Error Mining for Regular Expression Patterns

Giuseppe Castagna[1], Dario Colazzo[2], and Alain Frisch[3]

[1] CNRS, Ecole Normale Supérieure de Paris, France
[2] LRI, Université Paris Sud, Orsay, France
[3] INRIA, Rocquencourt, France

Abstract. In the design of type systems for XML programming languages based on regular expression types and patterns the focus has been over *result analysis*, with the main aim of statically checking that a transformation always yields data of an expected output type. While being crucial for correct program composition, result analysis is not sufficient to guarantee that patterns used in the transformation are correct. In this paper we motivate the need of static detection of incorrect patterns, and provide a formal characterization based on pattern matching operational semantics, together with locally exact type analysis techniques to statically detect them.

1 Introduction

Current type systems for query and transformation languages for XML data, such as those of XQuery [DFF^{+}05], ℂDuce [BCF03], C_{ω} [BMS05], XDuce [Hos00] mainly aim at result type analysis, that is at statically inferring the output type of a query or transformation function, starting from its structural requirements (XPath paths or ML-like patterns) and the input type.

Result analysis has a crucial importance as by statically knowing the output type, we can check if it is included in the input type required by some other application. Hence, being the output type an upper bound for values resulted by the query/function (type soundness), result type analysis constitutes a powerful tool for sound system composition.

Unfortunately, while result analysis is often sufficient for programming languages that deal with simple data structures, this is no longer true for languages manipulating complex data structures as it is the case for XML.

Working on XML trees requires two different powerful language primitives: (i) iterator primitives in order to navigate XML trees and (ii) deconstructing primitives (usually called patterns or templates) in order to capture subparts of their structure. The result analysis is often sufficient to verify correctness of iterators, but it is useless to spot errors hidden inside the deconstructing primitives. In the context of XML processing languages two different classes of deconstructing primitives can be found: path expressions (usually XPath paths, but also the “dot” navigation of C_{ω}) and regular expression patterns.

Path expressions are navigational primitives that finger where to capture data substructures. They closely resemble the homonymous primitives used by OQL in the contexts of OODB query languages with the difference that they return sets or sequences of elements, those that can be reached by the paths they define. These are at the basis of standard languages such as XSLT or XQuery.

More recently a new kind of deconstructing primitives was proposed, regular expression patterns [HP01], which extends by regular expressions the pattern matching as popularized by functional languages such as ML and Haskell. Regular expression patterns were first introduced in the XDuce [HP00] programming language and then adopted by other projects such as ℂDuce [BCF03] and its query language ℂQL [BCM05], Xtatic [GP03], Scala [OAC$^+$04], XHaskell [LS04] as well as the extension of Haskell proposed in [BFS04].

As we said result analysis is not sufficient to spot errors a programmer would have done in defining or writing paths or regular expressions patterns (from now on, "patterns" for short): indeed with the current technology a program containing errors in paths/patterns can (and in the absence of other errors, will) type check. In general, it is difficult to precisely characterize the class of wrong patterns or paths. An approximation is to consider as wrong those patterns/paths which contains subparts that are meaningless that is, roughly, that they are never be used whatever the input of the path/pattern is.

The problem of characterizing and detecting correctness of XPath expressions has been recently tackled by Colazzo et al. [Col04,CGMS04]. The authors provide quite a precise type analysis technique that, by checking the absence of matching between paths and input types, statically detects empty sub-queries of XQuery queries.

In this work we study the same problem for the other family of deconstructing primitives, that is regular expression patterns. In particular, we show how to formally define and statically detect patterns that contain subpatterns which are "never used". We develop our approach for the pattern algebra of the ℂDuce programming language since this algebra is the most general among those of the cited languages: the pattern algebras of the other languages are subsumed by the one of ℂDuce, therefore our technique can be straightforwardly adapted to them with few or no modifications.

To that end we study how such a kind of *local* errors can be (i) formally characterised in terms of operational semantics (hence, independently from a particular set of type rules) and (ii) statically detected by means of some improvements of the existing type systems. In particular, before defining the extended type system, we give several examples of practical and theoretical motivations of our study and, then, we give a formal characterization of the class of errors we want to mine. As we will see, the problem is not obvious to solve, due to possible high irregularities in types and patterns. However, the rich type algebra of ℂDuce will ensure a sound and complete analysis for a single pattern matching. The analysis reports a set of sub-patterns which are never used considering a given input type for the pattern. This analysis can be added to the ℂDuce type-checker. Of course, the analysis is then only locally exact (it is exact assuming that the type-checker gives the argument of the pattern matching a type which exactly denotes all the possible values of this argument at run-time), but globally sound (if it reports an unused sub-pattern, this sub-pattern is really useless and hence probably wrong).

