(pc) } ]
}
```

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

processes the content of a `person` label by the `proc_person_content` function (defined somewhere else), keeping the label itself.

2.3 Non-uniform sequence processing

Since we can use arbitrary combinations of regular expressions over clauses, this allows us to process a sequence in a more complex way than "for-each" iteration – such as processing elements with the same label in a different way depending on their positions, or operating on each group of elements in a sequence rather than on each individual element.

In the introduction, we have already seen a small example of non-uniform processing. Let us show here another, slightly more complex one. In the second example shown in Section 2.2, the output may mix `li`s and `p`s in the same sequence. However, since this is actually not allowed in XHTML, we want to put consecutive `li`s together in a `dl` label this time. For this, we need to process each consecutive list of `person`s and `company`s separately from intervening `comment`s. Our solution is to first view the type

```
(Person | Company | Comment)*
```

as the following equivalent type

```
((Person | Company)*, Comment)*, (Person | Company)*.
```

Here, we assume that the following type definitions are given.

```
type Person   = person[Info]
type Company  = company[Info]
type Comment  = comment[String]
```

Then, we write a filter that corresponds to the above regular expression.

```
filter v {
  (((Person|Company)* as s   {  dl[proc_mix(s)]  }),
   comment[Any as s]         {  p[s]  })*,
  ((Person|Company)* as s    {  dl[proc_mix(s)]  })
}
```

This filter calls the function `proc_mix` defined below (which takes an argument of type `(Person|Company)*`), which in turn uses another filter to process a sequence of `person`s and `company`s.

```
fun proc_mix ((Person|Company)* as seq) : ... =
  filter seq {
   ( person[Any as i]    {  li[...]  }
   | company[Any as i]   {  li[...]  } )*
  }
```

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

2.4 Almost-copying of trees

In practical XML processing, we often want to modify small bits of the input document, retaining the rest of the structure. For this purpose, regular expression filters are quite convenient.

For example, consider the following type definitions.

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

Suppose that we want to "clean up" a sequence of `persons` by processing the string of each `name`, `email`, or `tel` by a corresponding tidying function. Then, we can write a filter expression by copying the structure of the `Person` type and inserting an appropriate variable binder and a body expression after each `String` type.

```
filter ps {
  person[
    name[String as n  { tidy_name(n) }],
    email[String as e { tidy_email(e) }]*,
    tel[String as t   { tidy_tel(t) }]?
  ]*
}
```

(We assume that the `tidy_name` etc. functions from strings to strings are defined somewhere else.) Note that two Kleene stars are nested, around `person` and around `email`. If we had to write the same function only with pattern matching, we would need two recursive functions, which would be much more cumbersome.

The examples so far have been horizontal processing of XML data, but we would sometimes want vertical processing. For example, let us slightly change the `Person` type definition by allowing each `person` to have a sequence of `persons` recursively.

```
type Person = person[Name,Email*,Tel?,Person*]
```

Then, we would like to tidy leaf information in the same way as before, but, this time, the type involves recursion and therefore we would naturally want a filter traversing recursively on data of such type. For this, we provide a notation for writing recursively defined filters. In our example, we can first declare a filter named `tidy_persons` in the following way.

```
rule tidy_persons =
  person[
    name[String as n  { tidy_name(n) }],
    email[String as e { tidy_email(e) }]*,
    tel[String as t   { tidy_tel(t) }]?,
    tidy_persons
  ]*
```

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

That is, the filter `tidy_persons` processes a sequence of `persons` similarly to the previous paragraph, except that, for the sequence of `persons` appearing in the end of the content of each `person`, we apply the `tidy_persons` filter recursively. To invoke this filter for a given value, we simply refer to the filter's name in a filter expression as in the following.

```
filter ps { tidy_persons }
```

In writing an almost-copying program, we often want to retain the whole substructure of a value, delete it, or insert a substructure. For example, suppose that we have the following type definitions

```
type Person1 = person[Name,Email*]
type Person2 = person[Name,Tel]
```

and want to convert a value from type `Person1` to type `Person2`, retaining the `name` element, delete the sequence of `email` elements, and insert a "default" `tel` element. The following filter achieves this in a simple way.

```
filter p {
  person[
    Name,
    Email*  {  ()  },
    ()      {  tel["unknown"]  }
  ]
}
```

Here, we use the type `Name` as a filter, which simply retains the value matched by the type. Also, note that we use a clause with the empty sequence pattern, by which we can insert any value even though there is nothing corresponding in the original value.

2.5 Type inference

We now turn our attention to the type inference mechanism dedicated to our filter facilities. As mentioned in the introduction, the type inference has two parts: inference of types for bound variables and that of a type for result values.

2.5.1 Types for variables

The type inference for variables is *local* in the sense that it depends only on a type for input values and a subject filter (thus using no constraint from distant expressions). The inference is *locally precise* in the following sense. First, we conservatively assume that all the values from the input type may be passed to the filter. Then, under this assumption, we compute, for each pattern variable, a type that contains exactly the values that may be bound to the variable.

For example, consider the following filter where the input `bookcontent` has type `(Person|Company|Comment)*`.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

```
filter bookcontent {
   ()
   {  ()  }
 | AnyOne+ as c
   {  dl[
        filter c {
          (AnyOne as e { li[proc_each(e)] })+
        }]
   }
}
```

The purpose of the filter is to process each node in the input by the `proc_each` function (taking type `(Person|Company|Comment)` and defined somewhere else), put a `li` to each result, and enclose all the results by a `dl`. We need, however, to handle the case of the empty sequence specially since XHTML requires that a `dl` must contain one or more `li`s. In this case, we return the empty sequence as the whole result. From the fact that input values have type `(Person|Company|Comment)*` and the second clause matches only sequences of length one or more, the inference computes the type `(Person|Company|Comment)+` for the variable `c`. From this type, we further infer the type `(Person|Company|Comment)` for the variable `e`.

One benefit from this type inference is, of course, to avoid verbose type annotations. Another, potentially bigger benefit is that the inference can make program code robust against changes of types. For example, suppose we have changed the input type `(Person|Company|Comment)*` to `(Person|Company|Shop|Comment)*`. (In general, the most common way of evolving a type is to make it larger in denotation.) If we want to process the input exactly in the same way as before (except that the `proc_each` function should now handle the new case), then it is desirable that we need not change the above code fragment. If we did not have a precise inference, the type explicitly annotated for `c` has to manually be modified from `(Person|Company|Comment)+` to `(Person|Company|Shop|Comment)+` (and similarly for `e`'s type). With our precise inference, on the other hand, types for `c` and `e` are automatically computed and therefore such a modification is also automatic, which makes the code remain the same.

2.5.2 *Types for result values*

The inference for result types is also local in the sense that it depends only on a type for each body expression in addition to the input type and the filter. The type for each body expression may have been obtained by some typing algorithm from the types inferred for bound variables. We do not, however, assume any concrete typing algorithm for body expressions; rather, this is given as a parameter to our inference scheme.

Then, the inference is, again, locally precise. Before describing what we mean by this, let us show an example. Consider the following filter

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

```
filter v {
  (email[String as s]  { emailAddr[s] } )*
}
```

Suppose that we have the type `emailAddr[String]` for the body expression. What should be the result type of this filter? Naively, we may answer `emailAddr[String]*` from the structure of the filter. However, we could go further. That is, the result type can depend on the input type. For example, it can be `emailAddr[String]+` if the input type is `email[String]+` since the filter produces one output node for each input node. In general, we first conservatively assume that all the values in the input type may be passed to the filter, as before, and that all the values in the type of each body may be returned by it. (Again, some values from the body type may not actually be returned at run time.) Under these assumptions, we compute a type, for result values of the filter, that contains exactly the values that may be returned by the filter.

One may wonder why we need such a precision. Our answer is, again, robustness against type evolution. Let us consider the following hypothetical scenario. Suppose that a schema for address book documents is maintained by a (big) standard committee and you are an engineer in a (small) company writing filters from address books to address books. As in the reality, the committee often changes the type when there are enough external requirements. However, you may not want to change your filter programs every time they change the type especially when you write many different filters for address books. Therefore you may want to make your programs as general as possible in the first place so that they work after foreseeable type changes.

For concreteness, suppose that the address book schema contains the type `PersonInfo`, which is defined as follows at the beginning.

```
type PersonInfo = Name,Addr+,Email*,Tel?
```

You could write a filter with the same structure as this type, but this would not be robust against any type change. The committee might allow any `addr` to be omitted or allow more than one `tel` to be present. So it is better to write in the following way.

```
filter content {
  (name[Any as n]  {  name[tidy_name(n)]   }),
  (addr[Any as a]  {  addr[tidy_addr(a)]   })*,
  (email[Any as e] {  email[tidy_email(e)] })*,
  (tel[Any as t]   {  tel[tidy_tel(t)]     })*
}
```

So does this filter typecheck (with the expected type `Name,Addr+,Email*,Tel?`) before the anticipated type change actually happens? The naive inference would compute the result type as `Name,Addr*,Email*,Tel*` and makes the filter ill-typed since this type is larger than the expected type. On the other hand, our precise inference answers exactly the same type as the input type `Name,Addr+,Email*,Tel?` (which is the same as the expected type) since our inference recognizes, from the

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

input type, that there are only one or more `addrs` and zero or one `tel` in any input value, and therefore so in any output value.

If you further suspect that the committee might change the `PersonInfo` type in a way that allows an arbitrary order among `addrs`, `emails`, and `tels`, then you can make the filter more general in the first place as follows.

```
filter content {
 ( name[Any as n]  {  name[tidy_name(n)]   }
 | addr[Any as a]  {  addr[tidy_addr(a)]   }
 | email[Any as e] {  email[tidy_email(e)] }
 | tel[Any as t]   {  tel[tidy_tel(t)]     })*
}
```

The type inference still gives you the right type – `Name,Addr+,Email*,Tel?` – for the result. If you guess that they might allow more possible fields to be added, then you can write an even more general filter to ignore such fields.

```
filter content {
 ( name[Any as n]  {  name[tidy_name(n)]   }
 | addr[Any as a]  {  addr[tidy_addr(a)]   }
 | email[Any as e] {  email[tidy_email(e)] }
 | tel[Any as t]   {  tel[tidy_tel(t)]     }
 | ^(name|addr|email|tel)[Any]   { ()      } )*
}
```

The type inference still does the right job.

Caveat: our type inference has the restriction that it uses *only one* type for each body expression. For example, consider the following filter expression with the input type `a[T]|b[U]`.

