Type Inference for Unique Pattern Matching

STIJN VANSUMMEREN
Limburgs Universitair Centrum

Regular expression patterns provide a natural, declarative way to express constraints on semi-structured data and to extract relevant information from it. Indeed, it is a core feature of the programming language Perl, surfaces in various `UNIX` tools such as `sed` and `awk`, and has recently been proposed in the context of the XML programming language XDuce. Since regular expressions can be ambiguous in general, different disambiguation policies have been proposed to get a unique matching strategy. We formally define the matching semantics under both (1) the POSIX, and (2) the first and longest match disambiguation strategies. We show that the generally accepted method of defining the longest match in terms of the first match and recursion does not conform to the natural notion of longest match. We continue by solving the type inference problem for both disambiguation strategies, which consists of calculating the set of all subparts of input values a subexpression can match under the given policy.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*patterns*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*classes defined by grammars or automata (e.g., context-free languages, regular sets, recursive sets); operations on languages*; H.2.3 [**Database Management**]: Languages—*query languages; XML*

General Terms: Design, languages, theory, verification

Additional Key Words and Phrases: pattern matching, disambiguation policies, programming languages, XML

1. INTRODUCTION

The *Extensible Markup Language* (XML) [Yergeau et al. 2004] provides a standard syntax for describing tree-structured and semi-structured data. In the past few years it has become the standard format for the representation and exchange of data on the web. Although XML can describe arbitrary trees, most applications restrict themselves to a set of valid trees, described by a *schema*. The standard schema language promoted by the World Wide Web Consortium (W3C) is XML Schema [Thompson et al. 2001], although various other schema languages exist [Davidson et al. 1999; Clark and Makoto 2001; Møller 2003].

Recently, there has been growing interest to make XML transformations *type safe*: given a schema for the input trees, does the transformed output tree always adhere to some output schema [Suciu 2002]? One of the most influential treatments

The author is a Research Assistant of the Fund for Scientific Research - Flanders.
Author's address: Stijn Vansummeren, Limburgs Universitair Centrum, Universitaire Campus, Gebouw D, B-3590 Diepenbeek, Belgium.

of this typechecking problem was done by Hosoya et al., in the context of the XML programming language XDuce [Hosoya 2000; Hosoya and Pierce 2003; Hosoya et al. 2005]. They introduced a type system of *regular expression types* based on regular tree languages capable of expressing XML Schema, and gave an efficient subtyping algorithm. XDuce's type system has strongly influenced that of XQuery [Boag et al. 2005], the standard XML query language of the W3C.

In addition, XDuce proposed an extension of ML-style patterns, called *regular (hedge) expression patterns* to support data extraction on hedges (sequences of trees) [Hosoya 2000; Hosoya and Pierce 2002]. In order to support such patterns in a statically typed programming language, Hosoya and Pierce argued that the compiler has to infer the types of the variable bindings occurring in a pattern, otherwise the type annotations become too heavy.

The idea of regular expression pattern matching stems from traditional string manipulation languages such as Perl, and `UNIX` tools such as `sed` and `awk` [Dougherty and Robbins 1996]. These languages remain in frequent use today, as a lot of legacy semi-structured data is not tree-structured, but consists of ordinary string content. None of the above languages can guarantee the type safety of a transformation however. A study of regular expression pattern matching for strings and its associated type inference problem is hence an important first step towards type safe string transformations in those languages.

Syntactically, regular (hedge) expression patterns are regular (hedge) expressions annotated with variable binders. In general, regular expressions can be *ambiguous*, meaning that there are various ways of matching the input, resulting in multiple possible bindings of the variables. In order to obtain a unique matching semantics, one therefore needs to disallow ambiguous patterns [Book et al. 1971; Hosoya 2003], or define a *disambiguation policy*. Various disambiguation policies exist, and it is currently unclear which one is to be preferred:

—the XDuce policy, also employed by Perl;

—the *first and longest match*; and

—the POSIX policy, employed by all IEEE POSIX compliant tools, including `sed` and `awk`.

Especially the XDuce policy and its related type inference problem has been extensively studied. It was introduced by Hosoya and Pierce [Hosoya 2000; Hosoya and Pierce 2002], who also developed its first type inference algorithm. This algorithm is imprecise however, since it only computes precise types for tail variables. A precise algorithm was later developed in the context of the XML-centric general-purpose programming language ℂDuce [Frisch et al. 2002; 2003]. Both approaches consider the policy in a hedge-based setting. A type inference algorithm for the string-based setting was developed at the same time by Tabuchi et al. [2002].

The first and longest match policy was also (indirectly) introduced by Hosoya and Pierce [2000; 2002], as a means to intuitively explain the XDuce policy. We will show, however, that this generally accepted intuition is false. As a consequence, the first and longest match policy has not been studied before. To our knowledge, the POSIX policy [Institute of Electrical and Electronic Engineers 1992] has not been studied before either.

In this paper we will formalize the POSIX and first and longest match disambiguation policies for both strings and hedges, and develop precise type inference algorithms for them. Our aim here is to treat strings and hedges in a uniform manner, and to develop *declarative* type inference algorithms which specify the formal languages that need to be calculated, but do not rely on a concrete implementation strategy. This approach has two benefits: we get a better understanding of the the fundamental difficulties involved, and our solutions can be integrated in existing regular language frameworks (such as that of MONA [Elgaard et al. 1998; Klarlund and Møller 2001], XDuce, or ℂDuce).

We note that we are the first to demonstrate soundness and completeness of a type inference algorithm for regular expression pattern matching. Indeed, the XDuce and λ^{re} algorithms are imprecise [Hosoya and Pierce 2002; Hosoya 2000; Sumii 2003], while the correctness proof of ℂDuce is unpublished.

The rest of this paper is organized as follows. Section 2 introduces regular expression pattern matching, the importance of a disambiguation strategy to get a unique match, and the type inference problem. We formally define regular expression string patterns in Section 3. We then define the matching relation on strings according to the POSIX policy in Section 4, and solve its associated type inference problem in Section 5. The insights gained will help us formalize the matching relation on strings according to the first and longest match policy in Section 6 where we also discuss its difference with the XDuce policy. We then develop a precise type inference algorithm in Section 7. Section 8 introduces regular hedge expression patterns. Finally, we show how the matching process under the first and longest match policy and its associated type inference problem can be lifted to the hedge-based setting in Sections 9 and 10. The last section provides discussion and some pointers to future work.

2. BASIC CONCEPTS

2.1 Pattern Matching

Pattern matching in declarative programming languages such as Prolog [Sterling and Shapiro 1994] or ML [Ullman 1998] provides a means to describe constraints on values, at the same time allowing useful information to be extracted. Regular (hedge) expression patterns provide a similar feature if the values to be operated upon are strings or hedges (sequence of trees).

As an example of regular hedge expression patterns, consider the following ML-like match construct:

```
match $v with
    book[ title[$t], $a as (author[_])+, _* ] => result[$t, $a]
    book[ title[$t], $e as (ε|editor[_]), _* ] => result[$t, $e]
```

Here we have two rules. Each rule consists of a regular hedge expression pattern and an action to undertake when the pattern matches the value. Each rule is tried in turn, starting from the top, until a pattern is found for which the input hedge (in variable `$v`) matches. Matching a value against a pattern consists of two parts: (1) ensuring that the input belongs to the formal language defined by the pattern; and (2) associating with every subpattern the matching part of the input. The

obtained associations can then be used to undertake the associated action, which constructs the output.

In our example, the formal language of the first pattern consists of all (ordered) trees for which:

—the root node is labeled by `book`;

—the first child is labeled by `title`;

—this first child has one or more `author` nodes as right siblings; and

—those sibling nodes are followed by zero or more other nodes (the underscore _ denotes any tree).

If an input hedge belongs to this formal language, then variable `$t` should be bound to the children of the `title` node and variable `$a` should be bound to the `author` nodes matched by the `(author[_])`$^+$ subpattern. The result of this rule is constructed by creating a new node labeled `result`, with children `$t` and `$a`.

Likewise, the formal language of the second pattern consists of all trees for which:

—the root node is labeled by `book`;

—the first child is labeled by `title`;

—this first child is followed by an optional `editor` node (ε stands for the empty hedge pattern);

—which is followed by zero or more other nodes.

If an input hedge belongs to this formal language, then variable `$t` should be bound to the children of the `title` node and variable `$e` should be bound to the hedge matched by the `(`ε`|editor[_])` subpattern. The result of the second rule is constructed by creating a new node labeled `result`, with children `$t` and `$e`.

In general, patterns can be *ambiguous*, meaning that there are various ways of matching the input, resulting in multiple possible associations, and hence in multiple possible outputs.

Example 2.1. Indeed, consider the following input tree, which is depicted in Figure 1(a):

```
book[
  title["Data On The Web"],
  author["Abiteboul"], author["Buneman"], author["Suciu"],
  price[50]
]
```

It is clear that this tree belongs to the formal language defined by the first pattern. Note, however, that there are multiple ways of "parsing" the value by the pattern. For instance, we could parse the first `author` node by the `(author[_])`$^+$ subpattern, and we could parse its right siblings by the _* subpattern. Alternatively, we could parse the first two `author` nodes by the `(author[_])`$^+$ pattern and their right siblings by the _* subpattern. Finally, we could parse all `author` nodes by the `(author[_])`$^+$ subpattern, and only the `price` node by the _* subpattern. The following table summarizes the various associations for `$t` and `$a` corresponding to these possibilities.

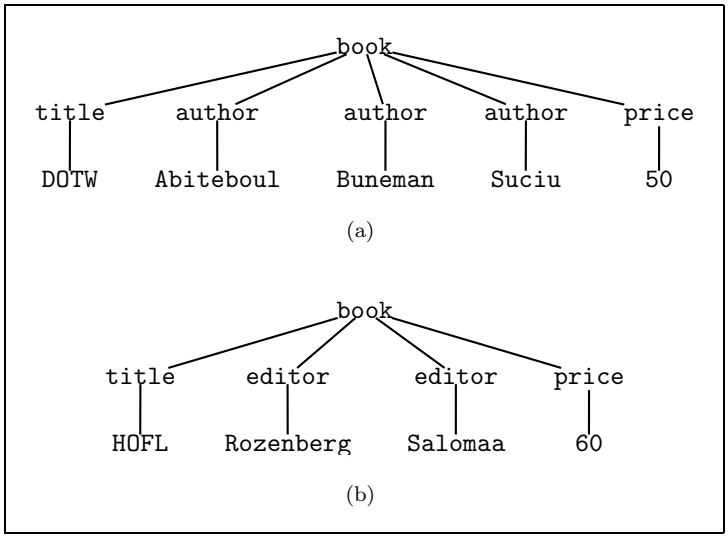

Fig. 1. (a) The input tree from Example 2.1. (b) The input tree from Example 2.2.

$t	$a
"DOTW"	author["Abiteboul"]
"DOTW"	author["Abiteboul"], author["Buneman"]
"DOTW"	author["Abiteboul"], author["Buneman"], author["Suciu"]

Note that we get a different output for each possible association. □

Example 2.2. Pattern two is also ambiguous. Consider the following input tree, which is depicted in Figure 1(b):

```
book[
  title["Handbook of Formal Languages"],
  editor["Rozenberg"], editor["Salomaa"],
  price[60]
]
```

It is clear that this tree belongs to the formal language defined by the second pattern. Here we could parse the empty hedge by the (ε|editor[_]) subpattern and the editor and price nodes by the $\_^*$ pattern; or we could parse the first editor node by the (ε|editor[_]) subpattern and its right siblings by $\_^*$. Note again that we get a different output for each possible association. □

When patterns are used in database query languages, it is common and desirable for a pattern to have many matches in the data, and to be able to retrieve all of them [Abiteboul et al. 1997; Neumann and Seidl 1998; Buneman et al. 2000; Neven and Schwentick 2001; Murata 2001; Boag et al. 2005]. However, in general-purpose programming using pattern matching as in ML [Ullman 1998] or Prolog [Sterling and Shapiro 1994] we normally want unique matching and a deterministic semantics.

One approach to the latter problem would be to simply disallow ambiguity by requiring the regular expressions to be *unambiguous* [Book et al. 1971; Hosoya 2003]. Another, more programmer-friendly approach is to allow arbitrary regular expression patterns, but to give a *disambiguation policy*, which ensures a unique matching semantics. It is this approach that is taken in Perl, `awk`, `sed`, and XDuce. Amongst these applications, various disambiguation policies exist:

— The first, followed by all IEEE POSIX compliant tools, including `awk` and `sed`, consists of a single rule which states that each subpattern should match as much of the input as possible while still allowing the rest of the pattern to match [Institute of Electrical and Electronic Engineers 1992]. Subpatterns starting earlier are given priority over those starting later. We will refer to this policy as the *POSIX policy*.

— The second, which was informally introduced in [Hosoya 2000; Hosoya and Pierce 2002], consists of two disambiguation rules: first match and longest match. The *first match rule* disambiguates a disjunction pattern $\mathtt{P}_1 + \mathtt{P}_2$ by giving higher priority to the first alternative $\mathtt{P}_1$. Moreover, disjunction distributes over concatenation. That is, when matching w against $(\mathtt{P}_1 + \mathtt{P}_2) \cdot \mathtt{P}_3$, w should be first matched against $\mathtt{P}_1 \cdot \mathtt{P}_3$ and it should only be matched against $\mathtt{P}_2 \cdot \mathtt{P}_3$ when this fails. The *longest match rule* disambiguates the Kleene closure in patterns of the form $\mathtt{P}_1^* \cdot \mathtt{P}_2$ by requiring that $\mathtt{P}_1^*$ matches as much of the input as possible, still allowing the rest of the pattern to match. We will refer to this policy as the *first and longest match policy*.

— The third, followed by Perl, XDuce, ℂDuce, and λ^{re} also consists of two rules: first match and greedy match [Hosoya 2000; Hosoya and Pierce 2002; Frisch et al. 2002]. The first match is the same as for the first and longest match policy. The *greedy match rule* disambiguates the Kleene closure in a pattern $\mathtt{P}_1^* \cdot \mathtt{P}_2$ by recursively rewriting it into $(\mathtt{P}_1 \cdot \mathtt{P}_1^* + \varepsilon) \cdot \mathtt{P}_2$. We will refer to this policy as the *XDuce policy*.

Example 2.3. Consider again the matching of the input tree in Figure 1(a) against the first pattern. Because the $(\mathtt{author[\_]})^+$ subpattern occurs before the $\mathtt{\_}^*$ subpattern, the POSIX policy requires us to match as many nodes by $(\mathtt{author[\_]})^+$ as possible. Hence, all `author` nodes are matched by this subpattern. As such, `$t` is associated with `"DOTW"` and `$a` is associated with

$$\mathtt{author["Abiteboul"]}, \mathtt{author["Buneman"]}, \mathtt{author["Suciu"]}.$$

Since the $(\mathtt{author[\_]})^+$ subpattern is a Kleene closure, the first and longest match policy and the XDuce policy also require to match as many nodes by $(\mathtt{author[\_]})^+$ as possible, resulting in the same associations. □

The associations obtained under the various polices differ when the input tree of Figure 1(b) is matched against the second pattern.

Example 2.4. Indeed, since the $(\varepsilon|\mathtt{editor[\_]})$ subpattern occurs before the $\mathtt{\_}^*$ subpattern, and since matching a single tree is considered longer than matching the empty hedge, the POSIX policy will require us to match the first `editor` node by $(\varepsilon|\mathtt{editor[\_]})$, and its right siblings by $\mathtt{\_}^*$. Hence, under the POSIX disambiguation policy, `$t` is associated with `"HOFL"` and `$a` is associated with

`editor["Salomaa"]`. The first and longest policy and the XDuce policy, however, will first try to match against the ε subpattern (which succeeds) before trying to match against the `editor[_]` subpattern. Hence, under these policies, `$t` is associated with `"HOFL"` and `$a` is associated with the empty hedge. We will show the difference between the first and longest match policy and the XDuce policy in Section 6. □

In the following sections we will formally describe the matching process under all three disambiguation policies by means of the matching relation $v \in \mathtt{P} \rightsquigarrow V$, signifying that (string or hedge) input value v is matched by (string or hedge) pattern $\mathtt{P}$ yielding associations V. We will view patterns as abstract syntax trees, and identify subpatterns by their corresponding nodes in the abstract syntax tree. This has the advantage that we do not have to mention variables explicitly in a pattern. We just have to reason about the node a variable is associated with. Hence, we can formally describe the associations V as a function from nodes n in $\mathtt{P}$ to subvalues of v or to the special symbol $\perp$. The matching relation will be defined such that $V(n) = v'$ if and only if the pattern rooted at node n is responsible for matching the subpart v' of v. It is $\perp$ if the subpattern is not responsible for recognizing any subpart of v.

We will not concern ourselves with the efficient implementation of the matching process under the various disambiguation policies, for which we refer to the literature [Laurikari 2000; 2001; Frisch and Cardelli 2004; Frisch 2004; Levin 2003].

2.2 Type Inference

XDuce [Hosoya 2000; Hosoya and Pierce 2003; 2002; Hosoya et al. 2005], ℂDuce [Frisch et al. 2003; 2002], and λ^{re} [Tabuchi et al. 2002] are programming languages that can statically verify whether a transformation is type-safe. They all use *regular expressions types* capable of representing *regular (hedge) languages* to achieve this goal. Regular (hedge) languages serve as a unifying model for many schema languages [Murata et al. 2001; Neven 2002]. In order to support regular expression pattern matching XDuce, ℂDuce, and λ^{re} employ a type inference algorithm that calculates, for each subpattern, the set of values it can be associated with given a type for the input. The idea is to use these sets to compute the type of all constructed output values, and to check that this type is a subtype of the given output type.

In the following sections we will introduce type inference algorithms for the $\mathbb{P}$ and first and longest match disambiguation policies on strings and on hedges. We abstract away from a particular syntax of regular expression types, and use regular word and hedge languages instead. We will use $\mathcal{T}^D(n, \mathtt{P}, C)$ to denote the set of all values the subpattern rooted at node n in $\mathtt{P}$ can be bound to under disambiguation policy D when the input values all belong to the set C:

$$\mathcal{T}^D(n, \mathtt{P}, C) := \{v' \mid \exists v \in C,\ v \in \mathtt{P} \rightsquigarrow V,\ V(n) = v'\}.$$

For the superscript D we will use $\mathbb{P}$ to denote the POSIX disambiguation policy, $\mathbb{FL}$ to denote the first and longest match disambiguation policy, and $\mathbb{XD}$ to denote the XDuce disambiguation policy.

3. REGULAR STRING EXPRESSION PATTERNS

In this section we define regular string expression patterns, and provide some general notation that will be used throughout the paper.

We assume given a fixed, finite alphabet Σ which does not contain the special symbols $\perp$ and $\Box$. Elements of Σ will be denoted by σ and words over Σ will be denoted by w throughout the rest of this paper. The empty word is denoted by λ. A *regular string expression pattern* $\mathtt{P}$ is a regular expression over Σ. That is, $\mathtt{P}$ is either of the form ε (with ε recognizing the empty word), σ (with $\sigma \in \Sigma$), $\mathtt{P}_1 + \mathtt{P}_2$, $\mathtt{P}_1 \cdot \mathtt{P}_2$, or $\mathtt{P}_1^*$, where $\mathtt{P}_1$ and $\mathtt{P}_2$ are already regular expression patterns. The *language* $L(\mathtt{P})$ of a pattern $\mathtt{P}$ is defined as usual. That is, $L(\varepsilon) = \{\lambda\}$, $L(\sigma) = \{\sigma\}$, $L(\mathtt{P}_1 + \mathtt{P}_2) = L(\mathtt{P}_1) \cup L(\mathtt{P}_2)$, $L(\mathtt{P}_1 \cdot \mathtt{P}_2)$ is the concatenation of $L(\mathtt{P}_1)$ and $L(\mathtt{P}_2)$, and $L(\mathtt{P}_1^*)$ is the Kleene closure of $L(\mathtt{P}_1)$. Because we want to identify the subexpressions of a pattern, we abuse notation slightly and identify $\mathtt{P}$ with the partial function $\mathtt{P} : \{1,2\}^* \to \{*, \cdot, +, \varepsilon\} \cup \Sigma$ such that

—if $\mathtt{P} = \varepsilon$ then $dom(\mathtt{P}) = \{\lambda\}$ and $\mathtt{P}(\lambda) = \varepsilon$;

—if $\mathtt{P} = \sigma$ with $\sigma \in \Sigma$ then $dom(\mathtt{P}) = \{\lambda\}$ and $\mathtt{P}(\lambda) = \sigma$;

—if $\mathtt{P} = \mathtt{P}_1 + \mathtt{P}_2$ then $dom(\mathtt{P}) = \{\lambda\} \cup \{1n \mid n \in dom(\mathtt{P}_1)\} \cup \{2n \mid n \in dom(\mathtt{P}_2)\}$ with $\mathtt{P}(\lambda) = +$, $\mathtt{P}(1n) = \mathtt{P}_1(n)$, and $\mathtt{P}(2n) = \mathtt{P}_2(n)$;

—if $\mathtt{P} = \mathtt{P}_1 \cdot \mathtt{P}_2$ then $dom(\mathtt{P}) = \{\lambda\} \cup \{1n \mid n \in dom(\mathtt{P}_1)\} \cup \{2n \mid n \in dom(\mathtt{P}_2)\}$ with $\mathtt{P}(\lambda) = \cdot$, $\mathtt{P}(1n) = \mathtt{P}_1(n)$, and $\mathtt{P}(2n) = \mathtt{P}_2(n)$; and

—if $\mathtt{P} = \mathtt{P}_1^*$ then $dom(\mathtt{P}) = \{\lambda\} \cup \{1n \mid n \in dom(\mathtt{P}_1)\}$, $\mathtt{P}(\lambda) = *$, and $\mathtt{P}(1n) = \mathtt{P}_1(n)$.