Overview The article is organized as follows. In the next section we provide some practical examples of the kind of errors we want to statically detect and that elude current type checkers technology. We also show the relevance of such errors and the importance to detect them when programming XML transformations. In Section 3 we formally define the class of errors we are interested in, together with a sound a complete analysis to statically detect them. In Section 4 we discuss the characteristic of our analysis and show how to embed it in existing type checkers.

2 Motivating examples

In writing programs that process typed XML data, programmers are very likely to specify in their patterns, only the part of the schema that is strictly necessary to recover desired data. This is almost always the case when writing programs that query XML data, but even in the context of XML transformation programs, very often, only a sub part of the input structure must be matched and processed.

Partial specification of structural requirements can be specified in regular expression patterns by using the wildcard pattern "_" which matches every value. This is of crucial importance as it enormously simplifies coding of programs and makes them more robust to possible evolutions of the data schemas. However, at the same time, the extensive use of the wildcard patterns is an important (but not exclusive) source of the kind of errors that we target in this paper: the common practise of a massive use of wildcard patterns, thus, makes the presence of undetected errors very likely, whence the importance of our analysis.

As we will explain, the presence of incorrect patterns may strongly compromise quality of system behavior, as incorrect patterns never match data, and, as a consequence, some desired data may end up to not contribute to partial and/or final results, without having the possibility of becoming aware of this problem at compile time. So, negative effects of this problem may be visible only by careful observing the results of the programs. This makes error detection quite difficult and the subsequent debugging very hard.

Let us see all of this on a standard example, and use it to introduce ℂDuce patterns. Consider the following schema:

```
type Bib    = <bib>[Book*]
type Book   = <book year=String>[Title (Author+|Editor+) Price?]
type Author = <author>[Last First]
type Editor = <editor>[Last First]
type Title  = <title>[PCDATA]
type Last   = <last>[PCDATA]
type First  = <first>[PCDATA]
type Price  = <price>[PCDATA]
```

The declarations above should not pose any problem to the reader familiar with XML, DTD, and XML Schema. The type `Bib` classifies XML-trees rooted at tag `bib` that delimits a possibly empty list of books. These are elements with tag `book`, an attribute `year`, and containing a sequence formed exactly by one element title, followed by either a non empty list of author elements, or a non empty list of editor elements, and ended by an optional element price. Title elements are tagged by `title` and contain a sequence of characters, that is, a string (in XML terminology "parsed character data", i.e. PCDATA). The other declarations have similar explanations.

The declarations above give a rather complete presentation of ℂDuce types: there are XML types, that are formed by a tag part and a sequence type (denoted by square brackets). The content of a sequence type is described by a regular expression on types, that is, by the juxtaposition, the application of `*`, `+`, `?` operators, and the union `|` of types. Besides these types there also are: (*i*) values which are considered singleton types, so for instance `"Colazzo"` is the type that contains only the string `"Colazzo"`, (*ii*) intersection of types, denoted by *s***&***t* that contains all the values that have both type *s* and type *t*, (*iii*) difference "\" of types, so that the type

```
<book year=String"1999">[Title (Author+|Editor+) Price?]
```

is the type of all books *not* published in 1999, (*iv*) the `Any` type, which is the type of all values and which is often denoted as "`_`", especially in patterns, and its complement the `Empty` type.

Patterns are just types enriched with capture variables. For instance the pattern `<bib>[(x::Book)*]` captures in `x` the sequence of all books of a bibliography. Indeed, the `*` indicates that the pattern `x::Book` must be applied to every element of the sequence delimited by `<bib>`. When matched against an element, the pattern `x::Book` captures this element in the sequence `x`, provided that the element is of type `Book`. Patterns can then be used in match expressions:

```
match biblio with   <bib>[ (x::Book)* ] -> x
```

This expression matches `biblio` against our pattern and returns `x` as result, thus it makes nothing but stripping the `<bib>` tag from `biblio`. Note that if we knew that `biblio` has type `Bib`, then we could have used the pattern `<bib>[(x::Any)*]` (or its syntactic sugar `<bib>[(x::_)*]` since we statically know that all elements have type `Book`.

