Typed and Unambiguous
Pattern Matching on Strings using Regular Expressions

Claus Brabrand
IT University of Copenhagen, Denmark
brabrand@itu.dk

Jakob G. Thomsen
Aarhus University, Denmark
jakob.g.thomsen@cs.au.dk

Abstract

We show how to achieve typed and unambiguous declarative pattern matching on strings using regular expressions extended with a simple recording operator.

We give a characterization of ambiguity of regular expressions that leads to a sound and complete static analysis. The analysis is capable of pinpointing all ambiguities in terms of the structure of the regular expression and report shortest ambiguous strings. We also show how pattern matching can be integrated into statically typed programming languages for deconstructing strings and reproducing typed and structured values.

We validate our approach by giving a full implementation of the approach presented in this paper. The resulting tool, `reg-exp-rec`, adds typed and unambiguous pattern matching to Java in a standalone and non-intrusive manner. We evaluate the approach using several realistic examples.

Categories and Subject Descriptors D.3.4 [*Processors*]: Parsing; F.4.2 [*Grammars and Other Rewriting Systems*]: Decision Problems, Parsing

General Terms Languages, Theory

Keywords Regular Expressions, Pattern Matching, Ambiguity, Disambiguation, Static Analysis, Type Inference, Parsing

1. Introduction

Syntactic analysis is an important and indispensable part of many applications that deal with dynamic data. Often, applications need to process data which is supplied at runtime according to rigorously structured data formats, but encoded as flat strings. The structure is only implicitly specified through various conventions and implicitly disambiguated through subtle combinations of spaces, delimiter characters, and sometimes, when we are lucky, balanced parentheses. Examples range from official standards such as URLs all the way to "home made" log files.

URLs, for instance, which have to be analyzed by Browsers and other Web-sensitive tools, pack lots of information into a string; e.g., information about a protocol (e.g., `http` or `https`), a symbolic server name or a numeric ip address, an optional port address, an optional username-and-password combination, and a query string whose arguments are, in turn, further structured as key-value pairs.

The process of analyzing such data is often referred to as *pattern matching* (on strings). It is essentially all about deconstructing implicitly structured strings into logical units of information based on conventions and/or specifications. It is important not to confuse this with pattern matching of (already) structured values for which there are lots of approaches, tools, and languages (e.g., TOM [1] and ML [32]).

There are many choices for specifying and/or programming such pattern matching; and many tools are available. Here, it is instructive to introduce the Chomsky Hierachy [8] dating back to 1956:

Chomsky Hierarchy	Type-3	Type-2	Type-1	Type-0
Language Classes	*Regular Languages*	*Context-Free Languages*	*Context-Sensitive Languages*	*Recursively Enumerable Languages*
Formalisms	***Regular Expressions***	*Context-Free Grammars*	*Context-Sensitive Grammars*	*Turing-Complete Programming*

In this paper we will look at the issue of pattern matching for *regular expressions* and contrast this to other formalisms and approaches. We will also look at a popular category which does not fit into the Chomsky Hierarchy; namely that of regular expressions with *capturing groups* and so-called *back-references*. This is the pattern matching mechanism found in `java.util.regex`, Perl, PHP, Python, Ruby, etc. We will use `java.util.regex` as a representative example from that category, and the Java programming language as representative example of Type-0 formalisms. (We will not look closely at Type-1 as it restricts expressivity, without contributing much in terms of safety.)

As we will show, *programming* pattern matching operationally in a Type-0 formalism is an error-prone and not always simple task. We will argue that for pattern matching, Type-0 Turing-Complete or Type-2 Context-Free expressivity is often not required; and, that it is possible to trade this excess expressivity for declarativity, simplicity, and static safety. Type-3 regular expressions are often enough and have nice closure and decidability properties.

1.1 Contributions

In this paper, we make the following contributions:

- a *syntax-directed characterization* of the ambiguity problem for regular expressions that leads to:
  - a *sound-and-complete* analysis of ambiguity capable of pinpointing ambiguities in terms of the *structure* of the regular expression and report shortest ambiguous strings; *and*

PPDP'10, July 26–28, 2010, Hagenberg, Austria.

- a concept of *local disambiguators* in the form of six locally disambiguated regular expression operators;

- we show *how* pattern matching can be integrated into a statically typed programming language (Java) for deconstructing strings and reproducing typed and structured values in a *stand-alone* and *non-intrusive* way;
- a *validation* and *evaluation* of the effectiveness of pattern matching on strings using regular expressions on realistic examples via a full implementation of everything presented (in the form of the tool, `reg-exp-rec`).

1.2 Outline

In Section 2, we introduce regular expressions along with a simple declarative recording construction for pattern matching. In Section 3, we show how to statically analyze ambiguity of regular expressions and how this analysis leads to disambiguation directives. (Note that ambiguity is a *structural* problem of *how* a language is defined, not a *linguistic* problem of *what* language is defined; i.e., along the lower row in the above table.) In Section 4, we show how to perform type inference on regular expressions with recordings to determine statically the type of all recordings (i.e., which strings they can match at runtime). Furthermore, we show how this type information can be reconciled with the static type systems of modern programming languages (e.g., Java). All this has been implemented as a stand-alone tool, `reg-exp-rec`, which essentially adds statically typed and unambiguous pattern matching to Java in a non-intrusive manner. Section 5 shows several realistic usage examples. Section 6 discusses parsing. Section 7 and 8 contain the evaluation and related work, respectively. Finally, Section 9 concludes.

2. Regular Expressions

Given a finite alphabet of symbols, Σ, we define the syntax of *regular expressions* by:

$$R \quad : \quad \emptyset \;\mid\; \varepsilon \;\mid\; c \;\mid\; R\,|\,R \;\mid\; R\cdot R \;\mid\; R*$$

where $\emptyset$ denotes the *empty language*, ε is the language containing only the *empty string* $\{\epsilon\}$, $c \in \Sigma$ is the *single character language* $\{c\}$, $R\,|\,R$ is the *union* of the two languages involved (aka., *choice*), $R \cdot R$ is the *concatenation* of the two languages involved, and $R*$ denotes zero or more self concatenations (aka., *iteration* or *Kleene star*) of the language R. (Syntactically, the infix concatenation operator, "·", is often omitted when writing a regular expression.) We denote by $\mathcal{R}$ the set of all regular expressions.

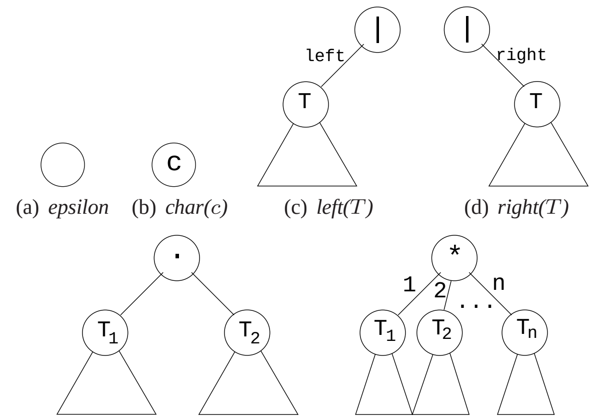
The semantics of regular expressions can then be captured by $\mathcal{L} : \mathcal{R} \to 2^{\Sigma^*}$ which defines the *language* of a regular expression, inductively:

$$\begin{array}{rcl}
\mathcal{L}(\emptyset) &=& \emptyset \\
\mathcal{L}(\varepsilon) &=& \{\, \epsilon \,\} \\
\mathcal{L}(c) &=& \{\, c \,\} \\
\mathcal{L}(R_1\,|\,R_2) &=& \mathcal{L}(R_1) \cup \mathcal{L}(R_2) \\
\mathcal{L}(R_1 \cdot R_2) &=& \mathcal{L}(R_1) \cdot \mathcal{L}(R_2) \\
\mathcal{L}(R*) &=& \mathcal{L}(R)^*
\end{array}$$

where $\cdot$ is language concatenation as in $L_1 \cdot L_2 = \{\, \omega_1\omega_2 \mid \omega_1 \in L_1,\ \omega_2 \in L_2\}$; and where L^* is the *Kleene star* operator on languages (reflexive transitive closure of self-concatenation) as in $L^* = \bigcup_{i\geq 0} L^i$ where $L^0 = \{\epsilon\}$ and $L^i = L \cdot L^{i-1}$, for $i > 0$.

We denote by $Reg_\Sigma = \{\, \mathcal{L}(R) \mid R \in \mathcal{R} \,\}$ the set of all *regular languages*. (Note that this set is closed under [25]; e.g., *union*, *concatenation*, *iteration*, *intersection*, *complement*, *restriction*, *homomorphisms*, *reversal*, *prefixing*, *suffixing*, etc.)