Intuitively, the function view of a pattern describes the abstract syntax tree of its regular expression, as shown in Figure 2. In general, an expression can have multiple parse trees. We therefore assume the usual precedence of operators in the previous definition: * binds tighter than $\cdot$, which has a higher precedence than $+$. Furthermore, $\cdot$ and $+$ are assumed to be right-associative. Elements of $\{1,2\}^*$ are called *nodes* and will be denoted by n, m, and their subscripted versions. We write $|\mathtt{P}|$ for the number of nodes of $\mathtt{P}$. Intuitively, nodes are used to identify subexpressions.

Since subpatterns inside a Kleene closure can match multiple subwords of an input word, we will not compute associations for such subpatterns. Therefore, a node $n \in dom(\mathtt{P})$ is a *bindable node* of $\mathtt{P}$ if it does not have an ancestor labeled with $*$. The set of bindable nodes of $\mathtt{P}$ is denoted by $bn(\mathtt{P})$.

As was already noted in Section 2, the matching process for a given disambiguation strategy is formally described by the matching relation $w \in \mathtt{P} \rightsquigarrow V$, signifying that w is matched by $\mathtt{P}$ yielding associations V. Here, V is a function from $bn(\mathtt{P})$ to subwords of w or to the special symbol $\perp$. The matching relation will be defined such that $V(n) = w'$ if and only if the pattern rooted at node n is responsible for matching the subword w' under the considered disambiguation policy. It is $\perp$ if the subpattern is not responsible for recognizing any subword of w.

Example 3.1. As we will further illustrate in Example 4.1, matching the word ab against the pattern $(a+a\cdot b)\cdot(b+\varepsilon)$ of Figure 2(a) under the POSIX disambiguation

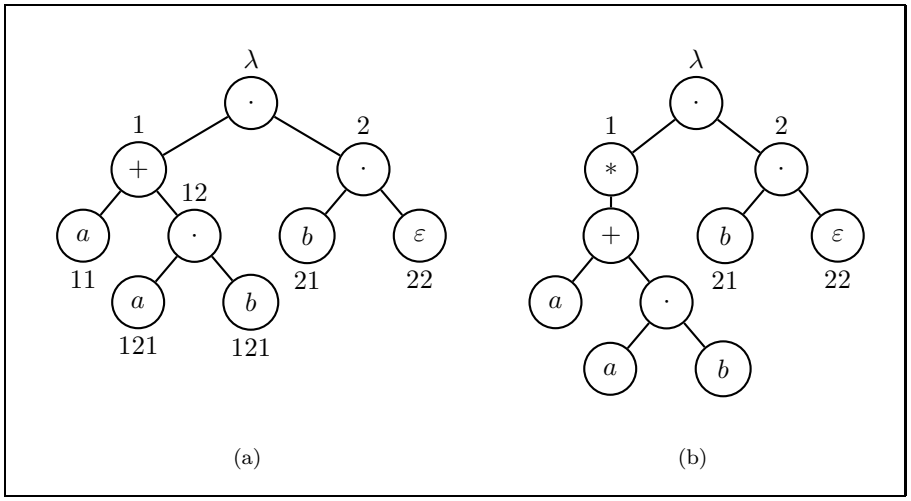

Fig. 2. The abstract syntax tree representation of $(a + a \cdot b) \cdot (b + \varepsilon)$ (left) and $(a + a \cdot b)^* \cdot (b + \varepsilon)$ (right). The bindable nodes have their addresses annotated.

strategy yields the associations V where

$$\begin{array}{lll} V(\lambda) = ab & V(1) = ab & V(11) = \bot \\ V(12) = ab & V(121) = a & V(122) = b \\ V(2) = \lambda & V(21) = \bot & V(22) = \lambda. \end{array}$$

On the other hand, matching ab against this pattern under the first and longest match disambiguation policy yields the associations V' where

$$\begin{array}{lll} V'(\lambda) = ab & V'(1) = a & V'(11) = a \\ V'(12) = \bot & V'(121) = \bot & V'(122) = \bot \\ V'(2) = b & V'(21) = b & V'(22) = \bot, \end{array}$$

as we will further illustrate in Example 6.1. □

To simplify the definition of matching relations we introduce the following notation. Let V_1 and V_2 be associations, and let $\mathtt{P}_1$ and $\mathtt{P}_2$ be patterns. We write $[\lambda \to w]$ to denote the function with domain $\{\lambda\}$ for which

$$[\lambda \to w](\lambda) = w.$$

We write $V_1 + \mathtt{P}_2$ to denote the function for which

$$(V_1 + \mathtt{P}_2)(n) = \begin{cases} V_1(1) & \text{if } n = \lambda; \\ V_1(m) & \text{if } n = 1m, m \in dom(V_1); \\ \bot & \text{if } n = 2m, m \in dom(\mathtt{P}_2). \end{cases}$$

We define $\mathtt{P}_1 + V_2$ similarly:

$$(\mathtt{P}_1 + V_2)(n) = \begin{cases} V_2(1) & \text{if } n = \lambda; \\ V_2(m) & \text{if } n = 2m,\ m \in dom(V_2); \\ \bot & \text{if } n = 1m, m \in dom(\mathtt{P}_1). \end{cases}$$

Finally, we denote by $V_1 \cdot V_2$ the function such that

$$(V_1 \cdot V_2)(n) = \begin{cases} V_1(\lambda) \cdot V_2(\lambda) & \text{if } n = \lambda, V_1(\lambda) \neq \bot, V_2(\lambda) \neq \bot; \\ \bot & \text{if } n = \lambda \text{ and } (V_1(\lambda) = \bot \text{ or } V_2(\lambda) = \bot); \\ V_1(m) & n = 1m, m \in dom(V_1) \\ V_2(m) & n = 2m, m \in dom(V_2). \end{cases}$$

Example 3.2. If V_1 is the association function with domain $\{\lambda, 1, 2\}$ such that

$$V_1(\lambda) = ab \qquad V_1(1) = a \qquad V_1(2) = b,$$

then $a + V_1$ is the association function W with domain $\{\lambda, 1, 2, 21, 22\}$ such that

$$W(\lambda) = ab \qquad W(1) = \bot \qquad W(2) = ab$$
$$W(21) = a \qquad W(22) = b.$$

Furthermore, if V_2 is the association function with domain $\{\lambda, 1, 2\}$ such that

$$V_2(\lambda) = \lambda \qquad V_2(1) = \bot \qquad V_2(2) = \lambda,$$

then $(a + V_1) \cdot V_2$ is the association function V from Example 3.1. I.e., it is the association obtained by matching the word ab against pattern `P` from Figure 2(a) under the POSIX disambiguation policy. □

4. MATCHING UNDER THE POSIX POLICY

As shown in Section 2, patterns can be *ambiguous*, meaning that there are various ways of matching an input word. In this section we formally introduce the POSIX disambiguation policy, employed by all IEEE POSIX standard compliant regular expression tools like `awk`, `sed`, ... It is easy to formalize and the techniques for its associated type inference algorithm, as developed in the next section, serve as a warmup for that of the first and longest match policy treated in the second half of this paper.

The POSIX disambiguation policy can be expressed as follows [Institute of Electrical and Electronic Engineers 1992; Laurikari 2001]:

> Subpatterns should match the longest possible substrings, where subpatterns that start earlier in the regular expression take priority over ones starting later. Hence, higher-level subpatterns take priority over their lower-level component subpatterns. Matching an empty string is considered longer than no match at all.

Let us clarify this rule with an example.

Example 4.1. Consider the matching of ab against the pattern $(a + a \cdot b) \cdot (b + \varepsilon)$ of Figure 2(a). Then the whole pattern matches ab. Because subpattern $(a + a \cdot b)$ starts earlier than $(b + \varepsilon)$, it should match as much of the input string as possible, still allowing the whole pattern to match. Hence, $(a + a \cdot b)$ matches ab and $(b + \varepsilon)$ matches λ. □

The matching relation $w \in \mathtt{P} \rightsquigarrow V$ under the POSIX policy is formally defined in Figure 3. Rules Empty and Lab are axioms allowing to match the empty sequence and a single symbol respectively. Rule Kleene allows matching a word against a

$$\textsc{Empty}\ \frac{}{\lambda \in \varepsilon \rightsquigarrow [\lambda \to \lambda]} \qquad \textsc{Lab}\ \frac{}{\sigma \in \sigma \rightsquigarrow [\lambda \to \sigma]} \qquad \textsc{Kleene}\ \frac{w \in L(\mathtt{P}^*)}{w \in \mathtt{P}^* \rightsquigarrow [\lambda \to w]}$$

$$\textsc{Or1}\ \frac{w \in \mathtt{P}_1 \rightsquigarrow V}{w \in \mathtt{P}_1 + \mathtt{P}_2 \rightsquigarrow V + \mathtt{P}_2} \qquad \textsc{Or2}\ \frac{w \in \mathtt{P}_2 \rightsquigarrow V \qquad w \notin L(\mathtt{P}_1)}{w \in \mathtt{P}_1 + \mathtt{P}_2 \rightsquigarrow \mathtt{P}_1 + V}$$

$$\textsc{Concat}\ \frac{w_1 \in \mathtt{P}_1 \rightsquigarrow V_1 \qquad w_2 \in \mathtt{P}_2 \rightsquigarrow V_2 \qquad \neg(\exists w_3 \neq \lambda, w_4 : w_3 w_4 = w_2 \wedge w_1 w_3 \in L(\mathtt{P}_1) \wedge w_4 \in L(\mathtt{P}_2))}{w_1 w_2 \in \mathtt{P}_1 \cdot \mathtt{P}_2 \rightsquigarrow V_1 \cdot V_2}$$

Fig. 3. The matching relation $w \in \mathtt{P} \rightsquigarrow V$ under the POSIX disambiguation policy.

Kleene closure pattern. Note that the resulting association function only provides an association for λ, which is the only bindable node of $\mathtt{P}^*$. For a disjunction $\mathtt{P}_1 + \mathtt{P}_2$, the POSIX disambiguation policy specifies that the whole pattern should match the longest possible substring. In order for the match to succeed, this would have to be the whole input word. Furthermore, since $\mathtt{P}_1$ starts earlier than $\mathtt{P}_2$, $\mathtt{P}_1$ is to be given precedence. Consequently, when matching w against $\mathtt{P}_1 + \mathtt{P}_2$, we should always try to match w to $\mathtt{P}_1$ first. This is expressed in rules Or1 and Or2, where Or2 can only be used if Or1 fails. Rule Concat specifies that in a concatenation $\mathtt{P}_1 \cdot \mathtt{P}_2$, pattern $\mathtt{P}_1$ should match as much as possible (since it occurs earlier), still allowing the entire pattern to match.

Theorem 4.2. *The matching relation of Figure 3 is well defined:*

(1) *The matching relation is semantically correct:* $w \in \mathtt{P} \rightsquigarrow V$ *iff* $w \in L(\mathtt{P})$*, and,*

(2) *The matching relation is unique: if* $w \in \mathtt{P} \rightsquigarrow V$ *and* $w \in \mathtt{P} \rightsquigarrow W$ *then* $V = W$*.*

Proof. (1). The "if" direction can be proved by a straightforward induction on $\mathtt{P}$. The "only if" direction can be proved by a straightforward induction on the matching derivation.

(2). By a straightforward induction on the matching derivation of $w \in \mathtt{P} \rightsquigarrow V$, with a case analysis on the last rule used. □

Example 4.3. The following is the matching derivation of ab against $(a + a \cdot b) \cdot (b + \varepsilon)$:

$$\dfrac{\dfrac{\dfrac{\textsc{Lab}\ \dfrac{}{a \in a \rightsquigarrow V_1 := [\lambda \to a]} \qquad \textsc{Lab}\ \dfrac{}{b \in b \rightsquigarrow V_2 := [\lambda \to b]}}{ab \in a \cdot b \rightsquigarrow V_1 \cdot V_2}\ \textsc{Concat}}{ab \in (a + a \cdot b) \rightsquigarrow a + (V_1 \cdot V_2)}\ \textsc{COr2} \qquad \dfrac{\textsc{Empty}\ \dfrac{}{\lambda \in \varepsilon \rightsquigarrow V_3 := [\lambda \to \lambda]}}{\lambda \in (b + \varepsilon) \rightsquigarrow b + V_3}\ \textsc{COr2}}{ab \in (a + a \cdot b) \cdot (b + \varepsilon) \rightsquigarrow (a + (V_1 \cdot V_2)) \cdot (b + V_3)}\ \textsc{Concat}$$

It is easily seen that the obtained association function $(a + (V_1 \cdot V_2)) \cdot (b + V_3)$ equals the association function V from Example 3.1. For example,

$$((a + (V_1 \cdot V_2)) \cdot (b + V_3))(1) = (a + (V_1 \cdot V_2))(\lambda) = V_1(\lambda) \cdot V_2(\lambda) = ab = V(1).$$

Likewise:

$$((a + (V_1 \cdot V_2)) \cdot (b + V_3))(21) = (b + V_3)(1) = \bot = V(21).$$

We note that we cannot match a by $(a + a \cdot b)$ and b by $(b + \varepsilon)$. Indeed, although it is possible to derive $a \in (a + a \cdot b) \rightsquigarrow W_1$ and $b \in b + \varepsilon \rightsquigarrow W_2$ for some associations W_1 and W_2, the third premise of rule Concat will disable us to conclude $ab \in (a + a \cdot b) \cdot (b + \varepsilon) \rightsquigarrow W_1 \cdot W_2$. Indeed, since $ab \in L(a + a \cdot b)$ and $\lambda \in L(b + \varepsilon)$ there exists a longer match. □

5. TYPE INFERENCE UNDER THE POSIX POLICY

The matching process described in the previous section is used in `UNIX` tools like `sed` and `awk` [Dougherty and Robbins 1996]. Solving its regular type inference problem can be seen as a first step towards making transformations in these languages type safe. The main result of this section can be stated as follows:

Theorem 5.1. *If C is a regular language then $\mathcal{T}^{\mathbb{P}}(m, \mathtt{P}, C)$ is also regular, and can be effectively computed.*

5.1 The Algorithm

Let us first introduce the algorithm by informal reasoning. We will formally prove its correctness later.

We observe that the type of the root node λ is exactly the set of words in C that can be matched by $\mathtt{P}$. Indeed, if w is successfully matched by $\mathtt{P}$ then λ is associated to w itself. If $m \neq \lambda$, then $\mathtt{P}$ is of the form $\mathtt{P}_1 + \mathtt{P}_2$ or $\mathtt{P}_1 \cdot \mathtt{P}_2$, since all other patterns contain only one bindable node: λ.

If $\mathtt{P} = \mathtt{P}_1 + \mathtt{P}_2$ then we observe that words can only be associated to subpatterns of $\mathtt{P}_1$ if they are subwords of some word in C matched by $\mathtt{P}_1$. Hence, if $m = 1n$ then we can calculate $\mathcal{T}^{\mathbb{P}}(1n, \mathtt{P}, C)$ simply by calculating $\mathcal{T}^{\mathbb{P}}(n, \mathtt{P}_1, C)$. Similarly, words can only be associated to subpatterns of $\mathtt{P}_2$ if they are subwords of some word in C matched by $\mathtt{P}_2$. We must take care however, since this word must not be matched against $\mathtt{P}_1$ because of the precedence of $\mathtt{P}_1$ over $\mathtt{P}_2$ in $\mathtt{P}$. Hence, we can calculate $\mathcal{T}^{\mathbb{P}}(2n, \mathtt{P}, C)$ by calculating $\mathcal{T}^{\mathbb{P}}(n, \mathtt{P}_2, C - L(\mathtt{P}_1))$.

If $\mathtt{P} = \mathtt{P}_1 \cdot \mathtt{P}_2$ then we observe that words can only be associated to subpatterns of $\mathtt{P}_1$ if they are subwords of a word w_1 matched by $\mathtt{P}_1$, for which there exists some w_2 matched by $\mathtt{P}_2$ such that $w_1 w_2 \in C$ and such that w_1 really is the longest possible prefix of $w_1 w_2$ that can be matched by $\mathtt{P}_1$, still allowing the corresponding suffix to be matched by $\mathtt{P}_2$. Formally this means that we cannot break w_2 in $w_3 \neq \lambda$ and w_4 with $w_1 w_3 \in L(\mathtt{P}_1)$ and $w_4 \in L(\mathtt{P}_2)$. Let us define the *left breaking* of C by languages L_1 and L_2, denoted by $\mathit{lbreak}(C, L_1, L_2)$, to be exactly the set of such words w_1:

$$\begin{aligned} \mathit{lbreak}(C, L_1, L_2) := \{ & w_1 \in L_1 \mid \exists w_2 \in L_2 : w_1 w_2 \in C \\ & \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3 w_4 = w_2 \wedge w_1 w_3 \in L_1 \wedge w_4 \in L_2)\}. \end{aligned}$$

Algorithm 1: Calculate $\mathcal{T}^{\mathbb{P}}(m, \mathtt{P}, C)$.

Input: A pattern $\mathtt{P}$; a node $m \in bn(\mathtt{P})$; and a regular context C.

Output: The type of m in $\mathtt{P}$ relative to C under the POSIX disambiguation policy.

```
if m = λ then
    return L(P) ∩ C
else
    switch P do
        case P1 + P2
            switch m do
                case 1n  return  T^P(n, P1, C)
                case 2n  return  T^P(n, P2, C − L(P1))
            end
        case P1 · P2
            switch m do
                case 1n  return  T^P(n, P1, lbreak(C, L(P1), L(P2)))
                case 2n  return  T^P(n, P2, rbreak(C, L(P1), L(P2)))
            end
    end
end
```

Then $\mathcal{T}^{\mathbb{P}}(1n, \mathtt{P}, C)$ equals $\mathcal{T}^{\mathbb{P}}(n, \mathtt{P}_1, \mathit{lbreak}(C, L(\mathtt{P}_1), L(\mathtt{P}_2)))$. Similarly, words can only be associated to subpatterns of $\mathtt{P}_2$ in $\mathtt{P}_1 \cdot \mathtt{P}_2$ if they are subwords of a word w_2 matched by $\mathtt{P}_2$ for which there exists some w_1 matched by $\mathtt{P}_1$ such that $w_1 w_2 \in C$ and such that w_1 really is the longest possible prefix of $w_1 w_2$ matched by $\mathtt{P}_1$, still allowing the corresponding suffix to be matched by $\mathtt{P}_2$. The formal requirement is the same as before. Let us define the *right breaking* of C by languages L_1 and L_2, denoted as $\mathit{rbreak}(C, L_1, L_2)$ to be exactly the set of such words w_2:

$$\begin{aligned}\mathit{rbreak}(C, L_1, L_2) := \{w_2 \in L_2 \mid \exists w_1 \in L_1 : w_1 w_2 \in C \\ \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3 w_4 = w_2 \wedge w_1 w_3 \in L_1 \wedge w_4 \in L_2)\}.\end{aligned}$$

Then $\mathcal{T}^{\mathbb{P}}(2n, \mathtt{P}, C)$ equals $\mathcal{T}^{\mathbb{P}}(n, \mathtt{P}_2, \mathit{rbreak}(C, L(\mathtt{P}_1), L(\mathtt{P}_2)))$.

As we will show below, the sets $\mathit{lbreak}(C, L_1, L_2)$ and $\mathit{rbreak}(C, L_1, L_2)$ are regular and can effectively be computed if C, L_1, and L_2 are regular. The type inference algorithm for the POSIX matching policy is then shown in Algorithm 1. It is well-defined if we start with a regular set C, since all used operations can effectively be computed for regular languages. Moreover, the algorithm is terminating since the depth of the nodes to be calculated get smaller upon each recursive call.

5.2 Computing the breaking of C

In order for Algorithm 1 to make any sense, we need a way to calculate the sets $\mathit{lbreak}(C, L(\mathtt{P}_1), L(\mathtt{P}_2))$ and $\mathit{rbreak}(C, L(\mathtt{P}_1), L(\mathtt{P}_2))$. We first need some auxiliary notions in order to develop a computation strategy.