Besides capture variables there is just one further difference between patterns and types, namely the union operator `|` is commutative for types while it obeys a first match policy in patterns. So for instance the following expression returns the sequence of all the books published in 1999:

```
match biblio with  <bib>[ ( (x::<book year="1999">_) | _ )* ] -> x
```

Again, the pattern `((x::<book year="1999">_) | _ )` is applied to each element of the sequence. This pattern first checks whether the element has the tag `<book year="1999">` whatever its sequence of elements is, and if it is so it captures it in `x`; otherwise it matches the element against the pattern "`_`", which always succeeds without capturing anything (in this way it discards the element). Note that, if we had instead used `<bib>[ (x::<book year="1999">_)* ]` this pattern would have succeeded only for bibliographies composed only by books published in 1999.

After this brief introduction to regular expression patterns, let us show the pattern errors we target in this work. Suppose that, for each book, we need to extract all titles, together with relative authors or editors. In ℂDuce we can write the following function:[4]

```
let extract(x : [Book*]) : [(Title (Author+|Editor+))*] =
  transform x with
    <book>[z::<title>_ y::(<author>_ |<editor>_ )+ _*] -> z @ y
```

The function `extract` takes a possibly empty sequence of books and returns a possibly empty sequence where a title alternates with a non-empty uniform sequence of authors or editors. The expression `transform` applies the pattern to each element of the sequence `x` and returns the concatenation of all the results of the patterns that have succeeded. The pattern captures the title in the sequence variable `z`, the sequence of authors or editors in the sequence variable `y`, and returns the concatenation of `z` and `y`.

Imagine now that the programmer had put a typo in the pattern, writing instead:

```
    <book>[z::<tite>_ y::(<author>_ |<editor>_ )+ _*] -> z @ y
```

[4] This is not the best way to write this function in ℂDuce but it serves to outline the problem.

then the ℂDuce compiler would signal an error (actually, a warning), since no book starts with a **<tite>** element, so this pattern cannot ever match. But if the typo had been in the author (or in the editor) pattern:

```
<book>[z::<title>_ y::(<autor>_ |<editor>_ )+ _*] -> z @ y
```

then no error would be signalled since the pattern can still match editors. However, all the books with authors would be filtered out from the result, which would then be of type `[(Title Editor+)*]`. If we had used a weaker pattern

```
<book>[z::<title>_ y::(<autor>_ |<editor>_ )* _*] -> z @ y
```

in which we traded a `+` for a `*`, then the transform would return all the titles but only the editor lists (the author lists being matched by the final `_*` pattern), yielding a result of type `[(Title Editor*)*]`. In this case an error would be signalled but just because we used a very precise type for the function: had we specified a less precise type such as `[(Title (Author|Editor)*)*]`, then the error would have passed unnoticed again.

This kind of errors is very frequent when using patterns to code XPath-like expressions. For instance in ℂDuce it is possible to write a XPath-like expression of the form e/t, which is syntactic sugar for

```
transform e with <_>[ (x::t|_)*] -> x
```

Thus for instance the following query extracts all titles from the database `biblio` of type `Bib`:

```
[biblio]/<book>_/<title>_
```

If we replace `title` with `tite`, thus introducing an incorrect pattern, ℂDuce type system correctly rises a warning stating that the pattern never matches, as emptiness of a whole expression can be directly checked by result analysis. However, if we want to extract each title together with the relative price, we can write

```
[bib]/<book>_/(<title>_ | <prize>_ )
```

which contains an error, as `prize` occurs instead of `price`. But since the result is not empty no warning is raised. Here, the error is hidden by the fact that the pattern is partially correct : it does find some match, even if, locally, `<prize>_` never matches, hence is incorrect. Note that, as price is optional, by looking at the query output, when seeing only titles, we do not know whether prices are not present in that database or something else went wrong . . . . This further motivates improvements of the type system in order to check at static time that each sub pattern will match in at least one evaluation. The subpattern `<prize>_` does not meet this property.

As previous examples showed, undetected wrong sub-patterns are mainly introduced by the pattern _ (i.e. Any), which always matches and, thus, covers surrounding failures. However, this is far from being the only case. The error in the last version of extract function was covered by the final _* which captured all the authors, and possibly the price, of a book. However, the error would have been hidden even in the absence of _*. Indeed, if we had written

```
<book>[z::<title>_ y::(<autor>_ |<editor>_ )* Price?] -> z @ y
```

then all the books with authors would have been filtered out, yielding a result of type [(Title Editor+)*]. But again no type warning would be issued.