With these basic operators it is easy to construct the usual extensions: *any character*, '`.`' (aka., Σ, as $c_1\,|\,c_2\,|\cdots|\,c_{|\Sigma|}$); *character ranges*, `[a-z]` (as `a|b|c|`$\cdots$`|z`); *one-or-more iterations*, R+ (as $R \cdot R*$); *optional regular expression*, R? (as $\varepsilon\,|\,R$); various *iterations* such as $R\{n\}$, $R\{n,\}$, and $R\{n,m\}$ (where, for instance, $R\{2,3\}$ corresponds to $R \cdot R \cdot R?$); and so on.



(a) *epsilon* (b) *char(c)* (c) *left(T)* (d) *right(T)*

(e) *concat(T_1,T_2)* (f) *star(T_1,T_2,...,T_n)*

Figure 1. The six different kinds of ASTs

2.1 Abstract Syntax Trees

Abstract Syntax Trees (ASTs) play a key role in our work, because regular expressions are used for structural substring matching and recording, not just for *yes/no*-recognition (aka., string membership). The regular expression operators give rise to six different kinds of ASTs (visualized in Figure 1):

$$\begin{array}{lll}
T & : & epsilon \;\mid\; char(c) \;\mid\; left(T) \;\mid\; right(T) \\
 & \mid & concat(T_1, T_2) \;\mid\; star(T_1, T_2, \ldots, T_n)
\end{array}$$

Figure 2 inductively defines the set of legal ASTs for a regular expression; we write $T \lhd R$, whenever T is a legal AST for a regular expression R. We denote by AST_R the set of all ASTs for R (i.e., $AST_R = \{\, T \mid T \lhd R \,\}$).

$$[\textsc{Epsilon}]\ \frac{}{epsilon \lhd \varepsilon} \qquad [\textsc{Char}]\ \frac{}{char(c) \lhd c}$$

$$[\textsc{Left}]\ \frac{T_1 \lhd R_1}{left(T_1) \lhd R_1\,|\,R_2} \qquad [\textsc{Right}]\ \frac{T_2 \lhd R_2}{right(T_2) \lhd R_1\,|\,R_2}$$

$$[\textsc{Concat}]\ \frac{T_1 \lhd R_1 \quad T_2 \lhd R_2}{concat(T_1, T_2) \lhd R_1 \cdot R_2}$$

$$[\textsc{Star}]\ \frac{T_1 \lhd R \quad T_2 \lhd R \quad \ldots \quad T_n \lhd R}{star(T_1, T_2, \ldots, T_n) \lhd R*}\ n \geq 0$$

Figure 2. Legal ASTs for regular expressions

We will also need a *flattening* operator, $\|\cdot\| : AST_R \to \Sigma^*$, which given an AST (for R), T, provides its corresponding string, $\|T\| = \omega$:

$$\begin{array}{rcl}
\|\, epsilon \,\| &=& \epsilon \\
\|\, char(c) \,\| &=& c \\
\|\, left(T) \,\| &=& \|\, T \,\| \\
\|\, right(T) \,\| &=& \|\, T \,\| \\
\|\, concat(T_1, T_2) \,\| &=& \|\, T_1 \,\|\ \|\, T_2 \,\| \\
\|\, star(T_1, T_2, \ldots, T_n) \,\| &=& \|\, T_1 \,\|\ \|\, T_2 \,\|\ \ldots\ \|\, T_n \,\|
\end{array}$$

Not surprisingly, we have that $\mathcal{L}(R) = \{\, \|T\| \mid T \in AST_R \,\}$.

2.2 The Recording Construction

We now extend the syntax of regular expressions with a *recording construction* for structural substring matching (aka., *capture variables*, cf. Section 8):

$$\texttt{<}x\texttt{=}R\texttt{>}$$

Here, x is an identifier taken from some finite alphabet of recording symbols. Semantically, the recording construction does not affect the language recognized; i.e.:

$$\mathcal{L}(\texttt{<}x\texttt{=}R\texttt{>}) = \mathcal{L}(R)$$

However, at runtime, matching produces a *side-effect* in that a substring matched by the regular expression R is *recorded*, the result of which can subsequently be dereferenced via the identifier x. (We will consider what happens if x is used in more than one recording shortly.) This is reflected in the syntax trees in that they will record the fact that x is associated with the AST for R:

$$[\textsc{Record}]\ \frac{T \lhd R}{record(x, T) \lhd \texttt{<}x\texttt{=}R\texttt{>}}$$

Flattening an AST recording node gives the string recorded for x:

$$\|\, record(x, T)\,\| = \|\, T\,\|$$

The recording construction is similar to the *capturing groups*, "(R)", known from Perl or `java.util.regex`. However, as we shall see later, our recording construction is much safer and much more flexible. For instance, a capturing group under a Kleene star will only record the last match. In contrast and as explained later, our construction will match multiple times and record *all* matches which will then be available as a *list*-structure.

As an example, consider the following recording-augmented regular expression of deliberately simplified email addresses (where we have underlined the recording identifiers):

```
<user = [a-z]+ > "@" <domain = [a-z]+ ("." [a-z]+)* >
```

When we match the above regular expression against the string "`obama@whitehouse.gov`", we get the following recordings: `user` = "`obama`" and `domain` = "`whitehouse.gov`".

Recordings can also be *nested* which gives rise to structured recordings as the following example shows:

```
<date =
    <day   = [0-9]{2} > "/"
    <month = [0-9]{2} > "/"
    <year  = [0-9]{4} >
>
```

Matching against the string "26/06/1992" will result in the recordings: `date.day = 26`, `date.month = 06`, `date.year = 1992`, but also `date = 26/06/1992`. The syntax "`x.y`" is just a naming convention for dereferencing a recording `y` nested within recording `x`. (We elaborate on structural matching in Section 4.)

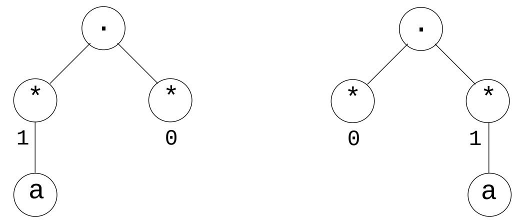
Finally, a recording can also give rise to *muliple values* which will be available as *lists*. This either happens if a regular expression contains multiple occurrences of the same recording, x, and/or if a recording is used under a star. The following example actually illustrates both cases; the recording, $name$, is used twice (one of which is under star). The regular expression will conveniently collect names (separated by ampersands) as a list:

```
<name = [a-z]+ > (" & " <name = [a-z]+ > )*
```

When matched against the string "`anna & bill & carl`", it will result in the *list* [`anna`,`bill`,`carl`] being recorded under the name, *`name`*. (We elaborate on list matching in Section 4.)

3. Ambiguity

We now define *ambiguity* of a regular expression:



(a) $concat(star(char(\texttt{a})),star())$ (b) $concat(star(),star(char(\texttt{a})))$

Figure 3. Two ASTs for string $\texttt{a} \in \mathcal{L}(\texttt{a*a*})$

DEFINITION 1 (Regular Expression Ambiguity).
A regular expression R is ambiguous *iff $\exists T, T' \in AST_R$ such that $T \neq T'$ and $\|T\| = \|T'\|$.*

For example, the regular expression, `a|a`, is ambiguous because it has two different ASTs for the same string, `a`:

$$\begin{array}{ccc} left(char(\texttt{a})) & \neq & right(char(\texttt{a})) \\ \|\, left(char(\texttt{a}))\,\| & = & \|\, right(char(\texttt{a}))\,\| \end{array}$$

Also, the regular expression `a*a*` is ambiguous, because it has two different ASTs for the same string, `a` (cf. Figure 3). The regular expressions, `a|aa` and `a*ba*`, on the other hand, are unambiguous.

Ambiguity is not a problem for *recognition*. That is, deciding the membership problem of whether or not a string, ω, is in the language defined by the regular expression, R (i.e., $\omega \in \mathcal{L}(R)$). However, it presents a real problem in the presence of recordings, when used for matching substrings. For example, when matching the regular expression `<x=a>|a` against the string `a`, gives rise to either *no match* for x or x=`a`. Similarly, matching `<x=a*>a*` against the string `aa`, x can ambigiously record either ϵ, `a`, or `aa`.

The ambiguity problem is *undecidable* for Context-Free Grammars [25], but *decidable* for Regular Expressions [3]. There exists a decision procedure for ambiguity of regular expressions via an ambiguity-preserving translation of regular expressions to Non-deterministic Finite Automata (NFAs) [3]. However, since ambiguities are reported in terms of NFAs, it is not easy to relate ambiguities back to the source of the problem in terms of the structure of an ambiguous regular expression.