```
filter v {
  ~[ Any as x  { c[x] } ]
}
```

This filter copies the input value, inserting an intermediate label `c` just under the top label `a` or `b`. One may expect the result type to be `a[c[T]]|b[c[U]]`. However, since we can give only one type to the body, the best type we can give is `c[T|U]`, and therefore the result type is `a[c[T|U]]|b[c[T|U]]`, which is larger than the expected one. We know that this limitation is undesirable and can have a negative impact on practice. Unfortunately, this seems the best we could do in the present local inference approach. Indeed, if we remove this restriction, then there is an example where we can obtain unboundedly better types by repeating the inference on the same body expression, which makes it difficult to define the specification of the inference. We will describe this point in Section 3.3 in more detail.

3 Formalization

In this section, we give the syntax and semantics of types, patterns, and filters, as well as the specification of our type inference.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

3.1 Syntax

We assume a (possibly infinite) set $\mathscr{A}$ of *labels*, ranged over by a. A *value* v is a sequence of labeled values, where a labeled value is a pair of a label and a value. We use the following syntax for writing values.

$$\begin{array}{lcll} v & ::= & \epsilon & \text{empty sequence} \\ & & a[v] & \text{labeled value} \\ & & v\,v & \text{concatenation} \end{array}$$

For brevity, we omit base values and types (such as strings) from the formalization. The changes required to add them are straightforward.

We assume a countably infinite set $\mathscr{S}$ of sets of labels. Each member of $\mathscr{S}$ is called *label set* and ranged over by L. Let the set $\mathscr{S}$ be closed under union, intersection, and complementation. We also assume countably infinite sets of pattern names ranged over by X, filter names ranged over by Y, variables ranged over by x, and body ids ranged over by e. Then, patterns P and filters F are defined as follows.[3]

$$\begin{array}{lcll} P & ::= & P \text{ as } x & \text{binder} \\ & & P\,P & \text{concatenation} \\ & & P \mid P & \text{alternation} \\ & & P^* & \text{repetition} \\ & & L[X] & \text{label} \\ & & \epsilon & \text{empty sequence} \end{array}$$

$$\begin{array}{lcll} F & ::= & P \to e & \text{clause} \\ & & F\,F & \text{concatenation} \\ & & F \mid F & \text{alternation} \\ & & F^* & \text{repetition} \\ & & L[Y] & \text{label} \\ & & \epsilon & \text{empty sequence} \end{array}$$

A *pattern grammar* is a finite mapping from pattern names to patterns and, similarly, a *filter grammar* is a finite mapping from filter names to filters. In either grammar, the names appearing in each pattern must be in the domain of the grammar. Throughout this paper, we assume fixed, global pattern grammar E and filter grammar G to be given. (Note that, in the formalization, we forbid nesting of labels and require names to occur only inside labels. This does not, however, lose generality since any definitions without these restrictions can be translated to ones with the restrictions[4].)

[3] In the formal definition, we slightly change the notation from the previous section to avoid confusion with standard mathematical notations, namely concatenation by juxtaposition instead of comma and clause by arrow instead of curly braces.

[4] More precisely, even if we drop the above-mentioned restrictions, we still need a restriction to ensure patterns not to have the power of context-free grammars, e.g.,

$$P = a[]\,P\,b[] \mid \epsilon.$$

A commonly adopted restriction is to require any recursive use of pattern names to appear inside a label. See Hosoya *et al.* (2000) and Hosoya (2003) for details.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

In the formal system here, we do not concretely specify what "body expressions" are, but rather treat them in an abstract way by using *body ids*; we've chosen this presentation for directly focusing on issues related to filters as well as for providing our techniques for filters as a package modularized from the rest of the programming language. We will show the operational semantics and the type inference specification for filters, which take evaluation and typing algorithms defined on body ids as parameters.

To ensure a given pattern to always bind the same set of variables, we impose a "linearity" restriction on them. Let $\mathbf{reach}(P)$ be the set of all variables reachable from P—that is, the smallest set satisfying the following:

$$\mathbf{reach}(P) \quad = \quad \mathbf{BV}(P) \cup \bigcup_{X \in \mathbf{FN}(P)} \mathbf{reach}(E(X)),$$

where $\mathbf{BV}(P)$ is the set of variables bound in P and $\mathbf{FN}(P)$ is the set of pattern names appearing in P. We say that a pattern P is *linear* iff, for any (reachable) subphrase P' of P, the following conditions hold.

- $x \notin \mathbf{reach}(P_1)$ if $P' = P_1$ as x.
- $\mathbf{reach}(P_1) \cap \mathbf{reach}(P_2) = \emptyset$ if $P' = P_1\, P_2$.
- $\mathbf{reach}(P_1) = \mathbf{reach}(P_2)$ if $P' = P_1 \mid P_2$.
- $\mathbf{reach}(P_1) = \emptyset$ if $P' = P_1{}^*$.

In what follows, we assume that (1) all patterns are linear, (2) for clauses $P_1 \to e_1$ and $P_2 \to e_2$ in a filter with $e_1 \neq e_2$, the same variable does not appear in P_1 and P_2, and (3) for $P_1 \to e_1$ and $P_2 \to e_2$ with $e_1 = e_2$, the sets of variables in P_1 and P_2 are the same.

Several features not appearing in the syntax can be expressed as shorthands. An optionality operator $F?$ can be rewritten to $F \mid \epsilon$ and a one-or-more-repetition filter F^+ to $F\, F^*$. (Note that each body appearing in F here is duplicated in the expanded form but still has the same id, e.g., no new id is allocated. This is important for ensuring the expansion not to break the restriction that each body in the original program is typechecked only once.)

As mentioned in Section 2.2, our alternation operator has the nondeterministic semantics, i.e., $F_1|F_2$ matches F_1 or F_2 nondeterministically, as opposed to the first-match semantics used in our previous framework (Hosoya & Pierce, 2002). However, since first-matching is often convenient for writing "default" clauses, we suggest providing (and indeed the current XDuce does support) a separate "first-match" alternation operator $F_1||F_2$, where F_2 matches only when F_1 does not match. One easy way to implement this is to convert the filter $F_1||F_2$ to $F_1|F_2'$ (using the nondeterministic alternation) where F_2' is a filter that behaves exactly the same as F_2 except it matches only values not matched by F_1. This conversion can easily be done by taking a "set-difference" between F_2 and F_1 preserving binding and action information in F_2. Further details are omitted in this paper. (The first-match alternation operator || suggested here does not cover the full expressiveness of our previous first-match semantics of patterns. In particular, the previous can express greedy matching, e.g., longest or shortest matching. However, we chose a

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

nondeterministic semantics since the first-match semantics introduces a significant complication in the language definition and implementation. Further discussion on matching policies is out of scope of this paper since the same argument already done elsewhere for pattern matching (Hosoya, 2003; Hosoya & Pierce, 2003) can be applied here and we believe that, if one wishes greedy-matching filters, one could transfer here previous techniques for greedy-matching patterns.)

A final remark is that, in Section 2.4, we have used a type expression itself as a filter expression, whose semantics is to retain the matched substructure. This is already in the syntax of filters since we allow a filter with no clause.

3.2 Evaluation

We now define the operational semantics of patterns and filters. As mentioned above, the semantics is parametrized over an evaluation algorithm $\mathbf{eval}(V, e)$ that takes an environment V and a body id e and returns a value. An environment is a finite mapping from variables to values, written $x_1 : v_1 \ldots x_n : v_n$.

The semantics of patterns is described by the matching relation $v \in P \Rightarrow V$, read "value v is matched by pattern P and yields environment V." The relation is defined by the following rules.[5]

$$\frac{v \in P \Rightarrow V}{v \in P \text{ as } x \Rightarrow (x : v)\, V}\ \textsc{PVar} \qquad \frac{\forall i.\ v_i \in P_i \Rightarrow V_i}{v_1\, v_2 \in P_1\, P_2 \Rightarrow V_1\, V_2}\ \textsc{PCat}$$

$$\frac{\exists i.\ v \in P_i \Rightarrow V}{v \in P_1 \mid P_2 \Rightarrow V}\ \textsc{POr} \qquad \frac{\forall i.\ v_i \in P \Rightarrow V_i}{v_1 \ldots v_n \in P^* \Rightarrow V_1 \ldots V_n}\ \textsc{PRep}$$

$$\frac{a \in L \qquad v \in E(X) \Rightarrow V}{a[v] \in L[X] \Rightarrow V}\ \textsc{PLab} \qquad \frac{}{\epsilon \in \epsilon \Rightarrow \emptyset}\ \textsc{PEps}$$

We write $v \in P$ when $v \in P \Rightarrow V$ for some V.

For the semantics of filters, a straightforward way would be to define a three-place relation on input values, filters, and output values, but we make a slight detour for the ease of formalizing the specification of type inference later. We first define the relation $v \in F \Rightarrow h$, read "from input value v, filter F yields *thunk* h." A thunk h is a sequence of pairs of environments and expressions or labeled thunks, as defined by:

$$\begin{array}{rcl} h & ::= & V \to e \\ & & h\, h \\ & & a[h] \\ & & \epsilon \end{array}$$

After evaluating a filter, we execute each body in the resulting thunk under the corresponding environment, and combine all the results by concatenation and

[5] Since we assume that patterns are linear, a repetition pattern P^* allows no variables in P and therefore the premise of rule PRep in fact always returns an empty environment. We chose this presentation only for symmetry.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

labeling. The filter evaluation relation $v \in F \Rightarrow h$ is defined by the following set of rules

$$\frac{v \in P \Rightarrow V}{v \in P \to e \Rightarrow V \to e}\ \textsc{FCla} \qquad \frac{\forall i.\ v_i \in F_i \Rightarrow h_i}{v_1\, v_2 \in F_1\, F_2 \Rightarrow h_1\, h_2}\ \textsc{FCat} \qquad \frac{\exists i.\ v \in F_i \Rightarrow h}{v \in F_1 \mid F_2 \Rightarrow h}\ \textsc{FOr}$$

$$\frac{\forall i.\ v_i \in F \Rightarrow h_i}{v_1 \ldots v_n \in F^* \Rightarrow h_1 \ldots h_n}\ \textsc{FRep} \qquad \frac{a \in L \qquad v \in G(Y) \Rightarrow h}{a[v] \in L[Y] \Rightarrow a[h]}\ \textsc{FLab}$$

$$\frac{}{\epsilon \in \epsilon \Rightarrow \epsilon}\ \textsc{FEps}$$

and the thunk evaluation relation $h \Rightarrow v$ by the following.[6]

$$\frac{}{V \to e \Rightarrow \mathbf{eval}(V, e)}\ \textsc{TCla} \qquad \frac{h_1 \Rightarrow v_1 \qquad h_2 \Rightarrow v_2}{h_1\, h_2 \Rightarrow v_1\, v_2}\ \textsc{TCat} \qquad \frac{h \Rightarrow v}{a[h] \Rightarrow a[v]}\ \textsc{TLab}$$

$$\frac{}{\epsilon \Rightarrow \epsilon}\ \textsc{TEps}$$

3.3 Type inference

Similarly to the operational semantics above, the specification of type inference is parameterized over a typing algorithm $\mathbf{type}(\Gamma, e)$ that takes a type environment Γ and a body id e and returns a type. Here, we define *types* T as patterns from which no variables are reachable, i.e., $\mathbf{reach}(T) = \emptyset$, and type environments Γ as mappings from variables to types.

The specification of inference consists of two steps. In the first step, we assume that a type T is given for input values and obtain a type environment Γ for the variables appearing in the target filter F. (Note that we compute one type environment for the whole filter. There is no danger of name clashes since, as already mentioned, patterns with different body ids have disjoint sets of variables.) We compute the type environment in such a way that, for each variable x, the type $\Gamma(x)$ contains exactly the set of values captured by x as a result of matching values from the input type T against the filter. Formally, we first define the set, written $h(x)$, of values assigned to x in a given thunk h.

$$\begin{array}{lcl} (V \to e)(x) & = & \{V(x)\} \\ (h_1\, h_2)(x) & = & h_1(x) \cup h_2(x) \\ (a[h])(x) & = & h(x) \\ (\epsilon)(x) & = & \emptyset \end{array}$$

[6] The following equivalences have been observed by Wadler:

$$\begin{array}{rcl} a[P \to e] & \equiv & a[P] \to a[e] \\ (P \to e)\,(P' \to e') & \equiv & P\ P' \to e\ e' \\ (P \to e \mid P' \to e')\,(P'' \to e'') & \equiv & P\ P'' \to e\ e'' \mid P'\ P'' \to e'\ e'' \end{array}$$

(where P, P', and P'' do not contain common variables).

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

Then, the type environment Γ satisfies:

$$\mathscr{L}(\Gamma(x)) \quad = \quad \bigcup\{h(x) \mid v \in T,\ \ v \in F \Rightarrow h\}$$

Here, the language $\mathscr{L}(T)$ of a type T is the set of values matched by T. In the second step of the inference, we assume that a type for each body id e is given by $\mathbf{type}(\Gamma, e)$ and obtain a type U for result values. We compute it in the way that U contains the set of values returned by the filter with the assumption that all values from T may be passed to the filter and all values from $\mathbf{type}(\Gamma, e)$ may be returned by the body e. Formally, we first define the set, written $h(\Gamma)$, of values resulted from a given thunk h.

$$\begin{array}{lcl}(V \to e)(\Gamma) & = & \mathscr{L}(\mathbf{type}(\Gamma, e)) \\ (h_1\, h_2)(\Gamma) & = & \{v_1\, v_2 \mid v_1 \in h_1(\Gamma),\ v_2 \in h_2(\Gamma)\} \\ (a[h])(\Gamma) & = & \{a[v] \mid v \in h(\Gamma)\} \\ (\epsilon)(\Gamma) & = & \{\epsilon\}\end{array}$$

Then, the result type U to compute satisfies:

$$\mathscr{L}(U) \quad = \quad \bigcup\{h(\Gamma) \mid v \in T,\ \ v \in F \Rightarrow h\}.$$

In the definitions of $\Gamma(x)$ and U above, it would still be sound if we had only containments ($\mathscr{L}(\Gamma(x)) \supseteq \ldots$ and $\mathscr{L}(U) \supseteq \ldots$). However, we further ensure equalities in order to provide the maximal flexibility to the user; this is why we say that our inference is precise. Despite this strong guarantee, we can compute it efficiently as an algorithm for it will be presented in the next section.

As mentioned before, our inference uses only one type for each body. This restriction is reflected in the fact that we obtain one type environment for the whole filter, from which the typechecking of each body can yield only one result type. What if we remove this restriction and allow more than once to typecheck each body? The answer is that there is an example where we can always get better types by typechecking the same body more times. Consider the filter

```
filter v {
  a[]* as x   {  x, b[], x  }
}
```

with the input type `a[]*`. The current scheme typechecks the body by giving the type `a[]*` to `x` and obtains the result type

```
a[]*,b[],a[]*.
```

However, if we split the input type into the case of the empty sequence and the case of sequences of length one or more, we obtain the result type