Finally, note that even if we used typos to introduce errors, other errors are possible, as well, of more conceptual nature. Imagine we want to select all books in which one author is either "Frisch" or "Colazzo", here is an example of hidden errors without any typo

```
let extract(x : [Book*]) : [Title*] =
  transform x with
    <book>[z::Title _*
           ( <author>[<last>"Colazzo"]
           | <author>[<last>"Frisch" _]) _*]  -> z
```

in this case no book with Colazzo as author will be selected since, contrary to the pattern for "Frisch", there is no pattern to match the <first>_ element. But again no error is signalled.

The technique to detect these errors will be presented in next section. We will work on binary trees to stay as close as possible to the implementation level (as these are the structures actually used in XDuce and ℂDuce to encode regular expressions) but also because the presentation will result far simpler. The whole theory can be then easily extended to general cases. As we will see, thanks to powerful type combinators of ℂDuce (union, negation, and intersection) the type rules that we provide are quite intuitive and simple. Also, the efficient implementation of ℂDuce type system ensures good performance of the newly introduced analysis, which relies on the same basic operators.

3 Error mining

Let us start by defining a simplified data model and type/pattern algebra. We are going to work with binary trees whose leaves are taken from a set of constants $\mathbb{C}$. We use the meta-variable c to range over constants. In ℂDuce, leaves can also be functions, and the trees have other kind of nodes (to deal with XML attributes and records).

Definition 1. *A* value *is a finite term produced by the following grammar:*

$$v \ ::= \ c \mid (v_1, v_2)$$

□

Now let us define the type and pattern algebra. For what concerns the contribution of this paper, namely the detection of useless sub-patterns, we do not need capture variables. This simplification allows us to give a common definition for types and patterns. However, we need an explicit way to localize sub-patterns. To do this, we annotate relevant sub-patterns with *marks* ranged over by the meta-variable ι. These marks can be thought as locations in the source code kept during the parsing phase and used to display error messages and warnings. Basic types are ranged over by the meta-variable b. A basic type denotes a set of constants. We write $(c : b)$ if the constant c belongs to the basic type b (the same constant can belong to many basic types).

Definition 2. *A* pattern *is a possibly infinite term produced by the following grammar:*

$$p \ ::= \ b \ \mid \ (p_1, p_2) \ \mid \ p_1|p_2 \ \mid \ p_1\boldsymbol{\&}p_2 \ \mid \ \neg p \ \mid \ \mathbf{0} \ \mid \ \mathbf{1} \ \mid \ p^{\iota}$$

with two additional requirements:

1. *(regularity) the term must be a regular tree (only but a finite number of different sub-terms);*
2. *(contractivity) any infinite branch must contain an infinite number of pair nodes* (p_1, p_2). □

Where b ranges overs basic types and **0** and **1** respectively represent the `Empty` and `Any` types. The infiniteness of patterns accounts for recursive types. Of course these types must be machine representable, therefore we impose a condition of regularity. The contractivity instead rules out meaningless terms such as $p = \neg p$ (that is, an infinite unary tree where all nodes are labeled by $\neg$). Both conditions are standard when dealing with recursive types (e.g. see [AC93]).

Note that these patterns are more than enough to encode all the types we used in Section 2: sequences can be encoded à la Lisp by pairs, pairs can also be used to encode XML types, while regular expression types are encoded by recursive patterns . So for instance if we do not consider attributes, the type

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

can be encoded as $Book = (\text{‘}book, (Title, X|Y)), X = (Author, X|(Price, \text{‘}nil)|\text{‘}nil)$ and $Y = (Editor, Y|(Price, \text{‘}nil)|\text{‘}nil)$, where *‘book* and *‘nil* are singleton (basic) types. For more details about the encoding, also in the presence of attributes and capture variables, see [BCF03].

We can now give the semantics for patterns. Intuitively, a pattern (without capture variable) applied to a value can succeed or fail. Since we want to identify useless sub-patterns, we will directly introduce an instrumented semantics which keeps track of sub-patterns that have indeed be used. Given a value v and a pattern p, the result of matching v against p is a pair $v/p = (\varepsilon, I)$ where $\varepsilon = 0$ denotes failure and $\varepsilon = 1$ denotes success, and I collects all the used ι marks. The definition is given by the following equations:

$$
\begin{array}{lll}
c/b & = (1,\varnothing) & \text{if } (c:b) \\
c/b & = (0,\varnothing) & \text{if } \neg(c:b) \\
(v_1,v_2)/b & = (0,\varnothing) & \\
c/(p_1,p_2) & = (0,\varnothing) & \\
(v_1,v_2)/(p_1,p_2) & = (\varepsilon, I_1 \cup I_2) & \text{if } v_1/p_1 = (1,I_1), v_2/p_2 = (\varepsilon, I_2) \\
(v_1,v_2)/(p_1,p_2) & = (0,I_1) & \text{if } v_1/p_1 = (0,I_1) \\
v/(p_1|p_2) & = (1,I_1) & \text{if } v/p_1 = (1,I_1) \\
v/(p_1|p_2) & = (\varepsilon, I_1 \cup I_2) & \text{if } v/p_1 = (0,I_1), v/p_2 = (\varepsilon, I_2) \\
v/(p_1 \mathbf{\&} p_2) & = (0,I_1) & \text{if } v/p_1 = (0,I_1) \\
v/(p_1 \mathbf{\&} p_2) & = (\varepsilon, I_1 \cup I_2) & \text{if } v/p_1 = (1,I_1), v/p_2 = (\varepsilon_2, I_2) \\
v/\neg p & = (1-\varepsilon, I) & \text{if } v/p = (\varepsilon, I) \\
v/\mathbf{0} & = (0,\varnothing) & \\
v/\mathbf{1} & = (1,\varnothing) & \\
v/p^{\iota} & = (0,I) & \text{if } v/p = (0,I) \\
v/p^{\iota} & = (1, I \cup \{\iota\}) & \text{if } v/p = (1,I)
\end{array}
$$

There are no overlapping cases in this definition, and it is well-founded. Indeed, the values in the right-hand side are smaller than or equal to the value in the left-hand side; when they are equal (which happens for the patterns $p_1|p_2$, $p_1\mathbf{\&}p_2$ and p^{ι}), the size of the patterns get strictly smaller, where the size of a pattern is defined by considering pair patterns as leaves (the size is finite because of the contractivity condition).

This instrumented semantics for pattern matching captures marks of sub-patterns which yield a successful match. A sequential traversal order has been chosen: the left sub-pattern in (p_1,p_2), $p_1\mathbf{\&}p_2$, $p_1|p_2$ is first considered, and the right sub-pattern is considered only when needed. For the alternation $p_1|p_2$, this corresponds to a natural naive implementation of a first-match policy; for (p_1,p_2) and $p_1\mathbf{\&}p_2$, this choice is arbitrary. In all cases, this sequential traversal order is just a way to formalize what are the used sub-patterns - and thus where to raise warnings for unused sub-patterns - and does not give any constraint on the actual run-time implementation of pattern matching.

Since patterns do not have capture variable in this presentation, they can be identified with types. We use the meta-variable t to range over types. The semantics of a type t is the set of values defined as:

$$[\![t]\!] = \{v \mid v/t = (1,I)\}$$

Note that the set of marks I is discarded in this definition. This semantics for types induces a natural equivalence relation: $t_1 \simeq t_2 \iff [\![t_1]\!] = [\![t_2]\!]$. From now on, we will identify types modulo this equivalence. Efficient algorithms have been developed to check inclusion between types; they obviously provide an effective and efficient way to check equivalence as well.

The pattern matching operation is intended to be used as a basic block in a programming language (such as ℂDuce). The type system for the language provides a static type for the argument of the pattern matching, which is an upper bound for the set of values that can actually flow to the pattern. The question we are interested in is to determine whether some part of the pattern is left unused for any value in this type.

Definition 3. *Let t be a type and p a pattern. The set of used marks when matching t against p is defined as:*

$$I(t,p) = \bigcup_{v \in [\![t]\!], (\varepsilon, I) = v/p} I$$

In words, a marked subpattern of p is used with respect to t if there exists a value v of t for which the marked subpattern is used when matching v against p.