3.1 Analysis of Ambiguity

$$[\textsc{Empty}]\ \frac{}{\Vdash \emptyset} \qquad [\textsc{Epsilon}]\ \frac{}{\Vdash \varepsilon} \qquad [\textsc{Char}]\ \frac{}{\Vdash c}$$

$$[\textsc{Choice}]\ \frac{\Vdash R_1 \quad \Vdash R_2}{\Vdash R_1 \,|\, R_2}\ \mathcal{L}(R_1) \cap \mathcal{L}(R_2) = \emptyset$$

$$[\textsc{Concat}]\ \frac{\Vdash R_1 \quad \Vdash R_2}{\Vdash R_1 \cdot R_2}\ \mathcal{L}(R_1) \bowtie \mathcal{L}(R_2) = \emptyset$$

$$[\textsc{Star}]\ \frac{\Vdash R}{\Vdash R*}\ \varepsilon \notin \mathcal{L}(R) \,\wedge\, \mathcal{L}(R) \bowtie \mathcal{L}(R*) = \emptyset$$

Figure 4. Analysis of Ambiguity

Figure 4 presents our *syntax-directed* analysis of ambiguity (inspired by previous work on context-free grammar ambiguity [4]) as a unary relation on regular expressions, $\Vdash\, \subseteq \mathcal{R}$, which inductively defines unambiguous regular expressions. The operator, $\bowtie$, is the so-called *language overlap* operator, $\bowtie : 2^{\Sigma^*} \times 2^{\Sigma^*} \to 2^{\Sigma^*}$,

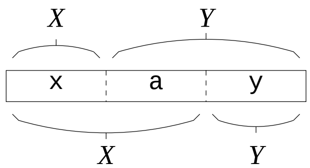


Figure 5. Illustration of string $\mathtt{xay} \in X \bowtie Y$.

defined by the following (and illustrated in Figure 5) where a is a non-empty word:

$$X \bowtie Y = \{\, xay \mid x, y \in \Sigma^* \wedge a \in \Sigma^+ \wedge x, xa \in X \wedge y, ay \in Y \,\}$$

The three base cases, $\emptyset$, ε, and c, are always unambiguous; in fact, the latter two only have one valid AST ($empty \triangleleft \varepsilon$ and $char(c) \triangleleft c$, respectively). The regular expression for choice, $R_1|R_2$, is unambiguous iff the two languages are disjoint (i.e., $\mathcal{L}(R_1) \cap \mathcal{L}(R_2) = \emptyset$). A concatenation, $R_1 \cdot R_2$, is unambiguous iff the two languages do not overlap (i.e., $\mathcal{L}(R_1) \bowtie \mathcal{L}(R_2) = \emptyset$). For Kleene star, $R*$, the unambiguity condition follows from the *unfolding equivalence*:

$$R* \quad \equiv \quad (R \cdot R*) \;\mid\; \varepsilon$$

It turns out that the analysis is both sound and complete and thus fully characterizes the ambiguity for regular expressions (which is captured by the following theorem):

THEOREM 1 (Characterization of Ambiguity).
A regular expression, R, is unambiguous iff $\Vert\!\models R$. (We refer to [5] for the proof of soundness and completeness of our analysis).

Note that ambiguity is inherently a *structural* problem depending on *how* a language is defined by a regular expression is defined, not on the *language* it defines. However, the above characterization permits a change of perspective in that it allows us to analyse ambiguity as a finite number of *linguistic* equations (of non-empty intersections and overlaps of *languages* and epsilon-containment). The total number of linguistic equations is linear in the size of the regular expression; there is exactly one equation for each of the composite constituents of the regular expression (i.e., one per *choice*, *concatenation*, and *star* construction).

The equations can easily be decided using automata, by inductively constructing the automata $\mathcal{L}(R)$, for all sub-expressions, R, in a bottom-up fashion. Further, since regular languages and automata are closed under intersection and overlap, we can get ambiguity violations (non-empty intersections and overlaps) in the form of automata, representing the possibly infinite set of ambiguous strings. From those, it is easy to extract *the* uniquely shortest (lexically least) ambiguous string and report it as along with the ambiguity warning/error. Here are three examples of such an ambiguity error being reported for the regular expressions, `(a|ab)·(ba|a)`, `a?b+|(ab)*`, and `(aa|aaa)*`, respectively:

```
*** ambiguous concatenation: (a|ab) <--> (ba|a)
    shortest ambiguous string: "aba"

*** ambiguous choice: a?b+ <-|-> (ab)*
    shortest ambiguous string: "ab"

*** ambiguous star: (aa|aaa)*
    shortest ambiguous string: "aaaaa"
```

Note that the ambiguity error is reported in terms of the *structure* of the regular expression and always with a concrete example. Obviously, such error messages make it easy for the programmer to locate the source of the ambiguity and take appropriate action (e.g., disambiguate, cf. Section 3.3).

The worst-case theoretical complexity of our analysis is exponential since we rely on minimized automata, however, this appears to not be a problem in practice. Also, the analysis can easily be optimized (e.g., by employing the techniques presented in Section 6 of [4]).

3.2 Ambiguous Recording

For a recording construction, <x=R>, it actually does not matter whether or not the regular expression, R, is internally ambiguous; that will not by itself give rise to multiple possibilities for x. Ambiguity, however, becomes a problem when a recording is put into a context alongside regular expressions that use one of the three constructions capable of introducing ambiguities. For instance, the regular expression, `<x=a>|a`, exemplifies this problem for *choice*; as previously explained, the string `a` can ambiguously give rise to either *no match* for x or x=`a`. For *concatenation*, this is exemplified by `a*·<x=a*>` where the string `a` can also ambiguously result in either x=ϵ or x=`a`. Finally, for *star*, `<x=a|aa>*`, the string `aaa` can ambiguously produce the following list recordings: x=[`aa`, `a`], x=[`a`, `aa`], and x=[`a`, `a`, `a`].

For this reason, we only have to analyze for ambiguity along the ancestor paths upwards from recording constructions (i.e., choices, concatenations, and stars above recordings in a given regular expression).

3.3 Disambiguation

Whenever the analysis pinpoints an ambiguity, the programmer has four ways of dealing with it: *i*) *manual rewrite* the regular expression; *ii*) use a *restriction* operator; *iii*) use *disambiguation directives*; and *iv*) ignore it and rely on *default disambiguation*.

i) Manual rewriting. The programmer can always rewrite the regular expression so that it is no longer ambiguous. In practice, however, this is sometimes cumbersome with regular expressions since they are declaratively and constructively specified which is why we have added three other options.

ii) Restriction. Regular expressions can be extended with a *restriction* operator, $R_1 \backslash R_2$, which is a convenient disambiguation tool by which unwanted possibilities can be explicitly ruled out. Note that restriction is inherently *constructive* (*intentional*) in its first argument in that it constructively gives rise to a value, whereas it is inherently *non-constructive* (*extensional*) in its second argument in that it does not give rise to a value (but merely filters out certain unwanted strings). Obviously, recordings do not make sense in non-constructive arguments. From restriction it is easy to define *complement*, $R^{\complement}$ (as $\Sigma^* \backslash R$) and then *intersection*, $R_1 \cap R_2$ (as $({R_1}^{\complement} \mid {R_2}^{\complement})^{\complement}$). Note that both operators are *non-constructive* in all their arguments. (Our tool supports intersection, complement, and restriction, but recordings are only permitted in the left operand of restriction.)

iii) Disambiguation directives. From our characterization of ambiguity we derive *left*- and *right*-disambiguated variants of the three operators that can potentially introduce ambiguities; i.e., $|_L$, $|_R$, $\cdot_L$, $\cdot_R$, $*_L$, $*_R$. Disambiguation is essentially a matter of choosing certain ASTs over others. This is conveniently done by introducing a partial order, "$\sqsubseteq$" on ASTs (i.e., "$\sqsubseteq$" $\subseteq AST \times AST$). For choice and concatenation we have the following rules:

$$\begin{array}{rcll} left(T_1) & \sqsubseteq & right(T_2) & \\ concat(T_1, T_2) & \sqsubseteq & concat(T_1', T_2') & \text{iff} \quad \Vert|T_1|\Vert \leq \Vert|T_1'|\Vert \end{array}$$

where $|\cdot|$ denotes the length of a string. The *left* disambiguators *minimize* the ordering; whereas the *right* disambiguators *maximize* the ordering (incidentally, $R_1 \,|_L\, R_2 \;\equiv\; R_1 \,|\, (R_2 \backslash R_1)$ and

$R_1 \,|_R\, R_2 \equiv (R_1 \backslash R_2) \,|\, R_2$). The star disambiguators are then easily defined in terms of the previous disambiguators:

$$\begin{array}{rcl} R{*}_L & \equiv & (R \cdot_L R{*}) \,|_L\, \varepsilon \\ R{*}_R & \equiv & (R \cdot_R R{*}) \,|_L\, \varepsilon \end{array}$$

Consistent use of only *left*-disambiguators corresponds to an *eager* (aka., *greedy*) matching strategy; whereas *right*-disambiguators yield *lazy* (aka., *reluctant*) matches. Note that all the disambiguation operators above are local and that a given regular expression may use combinations of all six variants, even in a nested fashion in which case they are resolved top-down on the ASTs.

iv) Default disambiguation. Any outstanding ambiguities are resolved using default disambiguation. In our tool, all constructions are, by default, *left*-disambiguated.