```
(),b[],()  |  a[]+,b[],a[]+
```

which is more specific than the previous one. In this way, by splitting the input type into more cases, we can always obtain strictly better types. In the limit, we could split the input type into all the cases of possible input values, where the set of result values would be

$$\{\texttt{a[]}^n\texttt{,b[],a[]}^n \mid n \geqslant 0\}.$$

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

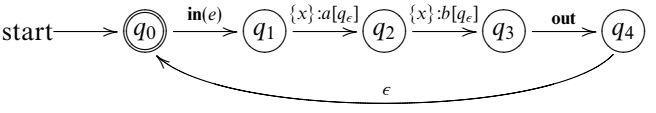
But this set is not expressive by our types (since it is not regular). (Note that this argument already applies to regular expression pattern matching: the problem appears when we consider the precision of the result type, not when we extend the language feature.) Although this limitation is quite disappointing, this seems inevitable as long as we stick to the current approach. We will continue this issue in Section 7, where we will show some possible future directions to address this problem.

4 Automata model

In this section, we introduce a notion of filter automata, which is a finite-state machine model corresponding to regular expression filters. We will use this for describing our type inference algorithm in the next section. The basic part of the model is standard, nondeterministic top-down tree automata accepting binary trees. We extend these to handle the additional functionalities provided by filters. Recall that a filter has a two-layered structure, i.e., the whole is a regular expression over clauses and each clause associates a body id with a pattern, which is a regular expression augmented with variable binders. We represent the whole structure by a tree automaton and add extra annotations to indicate which parts correspond to clauses or variable binders. Let us show an example before formalization. We represent the following filter in the surface language

```
((a[],b[]) as x  { e })*
```

by the filter automaton depicted below:



Here, we suppose that the state q_ϵ accepts the empty sequence. If we ignore **in**(e)- and **out**-transitions and variable sets on label transitions, we can see that the automaton accepts a sequence of alternating a and b nodes each containing the empty sequence. Then, to process each pair of a and b nodes in the input, we first follow an **in**(e)-transition before accepting an a and an **out**-transition after accepting a b. The subpart of the automaton between the **in**(e)- and the **out**-transitions corresponds to a clause in the filter. Inside the subautomaton, we bind the variable x to the sequence of a and b and, when exiting, we "invoke" the body e under the binding. We construct such binding by using label transitions annotated with variable sets. That is, while the automaton runs, we *accumulate* each label in the input whenever following a transition whose variable set contains the target variable. (We annotate a label transition with a set of variables rather than a single variable since variable binders may be nested in the surface language, e.g., `(a[] as x) as y`.)

Formally, a *filter automaton* A is a tuple $(Q, Q^{\mathbf{init}}, Q^{\mathbf{fin}}, T)$ where Q is a finite set of states, $Q^{\mathbf{init}} \subseteq Q$ is a set of initial states, $Q^{\mathbf{fin}} \subseteq Q$ is a set of final states, and T is a

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

set of transition rules of the form $q_1 \xrightarrow{\lambda} q_2$ where λ is defined by the syntax below, each q is a member of Q, and $\overline{x}$ is a set of variables.

$$\begin{array}{lcll} \lambda & ::= & \overline{x} : L[q] & \text{label} \\ & & \mathbf{in}(e) & \text{action-in} \\ & & \mathbf{out} & \text{action-out} \\ & & \epsilon & \text{silent} \end{array}$$

For a transition $q_1 \xrightarrow{\lambda} q_2$, we call q_1 the *source state* and q_2 the *sink state*; when $\lambda = \overline{x} : L[q]$, we call q the *content state*. (Note that, as in the surface language, we allow a label set in each label transition as opposed to standard automata formalisms using single labels.) To lighten the notation, we sometimes omit the set of variables from a label transition if the set is empty. A filter automaton is said *ϵ-free* if it does not have an ϵ-transition. A *tree automaton* is a filter automaton that has no action transition and whose each label transition has the empty variable set.

We make several syntactic restrictions on filter automata to reflect the two-layered structure of the surface language syntax. For this, we define two kinds of reachability—global and local—and their related concepts. Let a filter automaton $A = (Q, Q^{\mathbf{init}}, Q^{\mathbf{fin}}, T)$ given. Global reachability is the most standard one describing that a state can reach another by following any transitions. Formally, a *(global) path* from q_1 to q_n is a sequence $q_1, \ldots, q_n$ of states where, for $i = 1, \ldots, n-1$, either

C1 $q_i \xrightarrow{\lambda} q_{i+1} \in T$ for some λ, or

C2 $q_i \xrightarrow{\overline{x}:L[q_{i+1}]} q' \in T$ for some q', $\overline{x}$, and L.

In this case, q_n is said *(globally) reachable* from q_1; we define $\mathbf{reach}_A(q)$ to be the set of states reachable from q and $\mathbf{reach}_A(R)$ (for a set R of states) to be $\bigcup_{q \in R} \mathbf{reach}_A(q)$. We next introduce local reachability for precisely defining a subautomaton corresponding to a clause, which consists of a set of states that appear between an $\mathbf{in}(e)$- and **out**-transitions and that have no action transition among them. A *local path* from q_1 to q_n is a sequence $q_1, \ldots, q_n$ of states where, for $i = 1, \ldots, n-1$, either **C1** holds with λ having the form $\overline{x} : L[q']$ or ϵ, or **C2** holds. In this case, we say that q_n is *locally reachable* from q_1 and define $\mathbf{local}_A(q)$ to be the set of locally reachable states from q. A state q is in the *scope* of e if $q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T$ and $q \in \mathbf{local}_A(q_2)$ for some q_1 and q_2. Define $\mathbf{scope}_A(e)$ to be the set of states in the scope of e. A state q is said *in scope* when q is in the scope of some body id, and said *out of scope* when q is not in the scope of any body id. A state q_1 is an **in**-state when $q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T$ for some e, q_2, and an **out**-state when $q_1 \xrightarrow{\mathbf{out}} q_2 \in T$ for some q_2. Now, we impose the following restrictions on the given filter automaton for ensuring a "well-formedness" of the scope structure:

- Each state is either in the scope of a unique body id or out of scope.
- No state is an **out**-state if it is out of scope.
- No state is an **in**-state if it is in scope.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

- When $q_1 \xrightarrow{\overline{x}:L[q_2]} q_3 \in T$ with q_1 in scope, no state reachable from q_2 is either an **in**-state or an **out**-state.

The first condition is to avoid the same state being locally reachable from two different **in**-transitions. The second and third conditions are to prevent an **in**-transition with no corresponding **out**-transition or an **out**-transition with no corresponding **in**-transition. Also, scopes cannot be "nested," e.g., two **in**-transitions followed by two **out**-transitions are disallowed; the second, third, and last conditions altogether preclude such situation. We next give a restriction on variable binders. First, define the set $\mathbf{BV}_A(q)$ of variables bound after q:

$$\mathbf{BV}_A(q) = \bigcup\{\overline{x} \mid q_1 \in \mathbf{local}_A(q),\ \ q_1 \xrightarrow{\overline{x}:L[q_2]} q_3 \in T\}.$$

Then, we require the following to the given filter automaton.

When $q_1 \xrightarrow{\overline{x}:L[q_2]} q_3 \in T$ with q_1 in scope, we have that $\mathbf{BV}_A(q_2) \cap (\overline{x} \cup \mathbf{BV}_A(q_3)) = \emptyset$.

This corresponds to the linearity restrictions in the surface language. Note, however, that some non-linear patterns can be translated to automata with the above restriction, e.g.,