We will now give an algorithm to compute this set $I(t,p)$. First, we define a rewriting relation $\leadsto$ over type/pattern pairs:

$$\begin{array}{l}
(t,(p_1,p_2)) \leadsto (\pi_1(t),p_1) \\
(t,(p_1,p_2)) \leadsto (\pi_2(t \wedge (p_1,\mathbf{1})),p_2) \\
(t,p_1|p_2) \leadsto (t,p_1) \\
(t,p_1|p_2) \leadsto (t \wedge \neg p_1,p_2) \\
(t,p_1\mathbf{\&}p_2) \leadsto (t,p_1) \\
(t,p_1\mathbf{\&}p_2) \leadsto (t \wedge p_1,p_2) \\
(t,p^{\iota}) \leadsto (t,p) \\
(t,\neg p) \leadsto (t,p)
\end{array}$$

The type operators $\pi_i()$ are defined by the equation: $[\![\pi_i(t)]\!] = \{v_i \mid (v_1,v_2) \in [\![t]\!]\}$. It has been shown in previous work [Fri04] how to compute these operators effectively. The theory developed in this work also shows that, starting from a pair (t,p), the set of pairs (t',p') which are reachable under the reflexive and transitive closure of $\leadsto$ is finite. This comes from the regularity of types and patterns. This set is thus effectively computable. If we collect all the marks ι such that (t',p'^{ι}) is in this set, and such that some value in t' makes the pattern p' succeed, we obtain exactly the set $I(t,p)$.

Theorem 1. *Let t be a type and p a pattern. Then:*

$$I(t,p) = \{\iota \mid (t,p) \overset{\star}{\rightsquigarrow} (t',p'^{\iota}), t' \wedge p' \not\simeq \mathbf{0}\}$$

4 Discussion

4.1 Characteristics of the analysis

In the previous section we defined the set $I(t,p)$ of all the pattern marks that are used when matching the pattern p against values in t. We also showed that it is possible to compute this set by saturating the pair (t,p) with a rewriting that is assured to terminate by the regularity of patterns. Actually, the saturated set can be computed quite efficiently, by using the very same algorithms implemented in the ℂDuce type checker.

The computation of this set allows us to detect *all* the unused subparts of a pattern. Indeed if we mark all the occurrences of p, then a mark ι of p is not in $I(t,p)$ if and only if for all values v of type t the sub-pattern marked by ι is not used when matching v against p. In other words, there is no value in t for which this sub-pattern is useful.

The "if and only if" states that our analysis is exact: we cannot refine it further. Of course, as usual, it is just "locally" exact since its global precision depends on the precision of the host type system in inferring the t at issue. For instance, consider the expression:

```
match e with p -> e′
```

to check whether p is correct the type system will mark all the occurrences of p, deduce the type t of e, and check whether all the marks of p are in $I(t,p)$. Thus the precision of the deduction of the correctness of p depends on the precision of the type system in inferring t: a more precise inference for the type of e might detect more errors in p, so the analysis is not globally complete, although globally sound (a pattern detected as wrong is indeed wrong).

Local soundness and completeness were not easy to obtain. Our first attempt to define correctness of sub-pattern was based on `Empty` substitutions. According to that attempt a sub-pattern of a pattern p was considered wrong with respect to an input type t if for every value v of t there was no difference between matching v against p, or matching v against the same p in which the sub-pattern is replaced by $\mathbf{0}$ (i.e. the `Empty` type). Now, such a characterization captures all the examples we gave in Section 2 but it is not sound with respect to all possible wrong patterns since it signals as wrong some sub-patterns that should not be considered as such. The most trivial example is the pattern `Int|3`: for an input type 3 it signals the pattern `Int` as wrong. But that is a trivial case in which the right hand side of `|` is contained in the left hand-side. A subtler example where the two branches of the "`|`" pattern are independent (no inclusion) is `(Even,_)|(Int,Bool)`. With input type `(Even,Bool)|(Odd,Int)` the subpattern `(Even,_)` is considered wrong according to `Empty` substitution characterization, while the analysis of Section 3 correctly fingers as wrong the sub-pattern `Bool`.

It is important to stress that our definition of "used mark" hardcodes the intuition we have about errors. We already stressed in the previous section that our definitions reflect a sequential transversal order for the tree. So for instance if the pattern `Int|Int` is used, our analysis fingers as wrong the rightmost occurrence of `Int`; an analysis signalling the leftmost occurrence as wrong would be equally correct but, in our opinion, less intuitive. The same left to right analysis is applied to intersections and pairs, as well.

Also we wondered whether to consider as wrong a pattern such as `Int & Int`. Indeed, in our left to right perspective the rightmost `Int` is useless. Note however that here it is not the matter of being not used, but of being redundant (for instance, the `Empty` substitution argument does not apply). Thus it was clear to us that redundancy *must not* considered as an error since it would go against a common programming practise: programmers prefer to use redundant patterns so as to reuse previous type definitions and make the code simpler and more readable rather than to write the exact pattern that ensures the absence of any redundancy: we must not force her/him to use this second option.