The *left quotient* of language L by language K, denoted by $K \backslash L$, is defined as $\{s \mid \exists p \in K : ps \in L\}$. The *right quotient* of L by K, denoted by L/K, is defined

as $\{p \mid \exists s \in K : ps \in L\}$. It is well-known that regular languages are closed under both quotients [Hopcroft and Ullman 1979].

Let $\Box$ be a special symbol not in Σ. Let us write $\pi(w)$ for the word $w_1 w_2 \ldots w_n$ if $w = w_1 \Box w_2 \Box \cdots \Box w_n$. It is easy to see that if L is a regular language, then so is $\pi^{-1}(L) = \{w_1 \Box w_2 \Box \cdots \Box w_n \mid w_1 w_2 \ldots w_n \in L\}$ (modify a DFA for L to allow reading the letter $\Box$, which is then ignored).

The *breaking* of C by L_1 and L_2, denoted by $\mathit{break}(C, L_1, L_2)$, is defined as:

$$\begin{aligned}\mathit{break}(C, L_1, L_2) := \{w_1 \Box w_2 \mid\ & w_1 w_2 \in C \wedge w_1 \in L_1 \wedge w_2 \in L_2 \\ & \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3 w_4 = w_2 \wedge w_1 w_3 \in L_1 \wedge w_4 \in L_2)\}.\end{aligned}$$

Lemma 5.2. *If C, L_1, and L_2 are regular, then so are the breaking, left breaking and right breaking of C by L_1 and L_2. More specifically, with A abbreviating the language $\pi^{-1}(L_1) - (L_1 \cdot \{\Box\})$, we have:*

—$\mathit{break}(C, L_1, L_2) = \pi^{-1}(C) \cap ((L_1 \cdot \{\Box\} \cdot L_2) - A \cdot L_2)$,
—$\mathit{lbreak}(C, L_1, L_2) = \mathit{break}(C, L_1, L_2)/(\{\Box\} \cdot L_2)$, *and*
—$\mathit{rbreak}(C, L_1, L_2) = (L_1 \cdot \{\Box\}) \backslash \mathit{break}(C, L_1, L_2)$.

Proof. By definition, $(L_1 \cdot \{\Box\} \cdot L_2) - A \cdot L_2$ equals

$$\{w_1 \Box w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2 \wedge \neg(\exists v_1, v_2 : v_1 v_2 = w_1 \Box w_2 \wedge v_1 \in A \wedge v_2 \in L_2)\}.$$

Or, more elaborately,

$$\begin{aligned}&\{w_1 \Box w_2 \mid w_1 \in L_1 \wedge\ w_2 \in L_2 \\ &\quad \wedge \neg(\exists v_1, v_2 : v_1 v_2 = w_1 \Box w_2 \wedge \pi(v_1) \in L_1 \wedge (\forall p \in L_1 : v_1 \neq p\Box) \wedge v_2 \in L_2)\}.\end{aligned}$$

We show that this equals

$$\begin{aligned}\{w_1 \Box w_2 \mid\ & w_1 \in L_1 \wedge\ w_2 \in L_2 \\ & \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_2 = w_3 w_4 \wedge w_1 w_3 \in L_1 \wedge w_4 \in L_2)\}.\end{aligned}$$

We can see this as follows. Suppose that $w_1 \Box w_2$ is in the upper set and suppose that there do exist w_3 and w_4 such that $w_2 = w_3 w_4$, $w_3 \neq \lambda$, $w_1 w_3 \in L_1$ and $w_4 \in L_2$. Then take $v_1 = w_1 \Box w_3$ and $v_2 = w_4$ to see that $w_1 \Box w_2$ cannot be in the upper set, a contradiction. On the other hand, suppose $w_1 \Box w_2$ is in the lower set and suppose that there do exist v_1 and v_2 such that $v_1 v_2 = w_1 \Box w_2$, $\pi(v_1) \in L_1$, $\forall p \in L_1 : v_1 \neq p\Box$ and $v_2 \in L_2$. Since $v_2 \in L_2$ and since L_2 is a language over Σ, v_2 cannot contain the symbol $\Box$. Since $v_1 v_2 = w_1 \Box w_2$, v_2 must be a suffix of w_2. Hence, we can divide w_2 in w_3 and w_4 such that $v_1 = w_1 \Box w_3$ and $v_2 = w_4$. Since $v_1 \neq p\Box$ for any p, w_3 must be different from λ. Moreover, we immediately have $w_1 w_3 = \pi(v_1) \in L_1$ and $w_4 = v_2 \in L_2$, which gives us a contradiction.

As a consequence, $\pi^{-1}(C) \cap ((L_1 \cdot \{\Box\} \cdot L_2) - A \cdot L_2)$ must equal

$$\begin{aligned}\{w_1 \Box w_2 \mid\ & w_1 w_2 \in C \wedge w_1 \in L_1 \wedge w_2 \in L_2 \\ & \wedge \neg(\exists w_3, w_4 : w_3 \neq \lambda \wedge w_3 w_4 = w_2 \wedge w_1 w_3 \in L_1 \wedge w_4 \in L_2)\}.\end{aligned}$$

Hence, $\pi^{-1}(C) \cap ((L_1 \cdot \{\Box\} \cdot L_2) - A \cdot L_2) = \mathit{break}(C, L_1, L_2)$, as desired. With ϕ abbreviating $\neg(\exists w_3 \neq \lambda, w_4 : w_3 w_4 = w_2 \wedge w_1 w_3 \in L_1 \wedge w_4 \in L_2)$ we obtain the

other two desired equalities:

$$\begin{aligned}
\mathit{break}(C, L_1, L_2)/(\{\Box\} \cdot L_2) &= \{w_1 \mid \exists w_2 \in L_2 : w_1 \Box w_2 \in \mathit{break}(C, L_1, L_2)\} \\
&= \{w_1 \mid \exists w_2 \in L_2 : w_1 w_2 \in C \wedge w_1 \in L_1 \wedge \phi\} \\
&= \mathit{lbreak}(C, L_1, L_2) \\
(L_1 \cdot \{\Box\}) \backslash \mathit{break}(C, L_1, L_2) &= \{w_2 \mid \exists w_1 \in L_1 : w_1 \Box w_2 \in \mathit{break}(C, L_1, L_2)\} \\
&= \{w_2 \mid \exists w_1 \in L_1 : w_1 w_2 \in C \wedge w_2 \in L_2 \wedge \phi\} \\
&= \mathit{rbreak}(C, L_1, L_2)
\end{aligned}$$

□

5.3 Proof of Correctness

In this section we formally prove the correctness of Algorithm 1, thereby also proving Theorem 5.1.

LEMMA 5.3. *If $w \in \mathsf{P} \rightsquigarrow V$ then $V(\lambda) = w$.*

PROOF. By a straightforward induction on the matching derivation. □

PROPOSITION 5.4. *$\mathcal{T}^{\mathbb{P}}(\lambda, \mathsf{P}, C) = L(\mathsf{P}) \cap C$ for any pattern P.*

PROOF. By Lemma 5.3 and Theorem 4.2 it readily follows:

$$\begin{aligned}
w \in \mathcal{T}^{\mathbb{P}}(\lambda, \mathsf{P}, C) &\Leftrightarrow \exists w' \in C : w' \in \mathsf{P} \rightsquigarrow V \wedge V(\lambda) = w \\
&\Leftrightarrow w \in C \wedge w \in \mathsf{P} \rightsquigarrow V \\
&\Leftrightarrow w \in C \wedge w \in L(\mathsf{P})
\end{aligned}$$

□

PROPOSITION 5.5. *For $\mathsf{P} = \mathsf{P}_1 + \mathsf{P}_2$, the following equalities hold:*

(1) $\mathcal{T}^{\mathbb{P}}(1n, \mathsf{P}, C) = \mathcal{T}^{\mathbb{P}}(n, \mathsf{P}_1, C)$

(2) $\mathcal{T}^{\mathbb{P}}(2n, \mathsf{P}, C) = \mathcal{T}^{\mathbb{P}}(n, \mathsf{P}_2, C - L(\mathsf{P}_1))$

PROOF. We first note that the top of a derivation for $w' \in \mathsf{P} \rightsquigarrow V$ has two possible forms:

$$\frac{\dfrac{\cdots}{w' \in \mathsf{P}_1 \rightsquigarrow V_1}}{w' \in \mathsf{P}_1 + \mathsf{P}_2 \rightsquigarrow V_1 + \mathsf{P}_2}\ \textsc{Or1} \qquad \frac{\dfrac{\cdots}{w' \in \mathsf{P}_2 \rightsquigarrow V_2} \qquad w' \notin L(\mathsf{P}_1)}{w' \in \mathsf{P}_1 + \mathsf{P}_2 \rightsquigarrow \mathsf{P}_1 + V_2}\ \textsc{Or2}$$

Note that, if $w' \in \mathsf{P} \rightsquigarrow V$ and $V(1) \neq \bot$, then the derivation of $w' \in \mathsf{P} \rightsquigarrow V$ must be of the left form. Indeed, $V(1) = \bot$ for derivations of the right form. It is then easy to see that (1) holds:

$$\begin{aligned}
w \in \mathcal{T}^{\mathbb{P}}(1n, \mathsf{P}, C) &\Leftrightarrow \exists w' \in C : w' \in \mathsf{P} \rightsquigarrow V \wedge V(1n) = w \\
&\Leftrightarrow \exists w' \in C : w' \in \mathsf{P} \rightsquigarrow (V_1 + \mathsf{P}_2) \wedge V_1(n) = w \\
&\Leftrightarrow \exists w' \in C : w' \in \mathsf{P}_1 \rightsquigarrow V_1 \wedge V_1(n) = w \\
&\Leftrightarrow w \in \mathcal{T}^{\mathbb{P}}(n, \mathsf{P}_1, C)
\end{aligned}$$

Likewise, if $w' \in \mathtt{P}{\rightsquigarrow}V$ and $V(2) \neq \bot$, then the derivation of $w' \in \mathtt{P}{\rightsquigarrow}V$ must be of the right form. Indeed, $V(2) = \bot$ for derivations of the left form. It is then easy to see that (2) also holds:

$$\begin{aligned} w \in \mathcal{T}^{\mathbb{P}}(2n, \mathtt{P}, C) &\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}V \wedge V(2n) = w \\ &\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}(\mathtt{P}_1 + V_2) \wedge V_2(n) = w \\ &\Leftrightarrow \exists w' \in C : w' \notin L(\mathtt{P}_1) \wedge w' \in \mathtt{P}_2{\rightsquigarrow}V_2 \wedge V_2(n) = w \\ &\Leftrightarrow w \in \mathcal{T}^{\mathbb{P}}(n, \mathtt{P}_2, C - L(\mathtt{P}_1)) \end{aligned}$$

□

PROPOSITION 5.6. *When $P = P_1 \cdot P_2$ the following equalities hold:*

(1) $\mathcal{T}^{\mathbb{P}}(1n, P, C) = \mathcal{T}^{\mathbb{P}}(n, P_1, \mathit{lbreak}(C, L(P_1), L(P_2)))$

(2) $\mathcal{T}^{\mathbb{P}}(2n, P, C) = \mathcal{T}^{\mathbb{P}}(n, P_2, \mathit{rbreak}(C, L(P_1), L(P_2)))$

PROOF. Note that for any derivation of $w' \in \mathtt{P}{\rightsquigarrow}V$, the top must look like:

$$\dfrac{\dfrac{\cdots}{w_1 \in \mathtt{P}_1{\rightsquigarrow}V_1} \qquad \dfrac{\cdots}{w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2} \qquad \neg(\exists w_3 \neq \lambda, w_4 : w_2 = w_3 w_4 \wedge w_1 w_3 \in L(\mathtt{P}_1) \wedge w_4 \in L(\mathtt{P}_2))}{w' = w_1 w_2 \in \mathtt{P}{\rightsquigarrow}V = V_1 \cdot V_2}\ \textsc{Concat}$$

Using Theorem 4.2, equality (1) then readily follows:

$$\begin{aligned} & w \in \mathcal{T}^{\mathbb{P}}(1n, \mathtt{P}, C) \\ & \quad \Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}V \wedge V(1n) = w \\ & \quad \Leftrightarrow \exists w_1, w_2 : w_1 w_2 \in C \wedge w_1 \in \mathtt{P}_1{\rightsquigarrow}V_1 \wedge w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2 \wedge V_1(n) = w \\ & \qquad \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3 w_4 = w_2 \wedge w_1 w_3 \in L(\mathtt{P}_1) \wedge w_4 \in L(\mathtt{P}_2)) \\ & \quad \Leftrightarrow \exists w_1 \in L(\mathtt{P}_1), w_2 \in L(\mathtt{P}_2) : w_1 w_2 \in C \wedge w_1 \in \mathtt{P}_1{\rightsquigarrow}V_1 \wedge V_1(n) = w \\ & \qquad \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3 w_4 = w_2 \wedge w_1 w_3 \in L(\mathtt{P}_1) \wedge w_4 \in L(\mathtt{P}_2)) \\ & \quad \Leftrightarrow \exists w_1 \in \mathit{lbreak}(C, L(\mathtt{P}_1), L(\mathtt{P}_2)) : w_1 \in \mathtt{P}_1{\rightsquigarrow}V_1 \wedge V_1(n) = w \\ & \quad \Leftrightarrow w \in \mathcal{T}^{\mathbb{P}}(n, \mathtt{P}_1, \mathit{lbreak}(C, L(\mathtt{P}_1), L(\mathtt{P}_2))) \end{aligned}$$

Equality (2) can be obtained in a similar way. □

6. MATCHING UNDER THE FIRST AND LONGEST MATCH POLICY

In this section, we formally define the matching process on strings under the first and longest match disambiguation policy, and show that it guarantees a unique matching strategy. We also discuss the difference between the first and longest match policy and the XDuce policy.

Recall from Section 2.1 that the first and longest match policy consists of two disambiguation rules. The *first match rule* disambiguates a disjunction $\mathtt{P}_1 + \mathtt{P}_2$ by giving higher priority to the first alternative $\mathtt{P}_1$. Moreover, disjunction distributes over concatenation. That is, when matching w against $(\mathtt{P}_1 + \mathtt{P}_2) \cdot \mathtt{P}_3$, w should be first matched against $\mathtt{P}_1 \cdot \mathtt{P}_3$ and it should only be matched against $\mathtt{P}_2 \cdot \mathtt{P}_3$ when this fails. The *longest match rule* disambiguates the Kleene closure in patterns of the form $\mathtt{P}_1^* \cdot \mathtt{P}_2$ by requiring that $\mathtt{P}_1^*$ matches as much of the input as possible, still allowing the rest of the pattern to match.

Example 6.1. Consider the matching of ab against the pattern $(a+a\cdot b)\cdot(b+\varepsilon)$ of Figure 2(a). Then the whole pattern matches ab. Since disjunction distributes over concatenation, the first match rule requires us to first try to match ab against $a\cdot(b+\varepsilon)$. This obviously succeeds. Since a is matched by a and b by $(b+\varepsilon)$ in $a\cdot(b+\varepsilon)$, we associate $(a+a\cdot b)$ with a and $(b+\varepsilon)$ with b in $(a+a\cdot b)\cdot(b+\varepsilon)$. □

In contrast, $(a\cdot a+b)$ is associated with ab and $(b+\varepsilon)$ with ε under the POSIX disambiguation policy, as we have shown in Example 4.1. Thus, under the first and longest match policy we no longer require that $\mathtt{P}_1$ matches as much as possible in a concatenation $\mathtt{P}_1\cdot\mathtt{P}_2$, *unless* $\mathtt{P}_1$ is a Kleene closure.

The matching relation $w\in\mathtt{P}\rightsquigarrow V$ under the first and longest match policy is formally defined in Figure 4. Rules Empty, Lab, Kleene, Or1, and Or2 are the same as in Figure 3. The difference with the POSIX policy lies in the treatment of concatenation patterns $\mathtt{P}_1\cdot\mathtt{P}_2$, for which we use the auxiliary relation $(w_1,w_2)\in\mathtt{P}_1\cdot\mathtt{P}_2\rightsquigarrow(V_1,V_2)$. The intuitive meaning of this relation is that when matching w_1w_2 by $\mathtt{P}_1\cdot\mathtt{P}_2$ under the first and longest match policy, $\mathtt{P}_1$ will be responsible for matching prefix w_1 (with associations V_1), while $\mathtt{P}_2$ is responsible for matching suffix w_2 (with associations V_2). If $\mathtt{P}_1=\varepsilon$ or $\mathtt{P}_1=\sigma$, there is only one way to split the input word and no disambiguation is necessary, as expressed in rules CEmpty and CLab. Rules COr1 and COr2 express distribution of disjunction over concatenation, according to the first match rule. The longest match rule is expressed in CKleene. Note the resemblance of this rule with Concat of Figure 3. When matching w against patterns of the form $(\mathtt{P}_1\cdot\mathtt{P}_2)\cdot\mathtt{P}_3$, we first determine the prefix w_1 that is matched by $\mathtt{P}_1$ by matching w against $\mathtt{P}_1\cdot(\mathtt{P}_2\cdot\mathtt{P}_3)$. Then we determine which parts of the corresponding suffix are matched by $\mathtt{P}_2$ and $\mathtt{P}_3$ by matching this suffix against $\mathtt{P}_2\cdot\mathtt{P}_3$. The subword matched by $\mathtt{P}_1\cdot\mathtt{P}_2$ is then the concatenation of the subword matched by $\mathtt{P}_1$ and the subword matched by $\mathtt{P}_2$, as shown in rule CCon. Finally, rule Concat is used to convert from the auxiliary relation to the matching relation.

Example 6.2. As an example of the first match rule, consider the following matching derivation of ab against pattern $(a+a\cdot b)\cdot(b+\varepsilon)$:

$$\dfrac{\dfrac{\dfrac{\textsc{Lab}\ \dfrac{}{a\in a\rightsquigarrow V_1:=[\lambda\to a]} \qquad \dfrac{\dfrac{}{b\in b\rightsquigarrow V_2:=[\lambda\to b]}\ \textsc{Lab}}{b\in b+\varepsilon\rightsquigarrow V_2+\varepsilon}\ \textsc{Or1}}{(a,b)\in a\cdot(b+\varepsilon)\rightsquigarrow(V_1,V_2+\varepsilon)}\ \textsc{CLab}}{(a,b)\in(a+a\cdot b)\cdot(b+\varepsilon)\rightsquigarrow(V_1+(a\cdot b),V_2+\varepsilon)}\ \textsc{COr1}}{ab\in(a+a\cdot b)\cdot(b+\varepsilon)\rightsquigarrow(V_1+(a\cdot b))\cdot(V_2+\varepsilon)}\ \textsc{Concat}$$

It is easily seen that the obtained association function $(V_1+(a\cdot b))\cdot(V_1+\varepsilon)$ equals the association function V' from Example 3.1. For example,

$$((V_1+(a\cdot b))\cdot(V_2+\varepsilon))(1)=(V_1+(a\cdot b))(\lambda)=V_1(\lambda)=a=V'(1).$$

Likewise,

$$((V_1+(a\cdot b))\cdot(V_2+\varepsilon))(12)=(V_1+(a\cdot b))(2)=\ \perp=V'(12).$$

□

$$\textsc{Empty}\ \frac{}{\lambda \in \varepsilon \rightsquigarrow [\lambda \to \lambda]} \qquad \textsc{Lab}\ \frac{}{\sigma \in \sigma \rightsquigarrow [\lambda \to \sigma]} \qquad \textsc{Kleene}\ \frac{w \in L(\mathtt{P}^*)}{w \in \mathtt{P}^* \rightsquigarrow [\lambda \to w]}$$

$$\textsc{Or1}\ \frac{w \in \mathtt{P}_1 \rightsquigarrow V}{w \in \mathtt{P}_1 + \mathtt{P}_2 \rightsquigarrow V + \mathtt{P}_2} \qquad \textsc{Or2}\ \frac{w \in \mathtt{P}_2 \rightsquigarrow V \quad w \notin L(\mathtt{P}_1)}{w \in \mathtt{P}_1 + \mathtt{P}_2 \rightsquigarrow \mathtt{P}_1 + V}$$

$$\textsc{Concat}\ \frac{(w_1, w_2) \in \mathtt{P}_1 \cdot \mathtt{P}_2 \rightsquigarrow (V_1, V_2)}{w_1 w_2 \in \mathtt{P}_1 \cdot \mathtt{P}_2 \rightsquigarrow V_1 \cdot V_2} \qquad \textsc{CEmpty}\ \frac{w \in \mathtt{P} \rightsquigarrow V_2}{(\lambda, w) \in \varepsilon \cdot \mathtt{P} \rightsquigarrow ([\lambda \to \lambda], V_2)} \qquad \textsc{CLab}\ \frac{\sigma \in \sigma \rightsquigarrow V_1 \quad w \in \mathtt{P} \rightsquigarrow V_2}{(\sigma, w) \in \sigma \cdot \mathtt{P} \rightsquigarrow (V_1, V_2)}$$

$$\textsc{COr1}\ \frac{(w_1, w_2) \in \mathtt{P}_1 \cdot \mathtt{P}_3 \rightsquigarrow (V_1, V_2)}{(w_1, w_2) \in (\mathtt{P}_1 + \mathtt{P}_2) \cdot \mathtt{P}_3 \rightsquigarrow (V_1 + \mathtt{P}_2, V_2)} \qquad \textsc{COr2}\ \frac{(w_1, w_2) \in \mathtt{P}_2 \cdot \mathtt{P}_3 \rightsquigarrow (V_1, V_2) \quad w_1 w_2 \notin L(\mathtt{P}_1 \cdot \mathtt{P}_3)}{(w_1, w_2) \in (\mathtt{P}_1 + \mathtt{P}_2) \cdot \mathtt{P}_3 \rightsquigarrow (\mathtt{P}_1 + V_1, V_2)}$$

$$\textsc{CCon}\ \frac{(w_1, w_2 w_3) \in \mathtt{P}_1 \cdot (\mathtt{P}_2 \cdot \mathtt{P}_3) \rightsquigarrow (V_1, W) \quad (w_2, w_3) \in \mathtt{P}_2 \cdot \mathtt{P}_3 \rightsquigarrow (V_2, V_3)}{(w_1 w_2, w_3) \in (\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3 \rightsquigarrow (V_1 \cdot V_2, V_3)}$$

$$\textsc{CKleene}\ \frac{w_1 \in \mathtt{P}_1^* \rightsquigarrow V_1 \quad w_2 \in \mathtt{P}_2 \rightsquigarrow V_2 \quad \neg(\exists w_3 \neq \lambda, w_4 : w_3 w_4 = w_2 \wedge w_1 w_3 \in L(\mathtt{P}_1^*) \wedge w_4 \in L(\mathtt{P}_2))}{(w_1, w_2) \in \mathtt{P}_1^* \cdot \mathtt{P}_2 \rightsquigarrow (V_1, V_2)}$$

Fig. 4. The matching relation $w \in \mathtt{P} \rightsquigarrow V$ under the first and longest match disambiguation policy.