```
(a[] as x), (b[] as x).
```

In other words, we can safely relax our definition of linearity for patterns so that variables can capture non-consecutive sequences. This approach has been taken by CDuce (Benzaken *et al.*, 2003).

Next, we define the semantics of filter automata. Analogously to the surface language, a given automaton produces, from an input value, an intermediate structure called annotated value rather than directly emitting an output. The produced annotated value is, in fact, identical to the input value except that it is augmented with variable and action annotations. The annotated value is then processed for forming environments, executing bodies, and constructing a result value. Formally, an *annotated value* ρ is defined by the following syntax.

ρ	::=	$\rho\,\rho$	concatenation
		$a[\rho]^{\overline{x}}$	label with variable set
		ϵ	empty sequence
		$\mathbf{in}(e)$	action-in
		$\mathbf{out}$	action-out

An annotated value is **in-out-***free* when it does not contain $\mathbf{in}(e)$ or **out**. We define $\mathbf{BV}(\rho)$ by the union of the variable sets appearing in ρ. Now, the semantics of filter automata is described by the relation $A \vdash_q v \Rightarrow \rho$, read "automaton A in state q accepts value v and yields annotated value ρ," and inductively defined by the

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

following set of rules.

$$\frac{q \in Q^{\mathbf{fin}}}{A \vdash_q \epsilon \Rightarrow \epsilon}\ \textsc{Fin} \qquad \frac{q_1 \xrightarrow{\epsilon} q_2 \in T \qquad A \vdash_{q_2} v_1 \Rightarrow \rho}{A \vdash_{q_1} v_1 \Rightarrow \rho}\ \textsc{Eps}$$

$$\frac{a \in L \qquad q_1 \xrightarrow{\overline{x}:L[q_3]} q_2 \in T \qquad A \vdash_{q_3} v_1 \Rightarrow \rho_1 \qquad A \vdash_{q_2} v_2 \Rightarrow \rho_2}{A \vdash_{q_1} a[v_1]\, v_2 \Rightarrow a[\rho_1]^{\overline{x}}\, \rho_2}\ \textsc{Lab}$$

$$\frac{q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T \qquad A \vdash_{q_2} v \Rightarrow \rho}{A \vdash_{q_1} v \Rightarrow \mathbf{in}(e)\, \rho}\ \textsc{Enter} \qquad \frac{q_1 \xrightarrow{\mathbf{out}} q_2 \in T \qquad A \vdash_{q_2} v \Rightarrow \rho}{A \vdash_{q_1} v \Rightarrow \mathbf{out}\, \rho}\ \textsc{Exit}$$

We form an environment from an **in-out**-free annotated value by using the following **envof** function

$$\begin{aligned}
&\mathbf{envof}(a[\rho_1]^{\overline{x}}\, \rho_2)(x) \\
&= \begin{cases} a[\mathbf{erase}(\rho_1)]\, \mathbf{envof}(\rho_2)(x) & (x \in \overline{x}) \\ \mathbf{envof}(\rho_1)(x) & (x \notin \overline{x},\ x \in \mathbf{BV}(\rho_1)) \\ \mathbf{envof}(\rho_2)(x) & (x \notin \overline{x},\ x \notin \mathbf{BV}(\rho_1)) \end{cases} \\
&\mathbf{envof}(\epsilon)(x) \quad = \quad \epsilon
\end{aligned}$$

where $\mathbf{erase}(\rho)$ is the value after eliminating all variables from ρ:

$$\begin{aligned}
\mathbf{erase}(a[\rho_1]^{\overline{x}}\, \rho_2) &= a[\mathbf{erase}(\rho_1)]\, \mathbf{erase}(\rho_2) \\
\mathbf{erase}(\epsilon) &= \epsilon
\end{aligned}$$

To understand the definition of **envof**, first note that linearity ensures that, in any annotated value resulted from a matching, the same variables occur only in the same sequence. For example, we may have an annotated value

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

but never

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

Thus, when the **envof** function visits each node $a[\rho_1]^{\overline{x}}\, \rho_2$ of the given annotated value, there are only three cases. First, the node's variable set contains x, in which case we retain this node (by erasing all the annotations from its content ρ_1) and proceed to the remaining sequence ρ_2. In the second case, x occurs in ρ_1. Then, x does not in ρ_2, so we ignore ρ_2. In the third case, x does not occur in ρ_1. Then, x may occur in ρ_2, so we proceed to ρ_2.

Finally, to construct the result value from an annotated value, we use the relation $\rho \rightsquigarrow v$ defined as follows.

$$\frac{\mathbf{eval}(\mathbf{envof}(\rho), e) = v_1 \qquad \sigma \rightsquigarrow v_2}{\mathbf{in}(e)\, \rho\, \mathbf{out}\, \sigma \rightsquigarrow v_1\, v_2}\ \textsc{RCla} \qquad \frac{\sigma_1 \rightsquigarrow v_1 \qquad \sigma_2 \rightsquigarrow v_2}{a[\sigma_1]\, \sigma_2 \rightsquigarrow a[v_1]\, v_2}\ \textsc{RLab} \qquad \frac{}{\epsilon \rightsquigarrow \epsilon}\ \textsc{RFin}$$

In RCla, the annotated value ρ between $\mathbf{in}(e)$ and **out** is fed to $\mathbf{envof}(\rho)$, thus ensured to be **in-out**-free. The relation uses the same evaluation function **eval** as in the previous section.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

It is easy to translate regular expression filters to filter automata by using a variation of the standard translation algorithm from string regular expressions to string automata (Hopcroft & Ullman, 1979). Our concrete translation algorithm is given in Appendix A. Also, we can easily convert any filter automata to ϵ-free filter automata by using the usual ϵ-elimination technique (Comon *et al.*, 1999).

Since the presented semantics contains nondeterminism, a naive implementation of this with backtracking would be inefficient. Although we have not yet worked it out, we believe that we can construct a linear-time algorithm for evaluating filters by adapting existing linear-time algorithms for checking membership of tree automata (Murata *et al.*, 2001).

5 Inference algorithm

In this section, we describe our inference algorithm using filter automata introduced in the last section and give a formal discussion of its correctness.

5.1 Inference for variables

For bound variables, the inference takes as inputs a tree automaton A (representing the input type) and a filter automaton B. For simplicity, we assume that both automata are ϵ-free. The inference algorithm works in three steps. First, we specialize the filter automaton B with respect to the tree automaton A such that the resulting automaton D behaves exactly the same as the original B except that it accepts only values accepted by the automaton A. For this, we use a variation of standard product construction where we preserve the action and the variable binding behaviors in the automaton B. Second, we eliminate spurious transitions that are never used for any input. Third, we obtain, for each variable x, a tree automaton $H^{e,x}$ (where x is bound in the scope of e) such that $H^{e,x}$ accepts a value v if and only if the filter automaton D accepts some value from A and binds x to v. We compute the automaton $H^{e,x}$ from the filter automaton D by retaining all the transitions with variable sets containing x and eliminating all the other transitions.

Formally, we first construct a product automaton $C = (Q_C, Q_C^{\mathbf{init}}, Q_C^{\mathbf{fin}}, T_C)$ from $A = (Q_A, Q_A^{\mathbf{init}}, Q_A^{\mathbf{fin}}, T_A)$ and $B = (Q_B, Q_B^{\mathbf{init}}, Q_B^{\mathbf{fin}}, T_B)$ as follows.

$$
\begin{aligned}
Q_C &= Q_A \times Q_B \\
Q_C^{\mathbf{init}} &= Q_A^{\mathbf{init}} \times Q_B^{\mathbf{init}} \\
Q_C^{\mathbf{fin}} &= Q_A^{\mathbf{fin}} \times Q_B^{\mathbf{fin}} \\
T_C &= \left\{ \langle p_1, q_1 \rangle \xrightarrow{\overline{x}:(K \cap L)[\langle p_3, q_3 \rangle]} \langle p_2, q_2 \rangle \;\middle|\; p_1 \xrightarrow{K[p_3]} p_2 \in T_A,\ \ q_1 \xrightarrow{\overline{x}:L[q_3]} q_2 \in T_B,\ \ K \cap L \neq \emptyset \right\} \\
&\cup \left\{ \langle p, q_1 \rangle \xrightarrow{\mathbf{in}(e)} \langle p, q_1 \rangle \;\middle|\; p \in Q_A,\ \ q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T_B \right\} \\
&\cup \left\{ \langle p, q_1 \rangle \xrightarrow{\mathbf{out}} \langle p, q_2 \rangle \;\middle|\; p \in Q_A,\ \ q_1 \xrightarrow{\mathbf{out}} q_2 \in T_B \right\}
\end{aligned}
$$

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

The first clause of T_C follows the standard technique of product construction except that we take the intersection of the label sets from both transitions and that we copy the variable set in the transition from the automaton B. Note here that we need to check that the intersected label sets are disjoint since otherwise some states might become reachable by following "empty" transitions, making our inference imprecise. For the second and the third clauses, we keep the action annotation on the transition from the automaton B. Since the automaton A should not consume any input while the automaton B takes this action, the created transition connects the state $\langle p, q_1\rangle$ to $\langle p, q_2\rangle$ where the first component remains the same.

Next, we eliminate, from the automaton C, all the states that accept no tree and obtain the automaton $D = (Q_D, Q_D^{\mathbf{init}}, Q_D^{\mathbf{fin}}, T_D)$ defined as follows.

$$\begin{aligned}
Q_D &= \{r \mid C \vdash_r v \Rightarrow \rho\} \\
Q_D^{\mathbf{init}} &= Q_C^{\mathbf{init}} \cap Q_D \\
Q_D^{\mathbf{fin}} &= Q_C^{\mathbf{fin}} \cap Q_D \\
T_D &= \left\{ q_1 \xrightarrow{\overline{x}:a[q_3]} q_2 \in T_C \;\middle|\; q_1, q_2, q_3 \in Q_D \right\} \\
&\cup \left\{ q_1 \xrightarrow{\mathbf{in}(e)} q_2 \in T_C \;\middle|\; q_1, q_2 \in Q_D \right\} \\
&\cup \left\{ q_1 \xrightarrow{\mathbf{out}} q_2 \in T_C \;\middle|\; q_1, q_2 \in Q_D \right\}
\end{aligned}$$

The above definition does not directly give a concrete algorithm for empty state elimination. However, there is a standard linear-time algorithm that works for tree automata and can easily be transferred here (Comon *et al.*, 1999).

The final step is, from the automaton D, to extract, for each variable x, an automaton $H^{e,x}$ representing the set of values that x may capture. Note that such a captured value is the concatenation of the nodes that are matched by x-annotated label transitions (i.e., whose variable set contains x) between an **in**(e)-transition and an **out**-transition in the automaton D. Thus, the basic idea of the inference is to obtain an automaton after extracting only such label transitions from D. A subtlety here is, however, that each state in the automaton D can have two different behaviors. That is, after following an **in**-transition, we are in the "capture mode," where we skip the nodes matched by transitions without x; when we take a label transition *with* x, we get into the "duplicate mode" from the content state of the transition, where we completely retain the structure of the input. Thus, in creating the new automaton $H^{e,x}$, we copy two complete sets of states from the automaton D: a capture-mode state $\langle r, 0\rangle$ and a duplicate-mode $\langle r, 1\rangle$ for each state r of D. Formally, the final step is to compute $H^{e,x} = (Q_{H^{e,x}}, Q_{H^{e,x}}^{\mathbf{init}}, Q_{H^{e,x}}^{\mathbf{fin}}, T_{H^{e,x}})$ defined as follows.

$$\begin{aligned}
Q_{H^{e,x}} &= \left\{ \langle r, 0\rangle, \langle r, 1\rangle \;\middle|\; r \in Q_D \right\} \\
Q_{H^{e,x}}^{\mathbf{init}} &= \left\{ \langle r, 0\rangle \;\middle|\; r' \xrightarrow{\mathbf{in}(e)} r \in T_D,\ \ r' \in \mathbf{reach}_D(Q_D^{\mathbf{init}}) \right\} \\
Q_{H^{e,x}}^{\mathbf{fin}} &= \left\{ \langle r, 0\rangle,\ \langle r, 1\rangle \;\middle|\; r \in Q_D^{\mathbf{fin}} \right\} \\
&\cup \left\{ \langle r, 0\rangle \;\middle|\; r \xrightarrow{\mathbf{out}} r' \in T_D \right\}
\end{aligned}$$

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

$$
\begin{aligned}
T_{H^{e,x}} \;=\; & \left\{ \langle r_1, 0\rangle \xrightarrow{L[\langle r_3,1\rangle]} \langle r_2, 0\rangle \;\middle|\; r_1 \xrightarrow{\overline{x}:L[r_3]} r_2 \in T_D,\;\; x \in \overline{x} \right\} \\
\cup\; & \left\{ \langle r_1, 0\rangle \xrightarrow{\epsilon} \langle r_2, 0\rangle \;\middle|\; r_1 \xrightarrow{\overline{x}:L[r_3]} r_2 \in T_D,\;\; x \notin \overline{x},\;\; x \in \mathbf{BV}_D(r_2) \right\} \\
\cup\; & \left\{ \langle r_1, 0\rangle \xrightarrow{\epsilon} \langle r_3, 0\rangle \;\middle|\; r_1 \xrightarrow{\overline{x}:L[r_3]} r_2 \in T_D,\;\; x \notin \overline{x},\;\; x \notin \mathbf{BV}_D(r_2) \right\} \\
\cup\; & \left\{ \langle r_1, 1\rangle \xrightarrow{L[\langle r_3,1\rangle]} \langle r_2, 1\rangle \;\middle|\; r_1 \xrightarrow{\overline{x}:L[r_3]} r_2 \in T_D \right\}
\end{aligned}
$$

To form initial states of $H^{e,x}$, we take only the sink states of **in**(e)-transitions that are reachable from D's initial states since, as a result of eliminating empty states when computing D from C, some states can become unreachable from initial states. In the first clause of $T_{H^{e,x}}$, we copy all the x-annotated label transitions in the capture mode, where the content state is in the duplicate mode. In the second and third clauses, we create an ϵ-transition in the capture mode for each non-x-annotated label transition in D, where the sink state is either the remainder r_2 or the content r_3 depending on whether x is bound in r_2 or not – recall that the linearity restriction ensures that x is never bound both in r_2 and in r_3. The fourth clause copies all the label transitions in the duplicate mode. Note that a capture-mode state $\langle r, 0\rangle$ is final when r is either an **out**-state or final in D, whereas a duplicate-mode state $\langle r, 1\rangle$ is final only when r is final in D. This is because the automaton is always in the capture mode at the top level (i.e., at the same level as an **in**-transition), and moreover, when moving to the content of a label value, the automaton either continues the capture mode or gets in the duplicate mode depending on whether the variable x appears on the transition itself or inside its content state.

Let k be the number of variables appearing in B. The complexity of the inference algorithm for variables is $O(|T_A| \cdot |T_B| \cdot k)$ since the number of the generated transitions in the first phase (C) is proportional to $|T_A| \cdot |T_B|$ [7], the second phase (D) is linear, and the final phase ($H_{e,x}$) creates k automata each with a linear number of transitions relative to the one in the second phase.

5.2 *Inference for result values*