4.2 Extension to ℂDuce and other languages

The analysis developed in Section 3 applies directly to the cited languages based on regular expression patterns. Xtatic and recent versions of XDuce, however, require a slight modification to the definition of used sub-patterns since they use a non-deterministic semantics for the `|` pattern. This is very simple as it suffices to replace the two cases for $v/(p_1|p_2)$ by

$$v/(p_1|p_2) = (\varepsilon_1 \,||\, \varepsilon_2, I_1 \cup I_2) \qquad\qquad \text{if } v/p_i = (\varepsilon_i, I_i)$$

where $||$ denotes the logical or. The algorithm to compute used sub-patterns is simple to adapt as well. The rewriting rules for the `|` pattern are changed to:

$$\begin{array}{c}(t, p_1|p_2) \rightsquigarrow (t, p_1)\\(t, p_1|p_2) \rightsquigarrow (t, p_2)\end{array}$$

For what concerns the capture variables, we have to modify the definition of matching, since v/p must not only return a zero/one result but, in case of success, it also must return a substitution for the variables of the pattern. However, the analysis of Section 3 needs no change, since capture variables technically behaves the same as intersections with the type `Any`, as such they do not affect the analysis.

What it really remains to do in order to embed our analysis in the various languages at issue is to extend its definition to the other patterns present in these languages (for instance, in ℂDuce there also is a pattern for records), and to customize the typing rules of the languages so that they use the analysis. Let us discuss this last point for ℂDuce. We already hinted at how the typing rule for `match` expressions must be modified for taking into account the analysis. Formally this corresponds to having the following typing rule:

$$\begin{array}{c}\text{(for } t_i \equiv t \setminus \wr p_1 \int \setminus \ldots \setminus \wr p_{i-1} \int) \\ \dfrac{t \leq \wr p_1 \int \;|\; \ldots \;|\; \wr p_n \int \qquad I(t_1, p_i) = (\varepsilon, I_i) \qquad \Delta'_i = \mathit{marks}(p_i) \setminus I_i \\ \Gamma \vdash e : t \rightsquigarrow \Delta \qquad\qquad \Gamma, (t_i/p_i) \vdash e_i : s_i \rightsquigarrow \Delta_i}{\Gamma \vdash \textsf{match } e \textsf{ with } p_1 \rightarrow e_1 \;|\; \ldots \;|\; p_n \rightarrow e_n : \bigcup_{\{i | t_i \not\simeq \textsf{Empty}\}} s_i \rightsquigarrow \bigcup_{i=1 \ldots n} \Delta_i \cup \Delta'_i \cup \Delta}\end{array}$$

$\wr p_i \int$ denotes the exact type of all values that successfully match p_i (namely $\wr p \int = \{v \mid v/p \text{ succeeds}\}$), while $\Gamma \vdash e : t \rightsquigarrow \Delta$ means that, in the type environment Γ, e has type t and the labels in Δ denote unused sub-patterns in e; hence Δ is the error set computed by the type analysis (an expression is correct if the inferred error set is empty). The condition $t \leq \wr p_1 \int \;|\; \ldots \;|\; \wr p_n \int$ ensures that patterns are exhaustive with respect to all possible values e may produce. This ensures that well-typed terms never get stuck at run-time (at least one branch matches). (t_i/p_i) denotes the set of type assignments of p_i variables, computed by matching the pattern against the type t_i (see [FCB02]). Each t_i is computed by taking into account the first-match policy, so the e_i is typed in an environment in which each p_i is matched over values that cannot be matched by previous branches. Incorrect sub-patterns in p_i's are computed by subtracting from all marks of each p_i, denoted by $\mathit{marks}(p_i)$, the set of used patterns $I(t_i, p_i)$.

Error mining for iterators is not so straightforward, due to typing based on case analysis over the argument type. For example, if e is proved to have type `[`$S \,|\, U$`]`, then the `with` part of

```
transform e with p -> e'
```

is typed twice, once under the assumption that the argument has type S and once under the assumption that the type for the argument is U, thus inferring two types T_S and T_U, together with two errors sets Δ_S and Δ_U. The final inferred type is `[`$T_S \,|\, T_U$`]`, while the final errors set is $\Delta_S \cap \Delta_U$. This is because a sub-pattern in p is incorrect (unused) if it is so for both alternatives S and U (or, equivalently, it is correct if it is correct (used) with respect to at least one alternative among S and U). This technique was introduced and proved to be correct in [Col04,CGMS04]. Thus the complete formalization of error mining rules for iterators such as `transform`, follows those established for the XQuery iterator `for` in the cited papers, relying on the technique of Section 3 to infer incorrect fragments of patterns.