Example 6.3. As an example of the longest match rule, consider the following matching derivation of ab against the pattern $(a + a \cdot b)^* \cdot (b + \varepsilon)$ of Figure 2(b):

$$\textsc{Concat}\ \dfrac{\textsc{CKleene}\ \dfrac{\textsc{Kleene}\ \dfrac{ab \in L((a + a \cdot b)^*)}{ab \in (a + a \cdot b)^* \rightsquigarrow V_1 := [\lambda \to ab]} \quad \textsc{Or2}\ \dfrac{\textsc{Empty}\ \dfrac{}{\lambda \in \varepsilon \rightsquigarrow V_2 := [\lambda \to \lambda]} \quad \lambda \notin L(b)}{\lambda \in (b + \varepsilon) \rightsquigarrow \varepsilon + V_2}}{(ab, \lambda) \in (a + a \cdot b)^* \cdot (b + \varepsilon) \rightsquigarrow (V_1, \varepsilon + V_2)}}{ab \in (a + a \cdot b)^* \cdot (b + \varepsilon) \rightsquigarrow V_1 \cdot (\varepsilon + V_2)}$$

Here, ab itself is matched by $(a + a \cdot b)^*$, while $(b + \varepsilon)$ matches λ:

$$(V_1 \cdot (\varepsilon + V_2))(1) = V_1(\lambda) = ab,$$
$$(V_1 \cdot (\varepsilon + V_2))(2) = V_2(\lambda) = \lambda.$$

Matching a by $(a + a \cdot b)^*$ and b by $(b + \varepsilon)$ will not work. Indeed, although it is possible to derive $a \in (a + a \cdot b)^* \rightsquigarrow W_1$ and $b \in (b + \varepsilon) \rightsquigarrow W_2$ for some associations W_1 and W_2, the third premise of CKleene will disable us to derive $(a, b) \in (a + a \cdot b)^* \cdot (b + \varepsilon) \rightsquigarrow (W_1, W_2)$. □

As an analogy to Theorem 4.2 we have:

Theorem 6.4. *The matching relation of Figure 4 is well defined:*

(1) *The matching relation is semantically correct:* $w \in \mathsf{P}{\rightsquigarrow}V$ *iff* $w \in L(\mathsf{P})$*, and,*

(2) *The matching relation is unique: if* $w \in \mathsf{P}{\rightsquigarrow}V$ *and* $w \in \mathsf{P}{\rightsquigarrow}W$ *then* $V = W$*.*

Proof. (1). The "only if" direction can be obtained by a straightforward induction on the matching derivation of $w \in \mathsf{P}{\rightsquigarrow}V$, with a case analysis on the last rule used. We highlight the case where this last rule is Concat. In that case $\mathsf{P} = \mathsf{P}_1 \cdot \mathsf{P}_2$ and we can split up w into w_1 and w_2 such that $(w_1, w_2) \in \mathsf{P}_1 \cdot \mathsf{P}_2{\rightsquigarrow}(V_1, V_2)$ with $V = V_1 \cdot V_2$. A straightforward induction on the derivation of $(w_1, w_2) \in \mathsf{P}_1 \cdot \mathsf{P}_2{\rightsquigarrow}(V_1, V_2)$ shows that $w_1 \in L(\mathsf{P}_1)$ and $w_2 \in L(\mathsf{P}_2)$. Hence, $w_1 w_2 \in L(\mathsf{P})$, as desired.

The "if" direction can be obtained by well-founded induction [Baader and Nipkow 1998] on P according to the well-founded relation $\succ$. Here, $\succ$ relates a pattern P with immediate subpatterns if $\mathsf{P} \neq (\mathsf{P}_1 \cdot \mathsf{P}_2) \cdot \mathsf{P}_3$ and $\mathsf{P} \neq (\mathsf{P}_1 + \mathsf{P}_2) \cdot \mathsf{P}_3$. It relates $(\mathsf{P}_1 \cdot \mathsf{P}_2) \cdot \mathsf{P}_3$ with $\mathsf{P}_2 \cdot \mathsf{P}_3$ and with $\mathsf{P}_1 \cdot (\mathsf{P}_2 \cdot \mathsf{P}_3)$ and it relates $(\mathsf{P}_1 + \mathsf{P}_2) \cdot \mathsf{P}_3$ with $\mathsf{P}_1 \cdot \mathsf{P}_3$ and with $\mathsf{P}_2 \cdot \mathsf{P}_3$. The monotone embedding ϕ into the lexicographically ordered set $\mathbb{N} \times \mathbb{N}$ where $\phi(\mathsf{P}) = (|\mathsf{P}|, 0)$ if $\mathsf{P} \neq \mathsf{P}_1 \cdot \mathsf{P}_2$ and $\phi(\mathsf{P}_1 \cdot \mathsf{P}_2) = (|\mathsf{P}_1 \cdot \mathsf{P}_2|, |\mathsf{P}_1|)$ otherwise, shows that $\succ$ is well-founded [Baader and Nipkow 1998].

(2). By a straightforward induction on the matching derivation of $w \in \mathsf{P}{\rightsquigarrow}V$, with a case analysis on the last rule used. We highlight the case where this last rule is Concat. In that case, $\mathsf{P} = \mathsf{P}_1 \cdot \mathsf{P}_2$ and we can split up w into w_1 and w_2 such that $(w_1, w_2) \in \mathsf{P}_1 \cdot \mathsf{P}_2{\rightsquigarrow}(V_1, V_2)$ with $V = V_1 \cdot V_2$. Furthermore, since we also have $w \in \mathsf{P}{\rightsquigarrow}W$, we can also split up w into w_3 and w_4 such that $(w_3, w_4) \in \mathsf{P}_1 \cdot \mathsf{P}_2{\rightsquigarrow}(W_1, W_2)$ with $W = W_1 \cdot W_2$. A straightforward induction on the derivation of $(w_1, w_2) \in \mathsf{P}_1 \cdot \mathsf{P}_2{\rightsquigarrow}(V_1, V_2)$ then shows that $w_1 = w_3$, $w_2 = w_4$, $V_1 = W_1$, and $V_2 = W_2$. Hence, $\in L(\mathsf{P}_1)$ and $w_2 \in L(\mathsf{P}_2)$. Hence, $V = V_1 \cdot V_2 = W_1 \cdot W_2 = W$, as desired. □

6.1 Relation with the XDuce policy

The disambiguation policy employed in XDuce [Hosoya 2000; Hosoya and Pierce 2002], ℂDuce [Frisch et al. 2003], λ^{re} [Tabuchi et al. 2002], and Perl [Wall et al. 2000] consists of two rules: first match and greedy match. The *first match rule* is the same as in the first and longest match policy. The *greedy match rule* disambiguates a Kleene closure and is defined in terms of the first match policy and recursion. Formally, the matching relation under the XDuce policy is obtained from the matching relation of the first and longest match policy by replacing rule CKleene as follows [Tabuchi et al. 2002]:

$$\text{CKleene}' \quad \frac{(w_1, w_2) \in ((\mathsf{P}_1 \cdot \mathsf{P}_1^*) + \varepsilon) \cdot \mathsf{P}_2{\rightsquigarrow}(V_1, V_2)}{(w_1, w_2) \in \mathsf{P}_1^* \cdot \mathsf{P}_2{\rightsquigarrow}([\lambda \to V_1(\lambda)], V_2)}$$

Here, it is assumed without loss of generality that $\lambda \notin L(\mathsf{P}_1)$.

The behavior of the greedy match rule was informally explained in [Hosoya 2000; Hosoya and Pierce 2002] as being the longest match rule. The intuition behind this was that, when trying to derive $w \in \mathsf{P}_1^* \cdot \mathsf{P}_2{\rightsquigarrow}V$, we will be forced by the first match rule to consider $(\mathsf{P}_1 \cdot \mathsf{P}_1^*) \cdot \mathsf{P}_2$ before $\varepsilon \cdot \mathsf{P}_2$ at every expansion of $\mathsf{P}_1^* \cdot \mathsf{P}_2$. Since $\lambda \notin L(\mathsf{P}_1)$, this should require us to split w into $w_1 \in L(\mathsf{P}_1^*)$ and $w_2 \in L(\mathsf{P}_2)$ such that w_2 is the smallest suffix of w still matched by P_2. This is, however, a false intuition. Indeed,

because the first match strategy continues to be used in $\mathtt{P}_1$, it is possible that $\mathtt{P}_2$ is allowed to start matching before a longer matching alternative in $\mathtt{P}_1$ is considered. For example, consider the matching of ab against $\mathtt{P} = (a + a \cdot b)^* \cdot (b + \varepsilon)$. Under the first and longest match policy, the subpattern $(a + a \cdot b)^*$ is associated with ab, as we have shown in Example 6.3. Under the first and greedy match policy, however, this subpattern is associated with a, while the subpattern $(b + \varepsilon)$ is associated with b, as we show next. Let us abbreviate $(a + a \cdot b) \cdot (a + a \cdot b)^*$ by $(a + a \cdot b)^+$. We first derive:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{b \in b \rightsquigarrow V_3 := [\lambda \to b]}\,\textsc{Lab}}{b \in (b+\varepsilon) \rightsquigarrow V_3}\,\textsc{Or1}}{(\lambda, b) \in \varepsilon \cdot (b+\varepsilon) \rightsquigarrow ([\lambda \to \lambda], V_3)}\,\textsc{CEmpty} \qquad b \notin L((a + a \cdot b)^+ \cdot (b+\varepsilon))}{(\lambda, b) \in ((a + a \cdot b)^+ + \varepsilon) \cdot (b+\varepsilon) \rightsquigarrow (V_2, V_3)}\,\textsc{COr2}}{(\lambda, b) \in (a + a \cdot b)^* \cdot (b+\varepsilon) \rightsquigarrow (V_2, V_3)}\,\textsc{CKleene}'$$

Here, $V_2 = ((a + a \cdot b) \cdot (a + a \cdot b)^*) + [\lambda \to \lambda]$. Using this derivation of $(\lambda, b) \in \mathtt{P} \rightsquigarrow (V_2, V_3)$, we derive:

$$\dfrac{\dfrac{\dfrac{}{a \in a \rightsquigarrow [\lambda \to a]}\,\textsc{Lab} \qquad \dfrac{\dfrac{\cdots}{(\lambda, b) \in \mathtt{P} \rightsquigarrow (V_2, V_3)}\,\textsc{CKleene}'}{b \in \mathtt{P} \rightsquigarrow V_2 \cdot V_3}\,\textsc{Concat}}{(a, b) \in a \cdot \mathtt{P} \rightsquigarrow ([\lambda \to a], V_2 \cdot V_3)}\,\textsc{CLab}}{(a, b) \in (a + a \cdot b) \cdot \mathtt{P} \rightsquigarrow (V_1 := [\lambda \to a] + (a \cdot b), V_2 \cdot V_3)}\,\textsc{COr1}$$

Finally, we obtain:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\cdots}{(a, b) \in (a + a \cdot b) \cdot \mathtt{P} \rightsquigarrow (V_1, V_2 \cdot V_3)}\,\textsc{COr1} \qquad \dfrac{\cdots}{(\lambda, b) \in \mathtt{P} \rightsquigarrow (V_2, V_3)}\,\textsc{CKleene}'}{(a, b) \in (a + a \cdot b)^+ \cdot (b+\varepsilon) \rightsquigarrow (V_1 \cdot V_2, V_3)}\,\textsc{CCon}}{(a, b) \in ((a + a \cdot b)^+ + \varepsilon) \cdot (b+\varepsilon) \rightsquigarrow ((V_1 \cdot V_2) + \varepsilon, V_3)}\,\textsc{COr1}}{(a, b) \in (a + a \cdot b)^* \cdot (b+\varepsilon) \rightsquigarrow (V_1' := [\lambda \to ((V_1 \cdot V_2) + \varepsilon)(\lambda)], V_3)}\,\textsc{CKleene}'}{ab \in \mathtt{P} \rightsquigarrow V_1' \cdot V_3}\,\textsc{Concat}$$

Note that the subpattern $(a + a \cdot b)^*$ is associated with a and the subpattern $(b + \varepsilon)$ is associated with b, as we wanted to show:

$$(V_1' \cdot V_3)(1) = ((V_1 \cdot V_2) + \varepsilon)(\lambda) = V_1(\lambda) \cdot V_2(\lambda) = a,$$
$$(V_1' \cdot V_3)(2) = V_3(\lambda) = b.$$

The type inference problem for the XDuce policy has already been extensively studied [Hosoya and Pierce 2002; Hosoya 2000; Frisch et al. 2002; Tabuchi et al. 2002], and will not further be considered in this paper.

7. TYPE INFERENCE UNDER THE FIRST AND LONGEST MATCH POLICY

In this section we solve the type inference problem for the first and longest match policy, employing some of the techniques introduced in Section 5. We note that type inference algorithms developed for the XDuce policy cannot be used directly to do type inference for the first and longest match policy. Indeed, using the same counterexample pattern $\mathtt{P} = (a + a \cdot b)^* \cdot (b + \varepsilon)$ and word ab from Section 6.1, these algorithms must calculate $\mathcal{T}^{\mathbb{XD}}(1, \mathtt{P}, \{ab\}) = \{a\}$ and $\mathcal{T}^{\mathbb{XD}}(2, \mathtt{P}, \{ab\}) = \{b\}$. In contrast, as shown in Example 6.3, the first and longest match policy requires

the actual types to be $\{ab\}$ and $\{\lambda\}$ respectively. The main result of this paper can be stated as follows (to be proven later):

THEOREM 7.1. *If C is a regular language then $\mathcal{T}^{\mathbb{FL}}(m, \mathsf{P}, C)$ is also regular, and can be effectively computed.*

7.1 The algorithm

Algorithm 2 describes the type inference algorithm. As in Section 5.1, we will first explain the algorithm by informal reasoning, and prove its correctness later.

We observe that the type of the root node λ is exactly the set of words in C that can be matched by P. Indeed, if w is successfully matched by P then λ is associated to w itself. If we need to calculate the type for a node other than λ, then P must be of the form $\mathsf{P}_1 + \mathsf{P}_2$ or $\mathsf{P}_1 \cdot \mathsf{P}_2$, since λ is the only bindable node for the other patterns.

If $\mathsf{P} = \mathsf{P}_1 + \mathsf{P}_2$ then we can make the same observations as in Section 5.1. Hence, $\mathcal{T}^{\mathbb{FL}}(1n, \mathsf{P}_1 + \mathsf{P}_2, C)$ equals $\mathcal{T}^{\mathbb{FL}}(n, \mathsf{P}_1, C)$ and $\mathcal{T}^{\mathbb{FL}}(2n, \mathsf{P}_1 + \mathsf{P}_2, C)$ equals $\mathcal{T}^{\mathbb{FL}}(n, \mathsf{P}_2, C - L(\mathsf{P}_1))$.

If $\mathsf{P} = \mathsf{P}_1 \cdot \mathsf{P}_2$ then we need to make a further case analysis:

— If $\mathsf{P}_1 = \varepsilon$ or $\mathsf{P}_1 = \sigma$, then P_1 can only be associated with those words w_1 matched by P_1 for which there exists some word w_2 matched by P_2 such that $w_1 w_2 \in C$. Hence, $w_1 \in C/L(\mathsf{P}_2)$ and $\mathcal{T}^{\mathbb{FL}}(1, \mathsf{P}, C)$ equals $\mathcal{T}^{\mathbb{FL}}(\lambda, \mathsf{P}_1, C/L(\mathsf{P}_2))$. Likewise, subpatterns of P_2 can only be associated to those subwords of a word w_2 matched by P_2 for which there exists a w_1 matched by P_1 such that $w_1 w_2 \in C$. Hence, we can calculate $\mathcal{T}^{\mathbb{FL}}(2n, \mathsf{P}, C)$ by calculating $\mathcal{T}^{\mathbb{FL}}(n, \mathsf{P}_2, L(\mathsf{P}_1) \backslash C)$.

— For $\mathsf{P} = \mathsf{P}_1^* \cdot \mathsf{P}_2$, we again note the similarity between the POSIX and the first and longest match disambiguation policies. That is, P_1^* can only be associated to words w_1 matched by P_1^* for which there exists some w_2 matched by P_2 such that $w_1 w_2 \in C$ and such that w_1 really is the longest possible prefix of $w_1 w_2$ that can be matched by P_1^*, still allowing the corresponding suffix to be matched by P_2. Hence, $\mathcal{T}^{\mathbb{FL}}(1, \mathsf{P}, C)$ equals $\mathit{lbreak}(C, L(\mathsf{P}_1^*), L(\mathsf{P}_2))$. Similarly, $\mathcal{T}^{\mathbb{FL}}(2n, \mathsf{P}, C)$ equals $\mathcal{T}^{\mathbb{FL}}(n, \mathsf{P}_2, \mathit{rbreak}(C, L(\mathsf{P}_1^*), L(\mathsf{P}_2)))$.

— If $\mathsf{P} = (\mathsf{P}_1 + \mathsf{P}_2) \cdot \mathsf{P}_3$, then a word can only be associated to a subpattern of P_1 if it can be associated with P_1 in $\mathsf{P}_1 \cdot \mathsf{P}_3$. Hence, $\mathcal{T}^{\mathbb{FL}}(11n, \mathsf{P}, C)$ equals $\mathcal{T}^{\mathbb{FL}}(1n, \mathsf{P}_1 \cdot \mathsf{P}_3, C)$. Likewise, a word can only be associated with P_2 in P if it can be associated with P_2 in $\mathsf{P}_2 \cdot \mathsf{P}_3$ under context $C - L(\mathsf{P}_1 \cdot \mathsf{P}_3)$. Hence, $\mathcal{T}^{\mathbb{FL}}(21n, \mathsf{P}, C)$ equals

$$\mathcal{T}^{\mathbb{FL}}(1n, \mathsf{P}_2 \cdot \mathsf{P}_3, C - L(\mathsf{P}_1 \cdot \mathsf{P}_3)).$$

Finding the words that can be bound to $(\mathsf{P}_1 + \mathsf{P}_2)$ resolves to calculating the union of words that can be bound to P_1 or P_2. Words can be bound to subpatterns of P_3 if they are subwords of a word w_3 matched by P_3 for which there either exits a word w_1 matched by P_1 such that $w_1 w_3 \in C$, or a word w_2 matched by P_2 such that $w_2 w_3 \in C$ but $w_2 w_3 \notin L(\mathsf{P}_1 \cdot \mathsf{P}_3)$. Hence, $\mathcal{T}^{\mathbb{FL}}(2n, \mathsf{P}, C)$ equals

$$\mathcal{T}^{\mathbb{FL}}(2n, \mathsf{P}_1 \cdot \mathsf{P}_3, C) \cup \mathcal{T}^{\mathbb{FL}}(2n, \mathsf{P}_2 \cdot \mathsf{P}_3, C - L(\mathsf{P}_1 \cdot \mathsf{P}_3)).$$

— Calculating the types of subpatterns of P_1, P_2, or P_3 in $\mathsf{P} = (\mathsf{P}_1 \cdot \mathsf{P}_2) \cdot \mathsf{P}_3$ is simply a matter of calculating the type of the corresponding subpatterns in $\mathsf{P}' = \mathsf{P}_1 \cdot (\mathsf{P}_2 \cdot \mathsf{P}_3)$. The type of $(\mathsf{P}_1 \cdot \mathsf{P}_2)$ is a bit more difficult to find. By definition

Algorithm 2: Calculate $\mathcal{T}^{\mathbb{FL}}(m, \mathtt{P}, C)$.