For result values, the inference takes as inputs a tree automaton J_e for each body id e (representing the set of values returned by e), in addition to the filter automaton D obtained in the last subsection (which represents the target filter restricted to the input type). For simplicity, we assume that J_e is ϵ-free. (Note that D is also ϵ-free from its definition.) We then produce a tree automaton K representing the set of values returned by the filter automaton D.

To see what values should be contained in K, let us trace the behavior of the filter automaton D. Starting from its initial state, we retain all the labels matched by label transitions that are out of scope. After entering in the scope of a body id e, we perform variable binding. When exiting from the scope, we execute the body and emit the resulting value, which, as we have assumed, is contained in J_e. We then

[7] Here, the complexity of computing $K \cap L$ and checking its emptiness is not made clear. However, it does not, at least, depend on m or n

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

go back to the behavior of out-of-scope states. We continue these until we reach a final state.

Thus, in building the automaton K, we basically need to replace each subautomaton in D enclosed by an **in**(e)-transition and an **out**-transition by the corresponding automaton J_e. The replacement can be done by reconnecting the source state of each **in**(e)-transition to J_e's initial states and reconnecting J_e's final states to the sink state of the **out**-transition. However, there are two subtleties here. First, several subautomata in different places may have the same body id but may have different **in**- or **out**-transitions. In this case, we cannot simply replace each subautomaton with a single copy of J_e since this would mix up potentially different constraints expressed before the **in**-transitions (or the **out**-transitions). Instead, we need to replace each subautomaton with a separate copy of J_e and perform the reconnetion for each. Second, since we intend to *concatenate* a value resulting from e to the continuing value, we need to link only the "top-level" final states of J_e – horizontally reachable (formally defined below) from J_e's initial states – to the sink state of the **out**-transition. (Note that non-top-level final states are for the tails of *deeper* nodes of e's result value.) Therefore we duplicate the top-level states of the automaton J_e, but we leave the final states in deeper levels not to be reconnected. Thus, in creating K, we copy one complete set of states from D ("out-of-scope" states), one complete set of states from J_e for each subautomaton described above ("top-level" states), and one complete set of states from J_e ("deeper-level" states). Each state in the second set is written $\langle p, q\rangle$ where p is the sink state of the **in**(e)-transition (the "representative" state to identify which duplicate) and q is a state from D.

Let us formalize these ideas. First, for a given automaton $A = (Q, Q^{\mathbf{init}}, Q^{\mathbf{fin}}, T)$, we define a *horizontal path* from q_1 to q_n to be a sequence $q_1, \ldots, q_n$ of states where, for $i = 1, \ldots, n-1$, we have $q_i \xrightarrow{\lambda} q_{i+1} \in T$ with λ having the form $\overline{x} : L[q']$ or ϵ. In this case, we say that q_n is *horizontally reachable* from q_1 and define $\mathbf{horiz}_A(q)$ to be the set of horizontally reachable states from q. Then, from a given automaton $J_e = (Q_{J_e}, Q^{\mathbf{init}}_{J_e}, Q^{\mathbf{fin}}_{J_e}, T_{J_e})$ and the automaton D computed in the previous section, we obtain $K = (Q_K, Q^{\mathbf{init}}_K, Q^{\mathbf{fin}}_K, T_K)$ defined as follows.

$$
\begin{aligned}
Q_K &= Q_D \cup \textstyle\bigcup_e \{(Q_D \times Q_{J_e}) \cup Q_{J_e}\} \\
Q^{\mathbf{init}}_K &= Q^{\mathbf{init}}_D \\
Q^{\mathbf{fin}}_K &= Q^{\mathbf{fin}}_D \cup Q^{\mathbf{fin}}_{J_e} \\
T_K &= \left\{ p_1 \xrightarrow{L[p_3]} p_2 \in T_D \right\} \\
&\cup \left\{ p_1 \xrightarrow{\epsilon} \langle p_2, q_3\rangle \;\middle|\; p_1 \xrightarrow{\mathbf{in}(e)} p_2 \in T_D,\ \ q_3 \in Q^{\mathbf{init}}_{J_e} \right\} \\
&\cup \left\{ \langle p, q_1\rangle \xrightarrow{L[q_2]} \langle p, q_3\rangle \;\middle|\; p \in Q_D,\ \ q_1 \xrightarrow{L[q_2]} q_3 \in T_{J_e} \right\} \\
&\cup \textstyle\bigcup_e T_{J_e} \\
&\cup \left\{ \langle p_0, q_1\rangle \xrightarrow{\epsilon} p_3 \;\middle|\; q_1 \in Q^{\mathbf{fin}}_{J_e},\ \ p_2 \in \mathbf{horiz}_D(p_0),\ \ p_2 \xrightarrow{\mathbf{out}} p_3 \in T_D \right\}
\end{aligned}
$$

In the first clause of T_K, we simply copy all the transitions from D. In the second clause, we connect the source state of each **in**(e)-transition to each initial state of J_e

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

duplicated for the transition's sink state. The third clause copies the transitions of J_e for each duplicate, where we tag the duplicate's "representative" state on their source and sink states. The fourth clause copies J_e's transitions without tagging. Note that each transition in the third clause has a content state in the fourth clause. The fifth clause connects a final state of p_0's duplicate of J_e to the sink state of each corresponding **out**-transitions. Such a transition has the source state horizontally reachable from the state p_0.

Let l be the sum of the numbers of transitions in the automata J_e for all e. By noticing that each clause of T_K contains at most $|T_D| \cdot l$ transitions, we can easily see that the complexity of the inference algorithm for result values is $O(|T_A| \cdot |T_B| \cdot l)$.

5.3 Correctness

Let us first discuss the correctness of the first part of the inference – for bound variables. We prove that, for each variable x, the automaton $H^{e,x}$ accepts the values that x is bound to as a result of matching the filter automaton B against values from the input type A. To precisely state that x is bound to a value w, we actually need to say that the annotated value yielded by the matching contains an **in-out**-free annotated value ρ as a substructure enclosed by an **in**(e) and an **out**, and that the extraction of the x-marked subnodes from ρ results in the value w. Formally, we first define *contexts*, i.e., annotated values containing a single hole $[\cdot]$:

$$\begin{array}{rcl} S & ::= & a[S]^{\overline{x}}\,\rho \\ & & a[\rho]^{\overline{x}}\,S \\ & & [\cdot]\,\rho \\ & & \mathbf{in}(e)\,S \\ & & \mathbf{out}\,S \end{array}$$

We write $S[\rho]$ for the annotated value after replacing S's hole with ρ. Now, the main theorem for the first part of the inference is as follows.

Theorem 1
The following are equivalent.

1. $A \vdash v$ and $B \vdash v \Rightarrow S[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ where ρ is **in-out**-free and $\mathbf{envof}(\rho)(x) = w$.
2. $H^{e,x} \vdash w$.

This theorem follows from three lemmas shown below (Lemma 1, Lemma 2, and Lemma 5) corresponding to the three steps of the algorithm.

For the first step, we show that the created filter automaton C (in the state $\langle p, q\rangle$) exactly simulates the behavior of the original filter automaton B (in the state q) for the values accepted by the tree automaton A (in the state p).

Lemma 1
$A \vdash_p v$ and $B \vdash_q v \Rightarrow \rho$ if and only if $C \vdash_{\langle p,q\rangle} v \Rightarrow \rho$.

Proof
We first provethe "only if" direction by induction on the derivations of $A \vdash_p v$ and $B \vdash_q v \Rightarrow \rho$ with case analysis on the last rules applied. (To see that all cases are covered, note that A is an ϵ-free tree automaton and therefore it can

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

either terminates by FIN or transits by LAB; B and C are ϵ-free filter automata and therefore can additionally take an action transition.)

- Both A and B terminate by FIN, that is, $p \in Q_A^{\mathbf{fin}}$ and $q \in Q_B^{\mathbf{fin}}$. This implies that $\langle p, q\rangle \in Q_C^{\mathbf{fin}}$. Therefore C terminates by FIN.
- Both A and B transit by LAB. That is, v and ρ each have the form $a[v_1]\, v_2$ and $a[\rho_1]^{\overline{x}}\, \rho_2$. In addition, the last derivations of $A \vdash_p v$ and $B \vdash_q v \Rightarrow \rho$ are:

$$\frac{a \in K \qquad p \xrightarrow{K[p_1]} p_2 \in T_A \qquad A \vdash_{p_1} v_1 \qquad A \vdash_{p_2} v_2}{A \vdash_p a[v_1]\, v_2}$$

$$\frac{a \in L \qquad q \xrightarrow{\overline{x}:L[q_1]} q_2 \in T_B \qquad B \vdash_{q_1} v_1 \Rightarrow \rho_1 \qquad B \vdash_{q_2} v_2 \Rightarrow \rho_2}{B \vdash_q a[v_1]\, v_2 \Rightarrow a[\rho_1]^{\overline{x}}\, \rho_2}$$

  From the definition of C, we have $\langle p, q\rangle \xrightarrow{\overline{x}:(K\cap L)[\langle p_1,q_1\rangle]} \langle p_2, q_2\rangle \in T_C$. Also, by the induction hypothesis, $C \vdash_{\langle p_1,q_1\rangle} v_1 \Rightarrow \rho_1$ and $C \vdash_{\langle p_2,q_2\rangle} v_2 \Rightarrow \rho_2$. The result follows by LAB.
- The other cases are that B transits by ENTER and EXIT. The proof can be done by a straightforward use of the induction hypothesis.

We then prove the "if" direction by induction on the derivation of $C \vdash_{\langle p,q\rangle} v \Rightarrow \rho$ with case analysis on the last rule applied.

- C terminates by FIN. From the definition of C, we have $p \in Q_A^{\mathbf{fin}}$ and $q \in Q_B^{\mathbf{fin}}$. The result follows from FIN.
- C transits by LAB. That is, v and ρ each have the form $a[v_1]\, v_2$ and $a[\rho_1]^{\overline{x}}\, \rho_2$. In addition, the last derivation of $C \vdash_{\langle p,q\rangle} v \Rightarrow \rho$ is:

$$\frac{\begin{array}{c} a \in M \\ \langle p, q\rangle \xrightarrow{\overline{x}:M[\langle p_1,q_1\rangle]} \langle p_2, q_2\rangle \in T_C \end{array} \qquad C \vdash_{\langle p_1,q_1\rangle} v_1 \Rightarrow \rho_1 \qquad C \vdash_{\langle p_2,q_2\rangle} v_2 \Rightarrow \rho_2}{C \vdash_{\langle p,q\rangle} a[v_1]\, v_2 \Rightarrow a[\rho_1]^{\overline{x}}\, \rho_2}$$

  From the definition of C, we have $p \xrightarrow{K[p_1]} p_2 \in T_A$ and $q \xrightarrow{\overline{x}:L[q_1]} q_2 \in T_B$ with $M = K \cap L$. Also, by the induction hypothesis, we have:

$$A \vdash_{p_1} v_1 \qquad A \vdash_{p_2} v_2 \qquad B \vdash_{q_1} v_1 \Rightarrow \rho_1 \qquad B \vdash_{q_2} v_2 \Rightarrow \rho_2$$

  The result follows by LAB.
- The other cases are that C transits by ENTER and EXIT. The proof can be done by a straightforward use of the induction hypothesis. □

For the second step, we show that the automaton C and the automaton D (with no empty states) behave the same.

Lemma 2
$C \vdash_q v \Rightarrow \rho$ if and only if $D \vdash_q v \Rightarrow \rho$.

Proof
Both directions are trivial. □

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

For the third step, we need to prove two technical lemmas before showing the main lemma. First, we show that the automaton $H^{e,x}$ in a state $\langle r, 1\rangle$ accepts the erasures of the values that are produced by the automaton D in the state r. (The erasures are in fact exactly the same as the input values to D, but we formalize in this way for convenience in later lemmas.) The intention here is that the state r is in the duplicate mode. (Note that in such a state, the automaton D produces an annotated value without **in**- or **out**-transitions.)

Lemma 3
The following are equivalent.

1. $D \vdash_r v \Rightarrow \rho$ where ρ is **in-out**-free and $w = \mathbf{erase}(\rho)$.
2. $H^{e,x} \vdash_{\langle r,1\rangle} w$ for all x.

Proof
Both directions can be proved by straightforward induction. □

Next, we show that the automaton $H^{e,x}$ in a state $\langle r, 0\rangle$ accepts the values obtained by taking the **in-out**-free prefixes of the annotated values produced by the automaton D in the state r (in the scope of e) and then extracting the nodes marked x from these. The intention here is that the state r is in the capture mode.

Lemma 4
Let $r \in \mathbf{scope}(e)$. The following are equivalent.

1. $D \vdash_r v \Rightarrow \rho\,\rho'$ where ρ is **in-out**-free and ρ' is either ϵ or $\mathbf{out}\,\rho''$ with $w = \mathbf{envof}(\rho)(x)$.
2. $H^{e,x} \vdash_{\langle r,0\rangle} w$.

Proof
We first prove that (1) implies (2) by induction on the derivation of $D \vdash_r v \Rightarrow \rho\,\rho'$ with case analysis on the last rule applied. (Note that D is an ϵ-free filter automaton.)

Fin or Exit We have $\rho = \epsilon$ with either $r \in Q_D^{\mathbf{fin}}$ or $r \xrightarrow{\mathbf{out}} r' \in T_D$. In either case, we have $\langle r, 0\rangle \in Q_{H^{e,x}}^{\mathbf{fin}}$. The result holds from Fin.

Lab We have:

$$v = a[v_1]\,v_2 \qquad \rho = a[\rho_1]^{\overline{x}}\,\rho_2 \qquad a \in L$$
$$r \xrightarrow{\overline{x}:L[r_1]} r_2 \in T_D \qquad D \vdash_{r_1} v_1 \Rightarrow \rho_1 \qquad D \vdash_{r_2} v_2 \Rightarrow \rho_2$$

By the induction hypothesis, we obtain the following.

$$H^{e,x} \vdash_{\langle r_2,0\rangle} \mathbf{envof}(\rho_2)(x) \tag{1}$$
$$H^{e,x} \vdash_{\langle r_1,0\rangle} \mathbf{envof}(\rho_1)(x) \tag{2}$$

We have three subcases.

- $x \in \overline{x}$. This implies $w = a[\mathbf{erase}(\rho_1)]\,\mathbf{envof}(\rho_2)(x)$. Also, $\langle r, 0\rangle \xrightarrow{a[\langle r_1,1\rangle]} \langle r_2, 0\rangle \in T_{H^{e,x}}$. By Lemma 3, $H^{e,x} \vdash_{\langle r_1,1\rangle} \mathbf{erase}(\rho_1)$. With (1) and Lab, the result follows.
- $x \notin \overline{x}$ and $x \in \mathbf{BV}_D(\rho_1)$. This implies $w = \mathbf{envof}(\rho_1)(x)$. Also, $x \in \mathbf{BV}_D(r_1)$ from $x \in \mathbf{BV}_D(\rho_1)$, and therefore $\langle r, 0\rangle \xrightarrow{\epsilon} \langle r_1, 0\rangle \in T_{H^{e,x}}$. With (2) and Eps, the result follows.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

- $x \notin \overline{x}$ and $x \notin \mathbf{BV}_D(\rho_1)$. This implies $w = \mathbf{envof}(\rho_2)(x)$. We have further two subcases.
  - — $x \notin \mathbf{BV}_D(r_1)$. Then, $\langle r, 0\rangle \xrightarrow{\epsilon} \langle r_2, 0\rangle \in T_{H^{e,x}}$ and therefore the result follows from (1) and Eps.