4. Typing

In this section, we show how to do type inference for regular expression recordings, independent of a host language. In order to be able to type check pattern matching usage in programming languages such as Java, we need to associate *two* kinds of information with it. First, we need a *linguistic type* that tells us what are possible strings that could be recorded as a result of a matching at runtime. This is used to verify that recorded regular expressions, such as `[0-9]+`, `0|[1-9][0-9]*`, or even `[-+]?[0-9]+`, can always safely be assigned to an integer-typed variable. Second, we need a *structural type* to tell us *how* recordings are nested within each other. This is used to make sure that `person.age` corresponds to a matched structure which does indeed have a recorded field with name `age` within a recorded field with name `person`. The typing uses the following mathematical structures:

$$\begin{array}{lcll} \mathcal{T} & : & \mathcal{L} \times \mathcal{S} & \textit{overall type} \\ \mathcal{L} & : & \mathit{Reg}_\Sigma & \textit{linguistic type} \\ \mathcal{S} & : & \mathit{Id} \hookrightarrow (\mathcal{T} \times \mathcal{M}) & \textit{structural type} \\ \mathcal{M} & : & \{\texttt{<0>}, \texttt{<1>}, \texttt{<?>}, \texttt{<*>}\} & \textit{type modifier} \end{array}$$

Below we define type inference for regular expressions with recordings in the form of a typing judgement $\vdash$ as a relation of type $\vdash \;\subseteq\; \mathcal{R} \times \mathcal{T}$. We will write $\vdash R : (L, S)$ as a short-hand for $(R, (L, S)) \in \;\vdash$, meaning that R is typeable with linguistic type L and structural type S. For each recording, x, the structure S, will tell us: $S(x) = ((l, s), m)$, where l is the regular language of all strings that can be matched at runtime by x; s is a structural type describing recordings that are nested inside x, and m is a type modifier telling us how many times the recording, x, can occur (e.g.: `<1>`, for exactly once; `<?>`, for zero or once; and `<*>`, for any number of occurrences). For instance, in the ampersand-separated name-list example from Section 2.2, the recording `name` will have structural type: $S = [\mathit{name} \mapsto ((\texttt{[a-z]+}, []), \texttt{<*>})]$.

Figure 6 specifies how to infer such types for recording-augmented regular expressions. The rules are all straightforward. Consistent with the semantics of regular expressions, the three axioms define the language component, L, as $\emptyset$, $\{\epsilon\}$, and $\{c\}$, respectively. Since none of them have recordings, they all have empty recording structures, $[]$. The composite rules, just propagate the *language* of a regular expression (according to the semantics of regular expressions), while delegating *choice*, *concatenation*, and *star* onto corresponding operations on structures, undertaken by: $\obar$, $\odot$, and $\circledast$ (which are defined in the following). The only rule that does something beyond delegation is [RECORD]. For a recording `<`x`=`R`>`, it creates a new recording structure, S', for the recording, x, as: $S' = [x \mapsto ((L, S), \texttt{<1>})]$; i.e., for which the language is L, and where its structure is that of R (i.e., S), and with the fact that x occurs exactly once (i.e., `<1>`).

$$[\textsc{Empty}]\ \frac{}{\vdash \emptyset : (\emptyset, [])} \qquad [\textsc{Epsilon}]\ \frac{}{\vdash \varepsilon : (\{\epsilon\}, [])}$$

$$[\textsc{Char}]\ \frac{}{\vdash c : (\{c\}, [])}$$

$$[\textsc{Choice}]\ \frac{\vdash R_1 : (L_1, S_1) \quad \vdash R_2 : (L_2, S_2)}{\vdash R_1 \,|\, R_2 : (L, S)}\ \begin{array}{l} L{=}L_1{\cup}L_2, \\ S{=}S_1{\obar}S_2 \end{array}$$

$$[\textsc{Concat}]\ \frac{\vdash R_1 : (L_1, S_1) \quad \vdash R_2 : (L_2, S_2)}{\vdash R_1 \cdot R_2 : (L, S)}\ \begin{array}{l} L{=}L_1{\cdot}L_2, \\ S{=}S_1{\odot}S_2 \end{array}$$

$$[\textsc{Star}]\ \frac{\vdash R : (L, S)}{\vdash R{*} : (L', S')}\ \begin{array}{l} L'{=}L^*, \\ S'{=}S{\circledast} \end{array}$$

$$[\textsc{Record}]\ \frac{\vdash R : (L, S)}{\vdash \texttt{<}x\texttt{=}R\texttt{>} : (L, S')}\ S' = [x \mapsto ((L, S), \texttt{<1>})]$$

Figure 6. Type inference

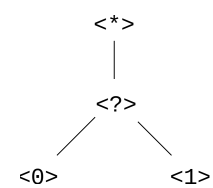


Figure 7. Partial-order among *type modifiers*.

The type modifiers induce a partial-ordering $\underset{M}{\sqsubseteq} \;\subseteq\; \mathcal{M} \times \mathcal{M}$ according to inclusions among the values they represent at runtime (cf. Figure 7). As usual, this order uniquely determines a least upper bound operator, $\underset{M}{\sqcup}$ which, not surprisingly, coincides with the choice (union) operator on modifiers, $\obar_m : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$ (defined in the following). Also concatenation on type modifiers, $\odot_m : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, is straightforward (and monotone):

$\obar_m$	<0>	<1>	<?>	<*>
<0>	<0>	<?>	<?>	<*>
<1>	<?>	<1>	<?>	<*>
<?>	<?>	<?>	<?>	<*>
<*>	<*>	<*>	<*>	<*>

$\odot_m$	<0>	<1>	<?>	<*>
<0>	<0>	<1>	<?>	<*>
<1>	<1>	<*>	<*>	<*>
<?>	<?>	<*>	<*>	<*>
<*>	<*>	<*>	<*>	<*>

Choice (union) on structures, $\obar : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$, becomes:

$(S_1 \obar S_2)(x) =$

$$\begin{cases} ((l_1 \cup l_2, s_1 \obar s_2), m_1 \obar_m m_2) & \text{if } x \in \mathit{dom}(S_1) \cap \mathit{dom}(S_2) \\ ((l_1, s_1), m_1 \obar_m \texttt{<0>}) & \text{if } x \in \mathit{dom}(S_1) \setminus \mathit{dom}(S_2) \\ ((l_2, s_2), m_2 \obar_m \texttt{<0>}) & \text{if } x \in \mathit{dom}(S_2) \setminus \mathit{dom}(S_1) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

where $((l_1, s_1), m_1) = S_1(x)$ and $((l_2, s_2), m_2) = S_2(x)$, whenever defined.

It basically performs a *least upper bound* operation on its constituents. For type modifiers, we union with `<0>` in cases where one of the structures in the choice does not have a recording. (The definition is slightly complicated by the fact that absent recordings are represented as *undefined* elements in the partial function S (i.e., $S(x) =$ *undefined*), rather than with explicit `<0>`-values as in: $[x \mapsto ((\emptyset, \perp_{\mathcal{S}}), \texttt{<0>})]$.)

Similarly, concatenation, $\bigodot : \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$, on structures is defined by:

$$(S_1 \bigodot S_2)(x) = \begin{cases} ((l_1 \cup l_2, s_1 \obar s_2), m_1 \bigodot_m m_2) & \text{if } x \in dom(S_1) \cap dom(S_2) \\ ((l_1, s_1), m_1 \bigodot_m \texttt{<0>}) & \text{if } x \in dom(S_1) \setminus dom(S_2) \\ ((l_2, s_2), m_2 \bigodot_m \texttt{<0>}) & \text{if } x \in dom(S_2) \setminus dom(S_1) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

where $((l_1, s_1), m_1) = S_1(x)$ and $((l_2, s_2), m_2) = S_2(x)$, whenever defined.