Input: A pattern $\mathtt{P}$; a node $m \in bn(\mathtt{P})$; and a regular context C.

Output: The type of m in $\mathtt{P}$ relative to C under the first and longest match disambiguation policy.

```
if m = λ then
    return L(P) ∩ C
else
    switch P do
        case P1 + P2
            switch m do
                case 1n  return  T^FL(n, P1, C)
                case 2n  return  T^FL(n, P2, C − L(P1))
            end
        case P1 · P2 with P1 = ε or P1 = σ
            switch m do
                case 1  return  T^FL(λ, P1, C/L(P2))
                case 2n  return  T^FL(n, P2, L(P1)\C)
            end
        case P1* · P2
            switch m do
                case 1  return  lbreak(C, L(P1*), L(P2))
                case 2n  return  T^FL(n, P2, rbreak(C, L(P1*), L(P2)))
            end
        case (P1 + P2) · P3
            let C' = C − L(P1 · P3)
            switch m do
                case 1  return  T^FL(1, P1 · P3, C) ∪ T^FL(1, P2 · P3, C')
                case 11n  return  T^FL(1n, P1 · P3, C)
                case 12n  return  T^FL(1n, P2 · P3, C')
                case 2n  return  T^FL(2n, P1 · P3, C) ∪ T^FL(2n, P2 · P3, C')
            end
        case (P1 · P2) · P3
            let P' = P1 · (P2 · P3)
            switch m do
                case 1  return  M(λ, P, C)/({□} · Σ*)
                case 11n  return  T^FL(1n, P', C)
                case 12n  return  T^FL(21n, P', C)
                case 2n  return  T^FL(22n, P', C)
            end
    end
end
```

of the matching relation, $(\mathtt{P}_1 \cdot \mathtt{P}_2)$ can only be associated to those words w_1w_2 for which there exists a w_3 such that $w_1w_2w_3 \in C$, $(w_1, w_2w_3) \in \mathtt{P}' \rightsquigarrow (V_1, W)$ and $(w_2, w_3) \in \mathtt{P}_2 \cdot \mathtt{P}_3 \rightsquigarrow (V_2, V_3)$. It is tempting to say that this means that the type of $(\mathtt{P}_1 \cdot \mathtt{P}_2)$ is exactly the right quotient of $L(\mathtt{P}) \cap C$ by $L(\mathtt{P}_3)$. This is incorrect however. Indeed, consider the pattern $(a \cdot (a \cdot b + a)) \cdot (b + \varepsilon)$ and context $C = \{aab, aabb\}$. Then $\{aab, aabb\}/\{b, \lambda\} = \{aa, aab, aabb\}$, which is too big since aa will never be associated with $(a \cdot (a \cdot b + a))$ under context C. Indeed, because of the first match policy, every word in C will first be matched against $a \cdot (a \cdot b) \cdot (b + \varepsilon)$, which always succeeds. Hence, the type of $(a \cdot (a \cdot b + a))$ is $\{aab, aabb\}$. In order to correctly calculate the type of $(\mathtt{P}_1 \cdot \mathtt{P}_2)$ in $(\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3$ we will use marked languages, which are defined as follows.

A *marked language* is a set of words of the form $w_1 \Box w_2$. The breaking of a context by two languages, as defined in Section 5.2, is an example of a marked language. Here, we will use the $\Box$ marker to record that matching w_1w_2 against a concatenation $\mathtt{P}_1 \cdot \mathtt{P}_2$ results in w_1 being matched by $\mathtt{P}_1$ and w_2 being matched by $\mathtt{P}_2$. We therefore define, for every pattern $\mathtt{P}$, the *marked language* $M(m, \mathtt{P}, C)$ of a node $m \in bn(\mathtt{P})$ with $\mathtt{P}(m) = \cdot$ under context C as follows:

$$M(m, \mathtt{P}, C) = \{w_1 \Box w_2 \mid \ \exists w' \in C, w' \in \mathtt{P} \rightsquigarrow V, V(m1) = w_1, V(m2) = w_2\}.$$

It is clear that, for $\mathtt{P} = (\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3$, $\mathcal{T}^{\mathbb{FL}}(1, \mathtt{P}, C) = M(\lambda, \mathtt{P}, C)/(\{\Box\} \cdot \Sigma^*)$. So, doing type inference for node 1 in $\mathtt{P}$ is simply a matter of calculating $M(\lambda, \mathtt{P}, C)$. We use Algorithm 3 for this purpose.

Algorithm 3 uses the following reasoning to compute $M(m, (\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3, C)$. Matching rule CCon states that if we want to know which part of word w is matched by $(\mathtt{P}_1 \cdot \mathtt{P}_2)$ when matching w by $\mathtt{P}$, then we first determine how it is broken up against $\mathtt{P}' = \mathtt{P}_1 \cdot (\mathtt{P}_2 \cdot \mathtt{P}_3)$. Suppose that $w = w_1v$, that $\mathtt{P}_1$ is responsible for matching w_1, and that $(\mathtt{P}_2 \cdot \mathtt{P}_3)$ is responsible for matching v when matching w by $\mathtt{P}'$. Next, we determine how v is broken up by the matching against $(\mathtt{P}_2 \cdot \mathtt{P}_3)$. Suppose that $v = w_2w_3$, that $\mathtt{P}_2$ is responsible for matching w_2, and that $\mathtt{P}_3$ is responsible for matching w_3. Then CCon states that w_1w_2 is matched by $(\mathtt{P}_1 \cdot \mathtt{P}_2)$ in $\mathtt{P}$ and w_3 by $\mathtt{P}_3$ in $\mathtt{P}$. Note that by definition, $w_1 \Box w_2w_3 \in M(\lambda, \mathtt{P}', C)$ and $w_2 \Box w_3 \in M(2, \mathtt{P}', C)$. Hence, if we already have $M(\lambda, \mathtt{P}', C)$ and $M(2, \mathtt{P}', C)$, it suffices to "link" these two sets correctly together in order to calculate $M(\lambda, \mathtt{P}, C)$. We therefore define the *redistribution* of two marked languages M_1 and M_2, denoted by $\mathit{redistrib}(M_1, M_2)$, to be the marked language

$$\mathit{redistrib}(M_1, M_2) := \{w_1w_2 \Box w_3 \mid w_1 \Box w_2w_3 \in M_1, w_2 \Box w_3 \in M_2\}.$$

By the reasoning made above, it is intuitively clear that

$$M(\lambda, \mathtt{P}, C) = \mathit{redistrib}(M(\lambda, \mathtt{P}', C), M(2, \mathtt{P}', C)).$$

We will prove this claim formally in the following section. Of course, we need a way to actually calculate the redistribution:

Lemma 7.2. *If M_1 and M_2 are regular, then so is $\mathit{redistrib}(M_1, M_2)$, which can effectively be computed.*

Algorithm 3: Calculate $M(m, \mathtt{P}, C)$.

Input: A pattern $\mathtt{P} = \mathtt{P}_1 \cdot \mathtt{P}_2$; a node $m \in bn(\mathtt{P})$ such that $m = 2^k$ for some $k \geq 0$ and $\mathtt{P}(m) = \cdot$; and a regular context C.

Output: The marked language of node m in $\mathtt{P}$.

```
switch P do
    case P1 · P2 with P1 = ε or P1 = σ
        switch m do
            case λ  return  T^FL(1, P, C) · {□} · T^FL(2, P, C)
            case 2n  return  M(n, P2, L(P1)\C)
        end
    case P1* · P2
        switch m do
            case λ  return  break(C, L(P1*), L(P2))
            case 2n  return  M(n, P2, rbreak(C, L(P1*), L(P2)))
        end
    case (P1 + P2) · P3
        return  M(m, P1 · P3, C) ∪ M(m, P2 · P3, C − L(P1 · P3))

    case (P1 · P2) · P3
        let P' = P1 · (P2 · P3)
        switch m do
            case λ  return  redistrib(M(λ, P', C), M(2, P', C))
            case 2n  return  M(22n, P', C)
        end
end
```

Proof. We introduce two operations on regular languages:

$$\iota(L) = \{w_1 \Box w_2 \Box w_3 \mid w_1 \Box w_2 w_3 \in L\},$$
$$\pi_1(L) = \{w_1 w_2 \Box w_3 \mid w_1 \Box w_2 \Box w_3 \in L\}.$$

It is clear that regular languages are closed under these two operations. For example, we can obtain an automaton for $\iota(L)$ by modifying an automaton for L to allow the reading of a second $\Box$ after the first, which is then ignored. The lemma then follows since $\mathit{redistrib}(M_1, M_2) = \pi_1(\iota(M_1) \cap (\Sigma^* \cdot \{\Box\} \cdot M_2))$. $\Box$

Now we have a way to calculate the marked language $M(\lambda, (\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3, C)$ if we can calculate $M(\lambda, \mathtt{P}_1 \cdot (\mathtt{P}_2 \cdot \mathtt{P}_3), C)$ and $M(2, \mathtt{P}_1 \cdot (\mathtt{P}_2 \cdot \mathtt{P}_3), C)$. Algorithm 3 calculates these marked languages by case analysis on $\mathtt{P}_1 \cdot (\mathtt{P}_2 \cdot \mathtt{P}_3)$, recursively calling itself when necessary. To do so, we only have to be able to calculate $M(m, \mathtt{P}'', C)$ for patterns $\mathtt{P}''$ of the form $\mathtt{P}''_1 \cdot \mathtt{P}''_2$ and nodes $m = 2^k \in bn(\mathtt{P}'')$ with $\mathtt{P}''(m) = \cdot$. That is, Algorithm 3 only needs to recursively call itself on such arguments.

Getting an understanding of this algorithm largely involves the same reasoning as for $\mathcal{T}^{\mathbb{FL}}(n, \mathtt{P}'', C)$. For instance, suppose $\mathtt{P}'' = \mathtt{P}''_1 \cdot \mathtt{P}''_2$ with $\mathtt{P}''_1 = \varepsilon$ or $\mathtt{P}''_1 = \sigma$. If $w \in C$ is matched by $\mathtt{P}''$, then w must be able to be split in words w_1 matched by $\mathtt{P}''_1$ and w_2 matched by $\mathtt{P}''_2$. Since $L(\mathtt{P}''_1)$ contains only one word, there can be no ambiguity in determining w_1 and w_2. Hence, $M(\lambda, \mathtt{P}'', C)$ equals $\mathcal{T}^{\mathbb{FL}}(1, \mathtt{P}'', C) \cdot \{\Box\} \cdot \mathcal{T}^{\mathbb{FL}}(2, \mathtt{P}'', C)$. Likewise, $M(2n, \mathtt{P}'', C)$ equals $M(n, \mathtt{P}''_2, L(\mathtt{P}''_1) \backslash C)$. For the other

cases, similar reasonings can be done and we will therefore not elaborate further on the working of Algorithm 3 here. Its correctness will be formally demonstrated in the next section.

7.2 Proof of Correctness

It is not immediately clear that Algorithms 2 and 3 terminate on every input. We will first prove they do:

PROPOSITION 7.3. *Algorithms 2 and 3 terminate on every input.*

PROOF. A relation $\succ$ on a set A *well-founded* (or terminating) if there is no infinite decreasing sequence $a_1 \succ a_2 \succ a_3 \succ \ldots$ [Baader and Nipkow 1998]. We will define a well-founded binary relation $\sqsupset$ on the set of all patterns. Termination of both algorithms then follows as they only recursively call themselves with "smaller" inputs (according to $\sqsupset$).

We define $\sqsupset$ to relate a pattern $\mathtt{P}$ with its immediate subpatterns if $\mathtt{P} \neq (\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3$ and $\mathtt{P} \neq (\mathtt{P}_1 + \mathtt{P}_2) \cdot \mathtt{P}_3$. It relates $(\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3$ with $\mathtt{P}_2 \cdot \mathtt{P}_3$ and with $\mathtt{P}_1 \cdot (\mathtt{P}_2 \cdot \mathtt{P}_3)$. It relates $(\mathtt{P}_1 + \mathtt{P}_2) \cdot \mathtt{P}_3$ with $\mathtt{P}_1 \cdot \mathtt{P}_3$ and with $\mathtt{P}_2 \cdot \mathtt{P}_3$. The monotone embedding ϕ into the lexicographically ordered $\mathbb{N} \times \mathbb{N}$ where $\phi(\mathtt{P}) = (|\mathtt{P}|, 0)$ if $\mathtt{P} \neq \mathtt{P}_1 \cdot \mathtt{P}_2$ and $\phi(\mathtt{P}_1 \cdot \mathtt{P}_2) = (|\mathtt{P}_1 \cdot \mathtt{P}_2|), |\mathtt{P}_1|)$ otherwise, shows that $\sqsupset$ is well-founded [Baader and Nipkow 1998].

Let $(m, \mathtt{P}, C)$ be a valid input of Algorithm 3. It is clear that Algorithm 3 directly calls itself only on inputs $(m', \mathtt{P}', C')$ with $\mathtt{P} \sqsupset \mathtt{P}'$. If $\mathtt{P} = \mathtt{P}_1 \cdot \mathtt{P}_2$ and $m = \lambda$ with $\mathtt{P}_1 = \varepsilon$ or $\mathtt{P}_1 = \sigma$ then Algorithm 3 calls Algorithm 2 with arguments $(1, \mathtt{P}, C)$ and $(2, \mathtt{P}, C)$. On these arguments, Algorithm 2 will call itself with arguments $(\lambda, \mathtt{P}_1, C/L(\mathtt{P}_2))$ and $(\lambda, \mathtt{P}_2, L(\mathtt{P}_1) \backslash C)$. On these recursive calls, Algorithm 2 terminates in one step. Hence, Algorithm 3 terminates on every input.

Let $(m, \mathtt{P}, C)$ be the input of Algorithm 2. It is clear that Algorithm 2 directly calls itself only on inputs $(m', \mathtt{P}', C')$ where $\mathtt{P} \sqsupset \mathtt{P}'$. If $\mathtt{P} = (\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3$ and $m = 1$, Algorithm 3 is called, which always terminates. Hence, Algorithm 2 terminates on every input. □

We will now formally prove the correctness of Algorithms 2 and 3, thereby also proving Theorem 7.1.

LEMMA 7.4. *If $w \in P \rightsquigarrow V$ then $V(\lambda) = w$, and if $(w_1, w_2) \in P_1 \cdot P_2 \rightsquigarrow (V_1, V_2)$ then $V_1(\lambda) = w_1$ and $V_2(\lambda) = w_2$.*

PROOF. By a straightforward induction on the matching derivation. □

PROPOSITION 7.5. *$\mathcal{T}^{\mathbb{FL}}(\lambda, P, C) = L(P) \cap C$ for any pattern P.*

PROOF. Similar to the proof of Proposition 5.4. □

PROPOSITION 7.6. *For $P = P_1 + P_2$, the following equalities hold:*

(1) $\mathcal{T}^{\mathbb{FL}}(1n, P, C) = \mathcal{T}^{\mathbb{FL}}(n, P_1, C)$
(2) $\mathcal{T}^{\mathbb{FL}}(2n, P, C) = \mathcal{T}^{\mathbb{FL}}(n, P_2, C - L(P_1))$

PROOF. Similar to the proof of Proposition 5.5. □

PROPOSITION 7.7. *If $P = P_1 \cdot P_2$ with $P_1 = \varepsilon$ or $P_1 = \sigma$, then the following equalities hold:*

(1) $\mathcal{T}^{\mathbb{FL}}(1, P, C) = \mathcal{T}^{\mathbb{FL}}(\lambda, P_1, C/L(P_2))$

(2) $\mathcal{T}^{\mathbb{FL}}(2n, P, C) = \mathcal{T}^{\mathbb{FL}}(n, P_2, (L(P_1)\backslash C))$

(3) $M(\lambda, P, C) = \mathcal{T}^{\mathbb{FL}}(1, P, C) \cdot \{\Box\} \cdot \mathcal{T}^{\mathbb{FL}}(2, P, C)$

(4) $M(2n, P, C) = M(n, P_2, L(P_1)\backslash C)$

Proof. We prove the case where $\mathtt{P}_1 = \sigma$, the case where $\mathtt{P}_1 = \varepsilon$ is similar. Note that if $\mathtt{P}_1 = \sigma$ the top of any matching derivation of $w' \in \mathtt{P}{\rightsquigarrow}V$ has the following form:

$$\dfrac{\dfrac{\dfrac{\cdots}{w_1 \in \sigma{\rightsquigarrow}V_1} \qquad \dfrac{\cdots}{w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2}}{(w_1, w_2) \in \mathtt{P}{\rightsquigarrow}(V_1, V_2)}\ \textsc{CLab}}{w' = w_1 w_2 \in \mathtt{P}{\rightsquigarrow}V = V_1 \cdot V_2}\ \textsc{Concat}$$

Equality (1) then readily follows by Theorem 6.4:

$$\begin{aligned}
w \in \mathcal{T}^{\mathbb{FL}}(1, \mathtt{P}, C) &\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}V \wedge V(1) = w \\
&\Leftrightarrow \exists w_1, w_2 : w_1 w_2 \in C \wedge w_1 \in \sigma{\rightsquigarrow}V_1 \wedge w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2 \wedge V_1(\lambda) = w \\
&\Leftrightarrow \exists w_1, w_2 : w_1 w_2 \in C \wedge w_1 \in \sigma{\rightsquigarrow}V_1 \wedge w_2 \in L(\mathtt{P}_2) \wedge V_1(\lambda) = w \\
&\Leftrightarrow \exists w_1 \in C/L(\mathtt{P}_2) : w_1 \in \sigma{\rightsquigarrow}V_1 \wedge V_1(\lambda) = w \\
&\Leftrightarrow w \in \mathcal{T}^{\mathbb{FL}}(\lambda, \sigma, C/L(\mathtt{P}_2))
\end{aligned}$$

Equalities (2) and (4) can be proven similarly. Equality (3) readily follows by Theorem 6.4 and equalities (1) and (2):

$$\begin{aligned}
v_1 \Box v_2 \in M(\lambda, \mathtt{P}, C) &\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}V \wedge V(1) = v_1 \wedge V(2) = w_2 \\
&\Leftrightarrow \exists w_1, w_2 : w_1 w_2 \in C \wedge w_1 \in \sigma{\rightsquigarrow}V_1 \wedge w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2 \\
&\qquad \wedge V_1(\lambda) = v_1 \wedge V_2(\lambda) = v_2 \\
&\Leftrightarrow \exists w_1, w_2 : w_1 w_2 \in C \wedge w_1 \in L(\sigma) \wedge w_1 \in \sigma{\rightsquigarrow}V_1 \\
&\qquad \wedge w_2 \in L(\mathtt{P}_2) \wedge w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2 \wedge V_1(\lambda) = v_1 \wedge V_2(\lambda) = v_2 \\
&\Leftrightarrow \exists w_1 \in C/L(\mathtt{P}_2), w_2 \in L(\sigma)\backslash C : w_1 \in \sigma{\rightsquigarrow}V_1 \wedge w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2 \\
&\qquad \wedge V_1(\lambda) = v_1 \wedge V_2(\lambda) = v_2 \\
&\Leftrightarrow v_1 \Box v_2 \in \mathcal{T}^{\mathbb{FL}}(\lambda, \sigma, C/L(\mathtt{P}_2)) \cdot \{\Box\} \cdot \mathcal{T}^{\mathbb{FL}}(\lambda, \mathtt{P}_2, L(\sigma)\backslash C) \\
&\Leftrightarrow v_1 \Box v_2 \in \mathcal{T}^{\mathbb{FL}}(1, \mathtt{P}, C) \cdot \{\Box\} \cdot \mathcal{T}^{\mathbb{FL}}(2, \mathtt{P}, C)
\end{aligned}$$

□

Proposition 7.8. *If $P = P_1^* \cdot P_2$, then the following equalities hold:*

(1) $\mathcal{T}^{\mathbb{FL}}(1, P, C) = \mathit{lbreak}(C, L(P_1^*), L(P_2))$

(2) $\mathcal{T}^{\mathbb{FL}}(2n, P, C) = \mathcal{T}^{\mathbb{FL}}(n, P_2, \mathit{rbreak}(C, L(P_1^*), L(P_2)))$