  - — $x \in \mathbf{BV}_D(r_1)$. Then, $x \notin \mathbf{BV}_D(r_2)$ and therefore $x \notin \mathbf{BV}_D(\rho_2)$, implying $w = \mathbf{envof}(\rho_2)(x) = \epsilon$. Also, $\langle r, 0\rangle \xrightarrow{\epsilon} \langle r_1, 0\rangle \in T_{H^{e,x}}$. On the other hand, from $x \notin \mathbf{BV}_D(\rho_1)$, we have $\mathbf{envof}(\rho_1)(x) = \epsilon$. The result follows from (2) and Eps.

We next prove that (2) implies (1) by induction on the derivation of $H^{e,x} \vdash_{\langle r,0\rangle} w$ with case analysis on the last rule applied. (Note that $H^{e,x}$ is a filter automaton possibly with ϵ-transitions.)

Fin We have $w = \epsilon$. From $\langle r, 0\rangle \in Q^{\mathbf{fin}}_{H^{e,x}}$, we have $r \in Q^{\mathbf{fin}}_D$. The result follows from Fin.

Lab We have:

$$\begin{array}{ll} w = a[w_1]\, w_2 & a \in L \\ \langle r, 0\rangle \xrightarrow{L[\langle r_1,1\rangle]} \langle r_2, 0\rangle \in T_{H^{e,x}} & \\ H^{e,x} \vdash_{\langle r_1,1\rangle} w_1 & H^{e,x} \vdash_{\langle r_2,0\rangle} w_2 \end{array}$$

By the induction hypothesis, $D \vdash_{r_2} v_2 \Rightarrow \rho_2\, \rho_2'$ for some v_2, **in-out**-free ρ_2, and ρ_2' with $\rho_2' = \epsilon$ or $\rho_2' = \mathbf{out}\, \rho_2''$. In addition, $w_2 = \mathbf{envof}(\rho_2)(x)$. From $\langle r, 0\rangle \xrightarrow{L[\langle r_1,1\rangle]} \langle r_2, 0\rangle \in T_{H^{e,x}}$, we have $r \xrightarrow{\overline{x}:L[r_1]} r_2 \in T_D$ and $x \in \overline{x}$. By Lemma 3, $D \vdash_{r_1} v_1 \Rightarrow \rho_1$ for some ρ_1 with $w_1 = \mathbf{erase}(\rho_1)$. From these and Lab, we obtain $D \vdash_r a[v_1]\, v_2 \Rightarrow a[\rho_1]^{\overline{x}}\, \rho_2\, \rho_2'$. Further, since $x \in \overline{x}$, we have $\mathbf{envof}(a[\rho_1]^{\overline{x}}\, \rho_2) = a[\mathbf{erase}(\rho_1)]\, \mathbf{envof}(\rho_2)(x) = a[w_1]\, w_2 = w$. The result follows.

Eps We have:

$$\langle r, 0\rangle \xrightarrow{\epsilon} \langle r_2, 0\rangle \in T_{H^{e,x}} \quad H^{e,x} \vdash_{\langle r_2,0\rangle} w$$

By the induction hypothesis, $D \vdash_{r_2} v_2 \Rightarrow \rho_2\, \rho_2'$ for some v_2, **in-out**-free ρ_2, and ρ_2' with $\rho_2' = \epsilon$ or $\rho_2' = \mathbf{out}\, \rho_2''$. Also, $w = \mathbf{envof}(\rho_2)(x)$. We have three subcases.

- $r \xrightarrow{\overline{x}:L[r_1]} r_2 \in T_D$ and $x \notin \overline{x}$ with $x \notin \mathbf{BV}_D(r_1)$. From the definition of D, we have $D \vdash_{r_1} v_1 \Rightarrow \rho_1$ for some v_1, ρ_1. Then, by Lab, we obtain $D \vdash_r a[v_1]\, v_2 \Rightarrow a[\rho_1]^{\overline{x}}\, \rho_2\, \rho_2'$ for some $a \in L$ (we know $L \neq \emptyset$ by the definition of C). Further, from $x \notin \mathbf{BV}_D(r_1)$, we have $x \notin \mathbf{BV}_D(\rho_1)$ and therefore $\mathbf{envof}(a[\rho_1]^{\overline{x}}\, \rho_2)(x) = \mathbf{envof}(\rho_2)(x) = w$.
- $r \xrightarrow{\overline{x}:L[r_2]} r_1 \in T_D$ and $x \notin \overline{x}$ with $x \in \mathbf{BV}_D(r_2)$. From the definition of D, we have $D \vdash_{r_1} v_1 \Rightarrow \rho_1$ for some v_1, ρ_1. Since $r \in \mathbf{scope}(e)$ and r_2 is the conent state of the transition $r \xrightarrow{\overline{x}:L[r_2]} r_1 \in T_D$, we can assume that $\rho_2' = \epsilon$. Then, by Lab, we obtain $D \vdash_r a[v_2]\, v_1 \Rightarrow a[\rho_2]^{\overline{x}}\, \rho_1$ for $a \in L$. We have further two subcases.
  - — When $x \in \mathbf{BV}_D(\rho_2)$, we have $\mathbf{envof}(a[\rho_2]^{\overline{x}}\, \rho_1)(x) = \mathbf{envof}(\rho_2)(x) = w$. The result follows.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

— When $x \notin \mathbf{BV}_D(\rho_2)$, we have $\mathbf{envof}(a[\rho_2]^{\overline{x}}\,\rho_1)(x) = \mathbf{envof}(\rho_1)(x)$. Further, since $x \in \mathbf{BV}_D(r_2)$ implies $x \notin \mathbf{BV}_D(r_1)$ by the variable restriction, we have $x \notin \mathbf{BV}_D(\rho_1)$ and therefore $\mathbf{envof}(\rho_1)(x) = \epsilon$. On the other hand, $x \notin \mathbf{BV}_D(\rho_2)$ implies $w = \mathbf{envof}(\rho_2)(x) = \epsilon$. The result follows. □

Finally, we show that the automaton $H^{e,x}$ accepts a value w if and only if the automaton D yields an annotated value that contains an **in-out**-free annotated value between an $\mathbf{in}(e)$ and an **out** and w is the extraction of the x-marked subnodes.

Lemma 5
The following are equivalent.

1. $D \vdash v \Rightarrow S[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ where ρ is **in-out**-free and $\mathbf{envof}(\rho)(x) = w$.
2. $H^{e,x} \vdash w$.

Proof
To show that (1) implies (2), we prove a stronger statement that, for all r, if $D \vdash_r v \Rightarrow S[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ where ρ is **in-out**-free and $\mathbf{envof}(\rho)(x) = w$, then $H^{e,x} \vdash w$. The proof proceeds by induction on the derivation of $D \vdash_r v \Rightarrow S[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ with case analysis on the last rule applied.

Lab $v = a[v_1]\,v_2$. We have two subcases.

- $S = a[S']\,\rho'$. We have $r \xrightarrow{L[r_1]} r_2 \in T_D$ and $a \in L$ with $D \vdash_{r_1} v_1 \Rightarrow S'[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ and $D \vdash_{r_2} v_2 \Rightarrow \rho'$. The result follows by the induction hypothesis.
- $S = a[\rho']\,S'$. Similar to the above case.

Enter We have two subcases.

- $S = [\cdot]\,\rho'$. We have $r \xrightarrow{\mathbf{in}(e)} r_2 \in T_D$ with $D \vdash_{r_2} v_2 \Rightarrow \rho\,\mathbf{out}\,\rho'$. The result follows by Lemma 4 and the definition of $Q^{\mathbf{init}}_{H^{e,x}}$.
- $S = \mathbf{in}(e)\,S'$. We have $r \xrightarrow{\mathbf{in}(e)} r_2 \in T_D$ with $D \vdash_{r_2} v_2 \Rightarrow S'[\mathbf{in}(e)\,\rho\,\mathbf{out}]$. The result follows by the induction hypothesis.

Exit $S = \mathbf{out}\,S'$. We have $r \xrightarrow{\mathbf{out}} r_2 \in T_D$ with $D \vdash_{r_2} v_2 \Rightarrow S'[\mathbf{in}(e)\,\rho\,\mathbf{out}]$. The result follows by the induction hypothesis.

We next prove that (2) implies (1). Let $H^{e,x} \vdash_{\langle r',0\rangle} w$ with $\langle r', 0\rangle \in Q^{\mathbf{init}}_{H^{e,x}}$. Then $r \xrightarrow{\mathbf{in}(e)} r' \in T_D$ and $D \vdash_{r'} v \Rightarrow \rho\,\mathbf{out}\,\rho'$ where **in-out**-free and $\mathbf{envof}(\rho)(x) = w$. By Enter, $D \vdash_r v \Rightarrow \mathbf{in}(e)\,\rho\,\mathbf{out}\,\rho'$. From the definition of $H^{e,x}$, the state r is reachable from a state $r_0 \in Q^{\mathbf{init}}_D$. Let $r_0, \ldots, r_n$ be a path from r_0 to r (where $r = r_n$). The result holds if, for all $0 \leqslant i \leqslant n$, we have $D \vdash_{r_i} v \Rightarrow S[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ for some S. The proof proceeds by mathematical induction on $n-i$. The base case is already shown. For the inductive cases, we show only the case $r_i \xrightarrow{\overline{x}:L[r'']} r_{i+1} \in T_D$ for some r'' and $\overline{x}$ since the other cases are similar. By the induction hypothesis, $D \vdash_{r_{i+1}} v' \Rightarrow S'[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ for some v' and S'. From the definition of D, we have $D \vdash_{r''} v'' \Rightarrow \rho''$ for some v'' and ρ''. Also, there is $a \in L$. By letting $v = a[v'']\,v'$ and $S = a[\rho'']^{\overline{x}}\,S'$, the result follows by Lab. □

Next, we turn our attention to the second part of the inference – for result values. First, we define a relation $\rightsquigarrow_J$ for constructing a result value from an annotated

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

value, which slightly modifies the relation $\rightsquigarrow$ defined in section 4 so as to incorporate the assumption that e's result values come from J_e (instead of **eval** applied to e with the environment $\mathbf{envof}(\rho)$).

$$\frac{J_e \vdash v_1 \qquad \sigma \rightsquigarrow_J v_2}{\mathbf{in}(e)\,\rho\,\mathbf{out}\,\sigma \rightsquigarrow_J v_1\,v_2}\ \textsc{JCla} \qquad \frac{\sigma_1 \rightsquigarrow_J v_1 \qquad \sigma_2 \rightsquigarrow_J v_2}{a[\sigma_1]\,\sigma_2 \rightsquigarrow_J a[v_1]\,v_2}\ \textsc{JLab} \qquad \frac{}{\epsilon \rightsquigarrow_J \epsilon}\ \textsc{JFin}$$

(Note that JCla ignores the annotated value ρ between the **in**(e) and **out**.) Clearly, the relation $\rightsquigarrow_J$ is more conservative than $\rightsquigarrow$ in the sense that $\sigma \rightsquigarrow v$ implies $\sigma \rightsquigarrow_J v$, provided $J_e \vdash \mathbf{eval}(\mathbf{envof}(\rho), e)$ for any ρ such that $\sigma = S[\mathbf{in}(e)\,\rho\,\mathbf{out}]$ for some S.

Then, the second main theorem shown below states that the tree automaton K accepts the values that are obtained by executing the filter automaton B for values from A and evaluating the produced annotated value σ by $\rightsquigarrow_J$.

Theorem 2
$A \vdash v$ and $B \vdash v \Rightarrow \sigma$ with $\sigma \rightsquigarrow_J w$ if and only if $K \vdash w$.

This theorem follows from Lemma 1 and 2 together with Lemma 6 shown below (which proves that K in the out-of-scope state p accepts the values that are obtained by executing the filter automaton D in the state p and evaluating the produced annotated value σ by $\rightsquigarrow_J$).

Lemma 6
Let $p \in Q_D$ be out of scope. Then, $D \vdash_p v \Rightarrow \sigma$ and $\sigma \rightsquigarrow_J w$ if and only if $K \vdash_p w$.

Proof
We first prove the "only if" direction by simultaneous induction on the derivations of $D \vdash_p v \Rightarrow \sigma$ and $\sigma \rightsquigarrow_J w$ with case analysis on the last rule applied to the former relation.

Fin The result immediately holds by Fin since $w = \epsilon$ and $p \in Q_D^{\mathbf{fin}} \subseteq Q_K^{\mathbf{fin}}$.

Lab The result follows from a straightforward use of the induction hypothesis and Lab.

Enter From the structural restrictions on filter automata, σ must have the form $\mathbf{in}(e)\,\rho\,\mathbf{out}\,\sigma'$ where ρ is **in-out**-free. This implies $p \xrightarrow{\mathbf{in}(e)} p'' \in T_D$ and $p''' \xrightarrow{\mathbf{out}} p' \in T_D$ with $D \vdash_{p'} v' \Rightarrow \sigma'$ for some p', p'', p''', and v'. Also, from $\sigma \rightsquigarrow_J w$, we have $J_e \vdash w_1$ and $\sigma' \rightsquigarrow_J w_2$ with $w = w_1\,w_2$. By the induction hypothesis, $K \vdash_{p'} w_2$. From the definition of K, we have both $\langle p'', q_1\rangle \xrightarrow{L[q_2]} \langle p'', q_3\rangle \in T_K$ and $q_1 \xrightarrow{L[q_2]} q_3 \in T_K$ for each $q_1 \xrightarrow{L[q_2]} q_3 \in T_{J_e}$. In addition, $p \xrightarrow{\epsilon} \langle p'', q\rangle \in T_K$ for each $q \in Q_{J_e}^{\mathbf{init}}$ and $\langle p'', q'\rangle \xrightarrow{\epsilon} p' \in T_K$ for each $q' \in Q_{J_e}^{\mathbf{fin}}$. Therefore the result $K \vdash_p w_1\,w_2$ follows from Lab and Eps.

We then prove the "if" direction by induction on the derivation of $K \vdash_p v \Rightarrow w$ with case analysis on the last rule applied.

Fin From $w = \epsilon$ and $p \in Q_K^{\mathbf{fin}} \cap Q_D = Q_D^{\mathbf{fin}}$, the result follows from Fin by choosing $v = \epsilon$ and $\sigma = \epsilon$.

Lab We have $w = a[w_1]\,w_2$ for some w_1 and w_2. Also, $a \in L$ and $p \xrightarrow{L[p_1]} p_2 \in T_K$ with $K \vdash_{p_1} w_1$ and $K \vdash_{p_2} w_2$ for some p_1, p_2, and L. From $p \in Q_D$ and the definition

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

of K, we have $p \xrightarrow{L[p_1]} p_2 \in T_D$. By the induction hypothesis, $D \vdash_{p_1} v_1 \Rightarrow \sigma_1$ and $\sigma_1 \rightsquigarrow_J w_1$; also, $D \vdash_{p_2} v_2 \Rightarrow \sigma_2$ and $\sigma_2 \rightsquigarrow_J w_2$. The result follows by Lab.

Eps From $p \in Q_D$, we have $p \xrightarrow{\epsilon} \langle p_1, q_1 \rangle \in T_K$ with $K \vdash_{\langle p_1, q_1 \rangle} w$ for some p_1 and q_1. Then, $p \xrightarrow{\mathbf{in}(e)} p_1 \in T_D$ and $q_1 \in Q_{J_e}^{\mathbf{init}}$. From $K \vdash_{\langle p_1, q_1 \rangle} w$, there is w_1 and w_2 such that $w = w_1 w_2$ where a horizontal path from $\langle p_1, q_1 \rangle$ to $\langle p_1, q_2 \rangle$ in K accepts w_1 and there is a transition $\langle p_1, q_2 \rangle \xrightarrow{\epsilon} p_3 \in T_K$ with $K \vdash_{p_3} w_2$. Therefore there is a horizontal path from q_1 to q_2 in J_e. Also, from $\langle p_1, q_2 \rangle \xrightarrow{\epsilon} p_3 \in T_K$, we have that $q_2 \in Q_{J_e}^{\mathbf{fin}}$ and there is p_2 horizontally reachable from p_1 in D such that $p_2 \xrightarrow{\mathbf{out}} p_3 \in T_D$. Therefore $J_e \vdash w_1$. From $K \vdash_{p_3} w_2$, the induction hypothesis yields $D \vdash_{p_3} v_2 \Rightarrow \sigma_2$ with $\sigma_2 \rightsquigarrow_J w_2$ for some v_2 and σ_2. Since $p \xrightarrow{\mathbf{in}(e)} p_1 \in T_D$ and p_2 is horizontally reachable from p_1, we obtain $D \vdash_p v_1 v_2 \Rightarrow \mathbf{in}(e)\, \rho\, \mathbf{out}\, \sigma_2$ for some $v_1 \rho$. Finally, from $J_e \vdash w_1$ and $\sigma_2 \rightsquigarrow_J w_2$, we obtain $\mathbf{in}(e)\, \rho\, \mathbf{out}\, \sigma_2 \rightsquigarrow_J w_1 w_2$, as desired. □

6 Related work

With the same motivation as ours, the CDuce language (Benzaken *et al.*, 2003) supports features that combine pattern matching and iteration. One notable difference between them and us is that their constructs are limited to operating on each individual element of a sequence and therefore are not capable of the non-uniform processing our filters allow. Another difference is that they provide three separate constructs with slightly different semantics corresponding to most common usages—`map` for operating each of the top-level elements and leaving out unmatched ones, `transform` for a similar operation but retaining unmatched elements, and `xtransform` for also a similar operation except that it retains the *label* of each unmatched element and recursively applies the same transformation to the content of the label. In contrast, our approach is to provide a single feature to support all purposes and indeed all of `map`, `transform`, `xtransform` can be encoded by our filters. For example, a CDuce `map`