Also here we combine with `<0>`-type-modifier-values in cases where recordings are absent. Note that for nested structures, s, inside recordings, S, we have to take the least upper bound, $\obar$. This is because when determining the kinds of values x can have in a concatenation, <x=R_1>·<x=R_2>, we have to take the union of the possibilities. That is, x is typed as a *list* whose elements can be in: $\mathcal{L}(R_1) \cup \mathcal{L}(R_2)$; i.e., the *union* of the two recordings of the same name. (This is analogous to typing a heterogeneous Java list, [`1, 2.5`], as `double[]`, because the least upper type bound on its elements yields: `double` = `int` $\sqcup$ `double`.)

Star on structures, $\circledast : \mathcal{S} \rightarrow \mathcal{S}$, is straightforward (it always produces *lists* of recordings, `<*>`):

$$(S\circledast)(x) = \begin{cases} ((l, s), \texttt{<*>}) & \text{if } x \in dom(S) \\ \textit{undefined} & \text{otherwise} \end{cases}$$

where $((l, s), m) = S(x)$, whenever defined.

Since regular languages are closed under *union*, *concatenation*, and *star*, the *language* part of the structure is exact.

The above type analysis will infer a type for a recording, <x=R>; say: $[x \mapsto ((L, S), m)]$. This means that the language of the regular expression recorded is, L (i.e., $L = \mathcal{L}(R)$); the structure, S, gives the typings of all sub-recordings nested within the recording (i.e., inside R); and the modifier, m, tells us how many times the recording can match at runtime as an interval (e.g., `<*>` means *zero or more*). In practice, the type inference provides the most specific Java type for recordings in all of our examples (cf. Section 5). The most specific type is found via deciding language containment via DFAs using the predefined regular expressions for the atomic types of Java (see below).

4.1 Host-language Embedding

After type inference, we have a three components for each recording, x, $[x \mapsto ((L, S), m)]$; a *linguistic type*, L, a *structural type*, S, and a *type modifier*, m. In the following we will explain how each of these can be used to provide a non-intrusive language embedding for Java.

Embedding linguistic types. The linguistic type is used to provide the most *specific* Java base type for a recording with respect to the following predefined regular expressions:

```
String  =  .* ;
char    =  . ;
float   =  [+-]?[0-9]*\.[0-9]+(E[+-]?[0-9]+)? ;
int     =  [+-]?[0-9]+ ;
boolean =  [Tt][Rr][Uu][Ee]  |  [Ff][Aa][Ll][Ss][Ee]  ;
```

Integers and characters are the only sets that overlap incomparably (on `[0-9]`). In that case, we prefer `int` over `char`. The most specific type means that for instance, `[0-9]+` is mapped to `int`, and not `float` or `String` (even though it is also contained in those).

Note that this ignores overflow in that the regular expression "`0|[1-9][0-9]+`" becomes a Java `int` (irrespective of overflow). The tool can be instructed to handle 32-bit integers (via the option `"-I 32"`) which replaces the above `int` with the appropriately bounded regular expression.

However, sometimes it is convenient to have "`00011101`" typed as a `String` instead of an integer (to prevent it from being mapped onto an integer as the number `11101`). To address this, we permit an optional *type override* annotation in the syntax for recordings as in "<τ x = R>", where τ is one of the atomic types mentioned above. This construction will be statically checked to verify that $\mathcal{L}(R) \subseteq \mathcal{L}(\tau)$. The linguistic type inferred will then be $\mathcal{L}(\tau)$. This means that the recording `bit8` in the regular expression: "`<String bit8 = [01]{8}>`", will result in a structure where the type of `bit8` is `String` such that the leading zeros in the above string will not disappear.

Embedding structural types. Each regular expression definition gives rise to a Java class and each of its toplevel recordings gives rise to field variables with appropriate types and type modifiers (as explained above). Consider for example, the following regular expression (`$Name` denotes inlining of the regular expression defined as `Name`):

```
Name = [a-z]+ ;
Person = <name = $Name > "(" <age = [0-9]+ > ")" ;
```

This will give rise to the following Java class:

```
class Person { // auto-generated
    String name;
    int age;

    String value;
    public static Person match(String s) { ... }
    public String toString() { ... }
}
```

Classes will always contain: a field, `value` (to hold strings matched at runtime); a static method, `match`, which takes a `String`, parses it according to the regular expression (stores it in `value`) and returns it; and a public `toString()` method (that returns `value`). Nested recordings give rise to nested Java classes (as shown in Section 5).

Embedding type modifiers. Type inferred modifiers (i.e., `<1>`, `<?>`, and `<*>`) give rise to Java type modifiers in a fairly straightforward way. Consider the following regular expression:

```
Points = $Name ":" <point = [0-9]+ >
         ("," <point = [0-9]+ > ) * ;
```

The recording, `point`, will be typed with a `<*>` type modifier and thus give rise to the field declaration: `int[] point;` in a Java class, `Points`. Since Java does not have *optional types* (as for instance Haskell), we also map `<?>`-types to arrays. (`<?>`-types could also be handled via the `null`-value, if we box primitive types; or by creating a special Java class with, say, an `isDefined()`-method.)

5. Examples

In the following, we demonstrate that our approach can be used to perform string-based pattern matching on realistic non-trivial examples in a purely declarative manner. We use it on URLs, Apache log files, and the DBLP publication database.

5.1 URLs

URLs are highly structured and pack a lot of different kinds of information used by many modern applications. As an example, consider the following URL:

```
http://www.google.com/search?q=record&hl=en
```

It contains a *protocol* (`http`), a *host* (`www.google.com`) a *path* (`search/`), and a *query string* (`q=record&hl=en`). The following (deliberately simplified) recording-augmented regular expression conveniently extracts these pieces of information:

```
Host  = <host = [a-z]+ ("." [a-z]+)* > ;
Path  = <path = [a-z/.]* > ;
Query = <query = [a-z&=]* > ;
URL   = "http://" $Host "/" $Path "?" $Query ;
```

The first three lines defines appropriate regular expression for *hosts*, *paths*, and *queries*. Note that they all contain recordings at the outermost level, that are all typed as `String`, and which will record the corresponding string values at runtime. The last line defines `URL` to be the constant prefix '`http://`', followed by the regular expression for `Host` (containing a recording), a constant slash character '`/`', whatever is defined for `Path`, a question mark '`?`', and finally the regular expression for *queries*. At runtime, when matched against a URL string, it will record all the relevant substrings.

We now improve on the regular expression by also extracting so-called *key-value pairs*, redefining `Query` as:

```
KeyVal = <key = [a-z]* > "=" <val = [a-z]* > ;
Query  = $KeyVal ("&" $KeyVal )* ;
```

Since the recordings, `key` and `val`, occur under a star, they will be typed (by the type inference, cf. Section 4) as *lists* of strings (i.e., `String[]` in Java). This means that the regular expression for URLs can now be used in Java in the following way; e.g.:

```
String u = "http://www.google.com/search?q=record&hl=en";

URL url = URL.match(u);
print("Host is: "  + url.host);
print("Path is: "  + url.path);
if (url.key.length>0) print("1st key is: "  + url.key[0]);
for (String val : url.val) print("Value = "  + val);
```

The method invocation, `URL.match(u)`, parses the string contained in `u` and the results is in the above example stored in the variable `url` of type `URL`. The parsing method and all of the classes have been automatically generated by the `reg-exp-rec` compiler from the regular expression. Note that all recordings are available as fields of that object (e.g., `url.host` contains whatever was matched as the hostname). Also, `url.key` and `url.val` are Java lists and can be transparently used as such. Note that if we wanted the keys and values as a list of *pairs*, we could simply wrap the keys and values in a pair-recording as in the following definition:

```
KeyVal = <pair = <key = [a-z]* > "=" <val = [a-z]* > > ;
```

5.2 Log files

Our next example deconstructs Apache's HTTP log files which look like:

```
13/02/2010 66.249.65.107 get /support.html
20/02/2010 42.16.32.64 post /search.html
```

This can easily be handled by the regular expression, reusing `Path` from above (assuming appropriate definitions of `Day` and `Month`, see later):

```
IP     = <ip = [0-9]{1,3} ("." [0-9]{1,3} ){3} > ;
Method = <method = "get"|"post" > ;
Date   = <date =
             <day = $Day > "/"
             <month = $Month > "/"
             <year = [0-9]{4} >
         > ;
Entry  = <entry = $Date " " $IP " " $Method " " $Path > ;
Log    = $Entry * ;
```