A similar technique must be used for overloaded functions: in ℂDuce an overloaded function is a function whose type is an intersection of arrows and the body of the function is typed once for each type in the intersection; of course are wrong only those occurrences of patterns that result unused in all these type deductions; once more an intersection applies. To implement this just a slight modification of the original typing rule is required, since we only need to add error sets to judgements and opportunely combine them in a way that strictly resembles error mining for iterators:

$$\frac{\Gamma, (x : t_i), (f : \bigwedge_{i=1..n} t_i \to s_i) \vdash e : s_i \rightsquigarrow \Delta_i \qquad i = 1..n}{\Gamma \vdash \mathsf{fun} f(t_1 \to s_1; \ldots; t_n \to s_n)(x) = e \,:\, (\bigwedge_{i=1..n} t_i \to s_i) \rightsquigarrow \bigcap_{i=1..n} \Delta_i}$$

The extension of other rules is even simpler, and omitted for space reasons.

It is worth observing that the presented error mining technique preserves the typing discipline in the hosting language, since error-mining depends on type-inference but not viceversa. In other words, the technique we have described is not intrusive and can be seen as an add-on of the hosting type system. However, in order to make error-mining more precise, that is to increase the number of errors detected at static time, one may consider to change the type discipline of the language. This may be needed when the type system infers a not-empty sequence type for expressions that instead always evaluate to the empty sequence. This can raise from an interaction between `/` and `transform`: for an example see [Col04] where the problem has been solved for XQuery. At this stage, we did not investigate this problem in the context of languages based on regular expression patterns, and we postpone this to future work.

Acknowledgments. This work was partially supported by the RNTL project "GraphDuce" and by the ACI project "Transformation Languages for XML: Logics and Applications".

References

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4), September 1993.

[BCF03] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.

[BCM05] V. Benzaken, G. Castagna, and C. Miachon. A full pattern-based paradigm for XML query processing. In *PADL 05, 7th International Symposium on Practical Aspects of Declarative Languages*, number 3350 in LNCS, pages 235–252. Springer, January 2005.

[BFS04] Niklas Broberg, Andreas Farre, and Josef Svenningsson. Regular expression patterns. In *ICFP '04: Proceedings of the ninth ACM SIGPLAN international conference on Functional programming*, pages 67–78, New York, NY, USA, 2004. ACM Press.

[BMS05] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In *ECOOP 2005*, LNCS, 2005. To appear.

[CGMS04] Dario Colazzo, Giorgio Ghelli, Paolo Manghi, and Carlo Sartiani. Types for Path Correctness for XML Queries. In *Proceedings of the ACM International Conference on Functional Programming (ICFP), Snowbird, Utah, USA*, 2004.

[Col04] Dario Colazzo. *Path Correctness for XML Queries: Characterization and Static Type Checking*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 2004.

[DFF$^+$05] Denise Draper, Peter Fankhauser, Mary Fernandez, Ashok Malhotra, Kristoffer Rose, Michael Rys, Jérôme Siméon, and Philip Wadler. XQuery 1.0 and XPath 2.0 Formal Semantics. Technical report, World Wide Web Consortium, February 2005. W3C Working Draft.

[FCB02] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic Subtyping. In *Proceedings, Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[Fri04] Alain Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML*. PhD thesis, Université Paris 7, December 2004.

[GP03] Vladimir Gapeyev and Benjamin C. Pierce. Regular object types. In *European Conference on Object-Oriented Programming (ECOOP), Darmstadt, Germany*, 2003. A preliminary version was presented at FOOL '03.

[Hos00] Haruo Hosoya. *Regular Expression Types for XML*. PhD thesis, The University of Tokyo, Japan, December 2000.

[HP00] Haruo Hosoya and Benjamin C. Pierce. XDuce: A typed XML processing language. In *Proceedings of Third International Workshop on the Web and Databases (WebDB2000)*, 2000.

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *The 25th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2001.

[LS04] K. Zhuo Ming Lu and M. Sulzmann. An implementation of subtyping among regular expression types. In *Proc. of APLAS'04*, volume 3302 of *LNCS*, pages 57–73. Springer-Verlag, 2004.

[OAC$^+$04] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, École Polytechnique Fédérale de Lausanne, 2004. Latest version at http://scala.epfl.ch.