```
map e {
  P1 -> e1
| ...
| Pn -> en
}
```

(where each pattern here is required to match only a single element) can be written as

```
filter e {
  ( P1      {  e1  }
  | ...
  | Pn      {  en  }
  || AnyOne {  ()  } )*
}
```

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

and similarly for a `transform`. For an `xtransform`

```
xtransform e {
  P1 -> e1
| ...
| Pn -> en
}
```

we can express it by first defining a recursive filter

```
rule F =
  ( P1      {  e1  }
  | ...
  | Pn      {  en  }
  || ~[ F ] )*
```

and apply it to the input:

```
filter e { F }
```

Another similar proposal is a simple for-each style iterator with a type-matching facility in XML Query (Fernández *et al.*, 2001; Fankhauser *et al.*, 2001). As briefly discussed in the introduction, since these constructs do not have a strong connection to regular expressions, it is quite difficult to process XML data with slightly unusual types such as `(a,b*)*` or process data in a non-uniform way as discussed in Sections 2.3 and 2.4.

A popular idiom found in many query languages for XML (Deutsch *et al.*, 1998; Cardelli & Ghelli, 2001; Abiteboul *et al.*, 1997; Fankhauser *et al.*, 2001) is a construct of the form `select` *e* `where` *p*, which collects the set of all bindings resulted from matching the input value against the pattern *p*, then evaluates the expression *e* under each binding, and finally concatenates all the results. Usually in this style of features, the "matching" part uses powerful pattern languages and therefore is quite expressive, whereas the "processing" part is not as satisfactory since it allows only one expression for any match. In particular, it is typically difficult to process different occurrences (or cases) of data in different ways, which is exactly what regular expression filters are good at.

A technique closely related to our type inference is CDuce's, which computes types both for bound variables and result values of `map`, `transform`, and `xtransform` constructs as well as pattern matches. The main difference from ours is, however, what types to be computed. First, they allow the inference to go over each body expression more than once, whereas we limit it to only once. As a result, they can obtain better types than ours. However, as argued in Section 3.3, it is impossible to obtain precise types if we remove our "only once" restriction. Thus, a difficulty arises how to give a specification of the inference; as of writing this paper, they have no accurate specification and this might make it hard for the user to figure out the reasons of type errors reported by the system.

Although handling the general case is difficult, some restrictions could yield a complete type inference technique. One approach is to restrict the language so

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

that precise analysis becomes decidable. For example, there is a large collection of work on precise type inference techniques for different computation models such as k-pebble tree transducers (Milo *et al.*, 2000), subsets of XSLT (Tozawa, 2001; Martens & Neven, 2003), models based on macro tree transducers (Perst & Seidl, 2004; Maneth *et al.*, 2005), and extended path expressions (Murata, 2001). Their work is, though, still in the theoretical level and actual feasibility is yet to be seen. More practical yet complete algorithms based on "type splitting" have been used first in the initial proposal of a type system for XQuery (Fernández *et al.*, 2001) and then in a typed-based analysis for detecting never-matching path expressions in XQuery programs (Colazzo *et al.*, 2004). Notably, both use a technique that splits the input type into several and typechecks the body expression multiple times. In the former work, splitting is done in a simple manner based on the syntax of the given input type and completeness is obtained for simple path expressions with only child axis. (The *current* specification of XML Query, however, uses an even simpler inference and has no desirable property of precision.) The latter work handles more general queries and achieves completeness with a certain restriction on types that ensures splitting to be finite.

7 Conclusions

We have shown that the simple idea of regular expressions on pattern clauses can yield significant expressiveness allowing non-trivial and useful programming idioms. Though, no single language feature is suitable for every purpose. Indeed, the kind of processing that filters permit is, roughly, the *map* operation as in usual functional languages; we cannot express *fold*-like processing, for example. However, rather than trying to extend our feature to allow as many programming patterns as possible, we prefer to keep it simple and easy to use. On the other hand, the effort required for implementing filters is not so big. The type inference is a series of simple operations on automata and the addition from our previous inference for pattern matching is rather moderate.

One dissatisfaction about the present proposal is the precision of the type inference, as discussed in Section 3—the computed types are sometimes not precise enough due to the restriction that the inference can use only one type for each body expression. For breaking through this obstacle, there are at least two directions. One possibility, taken by CDuce's map and transform (Benzaken *et al.*, 2003), is to give up having an accurate specification of type inference and compute *some* type that is more precise than ours. Whether this lack of specification is acceptable from the user's point of view is, however, yet to be seen. (There might be an intermediate solution that gives both a simple specification and a better precision, but whether it exists is still an open question.) Another possibility is to pursue a *backward inference* approach, which computes input types from output types (the opposite to our inference) (Milo *et al.*, 2000; Tozawa, 2001). This approach has successfully dealt with similar problems in several different settings, and therefore seems to be the most promising at the moment.

Acknowledgments

I would like to express my best gratitude to Vladimir Gapeyev, Michael Levin, Makoto Murata, and Alain Frisch for precious comments and useful discussions. The paper was greatly improved by the comments made by the anonymous reviewers of POPL'04, PLAN-X'04, and Journal of Functional Programming. This work was supported by Kayamori Foundation of Information Science Advancement, The Inamori Foundation, and Japan Society for the Promotion of Science.

A Translation from filters to filter automata

Given a pattern definition E, a filter definition G, and a "starting" filter name Y_0, we create an automaton $A^{\mathbf{all}} = (Q_{A^{\mathbf{all}}}, Q^{\mathbf{init}}_{A^{\mathbf{all}}}, Q^{\mathbf{fin}}_{A^{\mathbf{all}}}, T_{A^{\mathbf{all}}})$ where

$$\begin{aligned}
Q_{A^{\mathbf{all}}} &= \textstyle\bigcup_{X\in\mathbf{dom}(E)}(Q_{E(X),\emptyset} \cup \{q_X\}) \\
&\cup \textstyle\bigcup_{Y\in\mathbf{dom}(G)}(Q_{G(Y)} \cup \{q_Y\}) \\
Q^{\mathbf{init}}_{A^{\mathbf{all}}} &= q_{Y_0} \\
Q^{\mathbf{fin}}_{A^{\mathbf{all}}} &= \textstyle\bigcup_{X\in\mathbf{dom}(E)} Q^{\mathbf{fin}}_{E(X),\emptyset} \cup \bigcup_{Y\in\mathbf{dom}(G)} Q^{\mathbf{fin}}_{G(Y),\emptyset} \\
T_{A^{\mathbf{all}}} &= \textstyle\bigcup_{X\in\mathbf{dom}(E)}(T_{E(X),\emptyset} \cup (\{q_X\} \xrightarrow{\epsilon} Q^{\mathbf{init}}_{E(X),\emptyset})) \\
&\cup \textstyle\bigcup_{Y\in\mathbf{dom}(G)}(T_{G(Y),\emptyset} \cup (\{q_Y\} \xrightarrow{\epsilon} Q^{\mathbf{init}}_{G(Y),\emptyset}))
\end{aligned}$$

and $A_{P,\overline{x}} = (Q_{P,\overline{x}}, Q^{\mathbf{init}}_{P,\overline{x}}, Q^{\mathbf{fin}}_{P,\overline{x}}, Q^{\mathbf{fin}}_{P,\overline{x}})$ and $A_F = (Q_F, Q^{\mathbf{init}}_F, Q^{\mathbf{fin}}_F, Q^{\mathbf{fin}}_F)$ are inductively defined as follows.

$$\begin{aligned}
A_{\epsilon,\overline{x}} &= \langle \{q_1\},\ \{q_1\},\ \{q_1\},\ \emptyset \rangle \\
A_{L[X],\overline{x}} &= \langle \{q_1, q_2\},\ \{q_1\},\ \{q_2\},\ \{q_1 \xrightarrow{\overline{x}:L[q_X]} q_2\} \rangle \\
A_{(P_1\,P_2),\overline{x}} &= \langle Q_{P_1,\overline{x}} \cup Q_{P_2,\overline{x}},\ Q^{\mathbf{init}}_{P_1,\overline{x}},\ Q^{\mathbf{fin}}_{P_2,\overline{x}},\ T_{P_1,\overline{x}} \cup T_{P_2,\overline{x}} \cup (Q^{\mathbf{fin}}_{P_1,\overline{x}} \xrightarrow{\epsilon} Q^{\mathbf{init}}_{P_2,\overline{x}}) \rangle \\
A_{(P_1\,|\,P_2),\overline{x}} &= \langle Q_{P_1,\overline{x}} \cup Q_{P_2,\overline{x}},\ Q^{\mathbf{init}}_{P_1,\overline{x}} \cup Q^{\mathbf{init}}_{P_2,\overline{x}},\ Q^{\mathbf{fin}}_{P_1,\overline{x}} \cup Q^{\mathbf{fin}}_{P_2,\overline{x}},\ T_{P_1,\overline{x}} \cup T_{P_2,\overline{x}} \rangle \\
A_{P^*,\overline{x}} &= \langle Q_{P,\overline{x}} \cup \{q_1, q_2\},\ \{q_1\},\ \{q_2\},\ T_{P,\overline{x}} \cup ((Q^{\mathbf{fin}}_{P,\overline{x}} \cup \{q_1\}) \xrightarrow{\epsilon} (Q^{\mathbf{init}}_{P,\overline{x}} \cup \{q_2\})) \rangle \\
A_{(P\ \mathbf{as}\ x),\overline{x}} &= A_{P,\overline{x}\cup\{x\}}
\end{aligned}$$

$$\begin{aligned}
A_{\epsilon} &= \langle \{q_1\},\ \{q_1\},\ \{q_1\},\ \emptyset \rangle \\
A_{L[Y]} &= \langle \{q_1, q_2\},\ \{q_1\},\ \{q_2\},\ \{q_1 \xrightarrow{L[q_Y]} q_2\} \rangle \\
A_{(F_1\,F_2)} &= \langle Q_{F_1} \cup Q_{F_2},\ Q^{\mathbf{init}}_{F_1},\ Q^{\mathbf{fin}}_{F_2},\ T_{F_1} \cup T_{F_2} \cup (Q^{\mathbf{fin}}_{F_1} \xrightarrow{\epsilon} Q^{\mathbf{init}}_{F_2}) \rangle \\
A_{(F_1\,|\,F_2)} &= \langle Q_{F_1} \cup Q_{F_2},\ Q^{\mathbf{init}}_{F_1} \cup Q^{\mathbf{init}}_{F_2},\ Q^{\mathbf{fin}}_{F_1} \cup Q^{\mathbf{fin}}_{F_2},\ T_{F_1} \cup T_{F_2} \rangle \\
A_{F^*} &= \langle Q_F \cup \{q_1, q_2\},\ \{q_1\},\ \{q_2\},\ T_F \cup ((Q^{\mathbf{fin}}_F \cup \{q_1\}) \xrightarrow{\epsilon} (Q^{\mathbf{init}}_F \cup \{q_2\})) \rangle \\
A_{P\to e} &= \langle Q_{P,\emptyset} \cup \{q_1, q_2\},\ \{q_1\},\ \{q_2\},\ T_{P,\emptyset} \cup (\{q_1\} \xrightarrow{\mathbf{in}(e)} Q^{\mathbf{init}}_{P,\emptyset}) \cup (Q^{\mathbf{fin}}_{P,\emptyset} \xrightarrow{\mathbf{out}} \{q_2\}) \rangle
\end{aligned}$$

Here, $Q_1 \xrightarrow{\lambda} Q_2$ means $\{q_1 \xrightarrow{\lambda} q_2 \mid q_1 \in Q_1,\ q_2 \in Q_2\}$. Note that the construction for the empty sequence, labels, concatenations, alternations, and repetitions of both

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

patterns and filters is exactly the same as the standard one (Hopcroft & Ullman, 1979). The correctness proof of the translation is omitted.

B Complex regular expressions

In this section, we give a collection of regular expressions taken from real-world DTDs. These regular expressions have a certain complex structure that can defeat the use of a standard for-each style of language features mentioned in the introduction. Specifically, we choose the following characterization of such structure: a regular expression is *complex* if it has, as subexpression, either

- a concatenation of two expressions containing the same label, or
- a repetition containing a concatenation.

An example of the first is `((a,b),a)` and an example of the second is `(a,b)*`.

Below, we pick up six DTDs and quote their element and entity declarations with complex regular expressions. We omit the definitions of some of the entities referenced here when they are not important for the present purpose. The other entities are already expanded.

B.1 DocBook 4.2

The element declarations shown below can be found in the file `dbhierx.mod` (OASIS, 2002). Almost the same structure as the element `appendix` is used for the elements `chapter`, `preface`, `section`, `sect1`, `sect2`, `sect3`, `sect4`, and `sect5` (omitted here).