Since the *year* is defined by the regular expression, `[0-9]{4}`, it will be typed as a Java integer (i.e., `int`), as the generated code shows. The nested recordings give rise to nested inner classes:

```
public class Log { // auto-generated
    Entry[] entry;
    String value;

    public static Log match(String s) { ... }
    public String toString() { ... }

    public class Entry {
        String ip, method, path;
        Date date;
        String value;

        public String toString() { ... }

        public class Date {
            int day, month, year;
            String value;

            public String toString() { ... }
        }
    }
}
```

Now, we can, for instance, report all *leap day* accesses from a log file in the following way:

```
Log log = Log.match(log_file);
for (Entry e : log.entry)
    if (e.date.month == 02 && e.date.day == 29)
        print("Access on LEAP DAY from IP#: " + e.ip);
```

This type of use can be applied to a wide range of data formats (e.g., Java property files and Unix password files). Note that if we accidentally forget the slash between the day and month, our ambiguity checker complains and pinpoints the error with *the* shortest (lexicographically least) concrete ambiguous string:

```
*** ambiguous concatenation: <day> <--> <month>
    shortest ambiguous string: "101"
```

The error message tells us that the date can either be interpreted as January 1 (i.e., `1/01`) or January 10 (i.e., `10/1`). Here, we assumed that day and month are defined more sensibly than just `[0-9]+` (e.g., `Day = 0?[1-9] | 10 | 11 | 12` and `Month = 0?[1-9] | [1-2][0-9] | 30 | 31`). (If we were to define *day* and *month* more sloppily by, say, `[0-9]+`, then the ambiguous string reported would instead become: "`000`".)

5.3 DBLP

Our last example shows that our approach can provide safe, typed, and structural pattern matching to even highly structured XML data. As long as the structure has bounded depth which is the case for DBLP data, we are able to make a specification via regular expressions. Here is a hypothetical example of data in the DBLP format describing for two papers (one of which is published [8]):

```
<article>
    <author>Noam Chomsky</author>
    <title>Three Models for the
           Description of Language</title>
    <year>1956</year>
    <journal>IRE Transactions on
             Information Theory</journal>
</article>
<inproceedings>
    <author>Claus Brabrand</author>
```

```
    <author>Jakob G Thomsen</author>
    <title>Typed and Unambiguous Pattern Matching
           on Strings using Regular Expressions.</title>
    <year>2010</year>
    <booktitle>PPDP 2010</booktitle>
</inproceedings>
```

Again, this format is easily captured by our approach (even though it is rigourously structured XML):

```
Author = "<author>" <author = [a-z]* > "</author>" ;
Title  = "<title>" <title = [a-z]* > "</title>" ;
Article     = "<article>"
                   $Author* $Title .*
                "</article>" ;
Proceeding  = "<inproceedings>"
                   $Author* $Title .*
                "</inproceedings>" ;
Publication = $Proceeding | $Article ;
DBLP        = <pub = $Publication > * ;
```

This regular expression is ambiguous as pinpointed by our analysis:

```
*** ambiguous star: <pub>*
    shortest ambiguous string:
    "<article><title></title></article>
         <article><title></title></article>"
```

The string reported could either match, as intended, *two* distinct articles (under the star); or, it could match *one* article with an empty title (for which: "`</article><article><title></title>`" has unintendedly been eaten by the "`.*`" in `Article`).

As mentioned in Section 3.3, there are several ways to disambiguate. Using restriction (which is the binary infix minus operator in our tool; i.e.: R_1-R_2), `Article` can be re-written to disallow the unintended interpretation:

```
Article = "<article>"
              $Author* $Title (.* - (.* "</article>" .*))
          "</article>" ;
```

Note that since `Article` and `Proceeding` both have a `title` with language `[a-z]*` and type modifier `<1>`. This means that so will a publication (due to the definition of "$\obar$ " and "$\obar_m$", cf. Section 4); since $\mathcal{L}($`[a-z]*`$) \cup \mathcal{L}($`[a-z]*`$) = \mathcal{L}($`[a-z]*`$)$ and `<1>` $\obar_m$ `<1>` = `<1>`. Similarly, the recording `author` becomes the language $\mathcal{L}($`[a-z]*`$)$ with type modifier `<*>`. As a consequence, a publication then always has a *title* field, `title`, of type `String` and an *author* field, `author`, of type `String[]` (independent of whether it was an article or an inproceeding). The following code prints the titles of all publications:

```
DBLP dblp = DBLP.match(readXMLfile("DBLP.xml"));
for (Publication publication : dblp)
    print("Title: " + publication.title);
```

For more examples (including the iCalendar format), we refer to the the project homepage:

[http://www.cs.au.dk/~gedefar/reg-exp-rec/]

6. Parsing

Deterministic Finite Automata (DFAs) provide an efficient way of deciding the membership problem for Regular Expressions; i.e., given a string, ω, and a regular expression, R, deciding whether or not: $\omega \in \mathcal{L}(R)$. A regular expression can be compiled into a Finite Automaton [25], which can then be used to decide the membership problem at runtime in linear time in the size of the input string, $O(|\omega|)$. This works for recognition (i.e., *yes*/*no*-answers), but it does not work for recording and extracting substring matches since it does not provide structural information of *how* the string matched the regular expression.

Of course, determining structural information (syntactic analysis) is precisely the objective of *parsing*. However, most efficient parsing algoritms are devised for (Type-2) context-free grammars, *not* (Type-3) regular expressions.

6.1 Regular Expression Parsing

Regular Expressions can also be interpreted structurally using a *backtracking* algorithm which is a popular strategy in many languages (e.g., Perl, PHP, Python, Ruby, and `java.util.regex`). However, that strategy runs in $O(2^{|\omega|})$; matching, for instance, `a?{`n`}a{`n`}` against $\mathtt{a}^n$ runs *one million* times slower in Perl than an NFA-based approach, clocking in at 60 seconds for just n=29 [11].

There have been several attempts to extend automata with recording capabilities, but many are not able to deal with recordings under iteration [13, 30]. In [21], Frisch and Cardelli show how to perform greedily disambiguated pattern matching on a string, ω, via an automata-based approach while retaining structure (i.e., parsing) in linear time, $O(|\omega|)$. Nielsen and Henglein [34] have recently invented a simpler approach inspired by a *sound and complete axiomatization of regular expressions* [24] that is also based on DFAs and also runs in linear time.

Parsing Expression Grammars [19] (aka., PEGs) support EBNF-style regular expression operators (even intersection and complement) and have linear-time implementations (e.g., Packrat Parsing [18]). However, they have a *greedy, non-backtracking* semantics which is not able to deal with many regular expressions (e.g., it is incapable of matching against, `a*a`, as it will *never* abandon consuming `a`s under the star).

6.2 Context-Free Grammar Parsing

It is possible to use a context-free grammar parser for parsing regular expressions via *structure-preserving* transformations. A regular expression can be transformed into a CFG in such a way that parse-trees can be mapped back to the original structure of the regular expression. The Earley algorithm [12] runs in $O(|\omega|^3)$ (in $O(|\omega|^2)$ for deterministic grammars and in $O(|\omega|)$ for certain simple grammars). Note that the problem of CFG parsing reduces to matrix multiplication [31] for which the currently (although impractical) best-known worst-case complexity is $O(|\omega|^{2.376})$ [9]. In our tool tool, however, we currently rely on a CFG parser that uses a variant of Earley's algorithm [33].

Although linear-time CFG parsers exist (e.g., LL(k) and LR(k) can be done in $O(|\omega|)$), they work only for limited subsets of (unambiguous) grammars. In practice, regular expressions are often ambiguously specified and is thus something one needs to handle. (Recall that we automatically left-disambiguate any ambiguous operators.)

7. Evaluation

In this section we will evaluate our approach and relate it to other similar techniques. We have divided the evaluation of our approach into three dimensions: *flexibility*, *safety*, and *efficiency*. We will contrast each of these to approaches based on capturing groups (as used in e.g., Perl and `Java.util.regex`), CFG-based approaches, and unrestricted Turing-Complete programming. The evaluation is summarized in Figure 8.

Note that there are lots of parser generator tools (e.g., Yacc [29], SableCC [23], ANTLR [35]), but they are all based on CFGs and relate to our approach as depicted in Figure 8 and explained below. (The same applies for the source transformation language, TXL [10].)