(3) $M(\lambda, P, C) = \mathit{break}(C, L(P_1^*), L(P_2))$

(4) $M(2n, P, C) = M(n, P_2, \mathit{rbreak}(C, L(P_1^*), L(P_2)))$

Proof. Note that for any derivation of $w' \in \mathtt{P}{\rightsquigarrow}V$, the top must look like:

$$\dfrac{\dfrac{\dfrac{\cdots}{w_1 \in \mathtt{P}_1^*{\rightsquigarrow}V_1} \qquad \dfrac{\cdots}{w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2} \quad \neg(\exists w_3 \neq \lambda, w_4 : w_3w_4 = w_2 \wedge w_1w_3 \in L(\mathtt{P}_1^*) \wedge w_4 \in L(\mathtt{P}_2))}{(w_1, w_2) \in \mathtt{P}{\rightsquigarrow}(V_1, V_2)}\ \textsc{CKLeene}}{w' = w_1w_2 \in \mathtt{P}{\rightsquigarrow}V = V_1 \cdot V_2}\ \textsc{Concat}$$

Also note that $V_1(\lambda) = w_1$ and $V_2(\lambda) = w_2$ by Lemma 7.4. From these observations and Theorem 6.4 equality (1) readily follows:

$$\begin{aligned}
& w \in \mathcal{T}^{\mathbb{FL}}(1, \mathtt{P}, C) \\
& \quad \Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}V \wedge V(1) = w \\
& \quad \Leftrightarrow \exists w_1, w_2 : w_1w_2 \in C \wedge w_1 \in \mathtt{P}_1^*{\rightsquigarrow}V_1 \wedge w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2 \wedge V_1(\lambda) = w \\
& \qquad\quad \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3w_4 = w_2 \wedge w_1w_3 \in L(\mathtt{P}_1^*) \wedge w_4 \in L(\mathtt{P}_2)) \\
& \quad \Leftrightarrow \exists w_2 : ww_2 \in C \wedge w \in \mathtt{P}_1^*{\rightsquigarrow}V_1 \wedge w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2 \\
& \qquad\quad \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3w_4 = w_2 \wedge ww_3 \in L(\mathtt{P}_1^*) \wedge w_4 \in L(\mathtt{P}_2)) \\
& \quad \Leftrightarrow \exists w_2 : ww_2 \in C \wedge w \in L(\mathtt{P}_1^*) \wedge w_2 \in L(\mathtt{P}_2) \\
& \qquad\quad \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3w_4 = w_2 \wedge ww_3 \in L(\mathtt{P}_1^*) \wedge w_4 \in L(\mathtt{P}_2)) \\
& \quad \Leftrightarrow w \in \mathit{lbreak}(C, L(\mathtt{P}_1^*), L(\mathtt{P}_2))
\end{aligned}$$

Equality (3) can be obtained by a similar reasoning:

$$\begin{aligned}
& v_1 \Box v_2 \in M(\lambda, \mathtt{P}, C) \\
& \quad \Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}V \wedge V(1) = v_1 \wedge V(2) = v_2 \\
& \quad \Leftrightarrow \exists w_1, w_2 : w_1w_2 \in C \wedge w_1 \in \mathtt{P}_1^*{\rightsquigarrow}V_1 \wedge w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2 \\
& \qquad\quad \wedge V_1(\lambda) = v_1 \wedge V_2(\lambda) = v_2 \\
& \qquad\quad \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3w_4 = w_2 \wedge w_1w_3 \in L(\mathtt{P}_1^*) \wedge w_4 \in L(\mathtt{P}_2)) \\
& \quad \Leftrightarrow v_1v_2 \in C \wedge v_1 \in \mathtt{P}_1^*{\rightsquigarrow}V_1 \wedge v_2 \in \mathtt{P}_2{\rightsquigarrow}V_2 \\
& \qquad\quad \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3w_4 = v_2 \wedge v_1w_3 \in L(\mathtt{P}_1^*) \wedge w_4 \in L(\mathtt{P}_2)) \\
& \quad \Leftrightarrow v_1v_2 \in C \wedge v_1 \in L(\mathtt{P}_1^*) \wedge v_2 \in L(\mathtt{P}_2^*) \\
& \qquad\quad \wedge \neg(\exists w_3 \neq \lambda, w_4 : w_3w_4 = v_2 \wedge v_1w_3 \in L(\mathtt{P}_1^*) \wedge w_4 \in L(\mathtt{P}_2)) \\
& \quad \Leftrightarrow v_1 \Box v_2 \in \mathit{break}(C, L(\mathtt{P}_1^*), L(\mathtt{P}_2))
\end{aligned}$$

Equalities (2) and (4) can be proven similarly. □

Proposition 7.9. *If $P = (P_1 + P_2) \cdot P_3$, $P_1' = P_1 \cdot P_3$, and $P_2' = P_2 \cdot P_3$, then the following equalities hold:*

(1) $\mathcal{T}^{\mathbb{FL}}(1, P, C) = \mathcal{T}^{\mathbb{FL}}(1, P_1', C) \cup \mathcal{T}^{\mathbb{FL}}(1, P_2', C - L(P_1'))$
(2) $\mathcal{T}^{\mathbb{FL}}(11n, P, C) = \mathcal{T}^{\mathbb{FL}}(1n, P_1', C)$
(3) $\mathcal{T}^{\mathbb{FL}}(12n, P, C) = \mathcal{T}^{\mathbb{FL}}(1n, P_2', C - L(P_1'))$
(4) $\mathcal{T}^{\mathbb{FL}}(2n, P, C) = \mathcal{T}^{\mathbb{FL}}(2n, P_1', C) \cup \mathcal{T}^{\mathbb{FL}}(2n, P_2', C - L(P_1'))$
(5) $M(n, P, C) = M(n, P_1', C) \cup M(n, P_2', C - L(P_1'))$ *if* $n = 2^k$ *for some* $k \geq 0$

Proof. Note that for any derivation of $w' \in \mathtt{P}{\rightsquigarrow}V$, the top is either of the form

$$\dfrac{\dfrac{\dfrac{\cdots}{(w_1,w_2) \in \mathtt{P}_1 \cdot \mathtt{P}_3{\rightsquigarrow}(V_1,V_3)}}{(w_1,w_2) \in (\mathtt{P}_1+\mathtt{P}_2)\cdot \mathtt{P}_3{\rightsquigarrow}(V_1+\mathtt{P}_2,V_3)}\ \textsc{COr1}}{w' = w_1 \cdot w_2 \in \mathtt{P}{\rightsquigarrow}V = (V_1+\mathtt{P}_2)\cdot V_3}\ \textsc{Concat}$$

or of the form

$$\dfrac{\dfrac{\dfrac{\cdots}{(w_1,w_2) \in \mathtt{P}_2 \cdot \mathtt{P}_3{\rightsquigarrow}(V_2,V_3)} \qquad w_1 \cdot w_2 \notin L(\mathtt{P}_1 \cdot \mathtt{P}_3)}{(w_1,w_2) \in (\mathtt{P}_1+\mathtt{P}_2)\cdot \mathtt{P}_3{\rightsquigarrow}(\mathtt{P}_1+V_2,V_3)}\ \textsc{COr2}}{w' = w_1 \cdot w_2 \in \mathtt{P}{\rightsquigarrow}V = (\mathtt{P}_1+V_2)\cdot V_3}\ \textsc{Concat}$$

It is easily seen that hence $w' \in \mathtt{P}{\rightsquigarrow}(V_1+\mathtt{P}_2)\cdot V_3$ iff $w' \in \mathtt{P}'_1{\rightsquigarrow}V_1 \cdot V_3$ and that $w' \in \mathtt{P}{\rightsquigarrow}(\mathtt{P}_1+V_2)\cdot V_3$ iff $w' \in \mathtt{P}'_2{\rightsquigarrow}V_2 \cdot V_3$ and $w' \notin L(\mathtt{P}'_1)$. From these observations, equality (1) readily follows:

$$\begin{aligned}
w \in \mathcal{T}^{\mathbb{FL}}(1,\mathtt{P},C) &\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}V \wedge V(1) = w \\
&\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}(V_1+\mathtt{P}_2)\cdot V_3 \wedge V_1(\lambda) = w \\
&\quad \text{or } w' \in \mathtt{P}{\rightsquigarrow}(\mathtt{P}_1+V_2)\cdot V_3 \wedge V_2(\lambda) = w \\
&\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}'_1{\rightsquigarrow}V_1 \cdot V_3 \wedge V_1(\lambda) = w \\
&\quad \text{or } w' \in \mathtt{P}'_2{\rightsquigarrow}V_2 \cdot V_3 \wedge V_2(\lambda) = w \wedge w' \notin L(\mathtt{P}'_1) \\
&\Leftrightarrow w \in \mathcal{T}^{\mathbb{FL}}(1,\mathtt{P}'_1,C) \text{ or } w \in \mathcal{T}^{\mathbb{FL}}(2,\mathtt{P}',C - L(\mathtt{P}'_1))
\end{aligned}$$

Equalities (4) and (5) can be proven similarly. Note that, if $w' \in \mathtt{P}{\rightsquigarrow}V$ and $V(11n) \neq \bot$, then the matching derivation must be of the first form. Hence:

$$\begin{aligned}
w \in \mathcal{T}^{\mathbb{FL}}(1,\mathtt{P},C) &\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}V \wedge V(11n) = w \\
&\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}{\rightsquigarrow}(V_1+\mathtt{P}_2)\cdot V_3 \wedge V_1(n) = w \\
&\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}'_1{\rightsquigarrow}V_1 \cdot V_3 \wedge V_1(\lambda) = w \\
&\Leftrightarrow w \in \mathcal{T}^{\mathbb{FL}}(1,\mathtt{P}'_1,C)
\end{aligned}$$

Equality (3) can be proven similarly. □

Lemma 7.10. *If* $(w_1,w_2) \in \mathtt{P}_1 \cdot \mathtt{P}_2{\rightsquigarrow}(V_1,V_2)$ *then* $w_2 \in \mathtt{P}_2{\rightsquigarrow}V_2$

Proof. The proof goes by induction on the matching derivation $(w_1,w_2) \in \mathtt{P}_1 \cdot \mathtt{P}_2{\rightsquigarrow}(V_1,V_2)$ with a case analysis on the last rule used. In all the cases, the result either follows immediately from the premise of the last rule used, or follows immediately from the induction hypothesis. □

Proposition 7.11. *If* $\mathtt{P} = (\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3$ *and* $\mathtt{P}' = \mathtt{P}_1 \cdot (\mathtt{P}_2 \cdot \mathtt{P}_3)$*, then the following equalities hold:*

(1) $\mathcal{T}^{\mathbb{FL}}(1,\mathtt{P},C) = M(\lambda,\mathtt{P},C)\backslash(\{\Box\}\cdot\Sigma^*)$
(2) $\mathcal{T}^{\mathbb{FL}}(11n,\mathtt{P},C) = \mathcal{T}^{\mathbb{FL}}(1n,\mathtt{P}',C)$
(3) $\mathcal{T}^{\mathbb{FL}}(12n,\mathtt{P},C) = \mathcal{T}^{\mathbb{FL}}(21n,\mathtt{P}',C)$
(4) $\mathcal{T}^{\mathbb{FL}}(2n,\mathtt{P},C) = \mathcal{T}^{\mathbb{FL}}(22n,\mathtt{P}',C)$
(5) $M(\lambda,\mathtt{P},C) = \mathit{redistrib}(M(\lambda,\mathtt{P}',C), M(2,\mathtt{P}',C))$

(6) $M(2n, \mathsf{P}, C) = M(22n, \mathsf{P}', C)$

Proof. We start by stating the following properties of the matching derivations of P and P':

(A) If $w' \in \mathsf{P} \rightsquigarrow V$ then $V = (V_1 \cdot V_2) \cdot V_3$ and $V_3(\lambda) \neq \bot$.

(B) If $w' \in \mathsf{P}' \rightsquigarrow V'$ then $V' = V_1 \cdot (V_2 \cdot V_3)$.

(C) $w' \in \mathsf{P} \rightsquigarrow (V_1 \cdot V_2) \cdot V_3$ iff $w' \in \mathsf{P}' \rightsquigarrow V_1 \cdot (V_2 \cdot V_3)$.

Property (A) holds because the top of every matching derivation of $w' \in \mathsf{P} \rightsquigarrow V$ must look like:

$$\dfrac{\dfrac{\dfrac{\cdots}{(w_1, w_2w_3) \in \mathsf{P}_1 \cdot (\mathsf{P}_2 \cdot \mathsf{P}_3) \rightsquigarrow (V_1, W)} \qquad \dfrac{\cdots}{(w_2, w_3) \in \mathsf{P}_2 \cdot \mathsf{P}_3 \rightsquigarrow (V_2, V_3)}}{(w_1w_2, w_3) \in (\mathsf{P}_1 \cdot \mathsf{P}_2) \cdot \mathsf{P}_3 \rightsquigarrow (V_1 \cdot V_2, V_3)}\ \textsc{CCon}}{w' = w_1w_2w_3 \in (\mathsf{P}_1 \cdot \mathsf{P}_2) \cdot \mathsf{P}_3 \rightsquigarrow V = (V_1 \cdot V_2) \cdot V_3}\ \textsc{Concat}$$

By application of Lemma 7.10 on $(w_2, w_3) \in \mathsf{P}_2 \cdot \mathsf{P}_3 \rightsquigarrow (V_2, V_3)$ we have $w_3 \in \mathsf{P}_3 \rightsquigarrow V_3$. Then $V_3(\lambda) = w_3 \neq \bot$ by Lemma 7.4.

Property (B) holds because the top of every matching derivation of $w' \in \mathsf{P}' \rightsquigarrow V'$ must look like:

$$\dfrac{\dfrac{}{(w_1, w_2w_3) \in \mathsf{P}_1 \cdot (\mathsf{P}_2 \cdot \mathsf{P}_3) \rightsquigarrow (V_1, W)}}{w' = w_1w_2w_3 \in \mathsf{P}_1 \cdot (\mathsf{P}_2 \cdot \mathsf{P}_3) \rightsquigarrow V' = V_1 \cdot W}\ \textsc{Concat}$$

By application of Lemma 7.10 on $(w_1, w_2w_3) \in \mathsf{P}_1 \cdot (\mathsf{P}_2 \cdot \mathsf{P}_3) \rightsquigarrow (V_1, W)$ we have $w_2 \cdot w_3 \in \mathsf{P}_2 \cdot \mathsf{P}_3 \rightsquigarrow W$. This derivation must end with an application of rule Concat, so there must be a derivation of $(w_2, w_3) \in \mathsf{P}_2 \cdot \mathsf{P}_3 \rightsquigarrow (V_2, V_3)$ for some V_2, V_3 with $W = V_2 \cdot V_3$. Hence, V' is of the form $V_1 \cdot (V_2 \cdot V_3)$.

To prove property (C), suppose that $w' \in \mathsf{P} \rightsquigarrow V$. We then have $(w_1, w_2w_3) \in \mathsf{P}_1 \cdot (\mathsf{P}_2 \cdot \mathsf{P}_3) \rightsquigarrow (V_1, W)$ and $(w_2, w_3) \in \mathsf{P}_2 \cdot \mathsf{P}_3 \rightsquigarrow (V_2, V_3)$. Hence $w_2w_3 \in \mathsf{P}_2 \cdot \mathsf{P}_3 \rightsquigarrow W$ by application of Lemma 7.10. Furthermore, $w_2w_3 \in \mathsf{P}_2 \cdot \mathsf{P}_3 \rightsquigarrow V_2 \cdot V_3$ by application of rule Concat on $(w_2, w_3) \in \mathsf{P}_2 \cdot \mathsf{P}_3 \rightsquigarrow (V_2, V_3)$. Hence, $W = V_2 \cdot V_3$ by Theorem 6.4. Finally, $w' \in \mathsf{P}' \rightsquigarrow V_1 \cdot (V_2 \cdot V_3)$ by application of rule Concat on $(w_1, w_2w_3) \in \mathsf{P}_1 \cdot (\mathsf{P}_2 \cdot \mathsf{P}_3) \rightsquigarrow (V_1, V_2 \cdot V_3)$. Conversely, suppose that $w' \in \mathsf{P}' \rightsquigarrow V'$. By a reasoning similar to the one used to prove property (B) we obtain that $(w_1, w_2w_3) \in \mathsf{P}_1 \cdot (\mathsf{P}_2 \cdot \mathsf{P}_3) \rightsquigarrow (V_1, V_2 \cdot V_3)$ and $w_2w_3 \in \mathsf{P}_2 \cdot \mathsf{P}_3 \rightsquigarrow V_2 \cdot V_3$. By application of rule CCon on these subderivations we obtain $(w_1w_2, w_3) \in \mathsf{P} \rightsquigarrow (V_1 \cdot V_2, V_3)$. Finally, $w' \in \mathsf{P} \rightsquigarrow (V_1 \cdot V_2) \cdot V_3$ by application of rule Concat.

From property (A) equality (1) readily follows:

$$\begin{aligned} w \in \mathcal{T}^{\mathbb{FL}}(1, \mathsf{P}, C) &\Leftrightarrow \exists w' \in C : w' \in \mathsf{P} \rightsquigarrow V \wedge V(1) = w \\ &\Leftrightarrow \exists v \exists w' \in C : w' \in \mathsf{P} \rightsquigarrow V \wedge V(1) = w \wedge V(2) = v \\ &\Leftrightarrow \exists v : w \Box v \in M(\lambda, \mathsf{P}, C) \\ &\Leftrightarrow w \in M(\lambda, \mathsf{P}, C)/(\{\Box\} \cdot \Sigma^*) \end{aligned}$$

From all three properties equality (2) readily follows:

$$\begin{aligned} w \in \mathcal{T}^{\mathbb{FL}}(11n, \mathtt{P}, C) &\Leftrightarrow \exists w' \in C : w' \in \mathtt{P} \rightsquigarrow V \wedge V(11n) = w \\ &\Leftrightarrow \exists w' \in C : w' \in \mathtt{P} \rightsquigarrow (V_1 \cdot V_2) \cdot V_3 \wedge V_1(n) = w \\ &\Leftrightarrow \exists w' \in C : w' \in \mathtt{P}' \rightsquigarrow V_1 \cdot (V_2 \cdot V_3) \wedge V_1(n) = w \\ &\Leftrightarrow w \in \mathcal{T}^{\mathbb{FL}}(1n, \mathtt{P}', C) \end{aligned}$$

Equalities (3), (4), and (6) can be proven similarly. Let us abbreviate $M(\lambda, \mathtt{P}', C)$ by M_1 and $M(2, \mathtt{P}', C)$ by M_2. To prove equality (5) we observe:

$$\begin{aligned} & v \Box w_3 \in M(\lambda, \mathtt{P}, C) \\ & \quad \Leftrightarrow \exists w' \in C : w' \in \mathtt{P} \rightsquigarrow V \wedge V(1) = v \wedge V(2) = w_3 \\ & \quad \Leftrightarrow \exists w' \in C : w' \in \mathtt{P} \rightsquigarrow (V_1 \cdot V_2) \cdot V_3 \wedge (V_1 \cdot V_2)(\lambda) = v \wedge V_3(\lambda) = w_3 \\ & \quad \Leftrightarrow \exists w_1, w_2 \exists w' \in C : w' \in \mathtt{P} \rightsquigarrow (V_1 \cdot V_2) \cdot V_3 \wedge w_1 w_2 = v \\ & \qquad \wedge V_1(\lambda) = w_1 \wedge V_2(\lambda) = w_2 \wedge V_3(\lambda) = w_3 \\ & \quad \Leftrightarrow \exists w_1, w_2 \exists w' \in C : w' \in \mathtt{P}' \rightsquigarrow V_1 \cdot (V_2 \cdot V_3) \wedge w_1 w_2 = v \\ & \qquad \wedge V_1(\lambda) = w_1 \wedge V_2(\lambda) = w_2 \wedge V_3(\lambda) = w_3 \end{aligned}$$

We claim that the latter holds iff

$$\exists w_1, w_2 : w_1 w_2 = v \wedge w_1 \Box w_2 w_3 \in M_1 \wedge w_2 \Box w_3 \in M_2,$$

i.e., iff $v \Box w_3 \in \mathit{redistrib}(M_1, M_2)$. The "only if" direction is obvious. To prove the "if" direction, let us assume $w_1 \Box w_2 w_3 \in M_1$ and $w_2 \Box w_3 \in M_2$. By definition of M_1 and by property (B) there exists some $w' \in C$ with $w' \in \mathtt{P}' \rightsquigarrow V_1 \cdot (V_2 \cdot V_3)$ such that $V_1(\lambda) = w_1$ and $(V_2 \cdot V_3)(\lambda) = w_2 w_3$. Then, by Lemma 7.4:

$$w' = (V_1 \cdot (V_2 \cdot V_3))(\lambda) = V_1(\lambda) \cdot (V_2 \cdot V_3)(\lambda) = w_1 w_2 w_3.$$

Furthermore, since the derivation of $w' \in \mathtt{P}' \rightsquigarrow V_1 \cdot (V_2 \cdot V_3)$ must end with an application of rule Concat, we have $(w_1, w_2 w_3) \in \mathtt{P}' \rightsquigarrow (V_1, V_2 \cdot V_3)$. Hence, $w_2 w_3 \in \mathtt{P}_2 \cdot \mathtt{P}_3 \rightsquigarrow V_2 \cdot V_3$ by Lemma 7.10. Since $w_2 \Box w_3 \in M_2$ we have by definition of M_2 and property (B) that there must exist some $w'' \in C$ with $w'' \in \mathtt{P}' \rightsquigarrow V_1' \cdot (V_2' \cdot V_3')$, $V_2'(\lambda) = w_2$, and $V_3'(\lambda) = w_3$. Since the derivation of $w'' \in \mathtt{P}' \rightsquigarrow V_1' \cdot (V_2' \cdot V_3')$ must end with an application of rule Concat, we have $(w_1'', w_2'' w_3'') \in \mathtt{P}' \rightsquigarrow (V_1', V_2' \cdot V_3')$ for $w_1'' w_2'' w_3'' = w''$. Hence $w_2'' w_3'' \in \mathtt{P}_2 \cdot \mathtt{P}_3 \rightsquigarrow V_2' \cdot V_3'$ by Lemma 7.10. Furthermore, by Lemma 7.4:

$$w_2'' w_3'' = (V_2' \cdot V_3')(\lambda) = V_2'(\lambda) \cdot V_3'(\lambda) = w_2 w_3.$$

Since we now have $w_2 w_3 \in \mathtt{P}_2 \cdot \mathtt{P}_3 \rightsquigarrow V_2 \cdot V_3$ and $w_2 w_3 \in \mathtt{P}_2 \cdot \mathtt{P}_3 \rightsquigarrow V_2' \cdot V_3'$, we obtain $V_2 \cdot V_3 = V_2' \cdot V_3'$ by Theorem 6.4. Hence we have $w' \in \mathtt{P}' \rightsquigarrow V_1 \cdot (V_2 \cdot V_3)$ with $V_1(\lambda) = w_1$, $V_2(\lambda) = w_2$, and $V_3(\lambda) = w_3$. □

8. REGULAR HEDGE EXPRESSION PATTERNS

The true power of regular expression pattern matching comes into play when we introduce regular hedge expression patterns matching hedges. A hedge is a sequence of trees; hedges form the basic data model of XML [Murata 1999; Vianu 2001]. In this section we formally define hedges, regular hedge languages, and regular hedge expression patterns.