```
<!ELEMENT appendix
    (beginpage?,
     appendixinfo?,
     (%bookcomponent.title.content;),
     (toc|lot|index|glossary|bibliography)*,
     tocchap?,
     (%bookcomponent.content;),
     (toc|lot|index|glossary|bibliography)*)>

<!ELEMENT indexentry
    (primaryie, (seeie|seealsoie)*,
     (secondaryie, (seeie|seealsoie|tertiaryie)*)*)>

<!ELEMENT refmeta
    ((indexterm)*,
     refentrytitle, manvolnum?, refmiscinfo*,
     (indexterm)*)>
```

B.2 MusicXML 0.8

The elements `key` and `time` are declared in the file `attributes.dtd`, the element `harmony` in `direction.dtd`, and the elements `ornaments` and `lyric` in `note.dtd` (LLC., 2004).

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

```
<!ELEMENT key
    ((cancel?, fifths, mode?) |
     ((key-step, key-alter)*))>

<!ELEMENT time
    ((beats, beat-type)+ | senza-misura)>

<!ELEMENT harmony
    (((root | function), kind,
     inversion?, bass?, degree*)+)>

<!ELEMENT ornaments
    (((trill-mark | turn | delayed-turn |
       shake | wavy-line | mordent |
       inverted-mordent | schleifer |
       other-ornament),
      accidental-mark*)*)>

<!ELEMENT lyric
    ((((syllabic?, text),
       (elision, syllabic?, text)*, extend?) |
      extend | laughing | humming),
     end-line?, end-paragraph?)>
```

B.3 RecipeML 0.5

The following can be found in (FormatData, 2000).

```
<!ENTITY % amt.cont '(amt, (sep?, amt)*)'>

<!ENTITY % time.cont '(time, (sep?, time)*)'>

<!ENTITY % temp.cont '(temp, (sep?, temp)*)'>

<!ELEMENT equipment
    (equip-div+ | (note*, tool, (note | tool)*))>

<!ELEMENT equip-div
    (title?, description?, note*, tool,
     (note | tool)*)>

<!ELEMENT ingredients
    (ing-div+ | (note*, ing, (note | ing)*))>

<!ELEMENT ing-div
    (title?, description?, note*, ing,
     (note | ing)*)>
```

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

```
<!ELEMENT directions
    (dir-div+ | ((note | ing)*, step,
                 (note | ing | step)*))>

<!ELEMENT dir-div
    (title?, description?, (note | ing)*, step,
     (note | ing | step)*)>

<!ELEMENT amt
    ((qty | range)?, size?, unit?, size?)>
```

B.4 Adex 1.2

The following can be found in (Newspaper Association of America, 1999).

```
<!ELEMENT transfer-info
    (transfer-number, (from-to, company-id)+,
     contact-info)*>

<!ELEMENT days-and-hours
    (date, time)+>
```

B.5 SMIL 2.0

The following is in the file `smil-model-1.mod` (W3C, 2005).

```
<!ENTITY % SMIL.head.content
    (meta*,
     (customAttributes, meta*)?,
     (metadata, meta*)?,
     ((layout|switch), meta*)?,
     (transition+, meta*)?)>
```

B.6 W3C XML Specification 2.1

The following can be found in (W3C, 1998).

```
<!ELEMENT prod
    (lhs, (rhs, (com|wfc|vc|constraint)*)+)>
```

References

Abiteboul, S., Quass, D., McHugh, J., Widom, J. & Wiener, J. L. (1997) The Lorel query language for semistructured data. *Int. J. Digital Libraries*, **1**(1), 68–88.

Benzaken, V., Castagna, G. & Frisch, A. (2003) CDuce: An XML-centric general-purpose language. *Pages 51–63 of: Proceedings of the International Conference on Functional Programming (ICFP)*.

https://doi.org/10.1017/S0956796806005909 Published online by Cambridge University Press

Bray, T., Paoli, J., Sperberg-McQueen, C. M. & Maler, E. (2000) *Extensible markup language (XML™)*. http://www.w3.org/XML/.

Cardelli, L. & Ghelli, G. (2001) A query language for semistructured data based on the Ambient Logic. *Pages 1–22 of: Proceedings of 10th European Symposium on Programming*. LNCS, no. 2028.

Clark, J. (1999) *XSL Transformations (XSLT)*. http://www.w3.org/TR/xslt.

Clark, J. & Murata, M. (2001) *RELAX NG*. http://www.relaxng.org.

Cluet, S. & Siméon, J. (1998) Using YAT to build a web server. *Pages 118–135 of: Intl. Workshop on the Web and Databases (WebDB)*.

Colazzo, D., Ghelli, G., Manghi, P., & Sartiani, C. (2004) Types for path correctness of XML queries. *Pages 126–137 of: International Conference on Functional Programming (ICFP)*.

Comon, H., Dauchet, M., Gilleron, R., Jacquemard, F., Lugiez, D., Tison, S. & Tommasi, M. (1999) *Tree automata Techniques and Applications*. Draft book; available electronically on http://www.grappa.univ-lille3.fr/tata.

Deutsch, A., Fernandez, M., Florescu, D., Levy, A. & Suciu, D. (1998) *XML-QL: A Query Language for XML*. http://www.w3.org/TR/NOTE-xml-ql.

Fallside, D. C. (2001). *XML Schema Part 0: Primer, W3C Recommendation*. http://www.w3.org/TR/xmlschema-0/.

Fankhauser, P., Fernández, M., Malhotra, A., Rys, M., Siméon, J. & Wadler, P. (2001) *XQuery 1.0 Formal Semantics*. http://www.w3.org/TR/query-semantics/.

Fernández, M. F., Siméon, J. & Wadler, P. (2001) A semi-monad for semi-structured data. *Pages 263–300 of:* den Bussche, J. V. and Vianu, V. (eds.), *Proceedings of 8th International Conference on Database Theory (ICDT 2001)*. Lecture Notes in Computer Science, vol. 1973. Springer.

FormatData (2000) *RecipeML*. http://www.formatdata.com/recipeml/.

Frisch, A., Castagna, G. & Benzaken, V. (2002) Semantic subtyping. *Pages 137–146 of: Seventeenth Annual IEEE Symposium on Logic in Computer Science*.

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

Hopcroft, J. E. & Ullman, J. D. (1979) *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Hosoya, H. (2003) *Regular expression pattern matching — a simpler design*. Tech. rept. 1397. RIMS, Kyoto University.

Hosoya, H. & Pierce, B. C. (2002) Regular expression pattern matching for XML. *J. Funct. Program.* **13**(6), 961–1004. (Short version appeared in *Proceedings of The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 67–80, 2001.)

Hosoya, H. & Pierce, B. C. (2003) XDuce: A typed XML processing language. *ACM Trans. Internet Technol.* **3**(2), 117–148. (Short version appeared in *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, volume 1997 of Lecture Notes in Computer Science, pp. 226–244, Springer-Verlag.)

Hosoya, H., Vouillon, J. & Pierce, B. C. (2000) Regular expression types for XML. *Pages 11–22 of: Proceedings of the International Conference on Functional Programming (ICFP)*.

Hosoya, H., Frisch, A. & Castagna, G. (2005) Parametric polymorphism for XML. *Pages 50–62 of: The 32nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*.

LLC., Recordare (2004) *MusicXML*. http://www.recordare.com/xml.html.

Lu, K. Z. M. & Sulzmann, M. (2004) *XHaskell: Regular expression types for Haskell*. Manuscript.

Maneth, S., Perst, T., Berlea, A. & Seidl, H. (2005) XML type checking with macro tree transducers. *Proceedings of Symposium on Principles of Database Systems (PODS)*.

Martens, W. & Neven, F. (2003) Typechecking top-down uniform unranked tree transducers. *Pages 64–78 of: Proceedings of International Conference on Database Theory*.

Meijer, E. & Shields, M. (1999) *XMλ: A functional programming language for constructing and manipulating XML documents*. Manuscript.

Milo, T., Suciu, D. & Vianu, V. (2000) Typechecking for XML transformers. *Pages 11–22 of: Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM.

Murata, M., Lee, D. & Mani, M. (2001) Taxonomy of XML schema languages using formal language theory. *Extreme Markup Languages*.

Murata, M. (2001) Extended path expressions for XML. *Pages 126–137 of: Proceedings of Symposium on Principles of Database Systems (PODS)*.

Newspaper Association of America (1999) *NAA Classified Advertising Standards Task Force*. `http://www.naa.org/technology/clsstdtf/`.

OASIS (2002) *DocBook*. `http://www.docbook.org`.

Perst, T. & Seidl, H. (2004) Macro forest transducers. *Infor. Process. Lett.* **89**(3), 141–149.

Tozawa, A. (2001) Towards static type checking for XSLT. *Proceedings of ACM Symposium on Document Engineering*.

W3C (1998) *W3C XML Specification*. `http://www.w3.org/XML/1998/06/xmlspec-report.htm`.

W3C (2005) *Synchronized Multimedia Integration Language*. `http://www.w3.org/AudioVideo/`.