7.1 Flexibility

The examples from Section 5 show that our recording construction is sufficiently expressive for extracting structured information from

Criteria \ *Approach*		**Our Approach:** (`reg-exp-rec`)	**Capturing Groups:** (Perl/`java.util.regex`)	**CFG-Based Approaches:**	**Turing-Complete Programming:**
Expressivity (language class):		Regular Language (Type-3)	?	Context-Free Language (Type-2)	Recursively Enumerable (Type-0)
Flexibility	Declarative specification:	+	+	+	−
	Information extractable:	Tree(s) (bounded depth)	String(s) (fixed number)	Tree (unbounded depth)	Anything computable
	Structured values:	+	−	+	+
Safety	Guaranteed termination:	+	+	+	−
	Ambiguity decidable:	+	−	−	−
	Containment decidable:	+	−	−	−
Efficiency	Complexity of parsing:	$O(\|\omega\|)$	$O(2^{\|\omega\|})$	$O(\|\omega\|^{2.376})$	N/A

Figure 8. Evaluation summary of recording approaches (string-based pattern matching and extraction).

strings via declarative pattern matching based entirely on regular expressions. Here, we will argue that it is also simple and convenient (especially when contrasted to alternatives available).

Capturing groups. A capturing group is a construction for substring recording like our recordings. Syntactically, it is written as a set of parentheses the around a regular expression. Whatever is matched by the construction is then captured and is subsequently available as the regular expression; e.g., "$\backslash n$" (where `\1` refers to the lexically first capturing group, `\2` to the second, and so on). The construction is thus capable of matching any finite number of substrings. However, if a star is used around a capturing group, it will only record the *last* substring matched. Thus, it cannot be used for unbounded list matching. Also, the individual matches have no structure beyond that of a string. In contrast our construction is specified syntactically via identifiers (which is more descriptive than `\7`) and it can be used to match and record values over any tree structure with bounded depth).

Furthermore, capturing groups often come with so-called *back-references* which take them beyond regularity. They permit a dynamically recorded substring to appear in an otherwise static regular expression. For instance, the regular expression: `(a*)b\1`, matches the non-regular (but context-free) language: $\{\, \texttt{a}^n \ \texttt{b} \ \texttt{a}^n \mid n \geq 0 \,\}$. In fact, expressivity using back-references takes them beyond context-free languages; e.g.: the language $\{\, \omega c \omega \mid \omega \in \Sigma^*, c \in \Sigma \,\}$ is not context-free, but easily recognizable using back-references: `(.*).\1`. (It is unclear exactly what *class of languages*, abstractly speaking, is recognized by regular expressions with back-references.) Note also that the membership problem for regular expressions with back-references is NP-complete [7].

Context free grammars. Obviously, context free grammars are more expressive than regular expressions, but fewer properties are decidable (see Section 7.2 below). Grammars have recursive nonterminals which then make them capable of parsing tree structures of unbounded depth.

We hypothesize that regular expressions are easier to comprehend and use for novice programmers than context free grammars. Grammars are essentially regular expressions plus recursive nonterminals. (Curiously, although in no respect a valid scientific argument, a search on Google for "`regular expression`" vs. "`context free grammar`" reveals a factor of 13:1 in favor of regular expressions; context-free grammars "only" generate about a million hits (as of March 2010).

Finally, since CFGs are not closed under restriction, we cannot use restriction for disambiguation.

Unrestricted programming. Obviously any computable function can be expressed in a (*non-declarative*) Turing-Complete programming language such as Java. For a flexibility comparison of our approach versus that of a Type-0 language (Java), we have made a quantitative analysis of source code. We have compared the *full* URL specification written as a recording-augmented regular expression versus the standard implementation in Java, `java.net.URL` (version 1.136; April 30, 2009). Figure 9 lists the results of this comparison. For the code size, we see a *conciseness factor* of about almost 1:8 (45:347) in favor of our declarative approach. If we look at the *cyclomatic complexity* [36], we see that the total number of *branches* in the two formalisms (taken as choices for the regular expressions and `if`-conditionals in Java), we see a factor of 1:2. As for *iterations* (taken as R`*` and R`+` versus `while` and `for` loops), we see a 3:1 in favor of Java. However, a `while`-loop is operational and much more complicated than a simple and declarative Kleene star. In addition, the Java implementation has a whole range of highly operational features (96 method invocations, 9 `throw` statements, and 1 `try` statement with 2 `catch` handlers).

Curiously, and perhaps symptomatic of this code complexity, the current official implementation contains a known, but unresolved bug as indicated by a "`//FIX:`" comment in the source code. Figure 10 displays a short sample of bugs for `java.net.URL` (extracted from Sun's bug repository, `http://bugs.sun.com/`). The history of bugs spans a decade, which we take as an indication of complexity inherently associated with the (non-declarative) Turing-Complete approach for this kind of pattern matching. Obviously, some "errors" are due to natural evolution of the specification (i.e., adding new features), but the bug repository reveals no less than 88 bug entries for `java.net.URL`, a lot of which are real implementation bugs. We also conclude that *maintainability* (for which *legibility* is a prerequisite) is crucially important for this kind of pattern matching.

In contrast, we were able to use the official RFC for URLs [15, 16] directly and augment it with all relevant recordings in less than half an hour. Obviously, the regular expression version is much more concise, but also superior in terms of legibility and maintainability.

Feature	Regular Expressions		Programming (Java)	
code size	45 lines		347 lines	
branches	24 "`|`"	(choices)	50 `if`	(conditionals)
iterations	12 "`*`"	(stars)	5 `while`	(loops)
	6 "`+`"	(plusses)	1 `for`	(loop)
methods			96 $m(..)$	(invocations)
exceptions			9 `throw`	(errors)
			2 `catch`	(handlers)
			1 `try`	(handler)
bugs			1 unresolved "`fixme`" !	

Figure 9. Regular expressions vs. unrestricted programming (code: "`java.net.URL`") for URL matching.

Year	Bug ID	Synopsis
1998	4175737	`java.net.URL` should throw `MalformedURLException` on incorrect FILE
1999	4221439	`https` seen as invalid protocol by `java.net.URL`
1999	4264177	`HttpURL` does not handle URL of the format `http://user:passwd@host/`
...	...	...
2006	6506304	`java.net.MalformedURLException:` unknown protocol: c

Figure 10. Small sample of bugs for "`java.net.URL`" (source: "`http://bugs.sun.com/`" bug repository).

7.2 Safety

Safety is the paramount benefit of our approach. Termination is inherently guaranteed by the formalism; ambiguity is statically decidable with constructive counter-examples (cf. Section 3); it is possible to statically infer the language of recordings exactly (i.e., without precision loss); and language containment (i.e., $\mathcal{L}(R_1) \subseteq \mathcal{L}(R_2)$) is decidable. The latter makes it is possible to pick the most appropriate type among the types available in a host language, so that we can type a recording as, say, a list of integers. The analysis only abstracts away information that the Java type system does not address (e.g., the number of elements in an array and the exact types of individual elements in heterogeneous constant arrays such as: $[1, 2.5]$).

Termination is also guaranteed in the non-Turing-Complete alternatives, but this is as far as the other approaches go in terms of safety guarantees. This is because our approach is built on pure regular expressions (without non-regular extensions such as back-references), which means that virtually everything is not only decidable, but constructively so.

7.3 Efficiency

Section 6 discussed the worst-case asymptotic complexities of the different approaches (including that of the worst-case exponential-time interpretation-based strategy used in many language implementations). Note that for a Turing-Complete programming language, the worst-case complexity depends intimately on the particular language to be parsed and the algorithm implemented.

Figure 11 plots the result of parsing incorrect (i.e., rejecting) DBLP data strings of increasing length (along the x-axis) using Java's `java.util.regex` vs. our CFG-based approach. The numbers along the left hand side of the y-axis indicate the time for `java.util.regex` in seconds; along the right hand side of the y-axis, the time for our approach in milliseconds on a standard laptop PC (Intel Core 2 duo, 2.2 GHz, 2 GB memory, Windows). `java.util.regex` hits exponential growth on this example; parsing (and rejecting) a string of about 2,500 characters takes about two minutes (120 seconds). Our approach, on the other hand, exhibits a linear behavior and parses strings of 20,000 characters in about 0.06 seconds. In fact, our approach is able to parse (and reject) strings of 1.2 mio characters in about 6 seconds. In 6 seconds, it is able to parse correct DBLP data of strings of up to 800,000 characters.