A *hedge* over Σ is a sequence $\sigma_1[h_1]\ldots\sigma_n[h_n]$ where $n \geq 0$, $\sigma_1, \ldots, \sigma_n$ are symbols in Σ, and $h_1, \ldots, h_n$ are already hedges. The hedge with $n = 0$ is called the *empty hedge* and will be denoted by λ. Hedges with $n = 1$ are called *trees*. Hedges over Σ will be denoted by h, g, and their subscripted versions. Note that if h and g are hedges, then so is hg, the concatenation of h and g. Hedges of the form $\sigma[\lambda]$ will sometimes be abbreviated by σ.

Note that we cannot exactly model the trees in Figures 1(a) and 1(b) unless we put the actual data values ("Data On The Web", "Abiteboul", etc.) in the alphabet Σ. Putting all possible data values in our alphabet however, would result in an infinite alphabet. We will therefore abstract away from actual data values in pattern matching and assume Σ to contain a special element `data`, for which we will replace all data values. The tree of Figure 1(a) then corresponds to

```
book[
  title[data],
  author[data], author[data], author[data],
  price[data]
]
```

An actual programming language would provide features to retrieve the content of `data` nodes.

Just as regular word languages are defined as those languages that can be recognized by a finite word automaton, regular hedge languages are those languages that can be recognized by a *finite hedge automaton* [Brüggemann-Klein et al. 2001; Neven 2002]. A finite hedge automaton H over Σ is a tuple (Q, δ, F) where Q is a finite set of *states*; F is a regular language over Q; and δ is the *transition relation*: a possibly infinite set of triples (q, σ, w) with $q \in Q$ and w a word over Q, such for any q and σ the set $\{w \mid (q, \sigma, w) \in \delta\}$ is regular. We will denote this latter set by $\delta(q, \sigma)$. Since regular word languages are finitely representable by finite automata or regular expressions, the transition relation is also finitely representable. We associate a function δ^* with δ as follows: $\delta^*(\lambda) = \{\lambda\}$ and if $h = \sigma_1[h_1]\cdots\sigma_n[h_n]$ then

$$\delta^*(h) = \{q_1 \cdots q_n \mid \delta(q_1, \sigma_1) \cap \delta^*(h_1) \neq \emptyset, \ldots, \delta(q_n, \sigma_n) \cap \delta^*(h_n) \neq \emptyset\}.$$

A hedge h is accepted by a hedge automaton H if $\delta^*(h) \cap F \neq \emptyset$. The *language* $L(H)$ recognized by a hedge automaton H is the set of all hedges it accepts. A hedge language is *regular* if there exists some hedge automaton recognizing it. If $F \subseteq Q$ then H can only accept trees, and H is called a *finite tree automaton*. Its language is called a regular tree language. A hedge automaton is called *total* if $\delta^*(h) \neq \emptyset$ for all hedges h. Intuitively, a hedge automaton is total if it never gets "stuck" on any input. We can always make a hedge automaton total by adding a "garbage" state.

We will introduce regular hedge expression patterns next. While XDuce uses recursive patterns that allow the binding of nodes to subhedges which are arbitrarily deep in the input hedge, we will follow ℂDuce in the sense that we only allow to bind subhedges up to a certain depth. This will make the formalization considerably simpler. We still want our patterns to be able to recognize all regular hedge languages however, which can contain arbitrarily deep hedges. We therefore

assume to be given a fixed set $\mathcal{N}$ of *names*, together with an *environment* Δ. The set of names is assumed to be disjoint from Σ and does not contain the special symbols $\perp$ and $\Box$. The environment is a total function which relates every name $N \in \mathcal{N}$ with a regular *tree* language $\Delta(N)$. We will denote members of $\mathcal{N}$ by N, M and their subscripted versions.

A *regular expression hedge pattern* $\mathtt{P}$ is an expression of the form ε, N, $\sigma[\mathtt{P}_1]$, $\mathtt{P}_1 + \mathtt{P}_2$, $\mathtt{P}_1 \cdot \mathtt{P}_2$, or $\mathtt{P}_1^*$ where $\mathtt{P}_1$ and $\mathtt{P}_2$ are already hedge patterns. The hedge language $L(\mathtt{P})$ of a hedge pattern $\mathtt{P}$ is defined as follows:

$$\begin{array}{lcl} L(\varepsilon) & = & \{\lambda\} \\ L(\sigma[\mathtt{P}]) & = & \{\sigma[h] \mid h \in L(\mathtt{P})\} \\ L(N) & = & \Delta(N) \\ L(\mathtt{P}_1 + \mathtt{P}_2) & = & L(\mathtt{P}_1) \cup L(\mathtt{P}_2) \\ L(\mathtt{P}_1 \cdot \mathtt{P}_2) & = & L(\mathtt{P}_1) \cdot L(\mathtt{P}_2) \\ L(\mathtt{P}^*) & = & L(\mathtt{P})^* \end{array}$$

It is easy to see that $L(\mathtt{P})$ is always regular. As in Section 3, we identify $\mathtt{P}$ with the partial function $\mathtt{P} : \{1,2\}^* \to \{\varepsilon, +, \cdot, *\} \cup \Sigma \cup \mathcal{N}$ such that:

—if $\mathtt{P} = \varepsilon$ then $dom(\mathtt{P}) = \{\lambda\}$ and $\mathtt{P}(\lambda) = \varepsilon$;

—if $\mathtt{P} = N$ with $N \in \mathcal{N}$ then $dom(\mathtt{P}) = \{\lambda\}$ and $\mathtt{P}(\lambda) = N$;

—if $\mathtt{P} = \sigma[\mathtt{P}_1]$ with $\sigma \in \Sigma$ then $dom(\mathtt{P}) = \{\lambda\} \cup \{1n \mid n \in dom(\mathtt{P}_1)\}$ with $\mathtt{P}(\lambda) = \sigma$ and $\mathtt{P}(1n) = \mathtt{P}_1(n)$;

—if $\mathtt{P} = \mathtt{P}_1^*$ then we make a similar definition, only $\mathtt{P}(\lambda) = *$;

—if $\mathtt{P} = \mathtt{P}_1 + \mathtt{P}_2$ then $dom(\mathtt{P}) = \{\lambda\} \cup \{1n \mid n \in dom(\mathtt{P}_1)\} \cup \{2n \mid n \in dom(\mathtt{P}_2)\}$ with $\mathtt{P}(\lambda) = +$, $\mathtt{P}(1n) = \mathtt{P}_1(n)$, and $\mathtt{P}(2n) = \mathtt{P}_2(n)$; and

—if $\mathtt{P} = \mathtt{P}_1 \cdot \mathtt{P}_2$ we make a similar definition, only $\mathtt{P}(\lambda) = \cdot$.

Precedence of operators is the same as in Section 3. As before, the set of bindable nodes $bn(\mathtt{P})$ of a hedge pattern $\mathtt{P}$ are those nodes in its domain which do not have an ancestor node labeled with $*$.

We will use $\sigma[V]$ to denote the association function with domain $\{\lambda\} \cup \{1n \mid n \in dom(V)\}$ such that $(\sigma[V])(\lambda) = \sigma[V(\lambda)]$ and $(\sigma[V])(1n) = V(1n)$.

In XDuce two kinds of patterns were introduced: *external* patterns which largely correspond to the hedge patterns introduced above and *internal* patterns to which the external patterns are translated. The internal patterns are used to define the matching relation and to do type inference. These patterns can only recognize *ranked* trees, which are trees in which each label has a fixed number of children. It is therefore necessary to encode the *unranked* input trees (where a label can have an arbitrary number of children) into ranked trees before matching. We have chosen to work directly with the external, unranked, representation of patterns in this paper because our insights gained for regular string expression patterns can be directly extended to regular hedge expression patterns without using such internal patterns (as we will show in the following sections). This hugely simplifies the correctness proof of our type inference algorithm for hedges, as it largely follows from that of the algorithms given earlier.

$$\textsc{Empty}\quad \frac{}{\lambda \in \varepsilon \rightsquigarrow [\lambda \to \lambda]}$$

$$\textsc{Lab}\quad \frac{h \in \mathtt{P} \rightsquigarrow V}{\sigma[h] \in \sigma[\mathtt{P}] \rightsquigarrow \sigma[V]}$$

$$\textsc{Name}\quad \frac{\sigma[h] \in \Delta(N)}{\sigma[h] \in N \rightsquigarrow [\lambda \to \sigma[h]]}$$

$$\textsc{Kleene}\quad \frac{h \in L(\mathtt{P}^*)}{h \in \mathtt{P}^* \rightsquigarrow [\lambda \to h]}$$

$$\textsc{Or1}\quad \frac{h \in \mathtt{P}_1 \rightsquigarrow V}{h \in \mathtt{P}_1 + \mathtt{P}_2 \rightsquigarrow V + \mathtt{P}_2}$$

$$\textsc{Or2}\quad \frac{h \in \mathtt{P}_2 \rightsquigarrow V \qquad h \notin L(\mathtt{P}_1)}{h \in \mathtt{P}_1 + \mathtt{P}_2 \rightsquigarrow \mathtt{P}_1 + V}$$

$$\textsc{Concat}\quad \frac{(h_1, h_2) \in \mathtt{P}_1 \cdot \mathtt{P}_2 \rightsquigarrow (V_1, V_2)}{h_1 h_2 \in \mathtt{P}_1 \cdot \mathtt{P}_2 \rightsquigarrow V_1 \cdot V_2}$$

$$\textsc{CEmpty}\quad \frac{h \in \mathtt{P} \rightsquigarrow V_2}{(\lambda, h) \in \varepsilon \cdot \mathtt{P} \rightsquigarrow ([\lambda \to \lambda], V_2)}$$

$$\textsc{CLab}\quad \frac{\sigma[h_1] \in \sigma[\mathtt{P}_1] \rightsquigarrow V_1 \qquad h_2 \in \mathtt{P}_2 \rightsquigarrow V_2}{(\sigma[h_1], h_2) \in \sigma[\mathtt{P}_1] \cdot \mathtt{P}_2 \rightsquigarrow (V_1, V_2)}$$

$$\textsc{CName}\quad \frac{\sigma[h_1] \in N \rightsquigarrow V_1 \qquad h_2 \in \mathtt{P}_2 \rightsquigarrow V_2}{(\sigma[h_1], h_2) \in N \cdot \mathtt{P}_2 \rightsquigarrow (V_1, V_2)}$$

$$\textsc{COr1}\quad \frac{(h_1, h_2) \in \mathtt{P}_1 \cdot \mathtt{P}_3 \rightsquigarrow (V_1, V_2)}{(h_1, h_2) \in (\mathtt{P}_1 + \mathtt{P}_2) \cdot \mathtt{P}_3 \rightsquigarrow (V_1 + \mathtt{P}_2, V_2)}$$

$$\textsc{COr2}\quad \frac{\begin{array}{c}(h_1, h_2) \in \mathtt{P}_2 \cdot \mathtt{P}_3 \rightsquigarrow (V_1, V_2)\\ h_1 h_2 \notin L(\mathtt{P}_1 \cdot \mathtt{P}_3)\end{array}}{(h_1, h_2) \in (\mathtt{P}_1 + \mathtt{P}_2) \cdot \mathtt{P}_3 \rightsquigarrow (\mathtt{P}_1 + V_1, V_2)}$$

$$\textsc{CCon}\quad \frac{\begin{array}{c}(h_1, h_2 h_3) \in \mathtt{P}_1 \cdot (\mathtt{P}_2 \cdot \mathtt{P}_3) \rightsquigarrow (V_1, W)\\ (h_2, h_3) \in \mathtt{P}_2 \cdot \mathtt{P}_3 \rightsquigarrow (V_2, V_3)\end{array}}{(h_1 h_2, h_3) \in (\mathtt{P}_1 \cdot \mathtt{P}_2) \cdot \mathtt{P}_3 \rightsquigarrow (V_1 \cdot V_2, V_3)}$$

$$\textsc{CKleene}\quad \frac{\begin{array}{c}h_1 \in \mathtt{P}_1^* \rightsquigarrow V_1 \qquad h_2 \in \mathtt{P}_2 \rightsquigarrow V_2\\ \neg(\exists h_3 \neq \lambda, h_4 : h_2 = h_3 h_4 \wedge h_1 h_3 \in L(\mathtt{P}_1^*) \wedge h_4 \in L(\mathtt{P}_2))\end{array}}{(h_1, h_2) \in \mathtt{P}_1^* \cdot \mathtt{P}_2 \rightsquigarrow (V_1, V_2)}$$

Fig. 5. The matching relation $h \in \mathtt{P} \rightsquigarrow V$ for hedges under the first and longest match disambiguation policy.

9. HEDGE MATCHING UNDER THE FIRST AND LONGEST MATCH POLICY

In this section we lift the matching process under the first and longest match policy to hedges. Its associated type inference problem will be solved in the following section. We can lift the matching process and type inference algorithm for the POSIX policy in a similar way.

The matching relation for hedge regular expressions under the first and longest match policy is defined in Figure 5. Most of the rules are simple extensions to hedges of the rules in Figure 4. For example, rule Lab now allows us to match hedge $\sigma[h]$ against pattern $\sigma[\mathtt{P}]$ if h can be matched against $\mathtt{P}$. Note that if we view a word $\sigma_1 \ldots \sigma_n$ as a hedge $\sigma_1[\lambda] \ldots \sigma_n[\lambda]$, we get exactly the semantics of Figure 4. There are only two rules not occurring in the word case: Name and CName. Rule Name states that a tree is matched by a name N if the tree belongs to the associated tree language $\Delta(N)$. Rule CName is similar, but is used in concatenations.

The following theorem is the equivalent of Theorem 6.4:

Theorem 9.1. *The matching relation of Figure 5 is well defined:*

(1) *The matching relation is semantically correct:* $h \in \mathtt{P} \rightsquigarrow V$ *iff* $h \in L(\mathtt{P})$*, and,*
(2) *The matching relation is unique: if* $h \in \mathtt{P} \rightsquigarrow V$ *and* $h \in \mathtt{P} \rightsquigarrow W$ *then* $V = W$*.*

Proof. Completely analogous to the proof of Theorem 6.4. □

10. TYPE INFERENCE FOR HEDGES UNDER THE FIRST AND LONGEST MATCH POLICY

In this section we lift the type inference algorithm of Section 7 to the hedge setting. Concretely, we will show:

THEOREM 10.1. *If P is a hedge pattern and C is a regular hedge language then $\mathcal{T}^{\mathbb{FL}}(n, P, C)$ is also regular, and can be effectively computed.*

10.1 The algorithm

We obtain the type inference for hedges by a modification of Algorithm 2. Algorithm 2 uses quotient, breaking, and redistribution on word languages. The corresponding operations on hedge languages are defined in the obvious way. For example, the left quotient of hedge language L by hedge language K, denoted as $K \backslash L$, is the set $\{s \mid \exists p \in K : ps \in L\}$.

The main observations we made for the word setting can be transfered in a straightforward manner to the hedge setting. For example, $\mathcal{T}^{\mathbb{FL}}(\lambda, \mathtt{P}, C) = L(\mathtt{P}) \cap C$ for any $\mathtt{P}$. Likewise, if $\mathtt{P} = \mathtt{P}_1 + \mathtt{P}_2$ then $\mathcal{T}^{\mathbb{FL}}(1n, \mathtt{P}, C) = \mathcal{T}^{\mathbb{FL}}(n, \mathtt{P}_1, C)$ and $\mathcal{T}^{\mathbb{FL}}(2n, \mathtt{P}, C) = \mathcal{T}^{\mathbb{FL}}(n, \mathtt{P}_2, C - L(\mathtt{P}_1))$. The case where $\mathtt{P} = \mathtt{P}_1 \cdot \mathtt{P}_2$ with $\mathtt{P}_1 = \varepsilon$, $\mathtt{P}_1 = N$ or $\mathtt{P}_1 = \sigma[\mathtt{P}']$ can also be deduced using a reasoning similar to the word setting: $\mathcal{T}^{\mathbb{FL}}(1n, \mathtt{P}, C) = \mathcal{T}^{\mathbb{FL}}(n, \mathtt{P}_1, C/L(\mathtt{P}_2))$ and $\mathcal{T}^{\mathbb{FL}}(2n, \mathtt{P}, C) = \mathcal{T}^{\mathbb{FL}}(n, \mathtt{P}_2, L(\mathtt{P}_1) \backslash C)$. The other cases can also be lifted to the hedge setting.

The only case when we cannot fall back on our insights of the word setting is when we need to calculate the type of $1n$ in $\mathtt{P} = \sigma[\mathtt{P}_1]$. Intuitively, a hedge can only be associated to a subpattern of $\mathtt{P}_1$ in $\mathtt{P} = \sigma[\mathtt{P}_1]$, if it is a subhedge of a hedge h matched by $\mathtt{P}_1$ such that $\sigma[h] \in C$. Hence, if we define the *cut* of a hedge language L by a symbol σ, denoted by $cut(L, \sigma)$, as $\{h \mid \sigma[h] \in L\}$ then $\mathcal{T}^{\mathbb{FL}}(1n, \mathtt{P}, C)$ equals $\mathcal{T}^{\mathbb{FL}}(n, \mathtt{P}_1, cut(C, \sigma))$. Of course, we need to be able to calculate cuts:

LEMMA 10.2. *If L is a regular hedge language, then so is $cut(L, \sigma)$.*

PROOF. Since L is a regular hedge language, there exists a finite hedge automaton $H = (Q, \delta, F)$ such that $L(H) = L$. Let $S = F \cap Q$. Intuitively, S contains those states in F the automaton can be in after processing a tree. We then define $F' = \bigcup_{q \in S} \delta(q, \sigma)$ and $H' = (Q, \delta, F')$. It is easy to see that $\sigma[h] \in L(H)$ iff $h \in H'$. Hence $L(H') = cut(L, \sigma)$. □

The type inference algorithm for hedges is then obtained from Algorithm 2 by lifting all operations to the hedge setting, and adding the case for $m = 1n$ and $\mathtt{P} = \sigma[\mathtt{P}_1]$, as shown in Algorithm 4. The dots indicate the cases which are similar to the word setting.

10.2 Proof of correctness

Before we talk about the correctness of Algorithm 4, we need to show that the operations used are still computable for regular hedge languages.

It is well-known that hedge languages are closed under union, intersection and negation [Brüggemann-Klein et al. 2001]. It is also well-known that finite hedge languages are regular, as is the set of all hedges. Regular hedge languages are also closed under left and right quotient. Although the proof is straightforward, it has

Algorithm 4: Calculate $\mathcal{T}^{\mathbb{FL}}(m, \mathtt{P}, C)$ for hedges.

Input: A hedge pattern $\mathtt{P}$; a node $m \in bn(\mathtt{P})$; and a regular hedge context C.