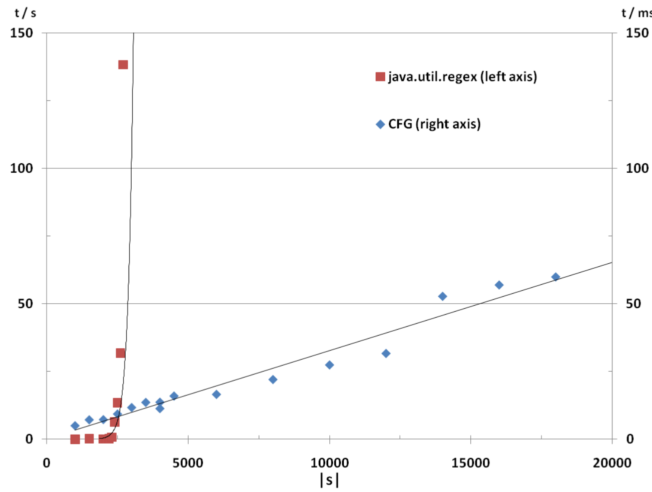


Figure 11. Parsing incorrect (i.e. rejecting) DBLP data using `java.util.regex` (in seconds) vs. our CFG-based approach (in milli-seconds).)

Although we see a mostly linear behavior, we should be able to speed up our tool using an automaton-based approach (e.g., [21, 34]).

8. Related Work

XML uses regular expressions as types constraining sequences of elements and for this reason regular expression pattern matching has been popular in XML-friendly languages such as XDuce [27, 28] and CDuce [2, 22]. The multi-paradigmatic language Scala [14] also features regular expression pattern matching and HaRP [6] adds regular expression patterns to Haskell via a lightweight preprocessor. All of these languages offer recording constructions, capable of recording lists that are more general than lists of characters (via the "x `as` R" construction in XDuce, "x `::` R" in CDuce, and "x`@`R" in Scala and HaRP).

In CDuce, regular expressions are added as syntactic sugar on top of an ML-style pattern matching (augmented with intersection, XML-friendly, and type-matching operators). This is achieved by compiling the regular expression patterns into Finite Automata which are then encoded as ML-style pattern matching (in normal form with cons-lists encoded as binary pairs). The paper [20] describes how to use automata in the compilation of regular expressions to core pattern matching, but only in terms of *recognition* (i.e., without recordings).

XDuce statically checks for ambiguities based on a product construction for tree automata [26]. However, since ambiguities are found in terms of tree automata, it is not easy to relate ambiguities back to the source of the problem in terms of the original regular expression. Our analysis, on the other hand, is syntax-directed, pinpoints the exact location of all ambiguities, and provides a unique shortest (and lexicographically least) ambiguous string in the case of ambiguities.

CDuce relies on disambiguation with left-bias for choice and with *greedy* iteration (but has a *non-greedy* variants). HaRP employs a *first-match* policy and disambiguates iteration *non-greedily* (but also has a *greedy* variant). Disambiguation policies can be subtle and are studied formally in [37] (including the POSIX and the *first-and-longest* match policies). These policies are all *global*. In contrast, we have six *locally* disambiguated regular expression operators that are all related to the potential origins of ambiguities (via the characterization), including two for concatenation. These are more general in that different disambiguation conventions can be used at different points in a regular expression.

XDuce and CDuce both have exact type inference and [37] provides a sound-and-complete proof of their regular expression type inference. We use our type inference, which provides the language and nesting structure of all recordings, to synthesize stand-alone pattern matching and container classes for Java in a stand-alone and non-intrusive way.

Finally, there are a number of other tools that are also made for processing unstructured or ad-hoc data (e.g., PADS [17]), but they go beyond regularity; hence, they are more expressive, but fewer properties are statically decidable.

9. Conclusion

We conclude that *if* regular expressions are sufficiently expressive, they provide a simple, flexible, and safe means for declarative pattern matching on strings, capable of extracting highly structural information in a statically type-safe and unambiguous manner. Also, regularity is enough for many realistic pattern matching tasks such as extracting structural information from URLs, log files, and even highly structured XML data of bounded depth (e.g., DBLP).

We have shown how to statically analyze ambiguity of regular expressions through a sound and complete analysis that pinpoints all ambiguities in terms of the structure of the regular expression, capable of reporting the shortest ambiguous string. If the grammar is ambiguous, we provide four ways in which it can be safely and locally disambiguated. We have also shown how to statically infer the type of structural information extracted which leads to convenient integration of pattern matching in languages such as Java in a completely stand-alone and non-intrusive way with acceptable runtime performance.

In conclusion, we have shown how expressivity can be traded for simplicity and static safety in the case of pattern matching on strings.

Acknowledgments

The authors would like to thank Peter Sestoft, Philip Bille, Michael Schwartzbach, Jurgen Vinju, Pierre-Etienne Moreau, Fritz Henglein, and Lasse Nielsen for valuable comments and suggestions. We would also like to thank the reviewers of PPDP 2010 for their comments, suggestions, and bibliographical indications.

References

[1] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 2007.

[2] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. Cduce: an XML-centric general-purpose language. In *ICFP*, pages 51–63, 2003.

[3] Ronald Book, Shimon Even, Sheila Greibach, and Gene Ott. Ambiguity in graphs and expressions. *IEEE Transactions on Computers*, 20(2):149–153, 1971.

[4] Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3), 2010. To appear. Earlier version in Proc. 12th International Conference on Implementation and Application of Automata, CIAA '07, Springer-Verlag LNCS vol. 4783.

[5] Claus Brabrand and Jakob G. Thomsen. Typed and unambiguous pattern matching on strings using regular expressions. Technical report, IT University of Copenhagen, 2010.

[6] Niklas Broberg, Andreas Farre, and Josef Svenningsson. Regular expression patterns. In *ICFP*, pages 67–78, 2004.

[7] C. Campeanu and N. Santean. On pattern expression languages. In *Proceedings AutoMathA*, 2007.

[8] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2:113–124, 1956.

[9] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[10] James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: a rapid prototyping system for programming language dialects. *Comput. Lang.*, 16(1):97–107, 1991.

[11] Russ Cox. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby, ...). `http://swtch.com/ rsc/regexp/regexp1.html`, January 2007.

[12] Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[13] Burak Emir. Compiling regular patterns to sequential machines. In *SAC*, pages 1385–1389, 2005.

[14] Martin Odersky et al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[15] T. Berners-Lee et. al. Uniform resource locators (`URL`). `http://www.ietf.org/rfc/rfc1738.txt`, 1994.

[16] T. Berners-Lee et. al. Uniform resource identifier (`URI`). `http://www.ietf.org/rfc/rfc3986.txt`, 2005.

[17] Kathleen Fisher and Robert Gruber. Pads: a domain-specific language for processing ad hoc data. *SIGPLAN Not.*, 40(6):295–304, 2005.

[18] Bryan Ford. Packrat parsing: simple, powerful, lazy, linear time. In *In Proc. International Conference on Functional Programming (ICFP'02)*, pages 36–47, 2002.

[19] Bryan Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Symposium on Principles of Programming Languages (POPL'04)*, pages 111–122. ACM Press, 2004.

[20] Alain Frisch. Regular tree language recognition with static information. In *IFIP TCS*, pages 661–674, 2004.

[21] Alain Frisch and Luca Cardelli. Greedy regular expression matching. In *ICALP*, pages 618–629, 2004.

[22] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping. In *LICS*, pages 137–146, 2002.

[23] Etienne M. Gagnon and Laurie J. Hendren. Sablecc, an object-oriented compiler framework. *Technology of Object-Oriented Languages, International Conference on*, 0:140, 1998.

[24] Fritz Henglein and Lasse Nielsen. Declarative coinductive axiomatization of regular expression containment and its computational interpretation (preliminary version). Topps d-report, Department of Computer Science, University of Copenhagen (DIKU), February 2010.

[25] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, April 1979.

[26] Haruo Hosoya. Regular expression pattern matching - a simpler design. Technical report, 2003.

[27] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for XML. In *POPL*, pages 67–80, 2001.

[28] Haruo Hosoya, Jerome Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *Proc. 5th ACM SIGPLAN International Conference on Functional Programming, ICFP '00*, September 2000.

[29] J.R.Levine, T. Mason, and D. Brown. Lex and yacc. O'Reilly and Associates, Inc., Sebastopol, CA, 1992.

[30] Ville Laurikari. Nfas with tagged transitions, their conversion to deterministic automata and application to regular expressions. In *SPIRE*, pages 181–187, 2000.

[31] Lillian Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, 2002.

[32] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[33] Anders Møller. `dk.brics.grammar`: Context-free grammars for java. `http://www.brics.dk/grammar/`, 2006.

[34] Lasse Nielsen and Fritz Henglein. Parsing with dfas (preliminary version). Topps d-report, Department of Computer Science, University of Copenhagen (DIKU), May 2010.

[35] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007.

[36] James F. Power and Brian A. Malloy. A metrics suite for grammar-based software: Research articles. *Journal of Software Maintenance and Evolution*, 16(6):405–426, 2004.

[37] Stijn Vansummeren. Type inference for unique pattern matching. *ACM Trans. Program. Lang. Syst.*, 28(3):389–428, 2006.