Output: The type of m in $\mathtt{P}$ relative to C under the first and longest match disambiguation policy.

```
if m = λ then
    return L(P) ∩ C
else
    switch P do
        ...
        case σ[P1]
            let n be such that m = 1n
            return  T^FL(n, P1, cut(C, σ))
        case P1 · P2 with P1 = ε, P1 = σ[P1'], or P1 = N
            switch m do
                case 1n  return  T^FL(n, P1, C/L(P2))
                case 2n  return  T^FL(n, P2, L(P1)\C)
            end
    end
end
```

not yet been explicitly given in the literature. For the sake of completeness we therefore provide it in Appendix A.

If L is a regular hedge language, then so is the language

$$\pi^{-1}(L) := \{h_1 \Box h_2 \Box \cdots \Box h_n \mid h_1 h_2 \ldots h_n \in L\}.$$

Indeed, we can add the transition $(q, \Box, \lambda)$ to an automaton $H = (Q, \delta, F)$ for L, where q is a new state. It then suffices to allow the reading of q at arbitrary places in F (modify a DFA for F to allow reading the letter q, which is then ignored). The closure of regular hedge languages under breakings then follows from Lemma 5.2.

Closure under redistribution follows by the following lemma:

Lemma 10.3. *If M_1 and M_2 are regular marked hedge languages, then so is redistrib(M_1, M_2), which can effectively be computed.*

Proof. We introduce two operations on hedge languages:

$$\iota(L) = \{h_1 \Box h_2 \Box h_3 \mid h_1 \Box h_2 h_3 \in L\},$$
$$\pi_1(L) = \{h_1 h_2 \Box h_3 \mid h_1 \Box h_2 \Box h_3 \in L\}.$$

We will show that if M is a marked hedge language then $\iota(M)$ is a regular hedge language and if N is a regular hedge language containing only hedges of the form $h_1 \Box h_2 \Box h_3$ then $\pi_1(N)$ is a regular marked hedge language. The lemma then follows since:

$$redistrib(M_1, M_2) = \pi_1(\iota(M_1) \cap (\mathcal{H}(\Sigma) \cdot \{\Box\} \cdot M_2).$$

Let M be a marked regular hedge language and $H = (Q, \delta, F)$ a hedge automaton recognizing M. Since M is a marked hedge language, every hedge in M is of the form $h_1 \Box h_2$ where the symbol $\Box$ does not occur in h_1 or h_2. Then every word in

F must be of the form w_1qw_2 with $\delta(q,\Box) = \{\lambda\}$ and $\delta(q,\sigma) = \emptyset$ for all σ. If this would be not the case, we can find a hedge h in M not containing $\Box$. We may then assume w.l.o.g. that there is exactly one such q (if there are more, we can group them all together in one new state). Let us define the language

$$\iota^q(F) := \{w_1qw_2qw_3 \mid w_1qw_2w_3 \in F\}.$$

This is clearly a regular language: modify an automaton for F such that an extra q can be read after the first one. Define $R = (Q, \delta, \iota^q(F))$. It is easy to see that $h_1\Box h_2h_3 \in L(H)$ iff $h_1\Box h_2\Box h_3 \in L(R)$. Hence $L(R) = \iota(M)$.

Let N be regular hedge language containing only hedges of the form $h_1\Box h_2\Box h_3$ and let $H = (Q, \delta, F)$ be a hedge automaton recognizing N. Then every word in F must be of the form $w_1qw_2q'w_3$ with $\delta(q,\Box) = \delta(q',\Box) = \{\lambda\}$ and $\delta(q,\sigma) = \delta(q',\sigma) = \emptyset$ for all σ. If this would be not the case, we can find a hedge h in M not containing $\Box$ or containing only one $\Box$. We may assume that there is only one such q and q' (if there are more we can group them all together in a new state). Then define

$$\pi_1^q(F) := \{w_1w_2qw_3 \mid w_1qw_2q'w_3 \in F\}.$$

This is clearly a regular language: modify an automaton for F to forget q'. Define $R = (Q, \delta, \pi_1^q(F))$. It is easy to see that $h_1\Box h_2\Box h_3 \in L(H)$ iff $h_1h_2\Box h_3 \in L(R)$. Hence $L(R) = \pi_1(M)$. □

The correctness of Algorithm 4 then follows from the fact that the propositions in Section 10.2 remain valid for the hedge setting, and the following two propositions:

Proposition 10.4. *If $\mathsf{P} = \mathsf{P}_1 \cdot \mathsf{P}_2$, then the following equalities hold:*

(1) $\mathcal{T}^{\mathbb{FL}}(1n, \mathsf{P}, C) = \mathcal{T}^{\mathbb{FL}}(n, \mathsf{P}_1, C/L(\mathsf{P}_2))$
(2) $\mathcal{T}^{\mathbb{FL}}(2n, \mathsf{P}, C) = \mathcal{T}^{\mathbb{FL}}(n, \mathsf{P}_2, (L(\mathsf{P}_1)\backslash C))$
(3) $M(\lambda, \mathsf{P}, C) = \mathcal{T}^{\mathbb{FL}}(1, \mathsf{P}, C) \cdot \{\Box\} \cdot \mathcal{T}^{\mathbb{FL}}(2, \mathsf{P}, C)$
(4) $M(2n, \mathsf{P}, C) = M(n, \mathsf{P}_2, L(\mathsf{P}_1)\backslash C)$

Proof. Similar to that of Proposition 7.7. □

Proposition 10.5. *If $\mathsf{P} = \sigma[\mathsf{P}_1]$, then $\mathcal{T}^{\mathbb{FL}}(1n, \mathsf{P}, C) = \mathcal{T}^{\mathbb{FL}}(n, \mathsf{P}_1, cut(C, \sigma))$*

Proof. Every matching derivation $h' \in \mathsf{P}{\rightsquigarrow}V$ must be of the form

$$\frac{\dfrac{\cdots}{h_1 \in \mathsf{P}_1{\rightsquigarrow}V_1}}{h' = \sigma[h_1] \in \mathsf{P}{\rightsquigarrow}V = \sigma[V_1]}\ \textsc{Lab}$$

The proposition readily follows:

$$\begin{aligned} h \in \mathcal{T}^{\mathbb{FL}}(1n, \mathsf{P}, C) &\Leftrightarrow \exists h' \in C : h' \in \mathsf{P}{\rightsquigarrow}V \wedge V(1n) = h \\ &\Leftrightarrow \exists h_1 : \sigma[h_1] \in C \wedge h_1 \in \mathsf{P}_1{\rightsquigarrow}V_1 \wedge V_1(n) = h \\ &\Leftrightarrow \exists h_1 \in cut(C, \sigma) : h_1 \in \mathsf{P}_1{\rightsquigarrow}V_1 \wedge V_1(n) = h \\ &\Leftrightarrow h \in \mathcal{T}^{\mathbb{FL}}(n, \mathsf{P}_1, cut(C, \sigma)) \end{aligned}$$

□

11. DISCUSSION AND FUTURE WORK

In this paper we have focussed on the longest match semantics in the POSIX and first and longest match disambiguation policies. One could also consider a *shortest* match disambiguation rule and even a mixture of longest and shortest match. Indeed, for the POSIX policy we could enrich the patterns with a *shortest match concatenation operator*, denoted by $\cdot^?$. The pattern $\mathtt{P}_1$ in $\mathtt{P}_1 \cdot^? \mathtt{P}_2$ then matches as little of the input as possible, still allowing the rest of the pattern to match. Likewise, for the first and longest match policy we could enrich the patterns with a *shortest match Kleene star operator*, denoted by $*^?$. The matching relation and type inference algorithm presented here can be extended in a straightforward manner to include these operators.

We note that in languages such as `sed` and `awk`, regular expression patterns are not required to match all of their input. They just have to start matching as early as possible in the input string, and can stop as soon as a match is found. Using the shortest-match concatenation operator introduced above, we can simulate this behavior for the POSIX disambiguation policy by transforming $\mathtt{P}$ into $\Sigma^* \cdot^? \mathtt{P} \cdot \Sigma^*$.

Whereas we restrict the bindable nodes of a pattern to those nodes not occurring in a Kleene closure, ℂDuce defines all nodes bindable, allowing patterns like:

```
match $v with
    (($a as author[_]) | _)* => result[$a]
```

Here, every subhedge matched by `author[_]` is concatenated to the value of `$a`. The XDuce policy continues to be used inside the Kleene closure to disambiguate if necessary. It is not immediately clear how our type inference techniques can be adapted to this setting. POSIX also define all nodes bindable, where variables inside the Kleene closure get bound to the last value matched. Again, it is not immediately clear how to adapt our type inference algorithm to this setting.

In this paper, we have focused on gaining fundamental insights into the type inference problem for unique pattern matching, and have not concerned ourselves with the practical implementation of our algorithms. We have also not considered the associated time and space requirements. To our knowledge, there has not yet been a formal investigation of the inherent time complexity bounds of the regular type inference problem. These bounds may depend on the way regular languages are represented (i.e. as finite automata, as regular expressions, or yet other formalisms). We note that any type inference algorithm using non-deterministic finite automata to represent regular sets must have has at least an exponential worst case running time. Indeed, $\mathcal{T}^{\mathbb{P}}(2, \mathtt{P} + \Sigma^*, \Sigma^*) = \mathcal{T}^{\mathbb{FL}}(2, \mathtt{P} + \Sigma^*, \Sigma^*) = \Sigma^* - L(\mathtt{P})$, the complement of $L(\mathtt{P})$. It is well-known that complementation of regular languages using non-deterministic automata can cause an exponential blow-up [Hopcroft and Ullman 1979].

As such, although in principle any finite (hedge) automaton library can be used to implement the type inference algorithms of this paper, it would be worthwhile to investigate which algorithms lend themselves to an acceptable performance in practice. A starting point here can be the work on MONA [Klarlund and Møller 2001; Elgaard et al. 1998], XDuce [Hosoya 2000; Hosoya et al. 2005], and ℂDuce [Frisch et al. 2002].

12. ACKNOWLEDGMENTS

I thank the anonymous referees, Jan Van den Bussche, Dirk Leinders, Wim Martens, and Frank Neven for inspiring discussions and for their constructive comments on a draft version of this paper.

REFERENCES

Abiteboul, S., Quass, D., McHugh, J., Widom, J., and Wiener, J. L. 1997. The Lorel query language for semistructured data. *International Journal on Digital Libraries 1,* 1, 68–88.

Baader, F. and Nipkow, T. 1998. *Term Rewriting and All That.* Cambridge University Press. Section 2.3.

Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J. 2005. *XQuery 1.0: An XML Query Language.* W3C Working Draft.

Book, R., Even, S., Greibach, S., and Ott, G. 1971. Ambiguity in graphs and expressions. *IEEE Transactions on Computers 20,* 2, 149–153.

Brüggemann-Klein, A., Murata, M., and Wood, D. 2001. Regular tree and regular hedge languages over unranked alphabets. Unpublished manuscript, version 1.

Buneman, P., Fernandez, M. F., and Suciu, D. 2000. UnQL: a query language and algebra for semistructured data based on structural recursion. *VLDB Journal: Very Large Data Bases 9,* 1, 76–110.

Clark, J. and Makoto, M. 2001. *RELAX NG Specification.* Organization for the Advancement of Structured Information Standards.

Davidson, A., Fuchs, M., Hedin, M., Jain, M., Koistinen, J., Lloyd, C., Maloney, M., and Schwarzhof, K. 1999. Schema for object-oriented XML 2.0. Tech. rep., Veo Systems Inc.

Dougherty, D. and Robbins, A. 1996. *Sed and Awk.* O'Reilly.

Elgaard, J., Klarlund, N., and Møller, A. 1998. Mona 1.x: new techniques for WS1S and WS2S. In *Computer Aided Verification, CAV '98, Proceedings.* LNCS, vol. 1427. Springer Verlag.

Frisch, A. 2004. Regular tree language recognition with static information. In *Exploring New Frontiers of Theoretical Informatics, IFIP 18th World Computer Congress, TCS 3rd International Conference on Theoretical Computer Science.* Kluwer, 661–674.

Frisch, A. and Cardelli, L. 2004. Greedy regular expression matching. In *Automata, Languages and Programming: ICALP 2004. Proceedings.* Lecture Notes in Computer Science, vol. 3142. 618–629.

Frisch, A., Castagna, G., and Benzaken, V. 2002. Semantic subtyping. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science.* IEEE Computer Society Press, 137–146.

Frisch, A., Castagna, G., and Benzaken, V. 2003. ℂDuce: an XML-centric general-purpose language. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming.* ACM Press, 51–63.

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

Hosoya, H. 2000. Regular expression types for XML. Ph.D. thesis, University of Tokyo.

Hosoya, H. 2003. Regular expression pattern matching - a simpler design. Tech. Rep. 1397, RIMS, Kyoto University.

Hosoya, H. and Pierce, B. C. 2002. Regular expression pattern matching for XML. *Journal of Functional Programming 13,* 6, 961–1004.

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

Hosoya, H., Vouillon, J., and Pierce, B. C. 2005. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems 27,* 1, 46–90.

Institute of Electrical and Electronic Engineers. 1992. Portable operating system interface (POSIX). IEEE Std 1003.2.

Klarlund, N. and Møller, A. 2001. *MONA Version 1.4 User Manual*. Basic Research In Computer Science (BRICS) Notes Series NS-01-1, Department of Computer Science, University of Aarhus.

Laurikari, V. 2000. NFAs with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *Symposium on String Processing and Information Retrieval (SPIRE)*.

Laurikari, V. 2001. Efficient submatch addressing for regular expressions. M.S. thesis, Helsinki University of Technology.

Levin, M. Y. 2003. Compiling regular patterns. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*. ACM Press, 65–77.

Møller, A. 2003. Document structure description 2.0. Tech. rep., Basic Research In Computer Science (BRICS), Department of Computer Science, University of Aarhus.

Murata, M. 1999. Hedge automata: a formal model for XML schemata. Available at `http://www.geocities.com/murata_makoto`.

Murata, M. 2001. Extended path expressions for XML. In *Proceedings of the twentieth ACM symposium on Principles of database systems*. ACM Press, 126–137.

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

Neumann, A. and Seidl, H. 1998. Locating matches of tree patterns in forests. In *Foundations of Software Technology and Theoretical Computer Science*. LNCS 1530. 134–145.

Neven, F. 2002. Automata theory for XML researchers. *ACM SIGMOD Record 31,* 3, 39–46.

Neven, F. and Schwentick, T. 2001. Automata- and logic-based pattern languages for tree-structured data. In *Semantics in Databases*. Vol. 2582. LNCS, Springer, 160–178.

Sterling, L. and Shapiro, E. 1994. *The Art of Prolog (second edition)*. MIT Press.

Suciu, D. 2002. The XML typechecking problem. *ACM SIGMOD Record 31,* 1, 89–96.

Sumii, E. May 2003. Personal Communication.

Tabuchi, N., Sumii, E., and Yonezawa, A. 2002. Regular expression types for strings in a text processing language (extended abstract). In *Workshop on Types in Programming (TIP'02)*. `http://web.yl.is.s.u-tokyo.ac.jp/~tabee/xperl/`.

Thompson, H. S., Beech, D., Maloney, M., and Mendelsohn, N. 2001. *XML Schema*. W3C Recommendation.

Ullman, J. D. 1998. *Elements of ML Programming*, Second ed. Prentice Hall.

Vianu, V. 2001. A web odyssey: from Codd to XML. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM Press, 1–15.

Wall, L., Christiansen, T., and Orwant, J. 2000. *Programming Perl*, 3rd ed. O'Reilly & Associates.

Yergeau, F., Bray, T., Paoli, J., Sperberg-McQueen, C. M., and Maler, E. 2004. *Extensible Markup Language (XML) 1.0 (Third Edition)*. W3C Recommendation.

A. CLOSURE OF REGULAR HEGDE LANGUAGES UNDER LEFT AND RIGHT QUOTIENTS

To our knowledge, the closure of regular hedge languages under left and right quotients has not yet been explicitly proven in the literature. We therefore prove it here.

Lemma A.1. *Regular hedge languages are closed under left and right quotient.*

Proof. Assume that L and K are regular hedge languages. By definition, there exist hedge automata $H = (Q_H, \delta_H, F_H)$ and $G = (Q_G, \delta_G, F_G)$ such that $L(H) = L$ and $L(G) = K$. We assume without loss of generality that H and G are total.

We will now show how to construct a hedge automaton that recognizes L/K. Let W_1 be a regular word language over Q_H and let W_2 be a regular word language over

Q_G. We define the *simultaneous product* $W_1 \times W_2$ of W_1 and W_2 as the language over $Q_H \times Q_G$ such that

$$W_1 \times W_2 = \{(q_1, s_1) \cdots (q_n, s_n) \mid q_1 \cdots q_n \in W_1, s_1 \cdots s_n \in W_2\}.$$

To prove the regularity of $W_1 \times W_2$, let us define the following two word languages:

$$\pi_1^{-1}(W_1) = \{(q_1, s_1) \cdots (q_n, s_n) \mid q_1 \cdots q_n \in W_1, s_j \in Q_G\},$$
$$\pi_2^{-1}(W_2) = \{(q_1, s_1) \cdots (q_n, s_n) \mid s_1 \cdots s_n \in W_2, q_i \in Q_H\}.$$

It is clear that $\pi_1^{-1}(W_1)$ and $\pi_2^{-1}(W_2)$ are regular word languages over $Q_H \times Q_G$ (we can modify an automaton for W_1 or W_2 to allow reading symbols in $Q_H \times Q_G$, taking into account the original symbol to be read). Then $W_1 \times W_2$ is also regular since $W_1 \times W_2 = \pi_1^{-1}(W_1) \cap \pi_2^{-1}(W_2)$.

Let $R = (Q, \delta, F)$ be the hedge automaton with

—$Q = Q_H \times Q_G$,
—$\delta((q, s), \sigma) = \delta_H(q, \sigma) \times \delta_F(q, \sigma)$,
—$F = \pi_1^{-1}(F_H)/(\pi_2^{-1}(F_G) \cap P^*)$.

Here, $P = \{(q, s) \mid q \in \delta_H^*(t), s \in \delta_G^*(t), t \text{ a tree}\}$, the set of reachable states of the tree automaton (Q, δ, Q), which can be computed by a standard reachability algorithm.

We will now show that $L(R) = L/K$. Suppose $h_1 \in L(R)$. Then $\delta^*(h_1) \cap F \neq \emptyset$ and we can take $(q_1, s_1) \cdots (q_k, s_k) \in \delta^*(h_1) \cap F$. By definition of F there exists a word $(q_{k+1}, s_{k+1}) \cdots (q_n, s_n) \in \pi_2^{-1}(F_G) \cap P^*$ such that

$$(q_1, s_1) \cdots (q_k, s_k)(q_{k+1}, s_{k+1}) \cdots (q_n, s_n) \in \pi_1^{-1}(F_H).$$

Hence, $q_1 \cdots q_n \in F_H$, $s_{k+1} \cdots s_n \in F_G$, and $(q_{k+1}, s_{k+1}) \cdots (q_n, s_n) \in P^*$. By definition of P, there hence exists a hedge h_2 such that $q_{k+1} \cdots q_n \in \delta_H^*(h_2)$ and $s_{k+1} \cdots s_n \in \delta_G^*(h_2)$. Then $h_2 \in L(G) = K$. Moreover, $h_1 h_2 \in L(H) = L$. Hence, $h_1 \in L/K$, and thus $L/K \subseteq L(R)$. Conversely, let $h_1 = \sigma_1[h'_1] \cdots \sigma_k[h'_k]$ be a hedge for which there exists some $h_2 = \sigma_{k+1}[h'_{k+1}] \cdots \sigma_n[h'_n] \in K$ such that $h_1 h_2 \in L$. Then $\delta_H^*(h_1 h_2) \cap F_H \neq \emptyset$ and $\delta_G^*(h_2) \cap F_G \neq \emptyset$. Hence we can choose $q_1, \cdots q_n \in \delta_H^*(h_1 h_2) \cap F_H$ and $s_{k+1} \cdots s_n \in \delta_G^*(h_2) \cap F_G$. Since G is total there is at least one string $s_1 \cdots s_k \in \delta_G^*(h_1)$. Hence, $s_1 \cdots s_n \in \delta_G^*(h_1 h_2)$ and $(q_1, s_1) \cdots (q_n, s_n) \in \delta^*(h_1 h_2)$. Since $(q_{k+1}, s_{k+1}) \cdots (q_n, s_n) \in \pi_2^{-1}(F_G)$ and since $(q_1, s_1) \cdots (q_n, s_n) \in \pi_1^{-1}(F_H)$ we have $(q_1, s_1) \cdots (q_k, s_k) \in \pi_1^{-1}(F_H)/(\pi_2^{-1}(F_G) \cap P^*)$. Hence, h_1 is accepted by R, and thus $L/K \subseteq L(R)$.

An automaton for $K \backslash L$ can be constructed in a similar way. □